Interactive Shape Editing

Luo Xiawu

SCHOOL OF COMPUTER ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

This report is presented for the master degree on Digital Media Technology program of Nanyang Technological University

2014

Contents

Contents of Figures

Contents of Tables

Abstract

Three-dimensional geometric models are the base data for applications in computer graphics, computer aided design, visualization, multimedia, and other related fields, the state-of-the-art interactive shape editing techniques become very popular in recent years, under such condition, we propose this dissertation to perform some research on this area.

The dissertation firstly introduces the background of the interactive shape editing technique, and states the problem in this research area, based on this, we propose the objective and scope of the research, and we draft the organization of this dissertation.

Secondly, we investigate the related works on "Interactive shape editing" and "free-form deformation (FFD)", by reading most relative papers, we know what resolutions have been proposed by most researchers; learn about the advantages and disadvantages of every solution. In order to deeply understand the popular method 'linearization' used in the deformation, we choose free-form deformation (FFD) approach to perform researching and implementation.

We mainly discuss the detailed implementation of the FFD application, includes the development environment, application's structure information, the workflow diagrams for displaying model, create control points, free-form deformation, multithreading functions. We also give the relationship diagram of classes that are created in this FFD application, and list some C++ pseudo code. During the implementation, we apply several advanced technologies, e.g., apply half edge data structure to store mesh data, which can improve the efficiency of getting vertex local information during deformation; On the basis of vertex local coordinate, we utilize the de Casteljau algorithm, to easily calculate the deformed coordinate with the Burnstein Polynomial form, etc.

After presenting the output of every function in this FFD application, we perform some discussion on 'how to improve the applicant's performance', and propose 'multithreading' technique to resolve the 'long time cost in calculating deformed coordinates' issue. Based on the experiment results, we conclude that the 'multithreading' technique can improve the performance of the FFD application very much, especially for large scale models.

Index Terms— interactive shape editing, free-form deformation (FFD), half edge, control points, de Casteljau algorithm, local coordinate, Burnstein Polynomial, multithreading

Acknowledgement

Foremost, I would like to express my deepest gratitude to my advisor, Dr. He Ying, for his excellent guidance, caring, patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation.

Secondly I would also like to thank my parents, my younger sisters, and elder brother. They were always supporting me and encouraging me with their best wishes.

Finally, I would like to thank my wife, Zhang Haiyun. She was always there cheering me up and stood by me through the good times and bad.

I am making this dissertation not only for the degree, but also to increase my knowledge, I learn much by performing this project.

THANKS AGAIN TO ALL WHO HELPED ME.

1 Introduction

1.1 Background of Interactive Shape Editing

In the past few decades, interactive shape editing techniques have become an active field of research in computer graphics, and most researches are mainly focusing on the following topics:

- How do we interactively edit the arbitrary meshes?
- How do we avoid changing the connectivity and positions of vertices?
- How do we efficiently edit on non-linear surface?
- How do we avoid the distortion during the deformation of the shape?
- How do we preserve the local details during deformation?
- How do we avoid the too costly computation in interactive mesh editing?
- How do we invent fast and robust techniques to tackle the complicated models?
- How do we tackle the issue: neighbouring vertex does not always define a local frame (due to linear dependency)?
- How do we prevent local self-intersection in the reconstructed surface?

In general, the input of such techniques is a triangle mesh that will be deformed, denoted by $M_{\mathrm{tri}}=(V_{\mathrm{tri}}, T_{\mathrm{tri}})$, which consists of n vertices $V_{\mathrm{tri}}=\{v_1,\dots v_n\}$ and m triangles $T_{\mathrm{tri}}=\{t_1,\dots t_m\}$. The objective is to efficiently edit and manipulate M_{tri} as naturally as possible by developing algorithms under the influence of a set of constraints inputted by the user.[1]

And in order to be interactive, editing methods will be based on easy-to-compute linear deformations which will generate acceptable deformation results, i.e. deformation is smooth or piecewise smooth and the result preserves the local appearance of the surface under deformation.

Recently, linear deformation methods based on differential representations have become more popular for they are robust, efficient to compute, and easy to implement, because it's associated with linear system in sparse. They use a local differential representation of the shape to perform modification, instead of directly modify the spatial location of each vertex in the mo

del, which encodes information about its size, local shape and orientation of the local details, to obtain a detail-preserving deformation result.

1.2 Problem Statement

In the interactive shape editing, deformation is manipulated by constructing a differential representation of the shape, performing it base on the inputted constraints, and finally reconstructing the shape from the modified differential representation.

And the resulting deformation is dependent on the particular embedding of the surface in space when applying these methods. During model manipulation, it may get unnatural deformation because its local representation is not updated. This happens when the surface deformation problem is inherently non-linear, because it requires the estimation of local rotation to be applied to the local differential representations.

In order to correct this limitation in the linearization process, a lot of approaches have been developed in the previous years to tackle this problem from different directions, like: Poisson-based approach, Laplacian-based technique, Interactive Volumetric Laplacian approach.

Under such background, in order to deeply understand the conception of interactive shape editing, and the linearization process run in the model's deformation, we adopt Free-Form Deformation (FFD) resolution as the main topic of the dissertation, by utilizing de Casteljau algorithm, to perform some research and implementation.

1.3 Objective and Scope

By this dissertation, we should deeply understand the concept and process of interactive shape editing, how model is free-form deformed, what are the background algorithms and techniques the free-form deformation uses, how the FFD application is implemented, and how to improve the performance of performing deformation on the large-scale models.

In this dissertation, free-form deformation will include four main processes: read model data from mesh file, create control points and grid lines, select the control point and move it, deform the model shape. During read model data from file phase, the application will pop up a standard Windows dialog to let users to choose the specific mesh file, then store them into half-edge data structure, and show the model in OpenGL UI. During the create control points phase, the application will create control points, indicate them with spheres and draw grid lines between those control points with the given planes number in *S*, *T*, *U* directions. During select the control point and move it phase, the application will apply OpenGL API 'GL_SELECT' etc. functions to determine which control point is selected, after the selected control point is moved, it will also trace the new position of the moved control point. And in the last deform model shape phase, the application will obtain the deformed global coordinate for every vertex by calculating its local coordinate *s*, *t*, *u* under the updated control points, then display the deformed model.

1.4 Organization

Firstly some related papers are reviewed in Chapter 2 which presents the related works of interactive shape editing and free-form deformation. In Chapter 3, free-form deformation preliminary and detailed algorithms, such as Bernstein polynomial, de Casteljau Algorithm, are introduced. In Chapter 4, FFD detailed implementation are proposed, which contains three parts: the detailed implementation, like: what developing tools are used, workflow diagrams of the application, classes diagrams of the application and some pseudo code; the results of the implementation and the discussion. What's more, some conclusions are drawn and suggestions on the future work are provided in Chapter 5.

1.5 Summary

In this chapter, we firstly introduce the background of the interactive shape editing technique, and then we state the problem in this research area, based on this, we propose the objective and scope of this dissertation, and at last we draft the organization of this dissertation.

2 Related Work

2.1 Interactive Shape Editing [1]

Interactive shape editing is an active field of research in computer graphics, and consequently a variety of different solutions were proposed to solve this problem.

Early methods, like free-form deformation [2] or space deformations [3-4] enable high-quality shape modelling by directly manipulating the 3D space where the object embedded, but this method has shortcoming, they typically fail to reproduce correct deformation results if it only uses a small number of constraints.

Some approaches propose to solve the computationally expensive non-linear surface deformation problem directly. [5-6] propose a non-linear differential coordinate setup, while [7] minimizes bending and stretching energies using a coupled shell of prisms. [8] employs a non-linear version of the volumetric graph Laplacian and [9] presents an extension of the non-linear Poisson-based deformation approach applied to mesh sequences. Alternatively, [10] combines Laplacian-based deformation with skeleton-based inverse kinematics.

In general, the main drawback of these non-linear methods is often that interactive deformation is only feasible on models of less complexity.

Interactive performance for more complex objects can be achieved by simplifying the inherent non-linear problem. One way to preserve geometric details under global deformations is to use multi-resolution techniques[11-14]. While these approaches are an effective tool for enhancing fine-scale detail preservation, the generation of the hierarchy can be expensive for complex models. Moreover, it is difficult to deal with large deformations in a single step. These limitations are the main reasons for differential-based deformation approaches, which represent the model using its local differential coordinates instead of using its spatial coordinates.

Typically, two differential representations can be used: Laplacian coordinates (or deformation gradients). Poisson-based methods use the input transformation constraints provided by the user to modify the surface gradient of the model. [9] presents a Poisson-based mesh editing method where the local transformations are propagated based on the geodesic distances. [1] introduces a way to replace the geodesic propagation scheme by harmonic field interpolation and shows that this leads to a better estimation of the local transformations.

Unfortunately, although these methods work well for rotations, since they are handed explicitly, they are insensitive to translations.

Laplacian-based methods represent vertex coordinates relative to their neighbours is introduced; although the original framework cannot correctly deal with rotations, recent improvements allow the methods to work similarly well for translations and rotations. [15] describes how local rotations can be estimated and incorporated to the original framework. [16] proposes to use the Laplacian representation combined with implicit transformation optimization. [17] presents a hybrid scheme combining implicit optimization with a two-step local transformation estimation.

In general, although these methods are able to estimate local rotations, the required linearization yields artifacts for large rotations.

And generally, most of these methods suffer from linearization problems: methods which use translational constraints are insensitive to rotations, whereas methods relying on rotational constraints exhibit insensitively to translations. In order to solve this problem, recent methods use skeleton-based techniques [18], multi-step approaches [19] or interactive approaches [20].

Most linear deformation methods rely on a triangle mesh representation. However, deforming a surface model may cause local self-intersections and shrinking effects. In order to prevent such artifacts, some methods use a volumetric structure as basis for the linear deformation [21].

Another class of approaches is able to manipulate an object while guaranteeing volume preservation by defining deformations based on vector fields [22]. Although it enables the definition of advanced implicit deformation tools, it’s still difficult to construct vector fields that satisfy the user-defined constraints.

2.2 Free Form Deformation (FFD) [23]

Efficient and intuitive methods for three-dimensional shape design, modification, and animation are becoming increasingly important areas in computer graphics. The model-dependent techniques initially used by designers to modify surfaces required each primitive type to have different parameters and/or control points that defined its shape.

Model designers had to consider this mathematical model when making the desired modifications, and shape design could be difficult in this case: making simple changes to the surface required the modification of many surface parameters. The process grew more difficult when local changes, such as adding arbitrarily shaped bumps, or global changes, such as bending, twisting, or tapering were necessary.

The free-form deformations [24-27] were designed to deal with some of these problems. These methods embed an object in a deformable region of space such that each point of the object has a unique parameterization that defines its position in the region. The region is then altered, causing recalculation of the positions based upon their initial parameterization. If a deformable space can be defined with great flexibility and with few control points relative to the number in the surface model, then complex models comprised of thousands of control points can be deformed in many interesting ways with very little user-interaction.

Barr [28] first introduced deformations by creating operations for stretching, twisting, bending and tapering surfaces around a central axis (x, y, or z). Operations that involved moving many control points could now be accomplished with the altering of as little as one parameter. However, this technique limits the possible definitions of the deformable space to that of a single coordinate system about an axis, and restricts the ways in which the space can be altered because the axis cannot be modified and the deformable space can only be modified by a few parameters.

Barr's deformations were followed by a more generalized approach to the problem, the Free-Form Deformations (FFDs) of Sederberg and Parry [27]. This method imposes an initial deformation lattice on a parallelepiped, and defines the deformable space as the trivariate B´ezier volume defined by the lattice points. The parallelepiped form of the lattice allows points of an embedded object to be quickly parameterized in the space of the lattice, and as the lattice is deformed, the deformed points can be calculated by simple substitution into the defining equations of the trivariate volume.

This method is widely used because of its ease of use and power to create many types of deformations with little user-interaction. Griessmair and Purgathofer [26] extended this technique by utilizing a trivariate B-Spline representation. Although both methods give the user many controls to alter the deformable space, both Sederberg and Parry's FFDs and Griessmair and Purgathofer's deformation techniques handle only a specific type of space definition, that defined initially by a parallelepiped lattice.

In order to generate free-form deformations for a more general lattice structure, Coquillart introduced Extended Free-Form Deformations (EFFD) [24-25]. This method uses the initial lattice points to define an arbitrary trivariate B´ezier volume, and allows the combining of many lattices to form arbitrary shaped spaces.

Modifying the points of the defining lattice creates a deformation of space where one trivariate volume is deformed into another. This extension allows a greater inventory of deformable spaces, but loses some of the flexibility and stability of Sederberg and Parry's FFDs: While the corner control points of the joined lattices are user-controllable, the internal control points are constrained to preserve continuity; and, calculating the parameterization of a point embedded in the initial trivariate volume requires numerical techniques.

A recent deformation technique developed by Chang and Rockwood [29] generalizes Barr's technique in a different manner. Instead of defining the space in a free-form manner, Chang's approach deals with increasing the flexibility of an axis-based approach by allowing modifications to the axis during the deformation. The technique allows the user to define the axis as a B´ezier curve with two user defined axes at each control point of the curve. Repeated affine transformations using a generalized de Casteljau approach are used to define the deformable space. This technique is very powerful, intuitive, and efficient, but again restricts the ways in which the space surrounding the curve can be altered.

[23] introduces a further extension to these techniques by establishing deformation methods defined on lattices of arbitrary topology. In this case, the deformable space is defined by using a volume analogy of subdivision surfaces [30-32]. In these subdivision methods, the lattice is repeatedly refined, creating a sequence of lattices that converge to a region in three-dimensional space. This refinement procedure is used to define a pseudo-parameterization of an embedded point. As the points of the lattice are modified, a deformation of the space is created, and the embedded points can be relocated within the deformed space.

A new free-form deformation technique is presented that generalizes previous methods by allowing 3-dimensional deformation lattices of arbitrary topology [23]. The technique uses an extension of the Catmull-Clark subdivision methodology that successively refines a 3-dimensional lattice into a sequence of lattices that converge uniformly to a region of 3-dimensional space.

Deformation of the lattice then implicitly defines a deformation of the space. An underlying model can be deformed by establishing positions of the points of them defined within the converging sequence of lattices and then tracking the new positions of these points within the deformed sequence of lattices. This technique allows a greater variety of deformable regions to be defined, and thus a broader range of shape deformations can be generated.

2.3 Summary

This chapter mainly investigates the related research works on interactive shape editing and free-form deformation (FFD), after reading relative papers, we know what solutions most researchers have proposed, and what are advantages and disadvantages of every solution, which will instruct us to design our FFD application in the following chapters.

3 Free-Form Deformation Algorithm

Free-Form Deformation (FFD) which based on trivariate Bernstein Polynomials is an approach to deform solid geometric models. The techniques involved can be used in CSG or B-rep solid modelling systems, to deform planes, quadrics, surfaces (B-spline parametric patches and implicit surface). It also has a property of preserving volume during deformations.

3.1 Preliminary

3.1.1 Half-Edge Data Structure

There are many popular data structures used to represent polygonal meshes; half-edge data structure is an edge-centred data structure capable of maintaining incidence information of vertices, edges and faces, it is a common way to represent a polygon mesh which is a shared list of vertices and a list of faces storing pointers for its vertices. This representation is both convenient and efficient for many purposes.

3.1.1.1 Definition

1.) Each edge is divided into two half-edges.

2.) Each half-edge has 5 references, as below:

① The face on left side (assume counter-clockwise order).
② The Previous half-edge in counter-clockwise order.
③ The next half-edge in counter-clockwise order.
④ The “twin” edge.
⑤ The starting vertex.

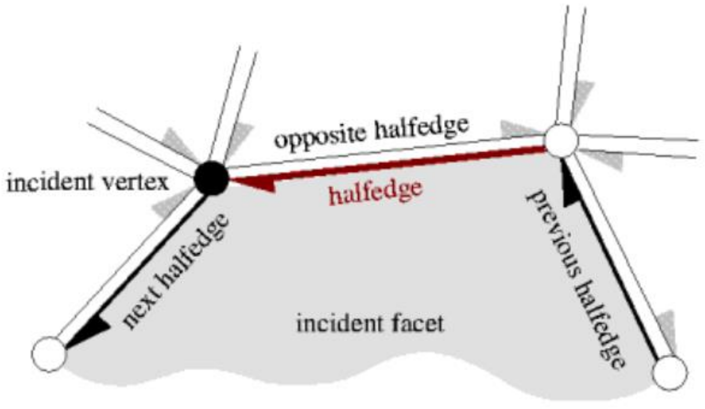


Fig. 1 Half edge

3.) Each face has a pointer to one of its edges.

4.) Each vertex has a pointer to a half edge that has this vertex as the start vertex.

3.1.1.2 One Example

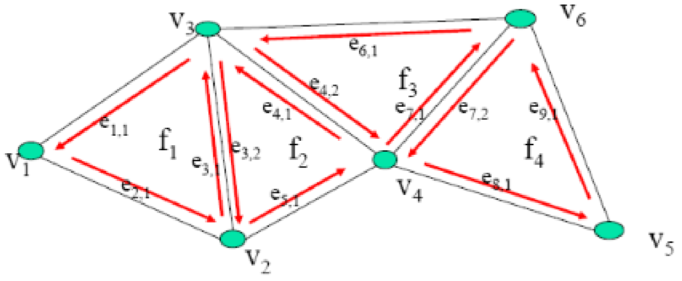
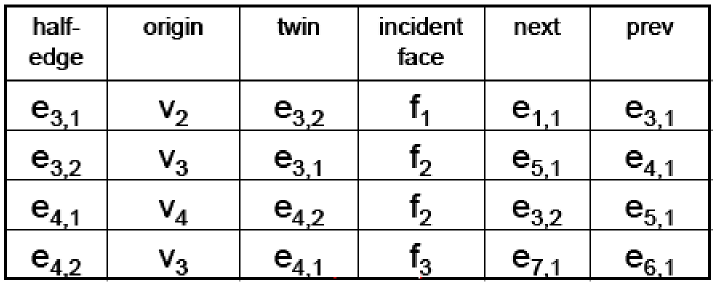


Fig. 2 Half edge example



half-edge	origin	twin	incident face	next	prev
$e_{3,1}$	v_2	$e_{3,2}$	f_1	$e_{1,1}$	$e_{3,1}$
$e_{3,2}$	v_3	$e_{3,1}$	f_2	$e_{5,1}$	$e_{4,1}$
$e_{4,1}$	v_4	$e_{4,2}$	f_2	$e_{3,2}$	$e_{5,1}$
$e_{4,2}$	v_3	$e_{4,1}$	f_3	$e_{7,1}$	$e_{6,1}$

Fig. 3 Half edge explanations

3.1.2 Bernstein Polynomial

A Bernstein polynomial is a polynomial in the Bernstein form, which is a linear combination of Bernstein basis polynomials.

Polynomials in Bernstein form were first used by Bernstein in a constructive proof for the Stone–Weierstrass approximation theorem. With the advent of computer graphics, Bernstein polynomials, restricted to the interval $x \in [0, 1]$, became important in the form of Bézier curves.

The Bernstein Polynomial definition is as below:

1.) The $n + 1$ **Bernstein basis polynomials** of degree n are defined as:

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \text{ , satisfying } \quad \sum_{i=0}^{n} B_i^n(t) \equiv 1 \qquad (1)$$

2.) Binomial coefficient ("*n*-choose-*i*") is:

$$\binom{n}{i} = \frac{n!}{(n-i)!i!} = \frac{n(n-1)(n-2)\cdots 2\cdot 1}{[(n-i)(n-i-1)\cdots 1]\cdot(i-1)\cdots 1} \tag{2}$$

3.) The Lerp of lower degrees is:

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \tag{3}$$

3.2 Algorithms Details

3.2.1 De Casteljau Algorithm

De Casteljau algorithm is a numerically stable way to evaluate polynomials in Bernstein form. And in the mathematical field of numerical analysis, de Casteljau's algorithm is a recursive method to evaluate polynomials in Bernstein form or Bézier curves.

3.2.1.1 Basic Process of Lerp

1.) Lerp – linear interpolation: $P(t) = (1-t)\ a + t\ b = a + t\ (b-a)$

2.) Notation: Lerp(a,b) = (1-t) a + t b

3.) Lerp(a,b) linearly interpolates a and b. As t varies between 0 and 1, Lerp(a,b) gradually changes from a to b, described as below:

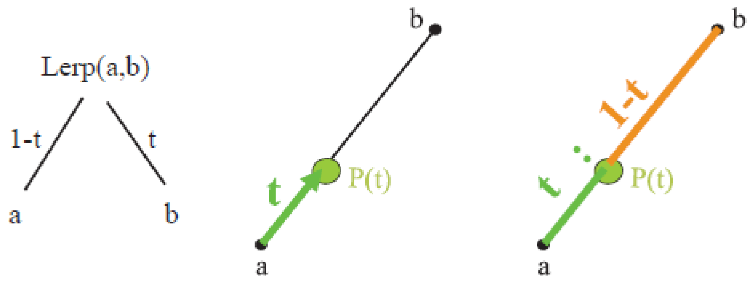
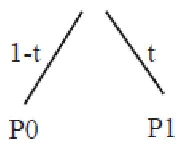
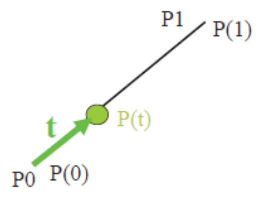
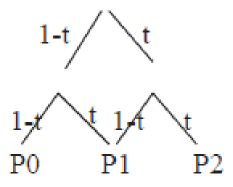
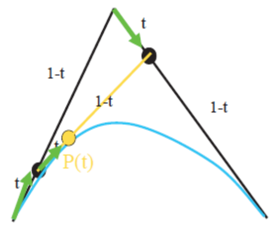
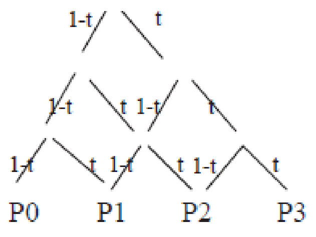
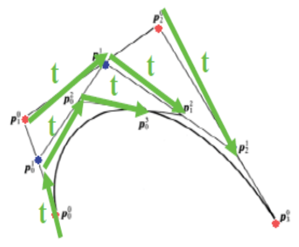


Fig. 4 Lerp (a, b) explanation

3.2.1.2 Lerps with Lower Degree

When degree n equals from 1 to 3, the solutions to resolve lerps problems can be described as in Tab. 1.

Degree	**Problems**	**Solutions**	**Diagrams Explanation**	**Example: Bézier Curves**
n =1	P(0) = P0, P(1) = P1, P (t) =?	P(t) = (1-t) P0 + t P1 = P0 + t(P1-P0)		
n=2	P(0) = P0, P(1) = P2, P (t) =?	P(t) = $(1\text{-}t)^2$ P0 + 2t(1-t) P1 + t^2 P2=(1-t) [(1-t)P0 + tP1] + t [(1-t)P1 + t P_2]		
n=3	P(0) = P0, P(1) = P3, P (t) =?	P(t) = $(1\text{-}t)^3$P0 +3t$(1\text{-}t)^2$ P1+3t^2(1-t) P2 + t^3P3 = (1-t)[(1-t)^2 P0 +2t(1-t) P1+t^2 P2]+ t [(1-t)^2 P1+2t(1-t) P2 + t^2 P3]	 1-t t 1-t t 1-t t 1-t t 1-t t 1-t t P0 P1 P2 P3	

Tab. 1 Solutions to solve lerps problems (*n* equals lower degree)

3.2.1.3 General Case

In general case, the de Casteljau algorithm can be regarded as repeated linear interpolation:

Step 1. Divide each leg into parameter proportions

Step 2. Connect with new segments

Step 3. Repeat with new segments

Step 4. The final point is on the curve.

3.2.1.4 Procedure of de Casteljau Algorithm

1.) Input: control polygon P0, P1 … Pn, and parameter value t, Output: P(t).

2.) Procedure:

① *Initialize* $P_0^0 = P_0, P_1^0 = P_1, \cdots, P_n^0 = P_n$

② *for k from 1 to n do*

```
③  for i from 0 to n – k do
④      $P_i^k(t) = (1-t)p_i^{k-1} + P_{i+1}^{k-1}$
⑤  end- for
⑥ end-for
⑦  return $P(t) = P_0^n$
```

3.2.2 Free Form Deformation (FFD)

Mathematically, the FFD is defined in terms of a tensor product trivariate Bernstein Polynomial. FFD is a technique for manipulating any shape in a free-form manner. It can be looked as a “rubber-like” deformation of space. Deformation can be modelled by deforming a regular grid; and it includes 2D-FFD and 3D-FFD. The deformation process can be described as below.

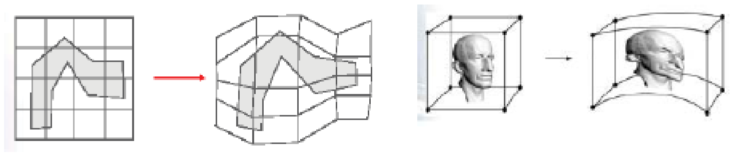


Fig. 5 2D-FFD and 3D-FFD

We will only discuss 3D-FFD in this chapter.

3.2.2.1 Local Coordinate System

1.) We can impose a local coordinate system on a parallel piped region, as shown below:

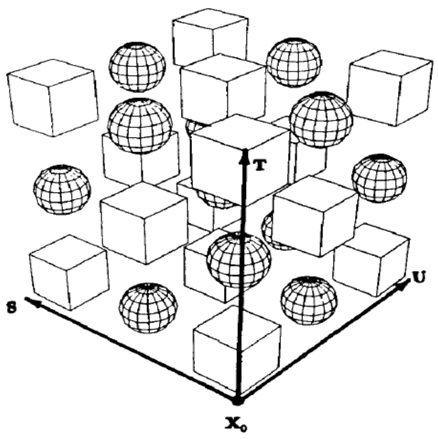


Fig. 6 *s*, *t*, *u* coordinate system [33]

2.) Any vertex position X can be expressed by the following equation with the local coordinate (s, t, u).

$$X = X_0 + sS + tT + uU \tag{4}$$

3.) The local coordinates (s, t, u) of X can be calculated by using the linear algebra, with the following equations.

$$s = \frac{T\times U\cdot(X-X_0)}{T\times U\cdot S}, t = \frac{S\times U\cdot(X-X_0)}{S\times U\cdot T}, u = \frac{S\times T\cdot(X-X_0)}{S\times T\cdot U} \tag{5}$$

($0<s<1$, $0<t<1$ and $0<u<1$)

3.2.2.2 Control Points

We impose a grid of control points P_{ijk} on the parallelepiped with $l + 1$ planes in S direction, $m + 1$ planes in T direction and $k + 1$ planes in U direction, looks at the grid of control points from X axes, it will be like below:

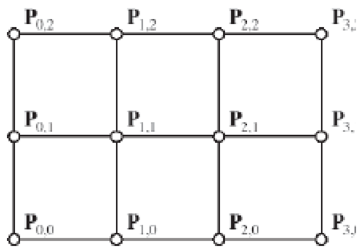


Fig. 7 Control points and grid (see from X axes)

And for every control point, its position can be calculated by the following equation:

$$P_{ijk} = X_0 + \frac{i}{l}S + \frac{j}{m}T + \frac{k}{m}U \tag{6}$$

3.2.2.3 Deformed Coordinate

The deformation is specified by moving the control point P_{ijk} from their un-displaced, latticial positions. And the deformed function is defined by a trivariate tensor product Bernstein Polynomial.

During deformation, the new point X_{ffd} of the arbitrary point X can be calculated by its local coordinates (s, t, u) according to the trivariate Bernstein Polynomial equation, as equation (7).

$$X_{ffd} = \sum_{i=0}^{l} \binom{l}{i}(1-s)^{l-i}s^t\left(\sum_{j=0}^{m} \binom{m}{j}(1-t)^{m-j}t^j\left(\sum_{k=0}^{n}\binom{n}{k}(1-u)^{n-k}u^k\, P_{ijk}\right)\right) \tag{7}$$

3.3 Summary

In this chapter, we learn the half edge data structure, the preliminary algorithms of Bernstein Polynomial. We also learn the concept of free-form deformation, local coordinates and control points; and at last we learn how to calculate the control points' positions, the deformed coordinate positions according to the equations respectively, with this knowledge, we can precede the implementation of FFD that will be discussed in the following Chapter 4.

4 Implementation and Results

In this chapter, we will describe: development environment, application basic information, features and workflow diagrams, results of running the application, the performance improvement discussion, and at last we provide a short summary of previous contents.

4.1 Development Environment

During implementing the application, we use the following development environment:

1.) Visual Studio 2010 Professional, C++ project.

2.) OpenGL Utility Toolkit, by introducing seven OpenGL 3rd party files, the file names and their locations are listed in the following table.

No	**3rd Party Files**	**Files Location**
1	glui.h	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\ gl\
2	glut.h	
3	glui32.lib	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib
4	glut.lib	
5	glut32.lib	
6	glut32.dll	C:\Windows\System32

Tab. 2 3rd party files location

4.2 Application Basic Information

1. The application name: FFD.
2. The solution name: FFD.sln.
3. The structure of the solution folder: FFD, we can refer to the below Fig. 8.

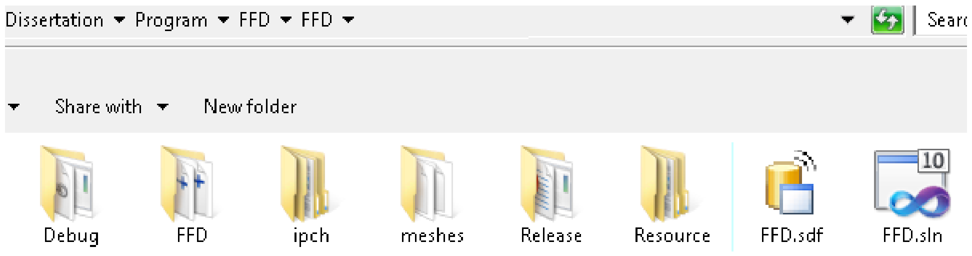


Fig. 8 FFD folder structure

4. There are 16 source code files under the source code folder \FFD, the details are listed as below.

No	**File List**	**Function**
1	FFD.cpp	Control whole process of application, OpenGL UI
2	FFD3DIO.cpp	Graphics related function, like: dotProduct(), crossProduct(), etc.
3	FFD3DIO.h	
4	FFDMath.h	Basic math function, like: Sqrt(), abs() etc.
5	FreeFormDef.cpp	Definition of FFD, calculation s,t,u, XffdI, multi-thread functions
6	FreeFormDef.h	
7	MultiThread.cpp	Implementation of C++ multithreading functions
8	MultiThread.h	
9	FFDUtil.cpp	Common functions that used in all classes of the application
10	FFDUtil.h	
11	HE_Edge.cpp	Half-Edge data structure to store mesh data, for showing model efficiently
12	HE_Edge.h	
13	HE_Face.cpp	
14	HE_Face.h	
15	HE_Vertex.cpp	
16	HE_Vertex.h	

Tab. 3 Source code files under FFD folder

5. Remained files information are listed in the below table.

No	Files	Location	Function
1	FFD.exe	\Debug and \Release	Executable EXE file
2	mesh files	\meshes	Show models
3	glui.h	Resource\1.HeaderFiles	All 3rd party (Open GL) head files
4	glut.h		
5	glui32.lib	Resource\2.LibFiles	All 3rd party (Open GL) lib files
6	glut.lib		
7	glut32.lib		
8	glut32.dll	Resource\3.DllFiles	Open GL 3rd party DLL file

Tab. 4 Remained files information

4.3 Features and Workflow Diagrams

The application implements three main features: open mesh file and show model; control points and free-form deformation. The application also utilizes 'C++ Multithreading' technique, creates several 'FFD classes', they are described in this chapter too, and at last 'workflow diagram of whole manipulation' is provided.

4.3.1 Open Mesh File and Display Model

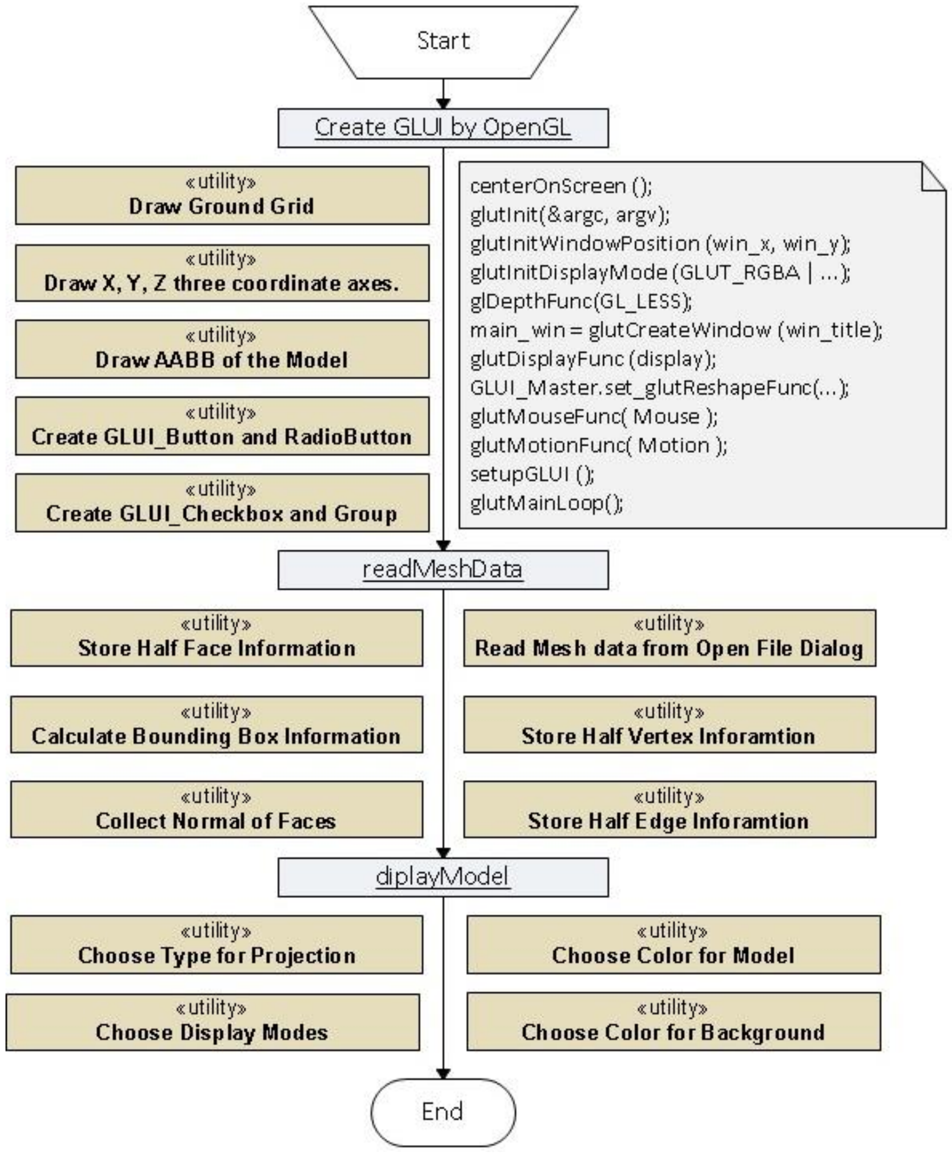
1. There are eleven functions are implemented in this feature, as shown in Tab. 5.

No	**Function**
1	Loading the Triangle Mesh: ① Use” iostream”, “Windows standard open file dialog”; ② Store vertex information; ③ Store Half Edges information (including: find and add pair Half edge for every half edge); and store Half Faces’ information; ④ Calculate Bounding box basic information.
2	Computing vertex normal; the averaged value is weighted by the area of the face.
3	Draw the ground grid (the xz-plane, Y=0), -5.0 ~ 5.0, 20 parallel lines, every 0.5.
4	Draw X, Y, Z three coordinate axes.
5	And draw the axis-aligned bounding box (*AABB*) of the model.
6	Draw models with colours, by popping up a colour dialog, let users to choose colours.
7	Draw the background colour, also by popping up a colour dialog, user can choose colour from it.
8	Changing the type of projection, it can let the user choose either orthogonal projection or perspective projection.
9	Implement four display modes, the user is allowed to choose one of the following rendering schemes: ①Points ②Wireframe rendering ③Flat shading ④Smooth shading
10	It implements a mouse-based camera control to allow the user to rotate, translate and zoom in on the object: ①To move control point, the user clicks the left button, and moves the mouse;

	②To translate objects or camera, users click the middle button, and moves the mouse; ③To zoom in/out, the user clicks the right button and moves the mouse.
11	Use GLUI to design a user-friendly GUI, includes: ①GLUI_Button ②GLUI_RadioButton ③GLUI_Checkbox ④GLUI_Seperator, etc.

Tab. 5 Eleven functions of displaying model

2. The workflow diagram of implementing this feature is as below Fig. 9.


Start
Create GLUI by OpenGL
«utility» Draw Ground Grid
«utility» Draw X, Y, Z three coordinate axes.
«utility» Draw AABB of the Model
«utility» Create GLUI_Button and RadioButton
«utility» Create GLUI_Checkbox and Group
readMeshData
«utility» Store Half Face Information
«utility» Read Mesh data from Open File Dialog
«utility» Calculate Bounding Box Information
«utility» Store Half Vertex Inforamtion
«utility» Collect Normal of Faces
«utility» Store Half Edge Inforamtion
diplayModel
«utility» Choose Type for Projection
«utility» Choose Color for Model
«utility» Choose Display Modes
«utility» Choose Color for Background
End

```
centerOnScreen ();
glutInit(&argc, argv);
glutInitWindowPosition (win_x, win_y);
glutInitDisplayMode (GLUT_RGBA | ...);
glDepthFunc(GL_LESS);
main_win = glutCreateWindow (win_title);
glutDisplayFunc (display);
GLUI_Master.set_glutReshapeFunc(...);
glutMouseFunc( Mouse );
glutMotionFunc( Motion );
setupGLUI ();
glutMainLoop();
```

Fig. 9 Workflow diagram of displaying models

4.3.2 Control Points

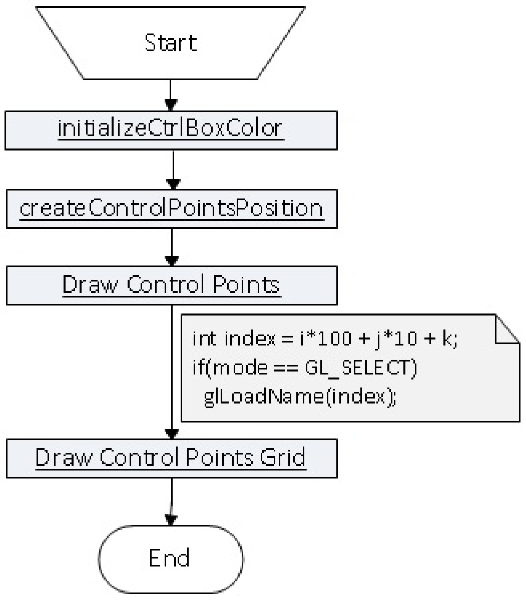
This feature is mainly implemented by using OpenGL Utility Toolkit, like the OpenGL API function ‘GL_SELECT’ and ‘GLuint buffer’, and this feature includes two parts: create, display control points and grids; move a control point.

4.3.2.1 Create, Display Control Points and Grids

This part includes four steps: Initialize control points colour, create control points position, draw control points and draw grid between control points.

1.) Initialize control points colour: set the default colour of all the control points to Red.
2.) Create control points position: calculate the position of all control points according to the equation (6).
3.) Draw control points: implemented by using OpenGL API function: glutSolidSphere() .
4.) Draw grid between control points: implemented by using OpenGL API function: GL_LINES.

The work flow diagram of the implementation is as below Fig. 10.



```
int index = i*100 + j*10 + k;
if(mode == GL_SELECT)
  glLoadName(index);
```

Fig. 10 Workflow diagram of creating control points

4.3.2.2 Move a Control Point

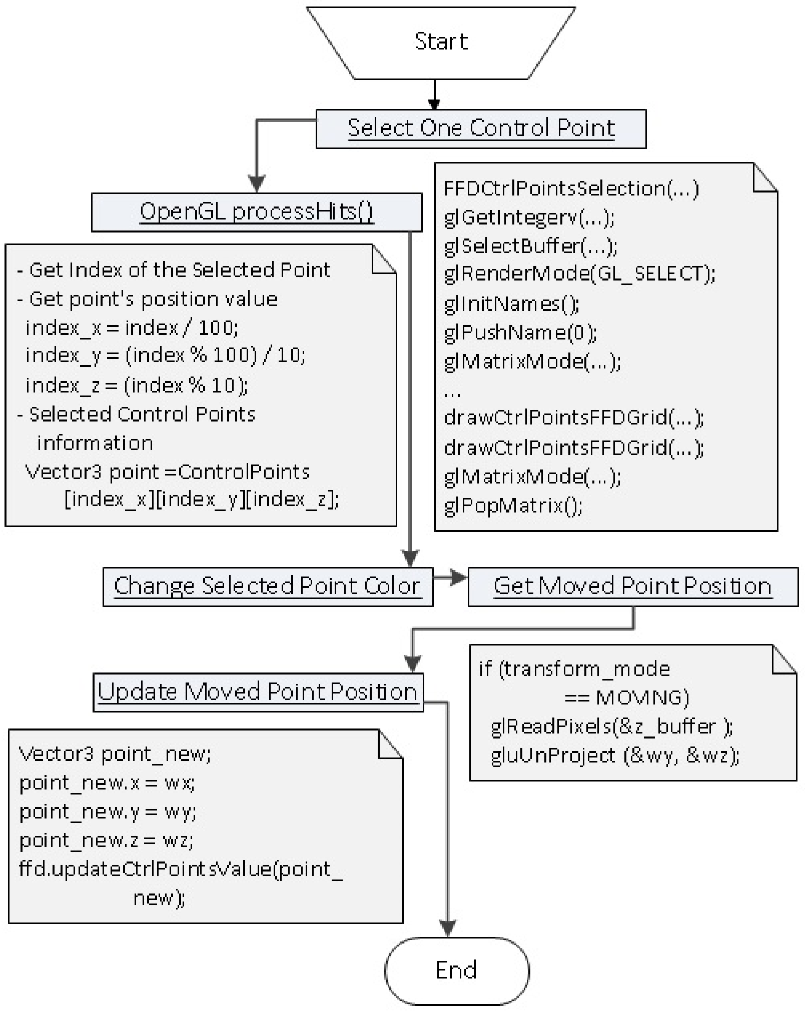
When a control point is moved, the following three scenarios will be implemented in this feature.

1.) The application shows which control point is selected by changing the selected control point's colour from Red to Green, implemented by OpengGL API function: glColor3f().

2.) When the user drags a control point to the new position, the application will re-draw the selected control points in the new position, implemented by OpengGL API function: glutSolidSphere() , glReadPixels().

3.) The application re-draws grid lines between the new updated control points, by using OpenGL API function: GL_LINES.

The workflow diagram of this implementation is shown in Fig. 11.



```
FFDCtrlPointsSelection(...)
glGetIntegerv(...);
glSelectBuffer(...);
glRenderMode(GL_SELECT);
glInitNames();
glPushName(0);
glMatrixMode(...);
...
drawCtrlPointsFFDGrid(...);
drawCtrlPointsFFDGrid(...);
glMatrixMode(...);
glPopMatrix();
```

```
- Get Index of the Selected Point
- Get point's position value
 index_x = index / 100;
 index_y = (index % 100) / 10;
 index_z = (index % 10);
- Selected Control Points
   information
 Vector3 point =ControlPoints
      [index_x][index_y][index_z];
```

```
if (transform_mode
          == MOVING)
  glReadPixels(&z_buffer );
  gluUnProject (&wy, &wz);
```

```
Vector3 point_new;
point_new.x = wx;
point_new.y = wy;
point_new.z = wz;
ffd.updateCtrlPointsValue(point_
              new);
```

Fig. 11 Workflow diagram of moving a control point

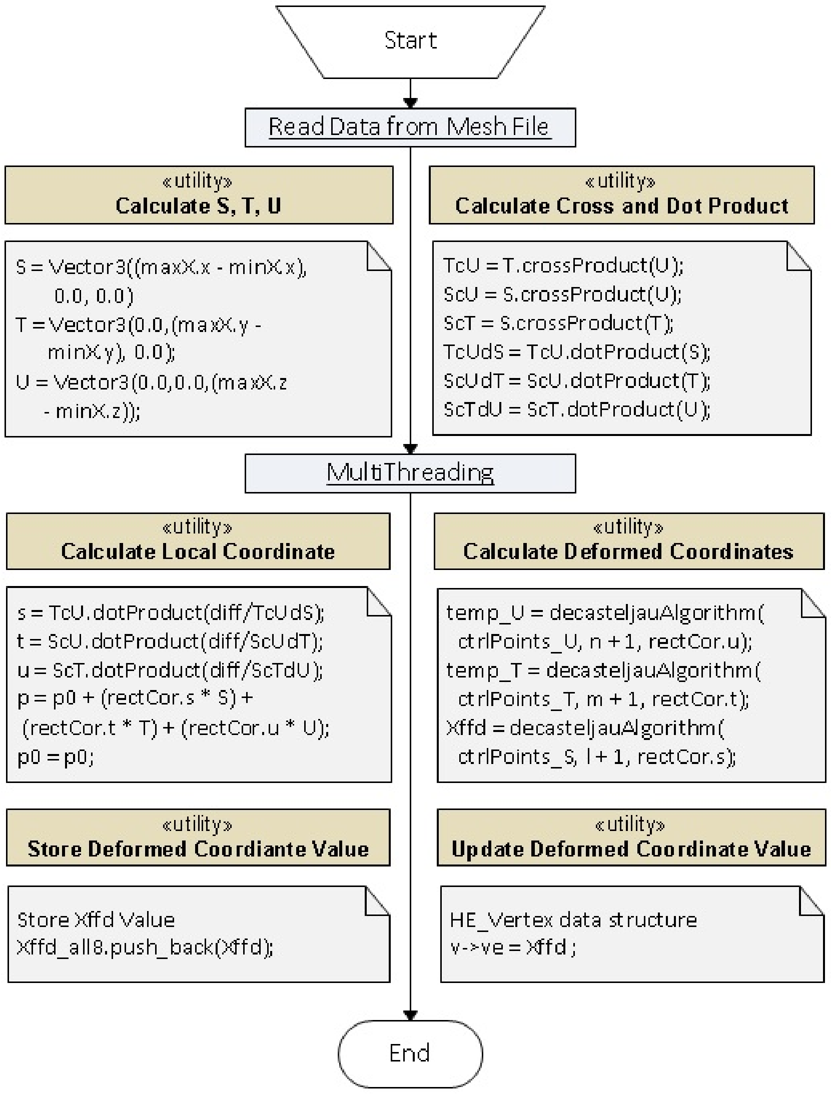
4.3.3 Free Form Deformation

4.3.3.1 Default FFD by Data from Mesh File

The application performs the following steps to calculate *S*, *T*, *U*, *s*, *t*, *u* and X_{ffd}.

- Read model data from mesh file; and store it into the half-edge data structure;
- Calculate *S*, *T*, *U* value, cross product *TcU*, *ScU*, *ScT*, and dot products *TcUdS*, *ScUdT*, *ScTdU*
- Calculate *s*, *t*, *u* according the equation (8) in 3.2.2.1.
- Calculate X_{ffd} according to the equation (9) in 3.2.2.3.

The work flow diagram the implementation is described as Fig. 12.


Start
Read Data from Mesh File
«utility» Calculate S, T, U
«utility» Calculate Cross and Dot Product
MultiThreading
«utility» Calculate Local Coordinate
«utility» Calculate Deformed Coordinates
«utility» Store Deformed Coordiante Value
«utility» Update Deformed Coordinate Value
End

```
S = Vector3((maxX.x - minX.x),
     0.0, 0.0)
T = Vector3(0.0,(maxX.y -
    minX.y), 0.0);
U = Vector3(0.0,0.0,(maxX.z
    - minX.z));
```

```
TcU = T.crossProduct(U);
ScU = S.crossProduct(U);
ScT = S.crossProduct(T);
TcUdS = TcU.dotProduct(S);
ScUdT = ScU.dotProduct(T);
ScTdU = ScT.dotProduct(U);
```

```
s = TcU.dotProduct(diff/TcUdS);
t = ScU.dotProduct(diff/ScUdT);
u = ScT.dotProduct(diff/ScTdU);
p = p0 + (rectCor.s * S) +
 (rectCor.t * T) + (rectCor.u * U);
p0 = p0;
```

```
temp_U = decasteljauAlgorithm(
  ctrlPoints_U, n + 1, rectCor.u);
temp_T = decasteljauAlgorithm(
  ctrlPoints_T, m + 1, rectCor.t);
Xffd = decasteljauAlgorithm(
  ctrlPoints_S, l + 1, rectCor.s);
```

```
Store Xffd Value
Xffd_all8.push_back(Xffd);
```

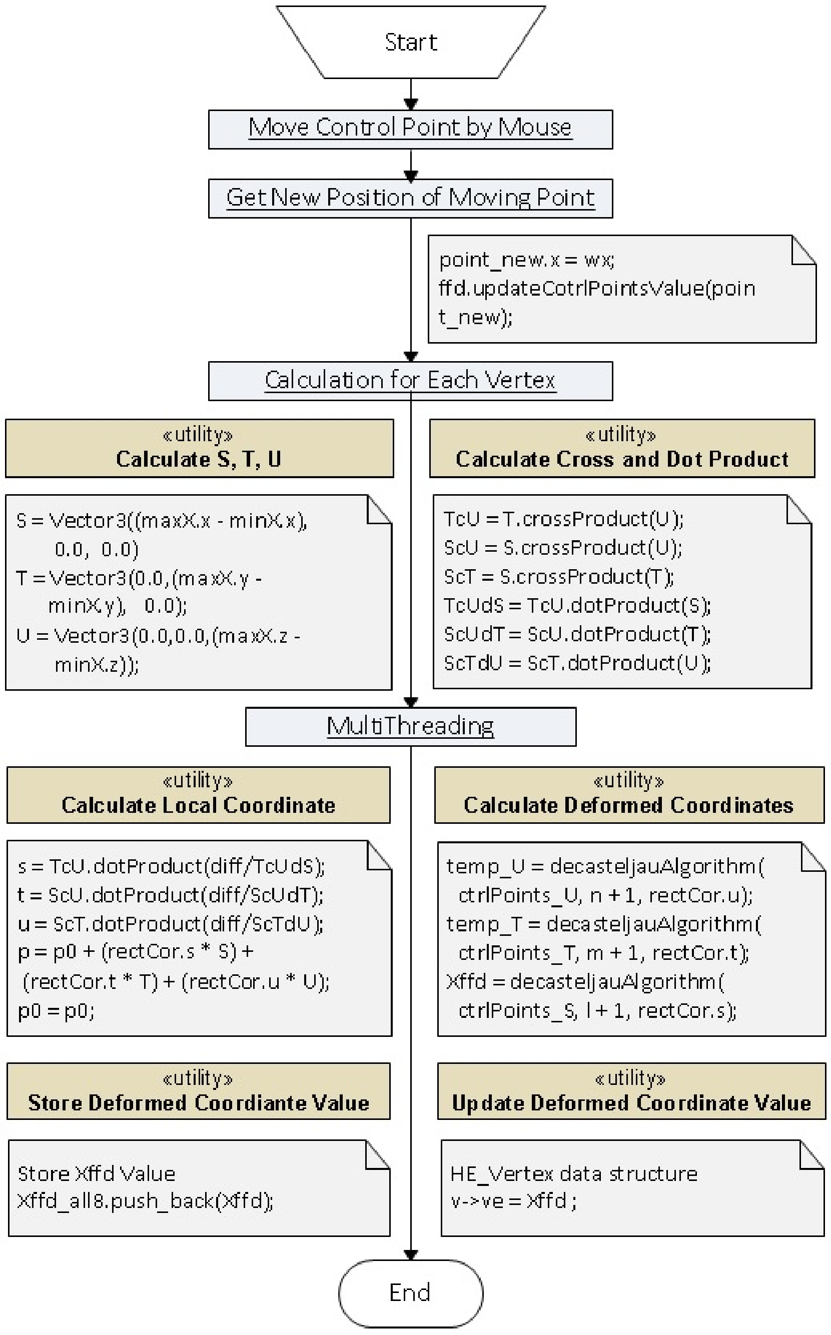
```
HE_Vertex data structure
v->ve = Xffd ;
```

Fig. 12 Workflow diagram of default free-form deformation

4.3.3.2 FFD from Dragging Control Point by Users

The application performs the following steps to calculate deformed coordinate X_{ffd} for every vertex when users drag a control point.

- The dragged control point will move to a new position.
- The application gets the new position of dragged control point.


Start
Move Control Point by Mouse
Get New Position of Moving Point
Calculation for Each Vertex
«utility» Calculate S, T, U
«utility» Calculate Cross and Dot Product
MultiThreading
«utility» Calculate Local Coordinate
«utility» Calculate Deformed Coordinates
«utility» Store Deformed Coordiante Value
«utility» Update Deformed Coordinate Value
End

```
point_new.x = wx;
ffd.updateCotrlPointsValue(poin
t_new);
```

```
S = Vector3((maxX.x - minX.x),
     0.0,  0.0)
T = Vector3(0.0,(maxX.y -
     minX.y),   0.0);
U = Vector3(0.0,0.0,(maxX.z -
     minX.z));
```

```
TcU = T.crossProduct(U);
ScU = S.crossProduct(U);
ScT = S.crossProduct(T);
TcUdS = TcU.dotProduct(S);
ScUdT = ScU.dotProduct(T);
ScTdU = ScT.dotProduct(U);
```

```
s = TcU.dotProduct(diff/TcUdS);
t = ScU.dotProduct(diff/ScUdT);
u = ScT.dotProduct(diff/ScTdU);
p = p0 + (rectCor.s * S) +
 (rectCor.t * T) + (rectCor.u * U);
p0 = p0;
```

```
temp_U = decasteljauAlgorithm(
  ctrlPoints_U, n + 1, rectCor.u);
temp_T = decasteljauAlgorithm(
  ctrlPoints_T, m + 1, rectCor.t);
Xffd = decasteljauAlgorithm(
  ctrlPoints_S, l + 1, rectCor.s);
```

```
Store Xffd Value
Xffd_all8.push_back(Xffd);
```

```
HE_Vertex data structure
v->ve = Xffd ;
```

Fig. 13 Workflow diagram of free-form deformation by moving a control point

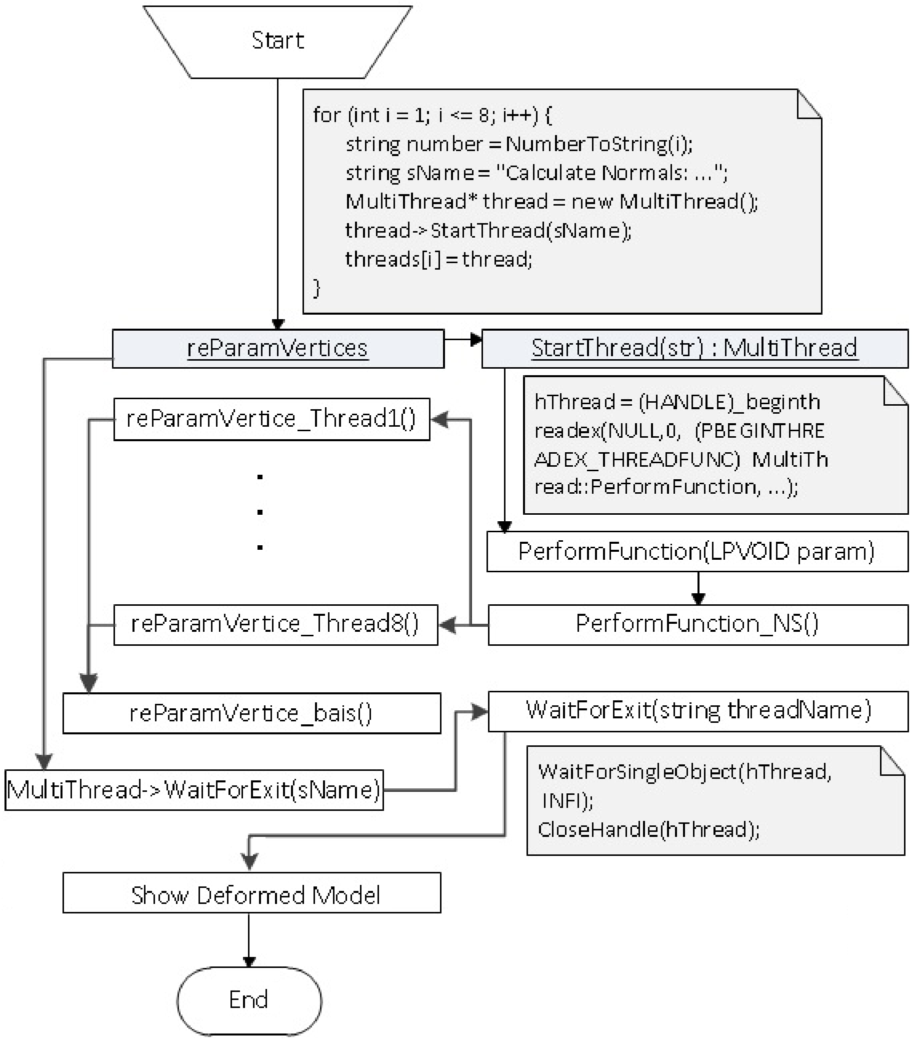
- The application will re-calculate the X_{ffd} value for every vertex according to the new control points' position value.

The workflow diagram of the implementation is described as Fig. 13.

4.3.4 Multithreading

To improve the performance of calculating deformed coordinates, we design and implement the FFD Multithreading base on C++ multithreading technique. The main steps in the implementation are as below:

1. Create a C++ class, named as 'MultiThread'.



```
for (int i = 1; i <= 8; i++) {
    string number = NumberToString(i);
    string sName = "Calculate Normals: ...";
    MultiThread* thread = new MultiThread();
    thread->StartThread(sName);
    threads[i] = thread;
}
```

```
hThread = (HANDLE)_beginth
readex(NULL,0, (PBEGINTHRE
ADEX_THREADFUNC) MultiTh
read::PerformFunction, ...);
```

```
WaitForSingleObject(hThread,
 INFI);
CloseHandle(hThread);
```

Fig. 14 Workflow diagram of Multithreading

2. Design and implement four main methods in the above class: StartThread(string threadName), WaitForExit(string threadName),WINAPI PerformFunction() and PerformFunction_NS(),their functions are described as below:
   - StartThread(string threadName): start a new thread, the parameter 'threadName' will record the thread name, the output log will use this thread name.
   - WaitForExit(string threadName): wait for one started thread to finish. The function of the parameter 'threadName' is same as StartThread(string threadName).
   - WINAPI PerformFunction(): is a static function, which will be a parameter called by StartThread(string threadName) method.
   - PerformFunction_NS():communicate with the class FreeFormDef, by calling the function like 'reParamVertice_Thread1()' defined in the class FreeFormDef, which performs the real manipulation, e.g., calculate deformed coordinates value.
3. The workflow diagram of multithreading implementation is described as Fig. 14.

4.3.5 FFD Classes

1. Functions of FFD Classes.

Totally, there are eight classes and one FFD.cpp file are created, their functions are listed in the following Tab. 6.

No	**Class Name**	**Function**
1	HE_Vertex	Store Model data from Mesh file
2	HE_Edge	
3	HE_Face	
4	FreeFormDef	Main functions for Free Form Deformation
5	MultiThread	Implementation of all multithreading functions
6	FFDUtil	Common functions
7	Vector3	Data structure for local & global coordinates
8	Math	Assistant functions for Vector3
9	FFD.cpp	Control the workflow of OpenGL UI implementation

Tab. 6 Functions of eight classes and FFD.cpp file

2. Hierarchy Diagram of Classes and the FFD.cpp file

The relationship of the eight classes and the FFD.cpp file are described as Fig. 15.

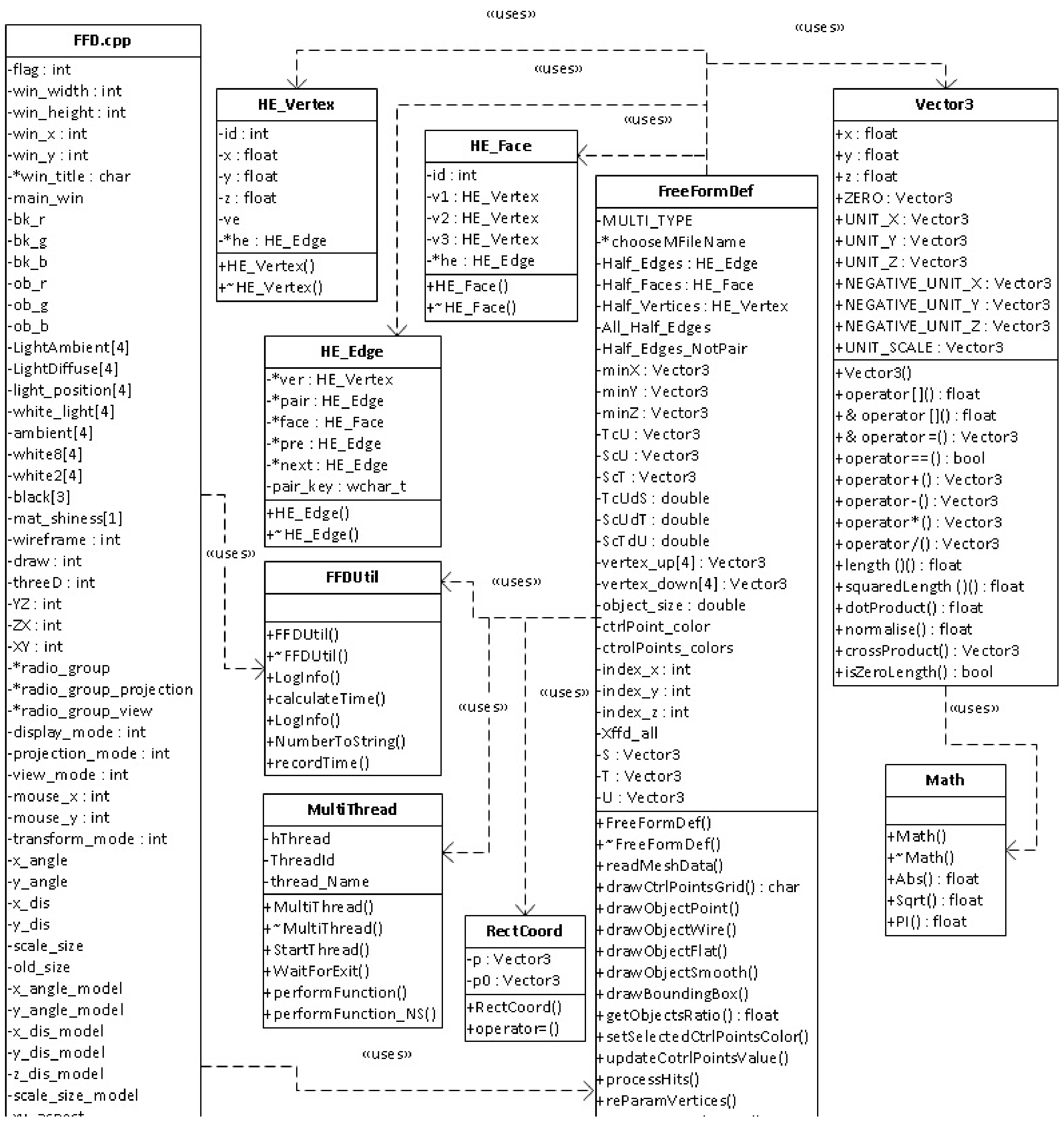


Fig. 15 FFD classes relationship

4.3.6 Whole Process of FFD Manipulation

The whole process of the application includes three main parts: create OpenGL UI, open mesh file and show model; create/show control points and free-form deformation. The implementation details of these three parts have been described in above chapters, in order to better understand the whole process, we provide a workflow diagram for the whole manipulation process here, as Fig. 16.

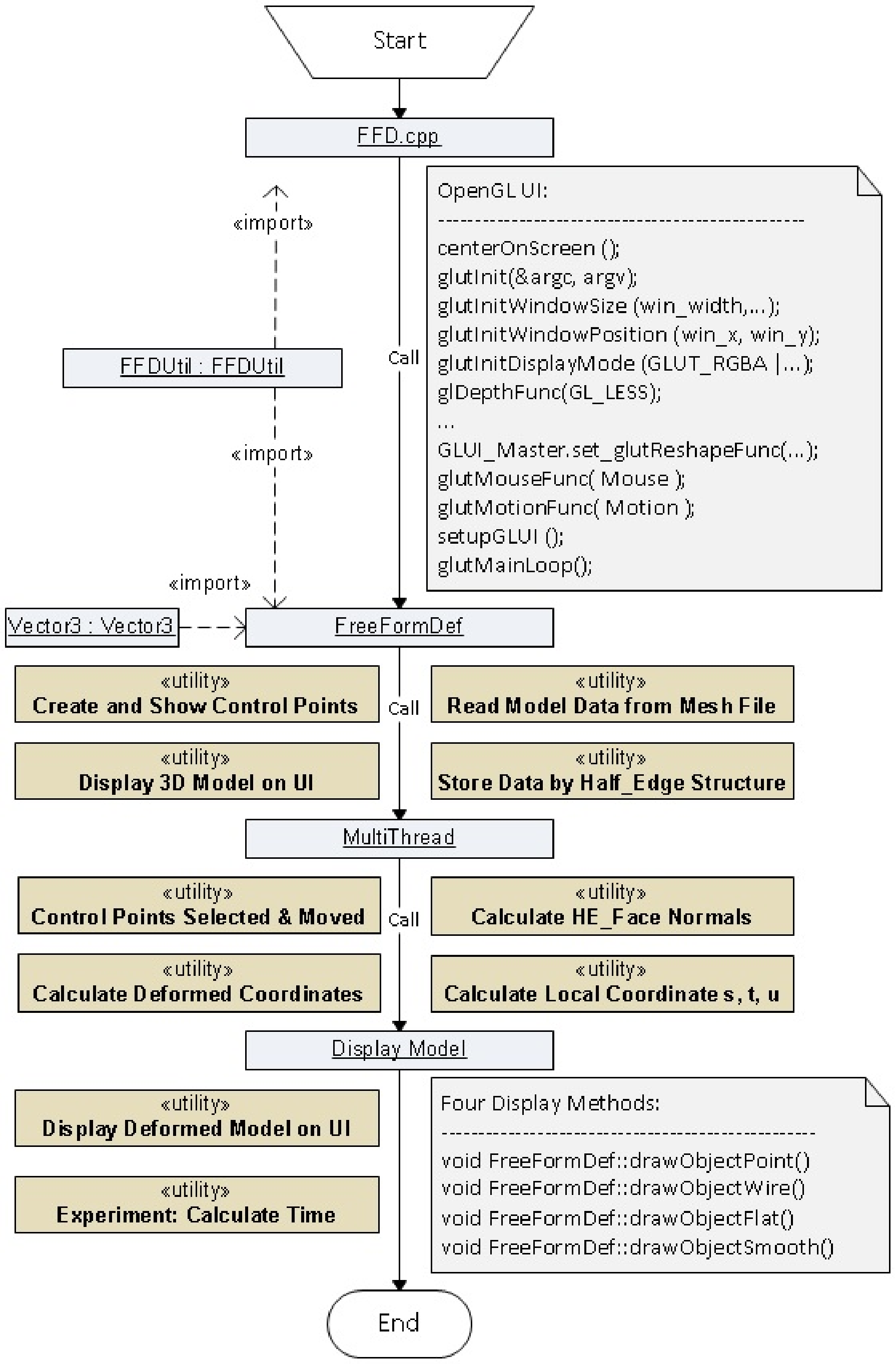
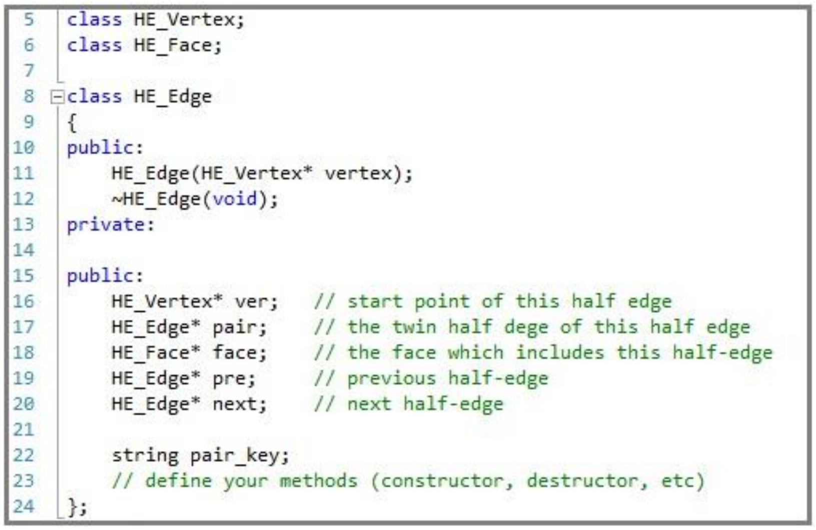

Start
FFD.cpp
«import»
FFDUtil : FFDUtil
«import»
«import»
Call
OpenGL UI:
centerOnScreen ();
glutInit(&argc, argv);
glutInitWindowSize (win_width,...);
glutInitWindowPosition (win_x, win_y);
glutInitDisplayMode (GLUT_RGBA |...);
glDepthFunc(GL_LESS);
...
GLUI_Master.set_glutReshapeFunc(...);
glutMouseFunc(Mouse);
glutMotionFunc(Motion);
setupGLUI ();
glutMainLoop();
Vector3 : Vector3
FreeFormDef
«utility» Create and Show Control Points
Call
«utility» Read Model Data from Mesh File
«utility» Display 3D Model on UI
«utility» Store Data by Half_Edge Structure
MultiThread
«utility» Control Points Selected & Moved
Call
«utility» Calculate HE_Face Normals
«utility» Calculate Deformed Coordinates
«utility» Calculate Local Coordinate s, t, u
Display Model
«utility» Display Deformed Model on UI
«utility» Experiment: Calculate Time
Four Display Methods:
void FreeFormDef::drawObjectPoint()
void FreeFormDef::drawObjectWire()
void FreeFormDef::drawObjectFlat()
void FreeFormDef::drawObjectSmooth()
End

Fig. 16 Workflow diagram of the FFD process

4.3.7 Some Pseudo Code

In this chapter, we will list some pseudo code for seven important parts of the FFD implementation.

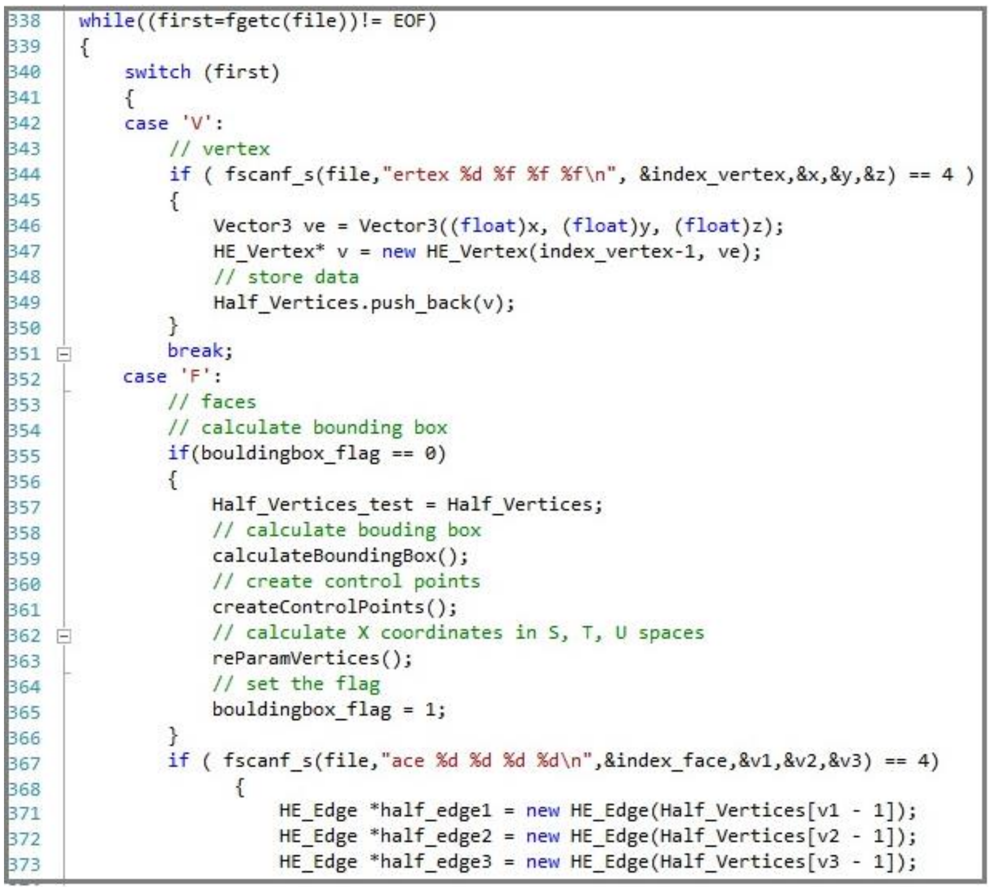
1. Half Edge data structure



```
class HE_Vertex;
class HE_Face;

class HE_Edge
{
public:
    HE_Edge(HE_Vertex* vertex);
    ~HE_Edge(void);
private:

public:
    HE_Vertex* ver;   // start point of this half edge
    HE_Edge* pair;    // the twin half dege of this half edge
    HE_Face* face;    // the face which includes this half-edge
    HE_Edge* pre;     // previous half-edge
    HE_Edge* next;    // next half-edge

    string pair_key;
    // define your methods (constructor, destructor, etc)
};
```

Fig. 17 Pseudo code: half edge data structure

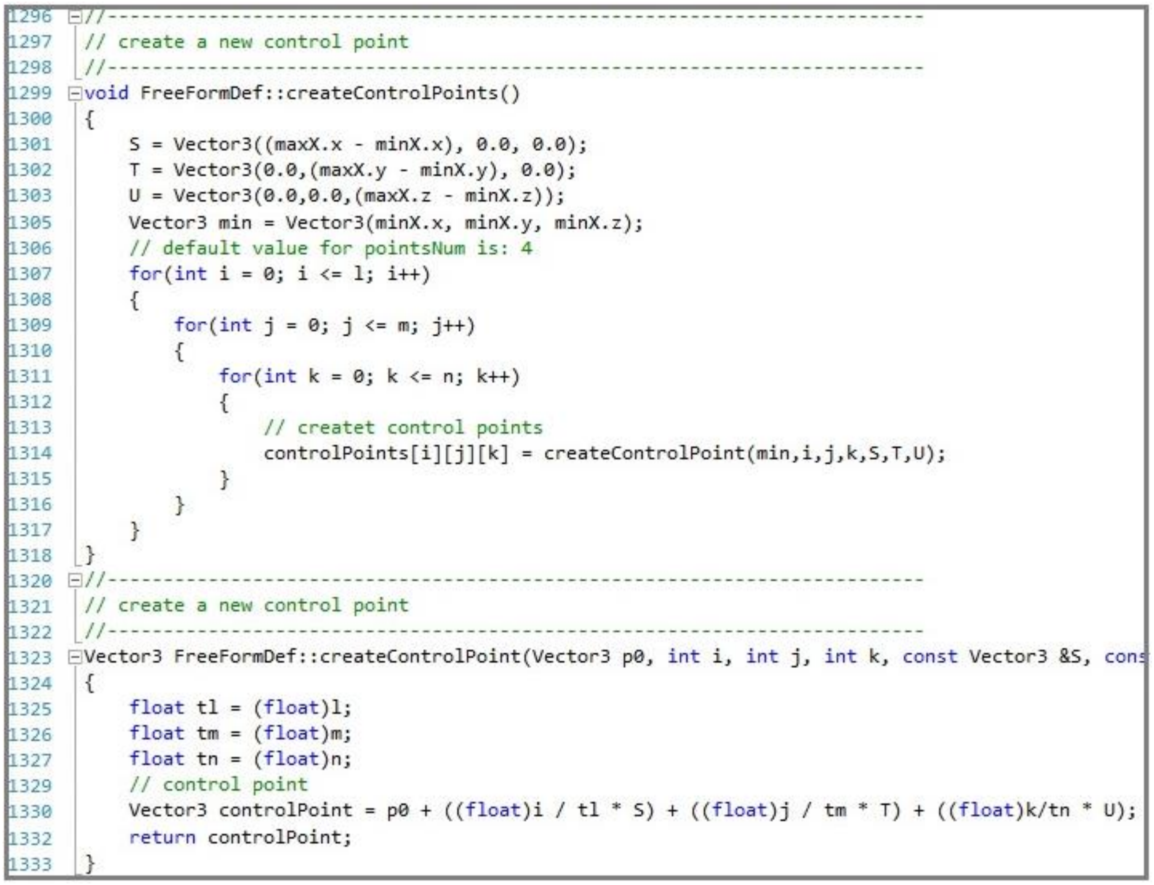
2. Read model data from the mesh file



```
while((first=fgetc(file))!= EOF)
{
    switch (first)
    {
    case 'V':
        // vertex
        if ( fscanf_s(file,"ertex %d %f %f %f\n", &index_vertex,&x,&y,&z) == 4 )
        {
            Vector3 ve = Vector3((float)x, (float)y, (float)z);
            HE_Vertex* v = new HE_Vertex(index_vertex-1, ve);
            // store data
            Half_Vertices.push_back(v);
        }
        break;
    case 'F':
        // faces
        // calculate bounding box
        if(bouldingbox_flag == 0)
        {
            Half_Vertices_test = Half_Vertices;
            // calculate bouding box
            calculateBoundingBox();
            // create control points
            createControlPoints();
            // calculate X coordinates in S, T, U spaces
            reParamVertices();
            // set the flag
            bouldingbox_flag = 1;
        }
        if ( fscanf_s(file,"ace %d %d %d %d\n",&index_face,&v1,&v2,&v3) == 4)
              {
                  HE_Edge *half_edge1 = new HE_Edge(Half_Vertices[v1 - 1]);
                  HE_Edge *half_edge2 = new HE_Edge(Half_Vertices[v2 - 1]);
                  HE_Edge *half_edge3 = new HE_Edge(Half_Vertices[v3 - 1]);
```

Fig. 18 Pseudo code: read model data from mesh file

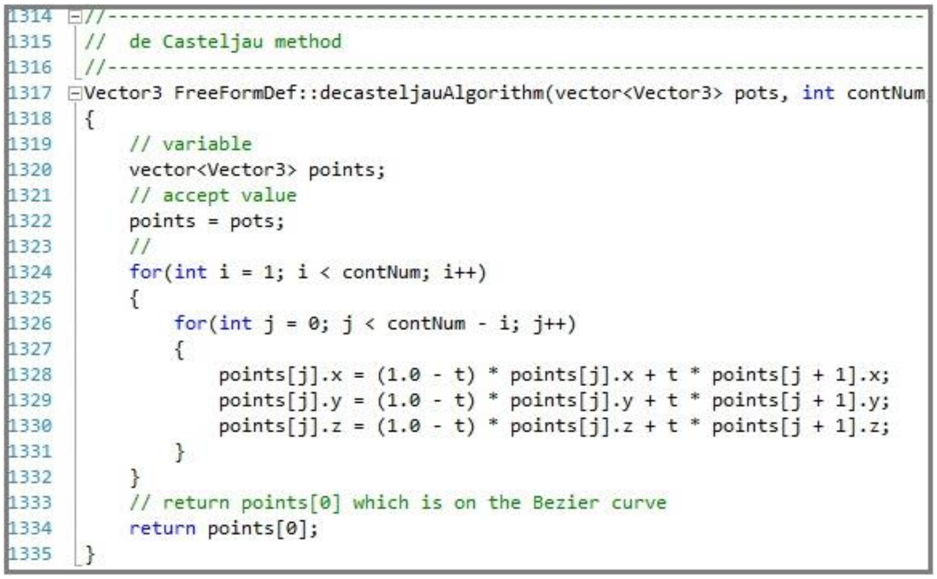
3. Create control points



```
//----------------------------------------------------------------------
// create a new control point
//----------------------------------------------------------------------
void FreeFormDef::createControlPoints()
{
    S = Vector3((maxX.x - minX.x), 0.0, 0.0);
    T = Vector3(0.0,(maxX.y - minX.y), 0.0);
    U = Vector3(0.0,0.0,(maxX.z - minX.z));
    Vector3 min = Vector3(minX.x, minX.y, minX.z);
    // default value for pointsNum is: 4
    for(int i = 0; i <= l; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            for(int k = 0; k <= n; k++)
            {
                // createt control points
                controlPoints[i][j][k] = createControlPoint(min,i,j,k,S,T,U);
            }
        }
    }
}
//----------------------------------------------------------------------
// create a new control point
//----------------------------------------------------------------------
Vector3 FreeFormDef::createControlPoint(Vector3 p0, int i, int j, int k, const Vector3 &S, cons
{
    float tl = (float)l;
    float tm = (float)m;
    float tn = (float)n;
    // control point
    Vector3 controlPoint = p0 + ((float)i / tl * S) + ((float)j / tm * T) + ((float)k/tn * U);
    return controlPoint;
}
```

Fig. 19 Pseudo code: create control points

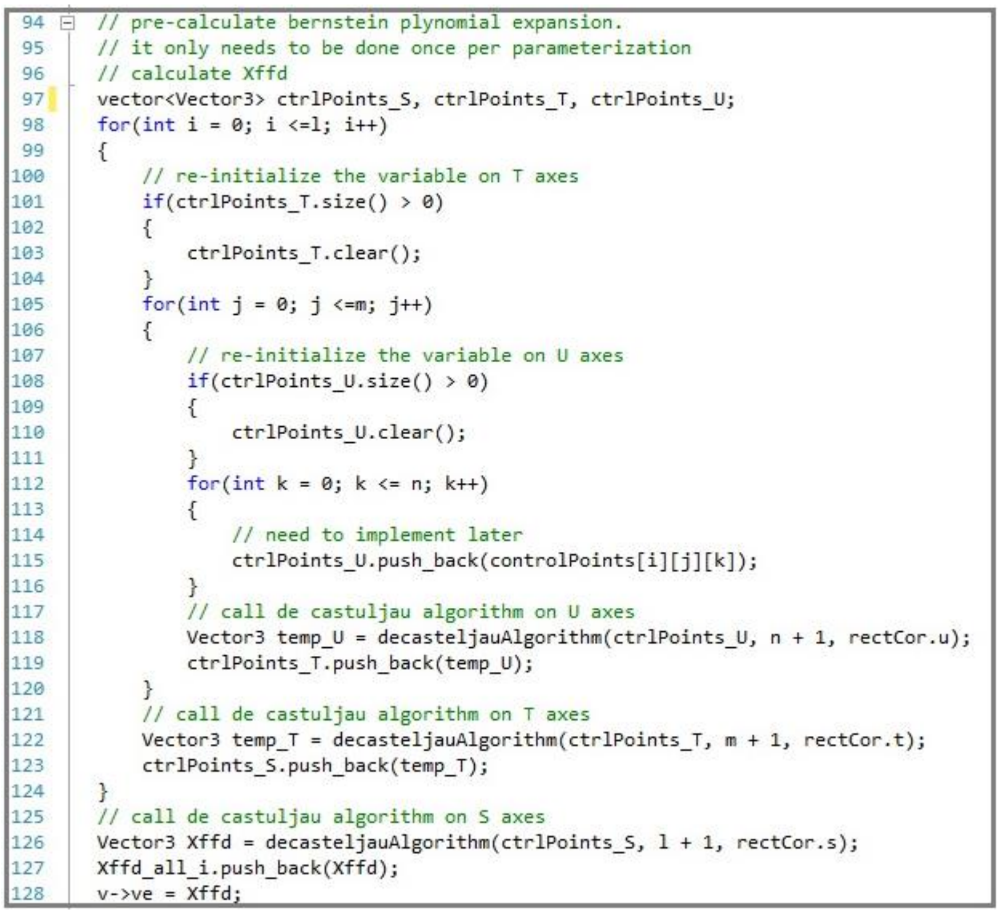
4. de Casteljau Algorithm



```
//----------------------------------------------------------------------
//  de Casteljau method
//----------------------------------------------------------------------
Vector3 FreeFormDef::decasteljauAlgorithm(vector<Vector3> pots, int contNum
{
    // variable
    vector<Vector3> points;
    // accept value
    points = pots;
    //
    for(int i = 1; i < contNum; i++)
    {
        for(int j = 0; j < contNum - i; j++)
        {
            points[j].x = (1.0 - t) * points[j].x + t * points[j + 1].x;
            points[j].y = (1.0 - t) * points[j].y + t * points[j + 1].y;
            points[j].z = (1.0 - t) * points[j].z + t * points[j + 1].z;
        }
    }
    // return points[0] which is on the Bezier curve
    return points[0];
}
```

Fig. 20 Pseudo code: de Casteljau algorithm

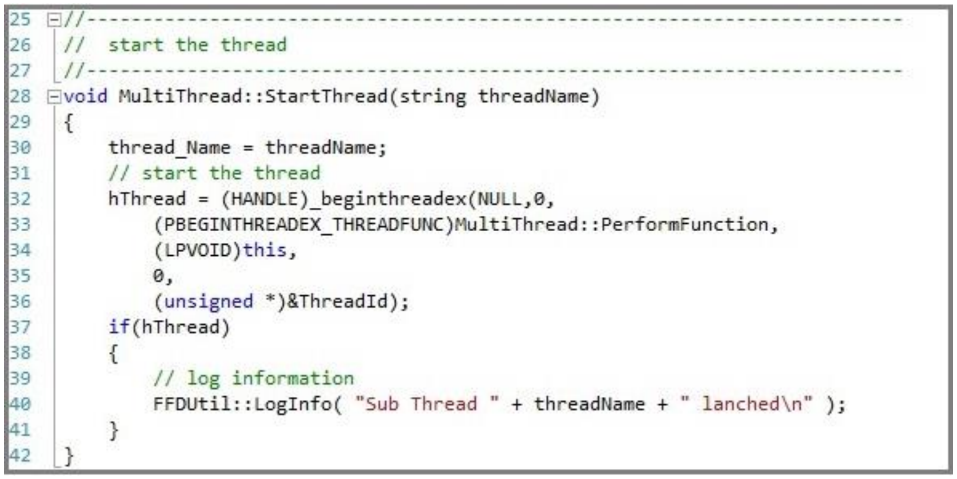
5. Calculate deformed coordinates



```
// pre-calculate bernstein plynomial expansion.
// it only needs to be done once per parameterization
// calculate Xffd
vector<Vector3> ctrlPoints_S, ctrlPoints_T, ctrlPoints_U;
for(int i = 0; i <=l; i++)
{
    // re-initialize the variable on T axes
    if(ctrlPoints_T.size() > 0)
    {
        ctrlPoints_T.clear();
    }
    for(int j = 0; j <=m; j++)
    {
        // re-initialize the variable on U axes
        if(ctrlPoints_U.size() > 0)
        {
            ctrlPoints_U.clear();
        }
        for(int k = 0; k <= n; k++)
        {
            // need to implement later
            ctrlPoints_U.push_back(controlPoints[i][j][k]);
        }
        // call de castuljau algorithm on U axes
        Vector3 temp_U = decasteljauAlgorithm(ctrlPoints_U, n + 1, rectCor.u);
        ctrlPoints_T.push_back(temp_U);
    }
    // call de castuljau algorithm on T axes
    Vector3 temp_T = decasteljauAlgorithm(ctrlPoints_T, m + 1, rectCor.t);
    ctrlPoints_S.push_back(temp_T);
}
// call de castuljau algorithm on S axes
Vector3 Xffd = decasteljauAlgorithm(ctrlPoints_S, l + 1, rectCor.s);
Xffd_all_i.push_back(Xffd);
v->ve = Xffd;
```

Fig. 21 Pseudo code: calculate deformed coordinates

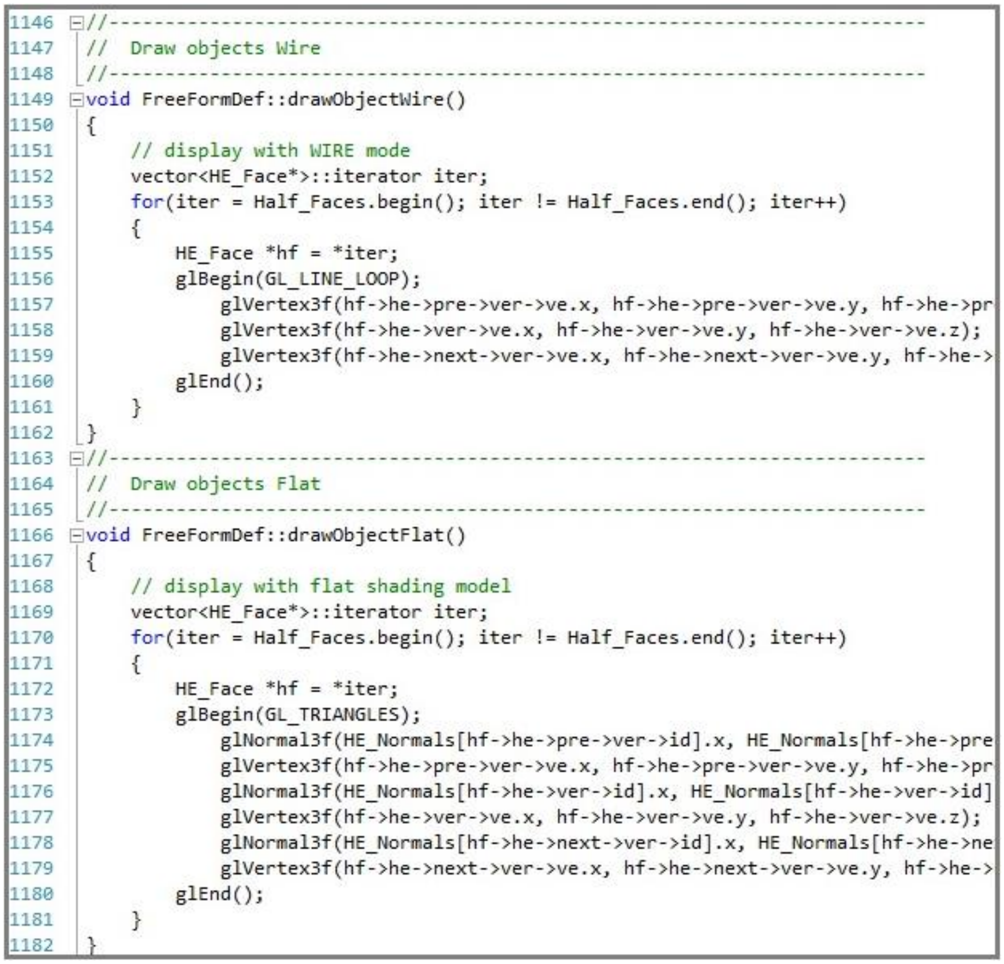
6. Multithreading implementation



```
//-----------------------------------------------------------------------
//  start the thread
//-----------------------------------------------------------------------
void MultiThread::StartThread(string threadName)
{
    thread_Name = threadName;
    // start the thread
    hThread = (HANDLE)_beginthreadex(NULL,0,
        (PBEGINTHREADEX_THREADFUNC)MultiThread::PerformFunction,
        (LPVOID)this,
        0,
        (unsigned *)&ThreadId);
    if(hThread)
    {
        // log information
        FFDUtil::LogInfo( "Sub Thread " + threadName + " lanched\n" );
    }
}
```

Fig. 22 Pseudo code: multithreading

7. Display model



```
//------------------------------------------------------------------------
//  Draw objects Wire
//------------------------------------------------------------------------
void FreeFormDef::drawObjectWire()
{
    // display with WIRE mode
    vector<HE_Face*>::iterator iter;
    for(iter = Half_Faces.begin(); iter != Half_Faces.end(); iter++)
    {
        HE_Face *hf = *iter;
        glBegin(GL_LINE_LOOP);
            glVertex3f(hf->he->pre->ver->ve.x, hf->he->pre->ver->ve.y, hf->he->pr
            glVertex3f(hf->he->ver->ve.x, hf->he->ver->ve.y, hf->he->ver->ve.z);
            glVertex3f(hf->he->next->ver->ve.x, hf->he->next->ver->ve.y, hf->he->
        glEnd();
    }
}
//------------------------------------------------------------------------
//  Draw objects Flat
//------------------------------------------------------------------------
void FreeFormDef::drawObjectFlat()
{
    // display with flat shading model
    vector<HE_Face*>::iterator iter;
    for(iter = Half_Faces.begin(); iter != Half_Faces.end(); iter++)
    {
        HE_Face *hf = *iter;
        glBegin(GL_TRIANGLES);
            glNormal3f(HE_Normals[hf->he->pre->ver->id].x, HE_Normals[hf->he->pre
            glVertex3f(hf->he->pre->ver->ve.x, hf->he->pre->ver->ve.y, hf->he->pr
            glNormal3f(HE_Normals[hf->he->ver->id].x, HE_Normals[hf->he->ver->id]
            glVertex3f(hf->he->ver->ve.x, hf->he->ver->ve.y, hf->he->ver->ve.z);
            glNormal3f(HE_Normals[hf->he->next->ver->id].x, HE_Normals[hf->he->ne
            glVertex3f(hf->he->next->ver->ve.x, hf->he->next->ver->ve.y, hf->he->
        glEnd();
    }
}
```

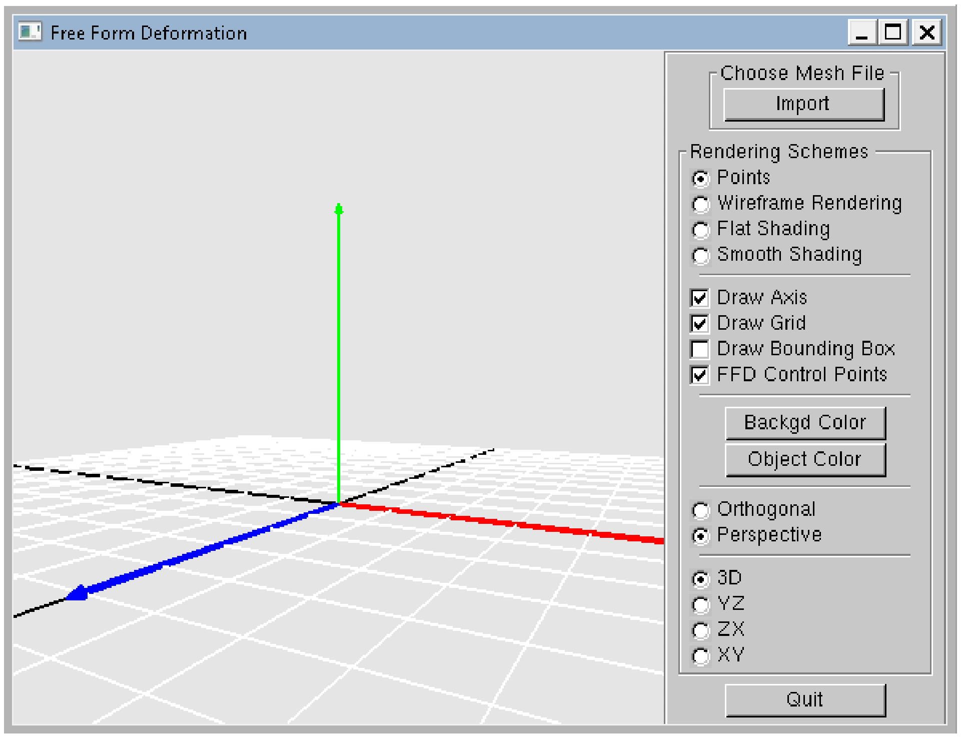
Fig. 23 Pseudo code: display model

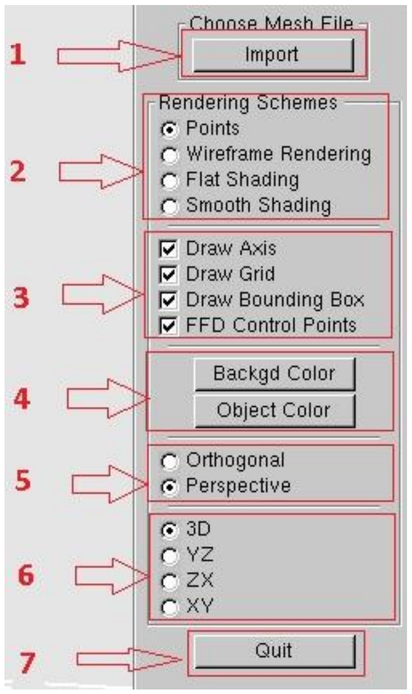
4.4 Results

This chapter mainly presents the results of every function in this FFD application, it includes five parts: how to use this application, the results of showing models by reading mesh data, creating control points, moving control points, deformation.

4.4.1 How to Use the Application

The FFD application can be launched by double clicking the executable file FFD.exe under: FFD\Release folder, the Main UI Framework is as Fig. 24 (a):





(a) (b)

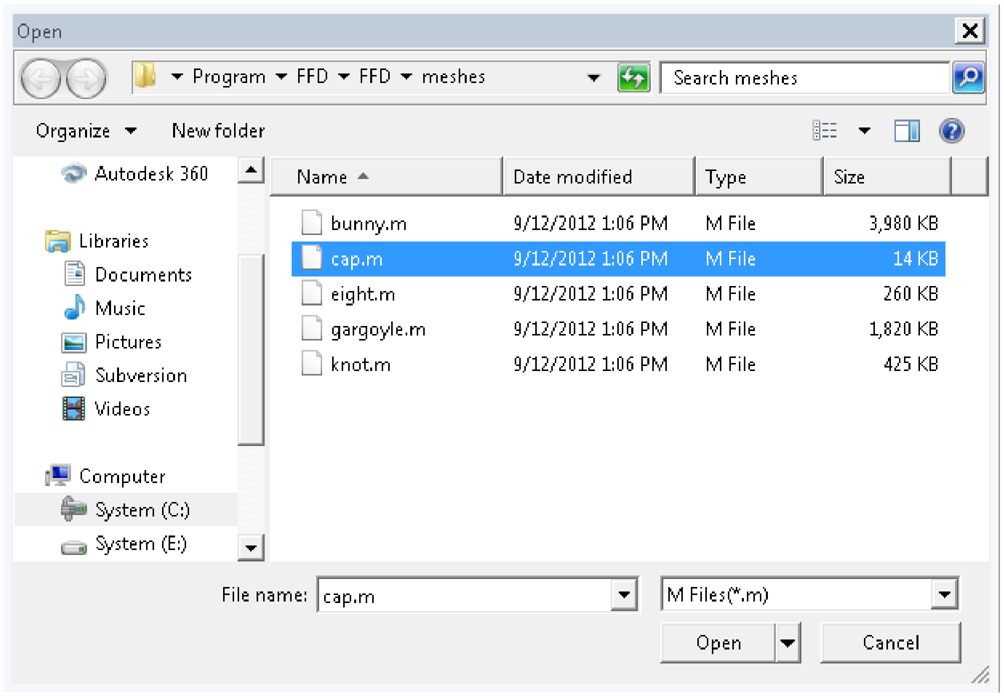
Fig. 24 Main UI Framework (a) and Main Menu (b)

Then perform the functions by choosing OpenGL GLUT controls in Fig. 24 (b).

1. Import the M file by clicking 'Import' button, a choose file dialog will pop up, e.g., Fig. 25.
2. Choose one of the four rendering schemes: Points, Wireframe Rendering, Flat Shading, and Smooth Shading.
3. Whether draw axis, grid, and bounding box, control points and grid, default values are true.
4. Choose background colour or Object Colour, a standard Windows colour dialog will pop up after the user clicks either of these two buttons.
5. Changing the type of projection, you can choose either orthogonal projection or perspective projection.
6. 3D: default view port, YZ: the object will be project to the YZ plane, ZX, the object will be projected to ZX plane, the last one, and Object will be projected to XY plane.
7. Exit the application.
8. To move the control point, by clicking the left button, and moving the mouse.
9. To translate the object or camera, by clicking the middle button, and moving the mouse.
10. To zoom in/out, by clicking the right button and moving the mouse.

4.4.2 Display Models by Reading Mesh Files

1. Choose mesh file from the popped up open file dialog, it’s a standard Windows open file dialog.


Open

Program ▸ FFD ▸ FFD ▸ meshes

Search meshes

Organize ▾ New folder

Autodesk 360

Libraries
Documents
Music
Pictures
Subversion
Videos

Computer
System (C:)
System (E:)

Name	Date modified	Type	Size
bunny.m	9/12/2012 1:06 PM	M File	3,980 KB
cap.m	9/12/2012 1:06 PM	M File	14 KB
eight.m	9/12/2012 1:06 PM	M File	260 KB
gargoyle.m	9/12/2012 1:06 PM	M File	1,820 KB
knot.m	9/12/2012 1:06 PM	M File	425 KB

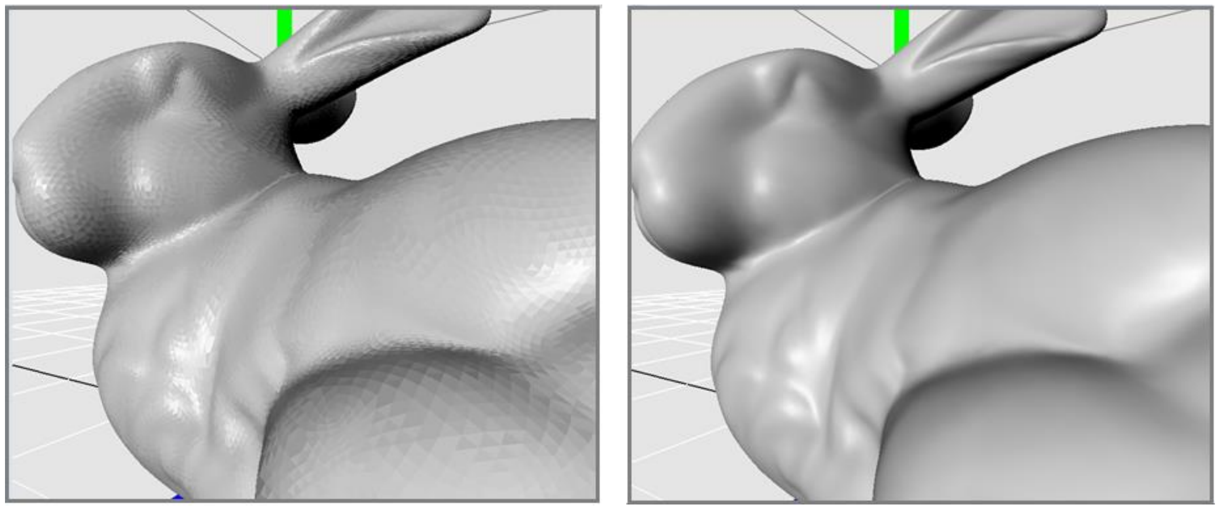
File name: cap.m

M Files(*.m)

Open Cancel

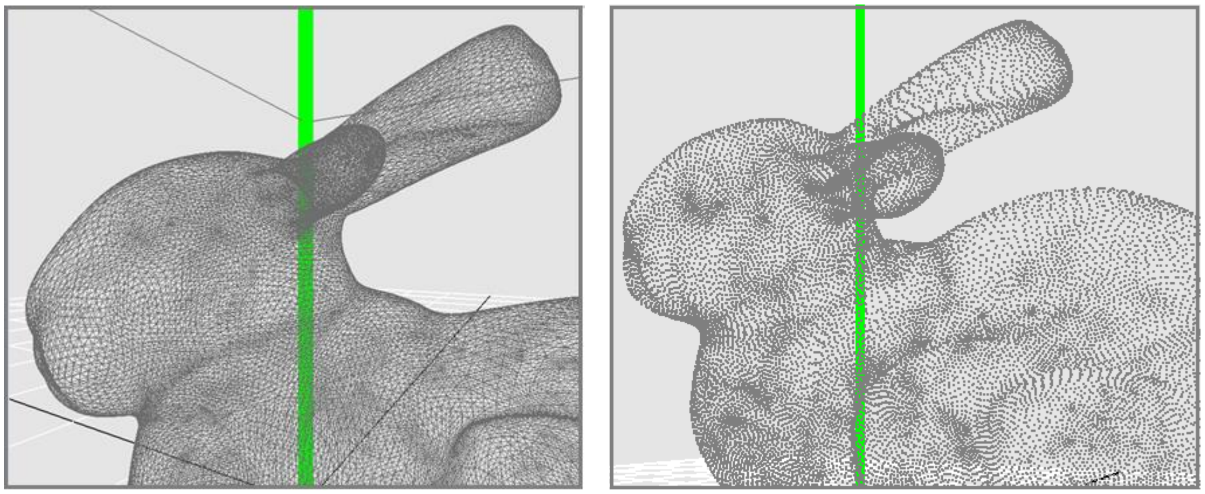
Fig. 25 Open file dialog

2. When we choose bunny.m file, whose file size is the largest, 3980kb, the display results of four rendering schemes: Flat Shading, Smooth Shading, Wireframe and Point are listed as below:



(a) (b)

Fig. 26 Flat shading effect of bunny (a) and Smooth shading effect of bunny (b)

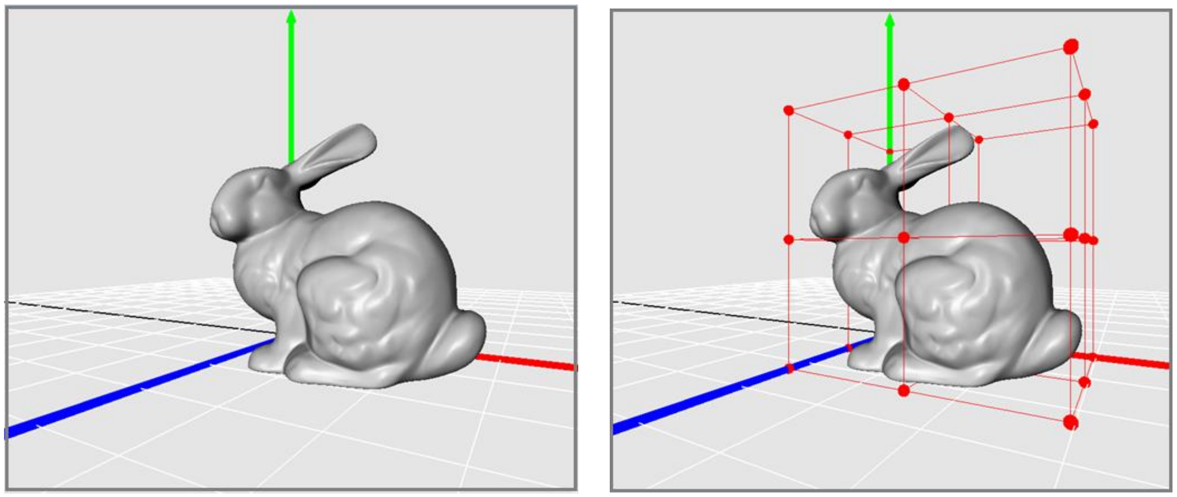


(a) (b)

Fig. 27 Wireframe shading effect of bunny (a) and Point shading effect of bunny (b)

4.4.3 Create/Display Control Points

By default, 3 × 3 × 3 control points will be generated with red colour spheres, and the grid lines between these control points will also be drawn with red colour, they looks as Fig. 28 (b).

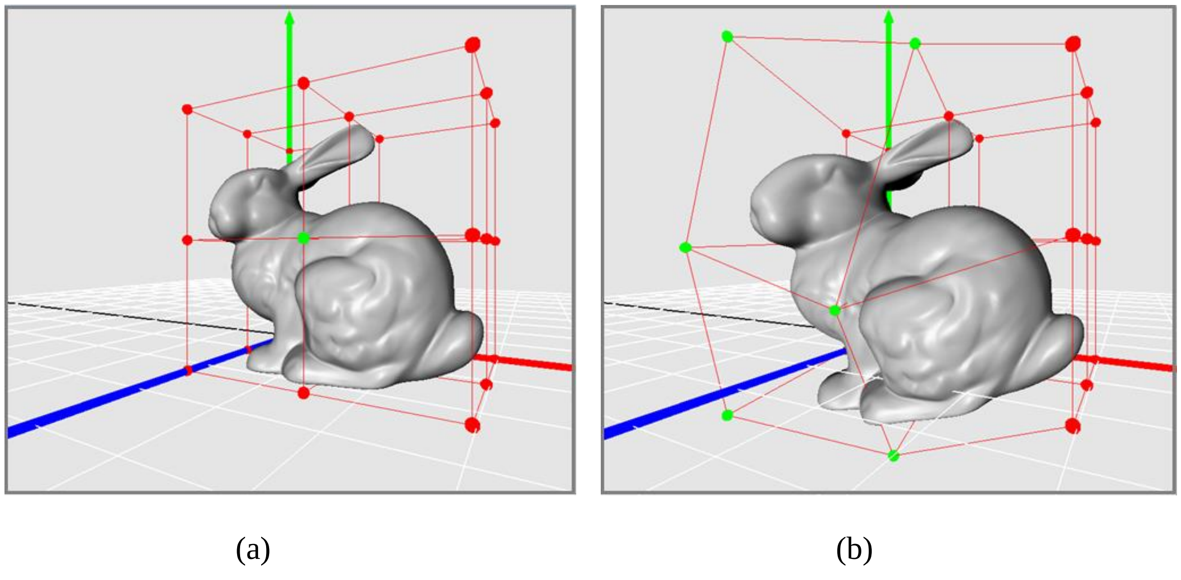


(a) (b)

Fig. 28 Bare bunny (a) and 3×3×3 Control points and grid lines (b)

4.4.4 Select/Move Control Points and the Deformation

When a control point is selected, its colour will change from Red to Green, as shown in Fig. 29 (a), when the control point is dragged to the new position, it will be re-drawn, together with the entire grid lines will also be redrawn according to the new control points' positions, as shown in Fig. 29 (b).



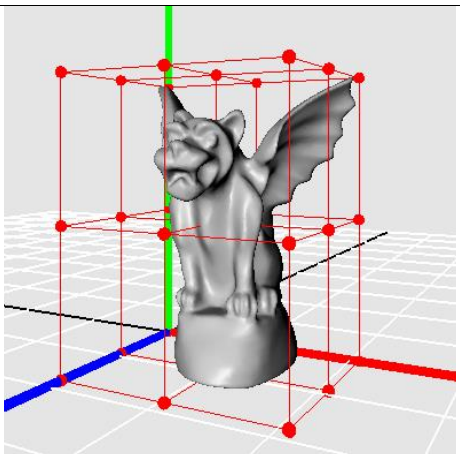
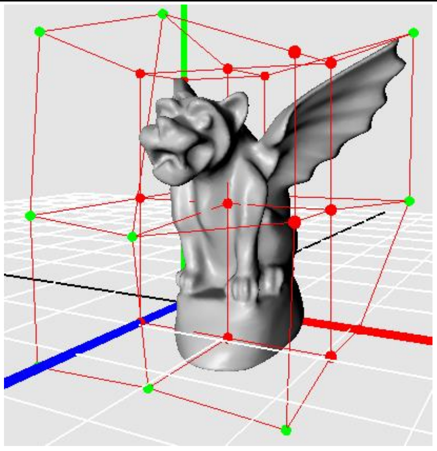
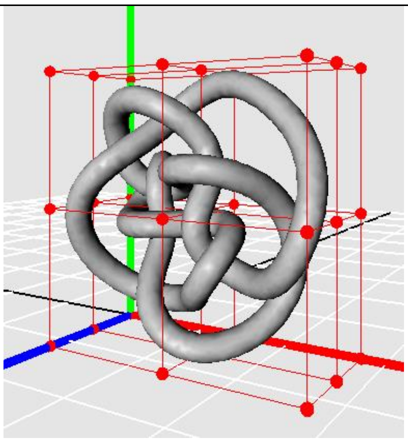
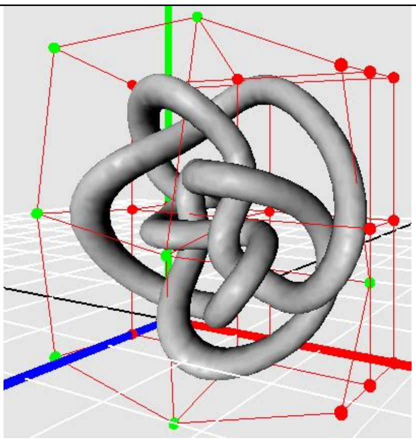
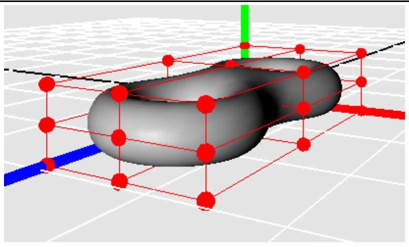
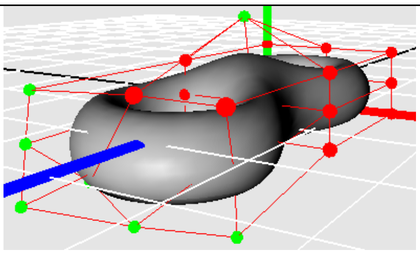
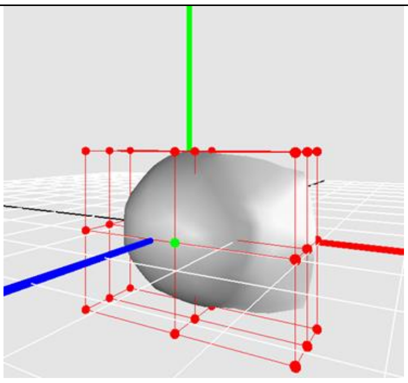
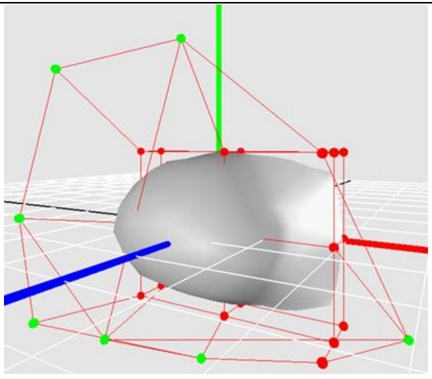
(a) (b)

Fig. 29 Select control point (a) and Move control points and FFD (b)

From above two result pictures, we can see the bunny is deformed with the free-form approach smoothly. If we provide more control points, then the user can change the bunny shape with more complexities, like performing deformation on the more detailed parts of bunny, e.g. mouse part, etc.

The deformation experiments on all the other models are also performed and their result effects are listed in Tab. 7.

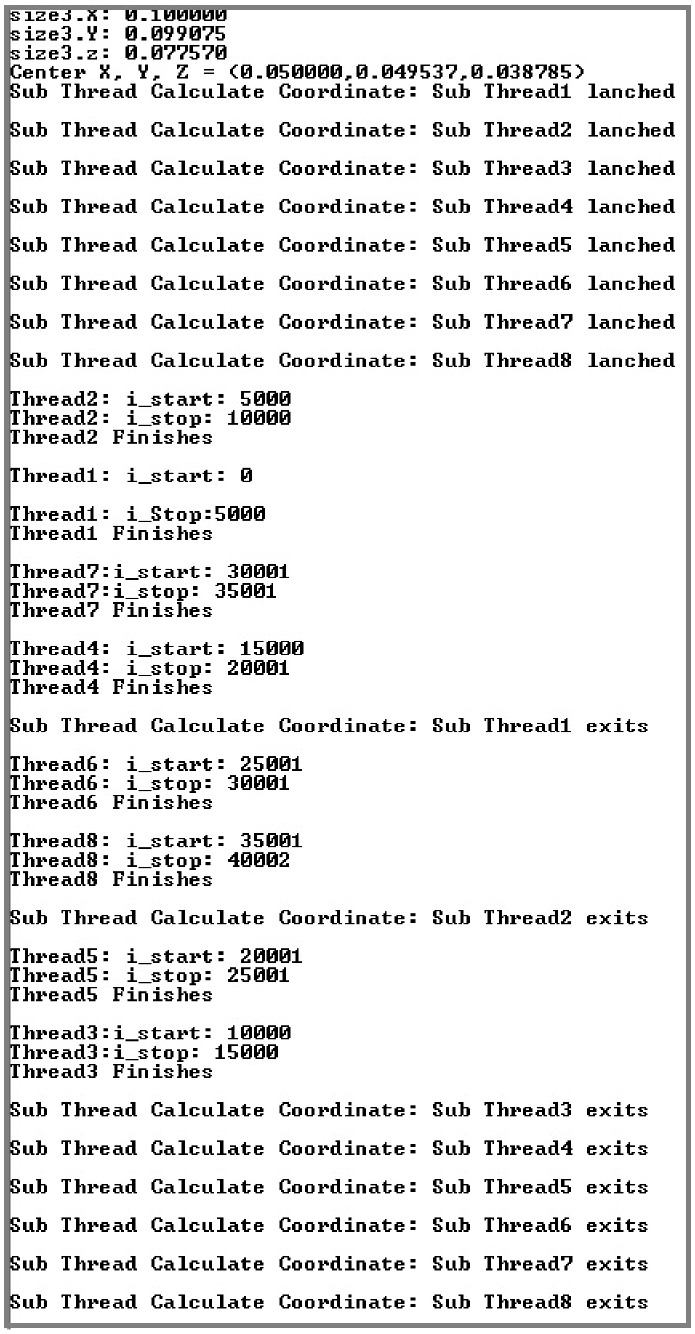
From the experiment results, we can see this FFD application can perform free-form deformation on all given models very well; it has achieved the objectives of this dissertation.

No	Model Name	File Size	Before FFD	After FFD
1	gargoyle	1820kb		
2	knot	425kb		
3	eight	260kb		
4	cap	14kb		

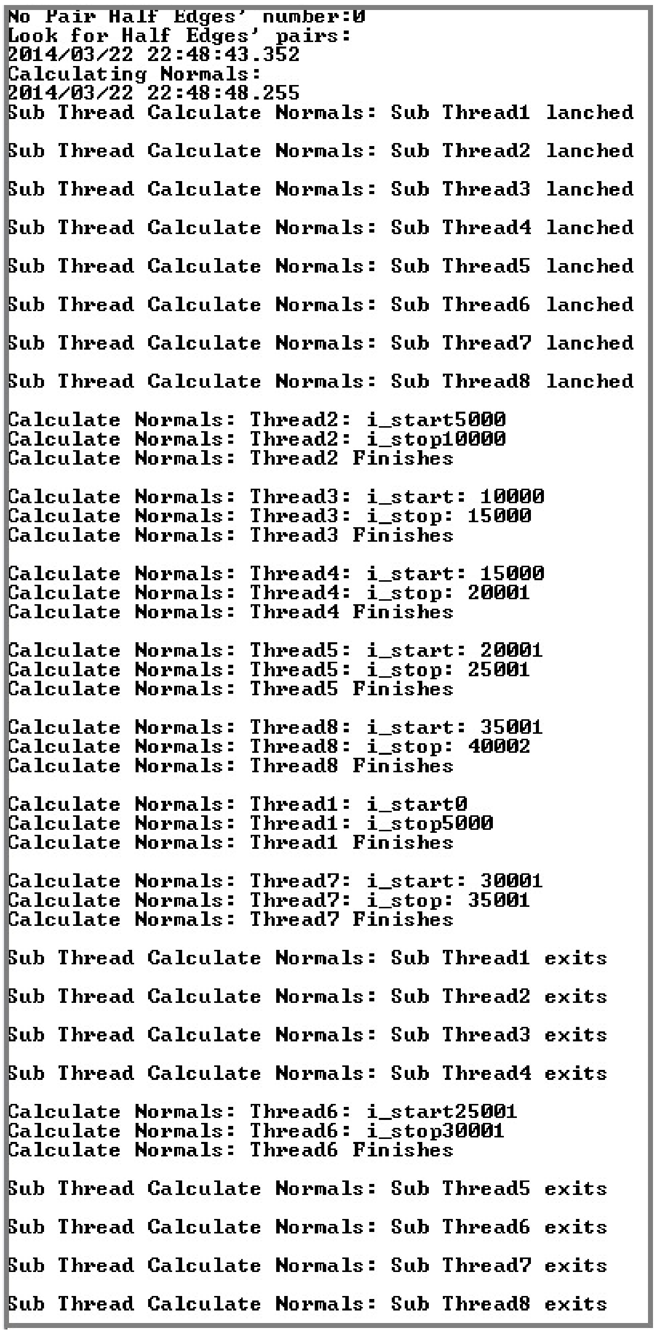
Tab. 7 Results of performing FFD on different models

4.4.5 Multithreading

When we perform FFD for models, the application will run multithreading to calculate the deformed coordinates for all vertices of the model, by applying this technique, it will speed up the performance of the manipulation. The following figures show the log of running multithreading during FFD.



```
size3.X: 0.100000
size3.Y: 0.099075
size3.z: 0.077570
Center X, Y, Z = (0.050000,0.049537,0.038785)
Sub Thread Calculate Coordinate: Sub Thread1 lanched

Sub Thread Calculate Coordinate: Sub Thread2 lanched

Sub Thread Calculate Coordinate: Sub Thread3 lanched

Sub Thread Calculate Coordinate: Sub Thread4 lanched

Sub Thread Calculate Coordinate: Sub Thread5 lanched

Sub Thread Calculate Coordinate: Sub Thread6 lanched

Sub Thread Calculate Coordinate: Sub Thread7 lanched

Sub Thread Calculate Coordinate: Sub Thread8 lanched

Thread2: i_start: 5000
Thread2: i_stop: 10000
Thread2 Finishes

Thread1: i_start: 0

Thread1: i_Stop:5000
Thread1 Finishes

Thread7:i_start: 30001
Thread7:i_stop: 35001
Thread7 Finishes

Thread4: i_start: 15000
Thread4: i_stop: 20001
Thread4 Finishes

Sub Thread Calculate Coordinate: Sub Thread1 exits

Thread6: i_start: 25001
Thread6: i_stop: 30001
Thread6 Finishes

Thread8: i_start: 35001
Thread8: i_stop: 40002
Thread8 Finishes

Sub Thread Calculate Coordinate: Sub Thread2 exits

Thread5: i_start: 20001
Thread5: i_stop: 25001
Thread5 Finishes

Thread3:i_start: 10000
Thread3:i_stop: 15000
Thread3 Finishes

Sub Thread Calculate Coordinate: Sub Thread3 exits

Sub Thread Calculate Coordinate: Sub Thread4 exits

Sub Thread Calculate Coordinate: Sub Thread5 exits

Sub Thread Calculate Coordinate: Sub Thread6 exits

Sub Thread Calculate Coordinate: Sub Thread7 exits

Sub Thread Calculate Coordinate: Sub Thread8 exits
```



```
No Pair Half Edges' number:0
Look for Half Edges' pairs:
2014/03/22 22:48:43.352
Calculating Normals:
2014/03/22 22:48:48.255
Sub Thread Calculate Normals: Sub Thread1 lanched

Sub Thread Calculate Normals: Sub Thread2 lanched

Sub Thread Calculate Normals: Sub Thread3 lanched

Sub Thread Calculate Normals: Sub Thread4 lanched

Sub Thread Calculate Normals: Sub Thread5 lanched

Sub Thread Calculate Normals: Sub Thread6 lanched

Sub Thread Calculate Normals: Sub Thread7 lanched

Sub Thread Calculate Normals: Sub Thread8 lanched

Calculate Normals: Thread2: i_start5000
Calculate Normals: Thread2: i_stop10000
Calculate Normals: Thread2 Finishes

Calculate Normals: Thread3: i_start: 10000
Calculate Normals: Thread3: i_stop: 15000
Calculate Normals: Thread3 Finishes

Calculate Normals: Thread4: i_start: 15000
Calculate Normals: Thread4: i_stop: 20001
Calculate Normals: Thread4 Finishes

Calculate Normals: Thread5: i_start: 20001
Calculate Normals: Thread5: i_stop: 25001
Calculate Normals: Thread5 Finishes

Calculate Normals: Thread8: i_start: 35001
Calculate Normals: Thread8: i_stop: 40002
Calculate Normals: Thread8 Finishes

Calculate Normals: Thread1: i_start0
Calculate Normals: Thread1: i_stop5000
Calculate Normals: Thread1 Finishes

Calculate Normals: Thread7: i_start: 30001
Calculate Normals: Thread7: i_stop: 35001
Calculate Normals: Thread7 Finishes

Sub Thread Calculate Normals: Sub Thread1 exits

Sub Thread Calculate Normals: Sub Thread2 exits

Sub Thread Calculate Normals: Sub Thread3 exits

Sub Thread Calculate Normals: Sub Thread4 exits

Calculate Normals: Thread6: i_start25001
Calculate Normals: Thread6: i_stop30001
Calculate Normals: Thread6 Finishes

Sub Thread Calculate Normals: Sub Thread5 exits

Sub Thread Calculate Normals: Sub Thread6 exits

Sub Thread Calculate Normals: Sub Thread7 exits

Sub Thread Calculate Normals: Sub Thread8 exits
```

(a) (b)

Fig. 30 Multithreading in calculating coordinates log (a) and Multithreading in calculating normal log (b)

4.5 Discussion: Performance Improvement

4.5.1 Background

When the application is ready, during testing, we find it works very well when performing FFD on small-scale models, like the ‘cap’ and ‘eight’ models. But when we are trying to perform FFD on the large-scale models like: ‘gargoyle’ and ‘bunny’, the speed becomes very slow, it will cost the user very long time to see the deformed effect of the models, when the user drags one control point. After carefully consideration, we think this is not acceptable, we need to refine this.

4.5.2 Issue Description

After the deep investigation on the whole process of implementation, we find the root reason for this issue is because: when the user moves a control point, the application will need to re-calculate the deformed coordinates for all the vertices, this will cost very long time to finish for large-scale models. We need to think out solutions to fix this.

4.5.3 Solutions

After performing some researches, we think whether we can divide the new vertices position calculation of the deformed model into several sub-steps, and manipulate every sub-step calculation simultaneously, then we decide to utilize C++ Multithreading technique to improve the application’s performance, get this issue solved.

We create a class called Multithread to implement this function, for the detailed description of the implementation, please refer to 3.2.1.1

4.5.4 Experiments and Results

We perform two sets of experiments on manipulating FFD.

4.5.4.1 Experiment 1: Utilize multithreading on calculating deformed coordinates

This experiment is performed with the following conditions:

1. The application doesn’t utilize multithreading technique.
2. The application utilizes multithreading technique.

3. We perform experiments on all models, ‘cap’, ‘eight’, ‘knot’, ‘gargoyle’ and ‘bunny’, from small-scale model to large-scale model.
4. The application records the time of running the step: calculating deformed coordinates.
5. We compare the results.

The time of calculating coordinates are recorded and the performance improved rate are listed in Tab. 8.

No	**Model Name**	**File Size**	**Multithreading**	**No-Multithreading**	**Improvement**
1	cap	14kb	00:00:00.157	00:00:00.56	-180.36%
2	eight	260kb	00:00:00.555	00:00:02.852	+80.54%
3	knot	425kb	00:00:00.882	00:00:06.291	+85.98%
4	gargoyle	1820kb	00:00:03.331	00:00:12.578	+73.52%
5	bunny	3980kb	00:00:06.733	00:00:56.455	+88.07%

[**Note**: Time unit on column 4# and 5# is ‘day: minutes: seconds: milliseconds’]

Tab. 8 Time of calculating coordinates and performance improved rate

From the above experiment results, we get the following conclusions:

1. For models with small-scale file size, like ‘cap’, the performance of utilizing multithreading is much slower than not using multithreading technique, we think this is because the time of creating eight threads is longer than the time cost in calculating the deformed coordinates of the small-scale model ‘cap’.

2. For models with middle-scale, large-scale file sizes like the models in row #2, #3, #4, #5 of the above table, the performance of utilizing multithreading is much faster than not using multithreading.

3. Though the performance in #1 is slower, but because the file size is very small, only 14kb, so the total cost time of utilizing multithreading is still acceptable for the model ‘cap’.

4. Multithreading technique can improve the performance of the application very much, especially for the large-scale models, improves about 73-88%.

4.5.4.2 Experiment 2: Utilize multithreading on calculating normal of model faces

We also perform an experiment on calculating normal of model faces with multithreading, before the experiment, we thought multithreading will also improve the performance of normal calculation much, but we find it's not, actually the performance of using multithreading is slower than no-multithreading, the experiment result is listed in Tab. 9.

No	Model Name	File Size	Multithreading	No-Multithreading	Improvement
1	cap	14k	00:00:00.180	00:00:00.006	-2900%
2	eight	260k	00:00:00.238	00:00:00.085	-64.29%
3	knot	425k	00:00:00.297	00:00:00.128	-132.03%
4	gargoyle	1820k	00:00:00.899	00:00:00.539	-66.79%
5	bunny	3980k	00:00:01.613	00:00:01.117	-44.40%

[**Note**: Time unit on column 4# and 5# is 'day: minutes: seconds: milliseconds']

Tab. 9 Time of calculating normal of faces on different model

From above experiment result, we also get some conclusions, listed as below:

1. Multithreading can't improve the performance of calculating normal for model faces.
2. Multithreading slower the performance of running application.
3. Base on the above experiment results, we adjust the implementation step; decide to use the single-threading to calculate normal of all faces, instead of using multithreading, which will result in the high performance of the application.

4.6 Summary

This chapter mainly discusses the implementation and results of the application; it includes 6 parts.

Part 1 introduces the development environment. Part 2 describes the application's name, solution's name etc. basic structure information. Part 3 introduces three main features of this application: open mesh file and show model; control points and free-form deformation. Provide the workflow diagram of every feature's implementation. In the same time, this part also describes the implementation of multithreading, draw the workflow diagram for it, and describes the relationship of the classes created for this FFD application. The 'workflow diagram of whole manipulation process' is also provided, and at last this part lists some pseudo code, like: half edge data structure, de Casteljau Algorithm etc.

Part 4 lists the output of the functions: how to use the application, display models by reading mesh files, create and display control points, free-form deformation.

Part 5 discusses the performance issue when performing FFD for large-scale models, provides the multithreading technique to solve this issue, perform some experiments on cases of utilizing multithreading and no-multithreading, records the time of deformed coordinates calculation, at last we conclude, multithreading can improve the performance of FFD application very much, especially for the large-scale models, improves about 73-88%.

5 Conclusions and Future Work

In this dissertation, we present the detailed process of developing a free-form deformation application for users to interactively edit models' shape. This FFD application contains three main processes: read model data from mesh file and display model in OpenGL UI, select and move control point, deform the model. The main features of the application include:

- ✓ Develop a friendly Windows UI by using OpenGL GLUT and Visual Studio 2010 C++ techniques, it includes a set of GLUT controls like: GL_Button, GL_Checkbox etc. for displaying models with more diversities.
- ✓ Read model data from mesh file, by popping up a standard Windows open file dialog and choosing mesh file, it's very convenient for users to manipulate if he/she is familiar with Widows OS.
- ✓ Apply the popular half edge data structure to store model's mesh data; it's very efficient in getting detailed local information of each vertex during FFD process.
- ✓ Create control points by applying free-form deformation techniques, like calculating *S*, *T*, *U* and *s*, *t*, *u*. Draw control points and grid lines between the controls points with Red colour.
- ✓ Select and move control point, by using 'GL_SELECT', 'GLint hits' and 'GLuint buffer' etc. OpenGL API functions. When control point is selected, its colour will change from Red to Green. The control points and the grid line between control points will be re-drawn when the control point is dragged to the new place.
- ✓ Re-calculate global coordinates for every vertex of the model, re-draw and display the deformed model when any control point is moved.
- ✓ Develop the de Casteljau algorithm, which is used in calculating the deformed coordinates of the model.
- ✓ Perform some discussion on the issue: it's very slow when manipulating FFD on the large-scale models, provide an approach 'C++ Multithreading' to solve this issue, the experiments' result shows this approach can improve the performance about 73-88% when manipulate FFD on large-scale models.
- ✓ Develop the multithreading algorithm, and wrap it into a separate C++ class.
- ✓ Generate a set of individual classes for the specific different function of FFD applicant according to the Design Model Theory, like we generate FFDUtil, MultiThread etc. class.

Although, this FFD application can interactively perform the free-form deformation on models powerfully, intuitively, and efficiently, especially for large-scale models, by applying a generalized de Casteljau approach to define the deformable space, utilizing OpenGL API functions to select and move the control points, it still has some parts need to improve, listed as below:

1. The de Casteljau algorithm has drawbacks, like it restricts the ways in which the space surrounding the curve can be altered, so in the future's work, we can try to find approaches to refine this, make this resolution more robust.
2. During the process of selecting control point, it utilizes OpenGL API 'GL_SELECT' method to implement this function, by the investigation, this approach has obsoleted, there are other new approaches to implement this function, like: Color coding, Selection ray (generic), if we have time, we can apply these new approaches to this application.
3. And there are also many other popular approaches can be utilized to perform the 'interactive shape editing', like below, if we have time and opportunities, we can continue to implement them.
    - Mesh Editing with Poisson-Based Gradient Field Manipulation [34]
    - Mean Value Coordinates for Closed Triangular Meshes [35]
    - Mesh Editing based on Discrete Laplace and Poisson Models [36]
    - Differential coordinates for interactive mesh editing [37]

6 References

[1] Edilson De Aguiar,: ‘Animation and Performance Capture Using Digitized Models’, ISBN-10: 3642103154, ISBN-13: 9783642103155, (2010)

[2] Sederberg, T.W., Scott, R.P.: ‘Free-form deformation of solid geometric models’. In: Proc. ACM SIGGRAPH, pp. 151-160 (1986)

[3] Milliron. T., Jensen, R.J., Barzel, R., Finkelstein, A.: ‘A framework for geometric warps and deformations’. ACM Trans. Graph. 21(1), 20-51 (2002)

[4] Botsch, M., Pauly, M., Wicke, M.: ‘Adaptive Space Deformations Based on Rigid Cells’. Computer Graphics Forum, Volume 26, Issue 3, pp. 339-347 (2007)

[5] Kraevoy, V., Sheffer, A.: ‘Mean-value geometry encoding’. International Journal of Shape Modeling, 29-46 (2006)

[6] Sheffer, A., Kraevoy, V.: ‘Pyramid coordinates for morphing and deformation’. In: Proc. 3D Data Processing, Visualization, and Transmission, pp. 68-75 (2004)

[7] Botsch, M., Pauly, M., Gross, M., Kobbelt, L.: ‘Primo: Coupled prisms for intuitive surface modeling’. In: Proc. SGP, pp.11-20 (2006)

[8] Huang, J., Shi, X., Liu, X., Zhou, K., Wei, L.Y., Teng, S.H., Bao, H., Guo, B., Shum, H.Y.: ‘Subspace gradient domain mesh deformation’. ACM Trans. Graph. 25(3), 1126-1134 (2006)

[9] Yizhou Yu,: ‘Mesh Editing with Poisson-Based Gradient Field Manipulation’. In: SIGGRAPH 2004. SIGGRAPH '04 ACM SIGGRAPH 2004, pp.644-651 (2004)

[10] Shi, X., Zhou, K., Tong, Y., Desbrun, M., Bao, H., Guo, B.: ‘Mesh puppetry: Cascading optimization of mesh deformation with inverse kinematics’. ACM SIGGRAPH 2007 (2007)

[11]Eck,M., DeRose,T., Duchamp,T., Hoppe,H., Lounsbery,M., Stuetzle,W.: ‘Multiresolution Analysis of Arbitrary Meshes’. In: SIGGRAPH '95 Proceedings of the 22nd annual conference on Computer graphics and interactive techniques,pp. 173-182 (1995)

[12] Kobbelt, L., Campagna, S., Vorsatz, J., Seidel, H.P.: ‘Interactive multi-resolution modeling on arbitrary meshes’. In: proc. SIGGRAPH 1998, pp. 105-114 (1998)

[13] Guskov, I., Sweldens, W., Schroder, P.: ‘Multi-resolution signal processing for meshes’. In: Proc. SIGGRAPH 1999, pp. 325-334 (1999)

[14] Lee, A., Moreton, H., Hoppe, H.: ‘Displaced subdivision surfaces. In: Proc. SIGGRAPH 2000, pp. 85-94 (2000)

[15] Lipman, Y., Sorkine, O., Cohen-Or. D., Levin, D., Rossl, C., Seidel, H.P.: ‘Differential coordinates for interactive mesh editing’, In: Proc. of Shape Modeling International , pp. 181-190 (2004)

[16] Alexa, M., Nealen, A.: ‘Mesh Editing based on Discrete Laplace and Poisson Models. In: Advances in Computer Graphics and Computer Vision’, Communications in Computer and Information Science. Pp.3-28 (2007)

[17] Fu, H., Au, O.K.C., Tai, C.L.: ‘Effective derivation of similarity transformations for implicit laplacian mesh editing’. Computer Graphics Forum 26(1), 34-45 (2007)

[18] Sundar, H.; Rutgers Univ.,USA; Silver,D.; Gagvani,N.; Dickinson, S.: ‘Skeleton Based Shape Matching and Retrieval’. In:Shape Modeling International, pp.130-139 (2003)

[19] Botsch, M., Sumner, R., Pauly, M., Gross, M.: ‘Deformation transfer for detail-preserving surface editing’. In: Proc. WMV, pp. 357-364 (2006)

[20]Sorkine, O., Alexa, M.: ‘As-right-as –possible surface modeling’. In: Proc. SGP, pp. 109-116 (2007)

[21] Kun, Z., Jin, H., Snyder, J., Hujun,B., Baining, G., Heung-Yeung, S,: ‘Large Mesh Deformation using Volumetric Graph Laplacian’. ACM SIGGRAPH, pp. 496-503 (2005).

[22] Von Funck, W., Theisel, H., Seidel, H.P.: ‘Vector field based shape deformations’. In:Proc. ACM SIGGRAPH, pp. 1118-1125 (2006)

[23] MacCracken, R.; Joy Kenneth I.: ‘Free-form deformations with lattices of arbitrary topology’.In: SIGGRAPH, pp. 181-188 (1996)

[24] Sabine Coquillart. : ‘Extended free-form deformation: A sculpturingtool for 3D geometricmodeling’.In: SIGGRAPH, pp. 187–196 (1990)

[25] Sabine Coquillart, Pierre Janc´ene.: ‘Animated free-form deformation:An interactive animation technique’. In: SIGGRAPH, pp. 23–26 (1991)

[26] Josef Griessmair, Werner Purgathofer.: ‘Deformation of solids with trivariate B-splines’. In Eurographics, pp,137–148 (1989)

[27] ThomasW. Sederberg and Scott R. Parry.: ‘Free-form deformation of solid geometric models’. In: SIGGRAPH, pp.151–160 (1986)

[28] Alan H. Barr.: ‘Global and local deformations of solid primitives’. In: SIGGRAPH, pp 21–30 (1984)

[29] Yu–Kuang Chang and Alyn P. Rockwood.: ‘A generalized deCasteljau approach to 3D free–Formdeformation’. In: SIGGRAPH, pp. 257–260 (1994)

[30] E. Catmull and J. Clark. : ‘Recursively generated B-spline surfaceson arbitrary topological meshes’. Computer-Aided Design,

pp:350–355 (1978)

[31] D. Doo. : ‘A subdivision algorithm for smoothing down irregularly shaped polyhedrons’. In Proceedings of the Int’l Conf.

Interactive Techniques in Computer AidedDesign, pp:157–165 (1978)

[32] D.Doo andM. Sabin.: ‘Behaviour of recursive division surfacesnear extraordinary points’. Computer-Aided Design, pp:356–360 (1978)

[33] Thomas W., Scott R.: ‘Free-Form Deformation of Solid Geometric Models’. ACM0-89791-196-2/86/008/0151 (1986)

[34] Yizhou Y., “Mesh Editing with Poisson-Based Gradient Field Manipulation’. In:SIGGRAPH, pp. 644-651 (2004)

[35] Tao J., Scott S., Joe W., ‘Mean Value Coordinates for Closed Triangular Meshes’. In:SIGGRAPH, pp. 561-566 (2005)

[36] Marc A., Andrew N., ‘Mesh Editing based on Discrete Laplace and Poisson Models’, Advances in Computer Graphics and Computer Vision, Communications in Computer and Information Science Volume 4, pp 3-28 (2007)

[37]Yaron L. , Olga S. , Daniel C. , David L. , Christian R. , Hans-peter S., ‘Differential Coordinates for Interactive Mesh Editing’, Proceeding of SMI '04 Proceedings table of contents ISBN:0-7695-2075-8 doi>10.1109/SMI. 2004.30, pp. 181 - 190, IEEE