5-2003

Extensions to OpenGL for CAGD.

Chunyan Ye
East Tennessee State University

Extensions to OpenGL for CAGD

A thesis
presented to
the faculty of the Department of Computer Science
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Masters of Science in Computer Science

---

by
Chunyan Ye
May 2003

Martin Barrett, Chair
Phil Pfeiffer
Michaele Duncan

Keywords: OpenGL, Triangular patch, Coons patch, Box splines patch

ABSTRACT

Extensions to OpenGL for CAGD

by

Chunyan Ye

Many computer graphic API's, including OpenGL, emphasize modeling with rectangular patches, which are especially useful in Computer Aided Geomeric Design (CAGD). However, not all shapes are rectangular; some are triangular or more complex. This paper extends the OpenGL library to support the modeling of triangular patches, Coons patches, and Box-splines patches. Compared with the triangular patch created from degenerate rectangular Bezier patch with the existing functions provided by OpenGL, the triangular Bezier patches can be used in certain design situations and allow designers to achieve high-quality results that are less CPU intense and require less storage space. The addition of Coons patches and Box splines to the OpenGL library also give it more functionality. Both patch types give CAGD users more flexibility in designing surfaces. A library for all three patch types was developed as an addition to OpenGL.

ACKNOWLEDGMENTS

I do really appreciate Dr. Martin Barrett, my adviser, for his helping with my thesis. Thanks to him for his time, patience, and effort.

Thanks to my committee members for help all the way, especially Dr. Phil Pfeiffer's patience and Michaele's support.

I do appreciate all teachers in Computer Science Department. Without their help, I could not finish.

CONTENTS

Page

ABSTRACT 2

ACKNOWLEDMENTS 3

LIST OF FIGURES 7

Chapter

1. INTRODUCTION 9

2. LITERITURE REVIW 11

Basics of Computer Aided Geometric Design (CAGD) 12

Mathematics of Image Representation 12

Preliminaries 12

Points and Vectors 12

Vector Space and Affine Space 13

Barycentric Combination 13

Parametric Curves and Parametric Surfaces 14

Homogeneous Coordinate System and Rational Curves 15

Tangent 15

Bernstein Polynomials 15

Basics 15

Definition of Bernstein Polynomials 16

Bezier Curve 17

Definition of Bezier Curves 17

Bezier Curve: De Casteljau's Algorithm 18

Bezier Surfaces 19

B-spline 20

B-spline Cuves and Basis 20

B-spline Surfacess 21

Tensor Product Surfaces 21

Splines vs Bezier 22

Rectangular Patches vs Triangular Patches 22
Rectangular Patches 22
Triangular Patches 22
Coons Patches 23
Box Splines 27
OpenGL Review 28
History of OpenGL 28
Advantages of OpenGL 29
Basic Libraries of OpenGL 29
How OpenGL Create Curves and Surfaces 29
Conclusion 30
3. TRIANGLE PATCHES 31
Motivation 31
Barycentric Coordinates in Triangles 31
Bernstein Polynomials and The de Casteljau Algorithm 33
Bernstein Polynomials 33
The de Casteljau Algorithm 33
Implementation of Triangular Patch in OpenGL 35
Prototype of Evaluators of Triangular Patch in OpenGL 35
Implementation of Evaluators of Triangular Patch in OpenGL 37
Comparison of Triangular Patch and Degenerated Cubic Patch 39
Conclusion 43
4. COONS PATCHES 44
Motivation 44
Bilinearly Blended Coons Patches 44
Implementation of Coons Patch in OpenGL 46
Prototype of Evaluators of Coons Patch in OpenGL 46
Implementation of Evaluators of Coons Patch in OpenGL 48
Comparison of Coons Patch and Rectangular Patch with the Same Control Points 50
Conclusion 52
5. BOX SPLINES 53
Motivation 53

Box Splines Definition 53
Algorithm 54
Implementation of Box Splines in OpenGL 56
Prototype of Evalutors of Box Splines in OpenGL 56
Implementation of Evaluators of Box Splines in OpenGL 57
Conclusion 59
6. CONCLUSION 60
Summary of Work 60
Conclusions 60
Future Work 61
BIBLIOGRAPHY 63
APPENDICES 65
Appendix A: myOpenGL.h 66
Appendix B: myOpenGL.cpp 68
Appendix C: A Example of Main Program 83
GROSSARY 89
VITA 95

LIST OF FIGURES

FIGURE Page

1. Cubic Bezier Curve Defined by Four Control Points 18

2. The de Casteljau's Algorithm: The point $b_0^n(t)$ is Obtained from Repeated Linear Interpolation 18

3. B-cubic Bezier Surface with 16 Control Points 19

4. B-spline Curve 21

5. Cubic Bezier Triangular Patch with 10 Control Points 23

6a. C1 and C2 Two Curves Built One Ruled Surface 24

6b. D1 and D2 Two Curves Built One Ruled Surface 24

6c. Coons Patch Built from C1, C2, D1, and D2 Four Curves 25

7. Bicubic Hermite Patches: Points and Vectors 26

8. The Box Spline B{(1, 0); (0,1)} 28

9. The triangle and Plane T Defined by Points (a, b, and c) TITLE OF FIGURE 2 32

10. The de Casteljau Algorithm in Triangular Patch 34

11. The Triangular Patch and Control Points' Order 36

12a. Triangular Patch in Point Mode 38

12b. Triangular Patch in Frame Mode 38

12c. Triangular Patch in Surface Mode 39

13. Degenerated Rectangular Patch Control Points(a) vs Triangular Patch Control Points 40

14a. The Right One is Degenerated Triangular, and the Left is Regular Triangular Patch in Surface Mode 41

14b. The Right One is Degenerated Triangular, and the Left is Regular Triangular Patch in Frame Mode.. 41

15a. New Designed Triangular Patch in Yellow 42

15b. Degenerated Triangular Patch in Yellow 42

16. Comparison of Triangular Patch with Degenerated Triangular Patch in a Designed Feature 43

17. The Bilinear Interpolant for Rcd..... 45

18. Coons Patch: A Bilinearly Blended Coons Patch 46

19. Data Input for 4x4x4x4 Coons Patch 47

20. Coons Patch in Surface Mode 49

21. Coons Patch in Frame Mode 49

22. Coons Patch in Point Mode 50

23. Data Sets for (a) a Rectangular Patch and (b) a Coons Patch 50
24. Rectangular Patch 51
25. Comparison of Rectangular and Coons Patch in Surface Mode 51
26. Comparison of Rectangular and Coons Patch in Frame Mode 52
27.The Subdivision Algorithm for Box Splines 56
28. Control Points Input with Dimension 4 and Dimension 3 57
29. Box Splines Surface in Point Mode 58
30. Box Splines Surface in Frame Mode..... 58
31. Box Splines Surface in Surface Mode 59

CHAPTER 1

INTRODUCTION

The goal of computer graphics is to produce pictures or images by computer with the help of mathematical computation. Computer graphics has developed quickly in recent years, with increased capabilities and reduced cost. Applications of computer graphics include display, design, simulation, and user interfaces. Successful applications of computer graphics in engineering are largely due to the progress of computer aided geometric design (CAGD), which provides the mathematical basis for describing and processing geometric shapes and data. CAGD as a field of computer-aided design (CAD) has developed considerably since its inception in the late 1950s. CAGD is extensively used in a large number of areas, including aerospace, automotive engineering, marine engineering, civil engineering, and electronic engineering. The use of parametric curves and surfaces can be deemed the mathematical base of CAGD (Farin 1993; Angel 2000).

Parametric curves and parametric surface patches are popular and powerful ways of representing curved objects. Several different but related methods exist for describing parametric curves and surfaces, including Bézier and B-spline techniques. The Bézier curve technique is a method for describing a polynomial curve in terms of Bernstein polynomials. A B-spline is another strategy for approximating curves that uses a piecewise polynomial function. For example, four “control points” define a cubic Bézier curve in two dimensions. Similarly, a grid of sixteen control points can define three-dimensional bi-cubic Bézier rectangular surface patches. A three-dimensional Bézier triangular surface patch can be defined as grid of ten control points. Similar methods are used for B-spline curves and surfaces (Farin 1993).

Because most of the early CAD work was developed for the car industry, the initial applications of CAM were for roof, door, hood, and other components with rectangular geometry. This emphasis on rectangles caused the theory of rectangle-based descriptive geometry to develop quickly. For example, he well-known de Casteljau's algorithm, which was developed in 1959, makes rectangular patches easy to compute. Therefore, rectangular patches are widely developed and used in many commercial CAD systems to model all kinds of surfaces (Farin 1986, 1993).

There are, however, some disadvantages to using rectangular Bezier patches to model any surfaces. Triangular surfaces can form among three rectangular Bezier patches in regions where natural

flows come together awkwardly, such as aircraft wings, suitcase corners, or the bulbous bow on ships. Triangular patches can be generated with rectangular patches, but the result is bad (Farin 1986).

Triangular patches generated with rectangular patches often cause problems in using standard algorithms for applying other graphic operations, such as plane/surface intersection or ray tracing. Other surfaces, such as Coons patches and box-splines, have been studied theoretically (Cohen 1984; Farin 1993; de Boor 1994; Farin 1999). They may be useful primitives in rendering surfaces. Therefore, it is a worthy goal to implement efficient tools to model non-rectangular surfaces.

CAGD is usually done with a graphic library or API. OpenGL is hardware independent and widely accepted software for developing graphic applications. It contains two basic libraries, GL and GLU. Some basic geometric primitives such as points, lines, and polygons are implemented in GL. The OpenGL Utility Library (GLU) provides more complex features, such as quadric rectangular surfaces and NURBS curves and surfaces. The algorithms used in GLU to render surfaces and NURBS curves and surfaces are Bézier basis. The OpenGL Utility Toolkit (GLUT), written by Mark Kilgard, is a window-system-independent toolkit. GLUT standardizes and simplifies window and event management. Its functions include initializing and creating a window, handling window and input events, loading the color map, initializing and drawing three-dimensional objects, managing a background process, and running the program. OpenGL produces surfaces based on Bezier rectangular patches (Woo 1999; Angel 2000).

OpenGL has only rectangular Bezier surface rendering functionality. This project proposes to add Bezier triangle patches, Coons patches and box-splines primitive functions to the OpenGL library.

The balance of this proposal is organized as follows. Chapter 2 reviews the basic mathematical theory of Bezier curves and surfaces used in computer graphics; discusses other surfaces, such as rectangular patches and triangular patches used in CAGD; and describes basic libraries in OpenGL. Chapter 3 discusses an implementation for Bezier triangular patch functions in OpenGL. Chapter 4 discusses coding for Coons' patches in OpenGL. Chapter 5 discusses coding for implementing box-splines in OpenGL. Chapter 6 provides a conclusion and areas of future research.

CHAPTER 2

LITERITURE REVIEW

Computer technology is widely used in daily activities, such as filmmaking, publishing, banking, engineering, and education. Many computer applications use a graphical interface to display information. Computer graphics is the study and realization of a complex process to produce pictures and images from a physical or conceptual object. Computer graphics creates synthetic images by programming the geometry and appearance of the contents of the images, and by displaying the results of that programming on appropriate display devices that support graphical output. The programming can be done with the support of a graphics Application Program Interface (API) that does most of the detailed work of rendering the scene that the program defines (Angel 2000).

A picture or image is composed of one or more geometric entities. The creation of a geometric entity has four major steps: modeling, geometric processing, rasterization, and display. In order to make a computer image, programmers develop codes to model the geometric entities; assemble these entities into an appropriate geometric space with proper relationships; define and present the appearance of the entities with assigned shades or colors; specify how the scene is to be viewed; and make it display on the graphic device appropriately (Angel 2000).

Advances in hardware and software and demands of the user community have led to improvements in computer graphics. Engineers and architects use computers to design images or products and computer-aided design (CAD) and computer-aided manufacture (CAM) are two important areas where computer graphics plays a central role. Both computer-aided design (CAD) and computer-aided manufacturing (CAM) are widely used in a large number of areas, including aerospace, automotive engineering, marine engineering, civil engineering, and electronic engineering. The application of computer graphics in engineering is the foundation of computer aided geometric design (CAGD), which provides the mathematical basis to describe and process geometric shapes and data. CAGD as a field of CAD has developed considerably since its inception in the late 1950's (Farin 1993; Angel 2000; Farin 2001).

Basics of Computer Aided Geometric Design (CAGD)

Computer-aided geometric design started in the late 1950s. Its actual application began with automated machinery to shape blocks of wood or steel and car parts (Farin 1993).

The most used descriptive methods of geometric shape are parametric curves and surfaces. Early developments in CAGD included the theory of Bézier surfaces and Coons patches, later combined with B-spline methods. Bézier curves and surfaces were introduced by P. de Casteljau at Citroën in 1959, then by P. Bézier at Rénault. De Casteljau's work slightly earlier than Bézier's, was never published. Citroën needed to convert data from 2D blueprint information into coordinates to drive a three-dimensional milling machine. De Castelijau invented "Courbes à Poles", known as Bézier curves today. Pierre Bézier, a mechanical engineer at Rénault, had learned about de Castelijau's work. He created a system with the same function, and Rénault let him publish it. Thus, the whole theory of polynomial curves and surfaces in Bernstein form now has Bézier's name and Bézier curves came to dominate CAGD (Farin 1993).

Besides Bezier curves and surfaces, two other techniques emerged from the automotive field: Coons' patches from Ford and Gordon surfaces from General Motors. They differ from Bezier methods, in that they "fill in" curve networks in order to create the surface instead of using control nets in Bézier or B-spline form (Farin 1993).

Another development was the introduction of splines. Based on the theory of interpolating piecewise cubic curves, or C^2 cubic splines, Ferguson developed a package for Boeing in the late 1950's. Splines were first studied by Schoenberg in 1946. De Boor of General Motors advanced the theory of B-spline curves and surfaces (Cohen 1984; Farin 2001).

Gordon and Reesenfeld, using de Boor's work, found that B-splines could be used in the same way that Bézier curves could. They showed that Bézier curves were just a special case of B-spline curves and made possible a unification of systems based on splines and those on Bézier curves. Today, Bézier and B-spline representations of curves and surfaces have become an industrial standard (Farin 1993, 2001).

Mathematics of Image Representation

Preliminaries

Points and Vectors. The fundamental 3-dimensional spatial entities that form the basis for all operations in computer graphics are points and vectors. A point is a location in space, and a vector is a

directed line segment in space. Points are elements in three-dimensional Euclidean space $\mathbf{E}^3$ and are described in capitalized bold letters such as **P** and **Q**, and vectors are elements in the three-dimensional real vector space $\mathbf{R}^3$ and are denoted as lower case letters with an arrow above such as $\vec{v}$ and $\vec{w}$. For each pair of points **P** and **Q**, there exists a unique vector $\vec{v}$ such that $\vec{v} = \mathbf{P} - \mathbf{Q}$, where $\mathbf{P}, \mathbf{Q} \in \mathbf{E}^3$, $\vec{v} \in \mathbf{R}^3$ (Angel 2000; Farin 1993).

Vector Space and Affine Space. A vector space has two distinct entities, vectors and scalars. A scalar is a real number and a unit of measurement. A vector $\vec{v} \in \mathbf{R}^3$ is an ordered triple as (x, y, z), where x, y, z are scalars. The length of $\vec{v}$ is calculated as $|\vec{v}| = \sqrt{x^2 + y^2 + z^2}$. $\vec{v}$ can be normalized into a unit vector $\vec{u} = \vec{v} / |\vec{v}|$ (Angel 2000; Watt 2000).

A vector space has vector-vector addition and scalar-vector multiplication operations. For n vectors, vector-vector addition can be defined as follows:

$$sum\vec{v} = \vec{v}_1 + \vec{v}_2 + ... + \vec{v}_n = (x_1 + x_2 + ... + x_n, y_1 + y_2 + ... + y_n, z_1 + z_2 + ... + z_n).$$

For a scalar-vector multiplication, scalar r multiplied times vector $\vec{v}$ is defined as $r * \vec{v}$ = (r*x, r*y, r*z). If a vector $\vec{v}$ has the same scalar as $\vec{u}$ but opposite direction, it is defined $\vec{v} = -\vec{u}$ or $\vec{u} = -\vec{v}$ and $\vec{u} + \vec{v} = 0$. Therefore, vector-vector subtraction can be defined as result $\vec{v} = \vec{v}_1 - \vec{v}_2 = \vec{v}_1 + (-\vec{v}_2)$ (Faux 1979; Angel 2000; Watt 2000).

An affine space extends a vector space by adding an additional object: the point. Affine addition is defined as $\mathbf{L} = \mathbf{P} + t\vec{v}$, where t is a scalar. This defines a line from **P** in the direction $\vec{v}$. An equivalent two-point form can be derived. For two points **R** and **Q** in a line, define the direction vector $\vec{v} = \mathbf{R} - \mathbf{Q}$. Then any point $\mathbf{P} \in \mathrm{E}^3$ on the line satisfies $\mathbf{P} = \mathbf{Q} + \alpha\vec{v} = \mathbf{Q} + \alpha(\mathbf{R}-\mathbf{Q}) = \alpha\mathbf{R} + (1-\alpha)\mathbf{Q}$. This is called the barycentric form of the line when rewritten as $\mathbf{P} = \alpha_1\mathbf{R} + \alpha_2\mathbf{Q}$, where $\alpha_1 + \alpha_2 = 1$ (Farin 1993; Angel 2000).

In affine space, the operations of translation, scaling, and rotation are invariant (Farin 1993).

Barycentric Combination. Barycentric combinations are weighted sums of points, where all the weights sum to 1. The general barycentric combination is

$$\mathbf{P} = \sum_{i=0}^{n} \alpha_i \, \mathbf{P_i} \quad \text{where } \mathbf{P_i} \in \mathbf{E}^3, \ \alpha_i \text{ is scalar, and } \alpha_1 + \alpha_2 + \ldots + \alpha_i = 1.$$

(Farin 1993).

The convex combination is an important special case of barycentric combinations. The weights or coefficients α_i of points of a convex combination are nonnegative. The set of points so defined is called the convex hull of the $\mathbf{P_i}$. A convex hull is the smallest tight-fitting stretched surface containing the given set of points. The concept of the convex hull is very important in computer graphic design. A convex combination of points is always "inside" those points (Farin 1993; Angel 2000).

Barycentric coordinates are another method of introducing coordinates into an affine space. Through barycentric combinations, a point can be checked if it is inside the convex hull. For example, a given triangle with vertices A, B, C and a given point M, $M = r_1A + r_2B + r_3C$, if $r_1 + r_2 + r_3 = 1$, if $0 \leq r_1 \leq 1$ for - = 1..3, M is inside the triangle, otherwise M is not inside the triangle (Farin 1993).

Parametric Curves and Parametric Surfaces

In describing curves or surfaces, an auxiliary parameter is used to represent the position of a point. This kind of curve or surface is parametric curve or surface. A parametric curve in space can be described as the following:

$x = x(u), y = y(u), z = z(u)$

where x(u), y(u), and z(u) are three functions (for example, polynomials) mapping a real value parameter u to a point in the curve (Faux 1979).

A parametric surface in space can be described as the following:

$x = x(u, v), y = y(u, v), z = z(u, v)$

where x(u, v), y(u, v), and z(u, v) are three functions mapping real value parameters u, v to a point in the surface. The parameter can be any real value, but, for simplicity, are often restricted to [0,1] (Faux 1979).

The parametric form of curves and surfaces is extensively used in computer graphics, because parametric curves and surfaces are easily manipulated (Faux 1979).

Homogeneous Coordinate System and Rational Curves

Programmers treat points as vertices described as (x, y, z) that define geometric objects in a user-given coordinate system. Every coordinate system has its origin. Usually the origin of a coordinate system is (0,0,0) (Angel 2000).

Every object has its own properties such as lines, angles, and position in a defined coordinate system. It is known that two points subtracted make a vector. The vector from (0,0,0) to (2,2,3) is the same as that from (1,1,1) to (3,3,4) with the same magnitude and direction. To distinguish them, there is a solution using homogeneous coordinates in describing a coordinate. Homogeneous coordinates use four-dimensional vectors; for example, if $P = a_1 v_1 + a_2 v_2 + a_3 v_3 + a_0$, then *P* may be represented as (a_1, a_2, a_3, a_0). Homogeneous coordinates permit rotation, scaling, reflection, and shearing transformations to be represented by matrix multiplication (Faux 1979; Angel 2000).

Tangent

For a fixed point A and a moving point B on a curve moving toward A, the vector from A to B approaches the tangent vector at A, and the line that contains the tangent vector is the tangent line.

To compute the tangent line at a point P (x_1, y_1), take

$y = y_1 + f'(x_1)(x - x_1)$

where $f'(x_1) = \frac{df}{dx}$ at $x = x_1$.

where $\frac{df}{dx}$ at $x = x_1$ and $\frac{df}{dy}$ at $y = y_1$.

(Faux 1979).

Bernstein Polynomials

Basics. Most geometric objects have complex curves or surfaces that are not easily represented by simple analytic functions. Complex curves are divided into small pieces and are designed in a piecewise manner. Complex surfaces are divided into patches (Faux 1979).

Polynomials allow curves and surfaces to be designed with ease, and manipulated in simple ways. It is easy to differentiate and integrate polynomials and polynomials. For this reason, piecewise curves and surface patches are usually represented with polynomial functions.

Any polynomial function that has degree less than or equal to, can be written as

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

which represents *P* in the power basis {1, x, x^2, …, x^n} (Faux 1979).

Definition of Bernstein Polynomials. A Bernstein polynomial of degree n is defined as:

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-1}$$

for i = 0, 1, …, n, where

$$\binom{n}{i} = \begin{cases} \dfrac{n!}{i!(n-i)!} & if \quad 0 \le i \le n \\ 0 & else \end{cases}$$

For a given n, the set of Bernstein polynomials form a basis. The Bernstein Basis has four properties: recursion, partition unity, non-negativity, and derivatives.

Recursion in Bernstein polynomials is given by

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

with $B_0^0(t) \equiv 1.$

and $B_i^n(t) \equiv 0 \quad \text{for } j \notin \{0,...,n\}.$

The proof:

$$\begin{aligned} B_i^n(t) &= \binom{n}{i} t^i (1-t)^{n-i} \\ &= \binom{n-1}{i} t^i (1-t)^{n-i} + \binom{n-1}{i-1} t^i (1-t)^{n-i} \\ &= (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \end{aligned}$$

The partition of unity property means that, for all t,

$$\sum_{j=0}^{n} B_j^n(t) \equiv 1$$

The proof:

$$1 = [t + (1-t)]^n = \sum_{j=0}^{n} \binom{n}{j} t^j (1-t)^{n-j} = \sum_{j=0}^{n} B_j^n(t)$$

The non-negativity property means that for $t \in [0,1]$, the Bernstein polynomial is non-negative:

$$B_i^n(t) \geq 0 \text{ , for } t \in [0,1]$$

The proof:

$$\binom{n}{i} \geq 0$$

$$t \geq 0 \qquad for \qquad 0 \leq t \leq 1$$

$$(1-t) \geq 0 \qquad for \qquad 0 \leq t \leq 1$$

The derivatives of a Bernstein polynomial are computed as

$$\frac{d}{dt} B_i^n(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t))$$

(Farin 2002).

Bézier Curve

Bezier curves and patches are among the most fundamental tools in computer graphics and computer aided modeling (Farin 1993).

Definition of Bézier Curves. A Bézier curve of degree n in Bernstein form is defined by the formula

$$P(u) = \sum_{i=0}^{n} B_i^n(u) p_i$$

where the p_i are control points, the $B_i^n(u)$ are the Bernstein basis functions, and u $\in [0,1]$.

A Bézier curve is an affine combination of its control points, and any affine transformation of a curve is the curve of the transformed control points (Farin 1993).

P(u) on a Bézier curve is the weighted average of all control points. Figure 1 shows a cubic Bézier curve defined by four control points p1, p2, p3, and p4 (Farin 1993; Angel 2000).

Figure 1. Cubic Bezier Curve defined by Four Control Points (Angel 2000)

A Bézier curve has all the properties of the Bernstein functions. In addition, a Bézier curve of degree n passes through P_0 and P_n. The first and last control points are as shown in the above figure, where the cubic curve passes through P1 and P4 respectively (Farin 1993).

Bézier Curve: De Casteljau's Algorithm. Finding a point p(*u*) on the curve with a particular *u* can be easily done with de Casteljau's algorithm described next.

Given a set of control points $b_0, b_1, \ldots, b_n \in \mathbf{E}^3$ for a Bézier curve *P(u)*, and a parameter $t \in \mathbf{R}$, set $b_i^r(t) = (1-t)b_i^{r-1}(t) + tb_{i+1}^{r-1}(t)$, where r = 1, …, n; i = 0, …, n-r, and $b_i^0(t) = b_i$. Then $b_0^n(t)$ is P(t). The figure 2 shows a cubic Bezier curve with the de Casteljau's algorithm at t=0.5 (Farin 1993).

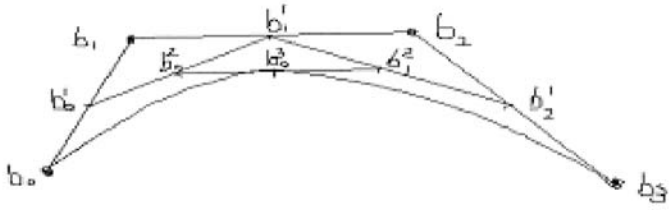

Figure 2. The de Casteljau's Algorithm: the point $b_0^n(t)$ is obtained from repeated linear interpolation. The cubic case n=3 is shown for t=1/2. (Farin 1993)

Bézier Surfaces

A tensor product Bézier surface in Bernstein form is described by a two-dimensional set of control points $\mathbf{p}_{i,j}$ with two parameters u and v. Its equation is shown below:

$$P(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u) B_j^n(v) P_{i,j}$$

Here, $B_i^m(u)$ and $B_j^n(v)$ are Bernstein functions in the *u*- and *v*- directions, respectively (Farin 1993).

Bezier surfaces have all properties of Bezier curves. De Casteljau's algorithm can be extended to Bézier surfaces and can compute the corresponding point on a Bézier surface as it does in a Bézier curve (Farin 1993).

Figure 3 is an example of Bezier rectangular patch, which is created with 16 control points.

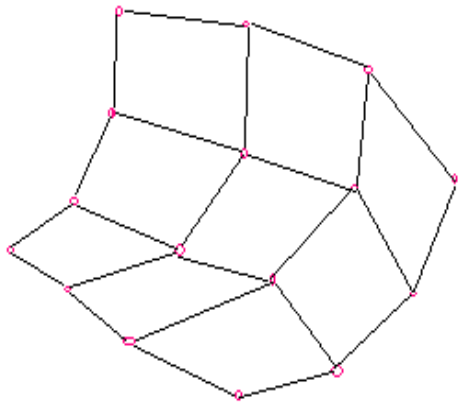

Figure 3. Bi-cubic Bezier Surface with 16 Control Points (Farin 1993)

B-spline

Bézier curves are a powerful tools in CAGD and widely used, but they have certain limitations: if a curve has a complicated shape, it either requires a higher degree Bézier curve or several piecewise Bézier curves together to model it. B-splines overcome these limitations (Angel 2000).

B-spline curves are polynomial curves and generalizations of Bézier curves and are developed to use lower degree curve segments. B-spline curves have higher degrees of freedom for curve design (Angel 2000).

B-spline Curves and Basis. A B-spline curve of degree m can be defined with $n + 1$ control points $p_0, p_1, ..., p_n$ and a knot vector $U = \{ u_0, u_1, ..., u_m \}$ in the following formula:

$$P(u) = \sum_{i=0}^{n} N_{i,m}(u) P_i$$

where $N_{i,m}(u)$ are B-spline basis functions are polynomials of degree m (Angel 2000).

B-spline curves and Bezier curves are very similar. The set of m + 1 non-decreasing real numbers, $u_0 <= u_2 <= u_3 <= ... <= u_m$ that subdivide the interval $[u_0, u_m]$ into knot spans are called knots. If the knots are separated equally ($u_{i+1} - u_i$ is a constant for $0 <= i <= m - 1$), it is called uniform; otherwise, it is non-uniform. All B-spline basis functions have their domain on $[u_0, u_m]$; the closed interval [0,1] is used for simplicity (Angel 2000; Faux 1979; Farin 1993).

The i-th B-spline basis function of degree p ($N_{i,m}(u)$) is defined recursively as shown below:

$$N_{i,0}(u) = \begin{cases} 1 & if \quad u_i \le u \le u_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

Figure 4 is an example of B-spline curve (Angel 2000).

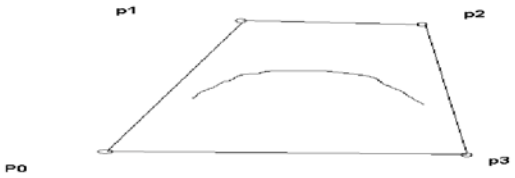

Figure 4. B-spline Curve (Angel 2000)

Non-Uniform Rational B-Splines (NURBS) are an extension of B-splines that can represent the quadric curves, including circles, ellipses, and many other curves that cannot be represented by polynomials. In NURBS weights w_i are added with corresponding control points $\mathbf{p}_i$ as the fourth component. If all weights are equal to 1, the NURBS curve is just a regular B-spline curve (Angel 2000).

B-spline Surfaces. The B-spline surface is defined as follows:

$$\mathbf{p}(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} N_{i,p}(u) N_{j,q}(v) \mathbf{p}_{i,j}$$

where $p_{i,j}$ is the control points in surface, and $N_{i,p}(u)$ and $N_{j,q}(v)$ are B-spline basis functions of degree p and q, respectively (Angel 2000).

Tensor Product Surfaces

The tensor product (or Cartesian product) technique constructs surfaces by "multiplying" two curves together. Given two Bézier curves, the tensor product method constructs a surface by multiplying the basis functions of the first curve with the basis functions of the second and uses the results as the basis functions for a set of two-dimensional control points. Surfaces generated in this way are called tensor product surfaces. Therefore, Bézier surfaces, B-spline surfaces, and NURBS surfaces are all tensor product surfaces (Faux 1979; Farin 1993).

Splines vs Bezier

Because the Bezier surface passes exactly through the control points and has the advantages of local control, easy subdivision, and easy computation, it is widely used. However, in some applications, higher continuity is required, and in that case, B-Splines are mandatory. The common part is that a Bezier curve is a just special case of B-spline curve (Farin 1993).

Rectangular Patches vs Triangular Patches

Most of the early CAD work was done in the car industry, where the applications of CAM were for roof, door, hood, and similar feature–all rectangular geometric objects. Accordingly, the theory of rectangular patches is widely developed and used in many commercial CAD systems to model surfaces (Farin 1993, 1986).

Rectangular Patches

Rectangular patches are simple applications of Benzier surfaces and B-spline surfaces (cf. § 2.2). The formula for rectangular patch is as follow:

$$P(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) B_{n,j}(v) P_{i,j}$$

where m = n = 4, and Bi,n and Bj,m are the cubic Bernstein polynomials (Angel 2000).

Triangular Patches

Not all geometric objects have rectangular appearances. Some sharp triangular shapes, such as wings of birds, are difficult to model with rectangular Bezier patches. Even though a degenerate rectangular patch may be used to create a triangular patch, the result is more like a rectangular patch rather than a triangular patch and can interfere with the operation of other algorithms in rendering, such as ray tracing (Farin 1986).

The development of the triangular patch will solve these problems. In linear barycentric terms, for a given triangle with vertices a, b, c, all in $\mathbf{E}^3$, any point p inside this triangle has form p = ua + vb +wc, where u +v + w = 1.

The form of a triangular Bezier patch, in Bernstein polynomials, is

$$P(t) = \sum_{|\mathbf{I}|=n} p_i B_i^n(t)$$

where $B_i^n(t) = \binom{n}{\mathrm{I}} u^i v^j w^k = \frac{n!}{i!\,j!k!} u_i v^j w^k ;|\,\mathrm{I}\,| = n$

Figure 5 is an example of a cubic Bezier triangle patch, created with 10 "control points" (Farin 2002).

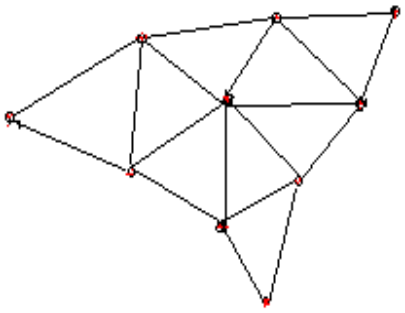

Figure 5. Cubic Bezier Triangular Patch with 10 Control Points (Farin 2002)

Coons Patches

A Coons' patch solves the problem of defining a surface from a given network of parametric curves. These curves do not have to be of the same type (Farin 1993). There are bilinearly blended Coons' patches and bicubically blended Coons' patches. The bilinearly Coons' patch interpolates to two boundary curves. It is defined as follows:

Given four curves C1(u), C2(u) and D1(v), D2(v), where u, $v \in [0,1]$, a surface X is found to have these four curves as boundary curves:

X(u,0) = C1(u), X(u, 1) = C2(u), X(0,v) = D1(v), X(1, v) = D2(v).

There are two ruled surfaces from above: C1,C2 two curves form a curve and D1, D2 form another one. They are denoted as Rc and Rd.

$$Rc(u,v) = (1-v)X(u,0) + uX(u,1) \text{ and } Rd(u,v) = (1-u)X(0,v) + uX(1,u)$$

The bilinear interpolant makes Rcd to the four corners:

$$Rcd(u,v) = [1-u \quad u]\begin{bmatrix} X(0,0) & X(0,1) \\ X(1,0) & X(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

These two ruled surfaces built up a bilinearly Coons' patch governed by the bilinear interpolant, described as: X = Rc + Rd – Rcd, and

$$X(u,v) = [1-u \quad u]\begin{bmatrix} X(0,v) \\ X(1,v) \end{bmatrix} + [X(u,0) \quad X(u,1)]\begin{bmatrix} 1-v \\ v \end{bmatrix} - [1-u \quad u]\begin{bmatrix} X(0,0) & X(0,1) \\ X(1,0) & X(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

see Figure 6a, 6b, and 6c.

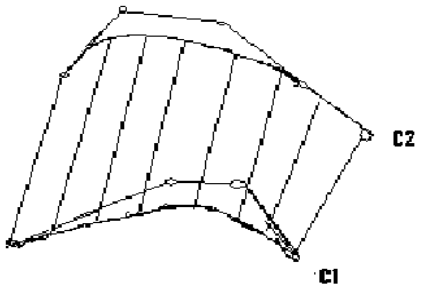

Figure 6a. C1 and C2 Two Curves Built One Ruled Surface (Farin 1993)

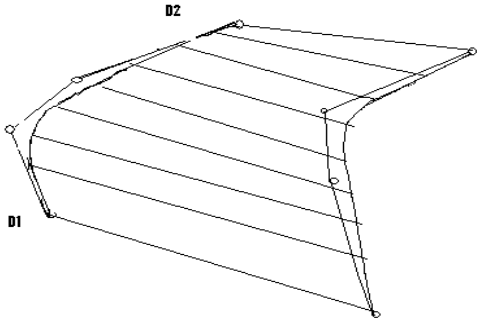

Figure 6b. D1 and D2 Two Curves Built One Ruled Surface (Farin 1993)

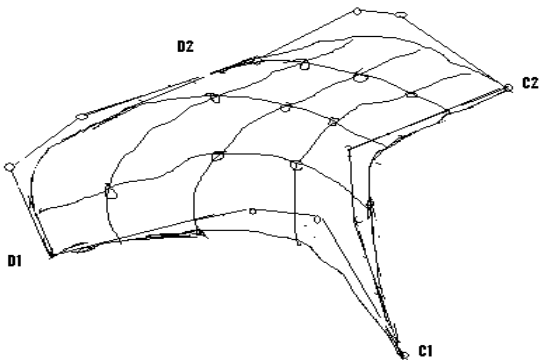

Figure 6c. Coons Patch Built from C1, C2, D1, and D2 Four Curves (Farin 1993)

The C1, C2, D1, and D2 four curves can be Bezier curves. If they have interpolants at four coners, this Coons patch is called bicubically blended Coons' patch (Farin 1993, 1999).

The Bicubically blended Coons' patch is a cubic Hermite interpolation that needs more input. Therefore, it is also called Bicubic Hermite Patch. The Bicubic Hermite Patch is described as

$$X(u,v)=\sum_{i=0}^{3}\sum_{j=0}^{3}h_{i,j}H_i^3(u)H_j^3(v); \qquad 0\le u,v\le 1,$$

where H_i^3 and H_j^3 are the cubic Hermite fuctions that are described in Bernstein form:

$$\begin{aligned}
H_0^3(t) &= B_0^3(t)+B_1^3(t),\\
H_1^3(t) &= \frac{1}{3}B_1^3(t),\\
H_2^3(t) &= -\frac{1}{3}B_2^3(t),\\
H_3^3(t) &= B_2^3(t)+B_3^3(t).
\end{aligned}$$

The $h_{i,j}$ is computed as

$$[h_{i,j}]=\begin{bmatrix}
X(0,0) & X_v(0,0) & X_v(0,1) & X(0,1)\\
X_u(0,0) & X_{uv}(0,0) & X_{uv}(0,1) & X_u(0,1)\\
X_u(1,0) & X_{uv}(1,0) & X_{uv}(1,1) & X_u(1,1)\\
X(1,0) & X_v(1,0) & X_v(1,1) & X(1,1)
\end{bmatrix}$$

However, the Hermite form is sensitive to the $u-$ and $v-$ parameter intervals. If they are not in [0,1], but $a \le u \le b$, $c \le v \le d$, then the above definition of the Bicubic Hermite Patche becomes

$$X(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} h_{i,j} H_i^3(s) H_j^3(t); \qquad 0 \le s,t \le 1,$$

where s and t are local coordinates of the intervals [a,b] and [c,d]. The $h_{i,j}$ is given as:

$$[h_{i,j}] = \begin{bmatrix} X(0,0) & \Delta_v X_v(0,0) & \Delta_v X_v(0,1) & X(0,1) \\ \Delta_u X_u(0,0) & \Delta_v \Delta_u X_{uv}(0,0) & \Delta_v \Delta_u X_{uv}(0,1) & \Delta_u X_u(0,1) \\ \Delta_u X_u(1,0) & \Delta_v \Delta_u X_{uv}(1,0) & \Delta_v \Delta_u X_{uv}(1,1) & \Delta_u X_u(1,1) \\ X(1,0) & \Delta_v X_v(1,0) & \Delta_v X_v(1,1) & X(1,1) \end{bmatrix}$$

where $\Delta_u = b-a$ and $\Delta_v = d-c$. Figure 7 shows the coefficients of the Hermite form (Farin 2002).

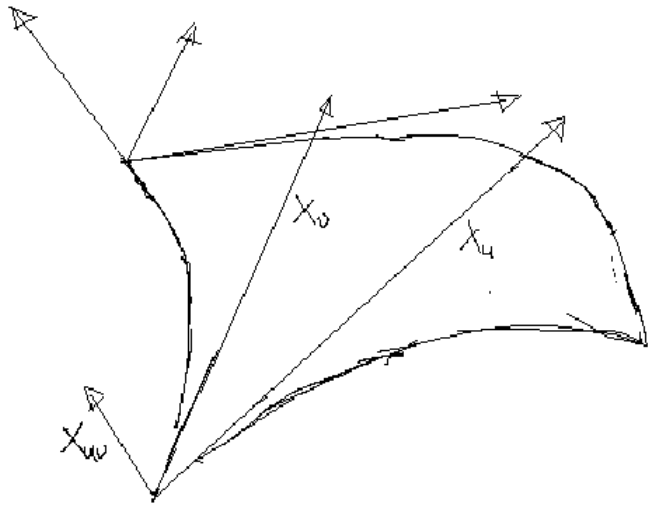

Figure 7. Bicubic Hermite Patches: points and vectors (Farin, 2002)

Different continuity orders between patches can be done with a higher degree of Coons patches. The Coons patch is actually a tensor-product patch. The advantage of a Coons' patch is that it is easy to match derivatives and twist vectors across the boundary with adjacent patches. It may meet in nonrectangular networks and is useful for combining irregular shapes. The Coons patch is easy to implement but not able to control internal shape (Faux 1979).

Box Splines

Box splines were introduced by de Boor and DeVore. They are multivariate splines derived as a generalization of univariate cardinal splines. A particular example of box splines is the B-splines with equidistant knots. In general, box splines consist of regularly arranged polynomial pieces. Box splines have a useful geometric interpretation. They can be viewed as density functions of the shadows of higher dimensional boxes and half-boxes. Of particular interest for CAGD are box spline surfaces that consist of triangular polynomial pieces. These box spline surfaces have planar domains, but it is quite simple to construct arbitrary two-dimensional surfaces—i.e., manifolds—with these box splines (Cohen 1984; de Boor et al. 1994).

The Box spline B(x) : R → R is defined as a "shadow " of a translucent box, B($\vec{x} = vol_n\{Q \cap P^{-1}\vec{x}\}$, where Q is a convex polyhedron in R^{n+2} , which is defined by the convex hull of vectors in U. P : R^{n+2} → R^2 is a operator for projection. When the convex hull of U is cubical, B(x), the n-dimentional volume of a cross section of Q is called a box spline (de Boor et al. 1994).

The box spline Bv is defined as:

$$(BV, \varphi) := \int_{[-1/2,1/2]^{\#V}} \varphi \quad (\sum_{v \in V} t_v v)dt \qquad for \quad \varphi \in C_0(IR^d)$$

where V is a collection of objects of vectors in IR^d with integer components. #V is the number of vectors in V. This definition can be written in the geometric way as:

$$BV(x) = vol_n\{t \in [-1/2,\ 1/2]^{\#V} : \sum_{v \in V} t_v v = x\},\ \text{and}\ (BV, \varphi) = \int BV(x)\varphi(x)dx$$ (Höllig 1986).

The box spline has its own properties: positive, supp $BV = \{\sum_V t_v v : -1/2 \le t_v \le 1/2\}$, a piecewise polynomial of degree $\le n := |V| - d$, and ϑ times continuously differentiable with $\vartheta := \min\{\#W : (V/WE) \ne IR^d\} - 2$. The derivative of a box spline is obtained by subtracting two lower degree box splines, and the convolution of two box splines produces a higher degree box spline. Averaging a box spline in a direction υ can obtain its identity box spline, and these two box splines form box splines. It can be denoted as

$$BV \cup v(x) = \int_{-1/2}^{1/2} BV(x+tv)dt$$

For example, the box spline $B_{\{\{1.0),(0.1)\}}$ is the characteristic function of the square $[-1/2,1/2]^2$. Applying the above identity function with $v = (1,1)$, three vectors (1,0), (0,1), and (1,1) are obtained and a linear box spline is produced. Then similarly averaging the linear box spline with $v = (-1,1)$ makes the quadratic box spline. The box spline $B_{\{\{1.0),(0.1)\}}$ and the corresponding mesh through computing is shown in Figure 8 (Höllig 1986).

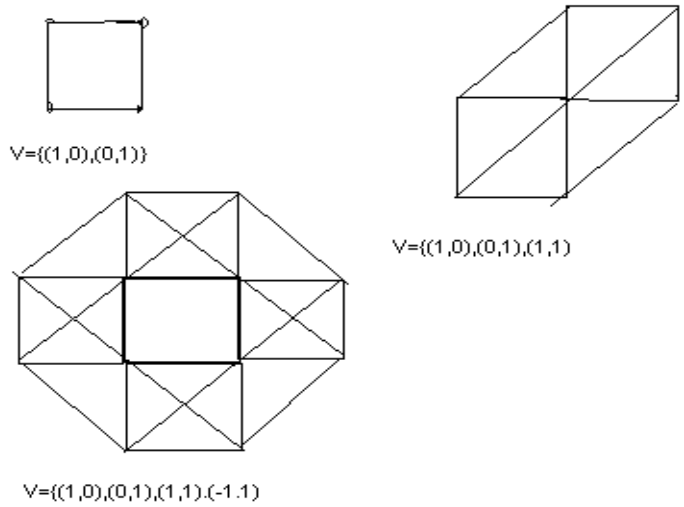

Figure 8. The Box Spline $B_{\{(1,0),(0,1)\}}$ (Höllig 1986)

OpenGL Review

History of OpenGL

OpenGL provides an environment for developing portable, interactive 2D and 3D graphics applications. OpenGL has become the most widely used and supported 2D and 3D graphics application programming interface (API) since it was introduced in 1992. OpenGL integrates a broad set of rendering, texture mapping, special effects, and other powerful visualization functions (Angel 2000).

Advantages of OpenGL

OpenGL API-based applications can be done on systems ranging from consumer electronics such as HBTV and WBTV to PCs, workstations, and supercomputers. OpenGL is an evolving, extensible, easy to use, and well-documented software application. OpenGL has become a widely accepted industry standard with broad industry support -- a vendor-neutral, multiplatform graphics standard. OpenGL is stable, reliable, and portable, regardless of operating system or window system. All OpenGL applications produce consistent visual display results on any OpenGL API-compliant machines (Woo 1999; Angel 2000).

Basic Libraries of OpenGL

OpenGL can render geometric objects ranging from a simple geometric point, line, or filled polygon to complex lighted and texture-mapped NURBS curved surfaces. It uses high-level languages such as C or C++ to do the applications. It contains two basic libraries, GL and GLU. Some basic geometric primitives such as points, lines, and polygons are implemented in GL. The OpenGL Utility Library (GLU) provides more complex features, such as quadric rectangular surfaces and NURBS curves and surfaces. OpenGL mainly uses cubic patches to implement curves and surfaces in the GLU library (Woo 1999; Angel 2000).

How OpenGL Create Curve and Surfaces

The algorithms used in GLU to create curves and surfaces are based on the Bezier basis. The mechanisms used in OpenGL to create Bézier curves and surfaces are evaluators. It is not necessary to use uniform spacing of control points because it is easy to use Bézier curves and surfaces to generate other types of polynomial curves and surfaces by making new control points. Using evaluators in OpenGL can generate one-, two-, three-, and four-dimensional curves and surfaces. OpenGL supports tensor product surfaces, making surfaces created in OpenGL rectangular based. The OpenGL evaluator functions also provide color, normal, and texture mapping (Woo 1999).

Conclusion

As noted earlier, rectangular surfaces are dominant in commercial CAGD applications. Thus, many graphic APIs including OpenGL emphasize modeling with rectangular patches. Some graphics rendering such as ray tracing are well developed for rectangular patches. However, not all shapes are rectangular; some are triangular or more complex, such as the wings of bird or a sharp corner. These complex shapes can be rendered using degenerated rectangular patches, but the outputs are not good. Also, degenerate rectangular patches cause problems in the other algorithms such as finding intersections or ray tracing. Extending OpenGL library is important to support other strategies for modeling, including some primitives like triangular patches, Coons patches, and box-splines (Farin 1986).

Compared with rectangular patches, Bezier triangular patches are higher CPU-intensive. Therefore, creating other surfaces may raise the cost of computing. However, new alternative primitives will reduce the time needed by the designer to design complex surfaces and improve output. Saving design time and getting better result versus requiring higher CPU-intensity is a significant trade-off (Farin 1986).

CHAPTER 3

TRIANGULAR PATCHES

There are certain situations in 3D modeling where rectangular Bezier patches are inappropriate. Triangular Bezier patches are defined in a way that is similar to rectangular patches, yet offer certain advantages over rectangular patches when applied correctly. This chapter describes the motivation behind triangular Bezier patches and the additions to the OpenGL library that provide an implementation of triangular patches that follows the existing rectangular patches' functionality and form. Four new functions for triangular patches are described.

Motivation

Many CAGD applications, such as automobile design, require the use of rectangular patches. For example, automobile parts (car doors, hoods, and body panels) are well suited to design with rectangular patches. The theory of, and software for, rectangular surfaces are well documented; this was described in Chapter 2. However, even designs that primarily use rectangular patches may still require triangular components. For example, if three rectangular patches come together at a corner, the remaining space is triangular. A triangular patch can be obtained from a degenerated cubic patch; however, for better output and to alleviate a designer's effort, a triangular patch is desirable (Farin 1986).

OpenGL is industry-standard software. It has only rectangular patches, both Bezier and NURBS, and the ability to render such surfaces. OpenGL lacks a triangular patch rendering functionality. This chapter discusses an addition to the OpenGL library for such a patch.

Barycentric Coordinates in Triangles

Three non-collinear points (a, b, and c) in the space define a non-degenerate triangular plane *T*. Any point *P* in the plane *T* can be expressed sum of these three points (a, b and c) with their weighted coefficients (*u*, *v*, and *w*), called the barycentric coordinates of *P* in plane *T*. The barycentric coordinates provide affine invariance for this triangular plane. The point P is expressed as below:

$P = u\mathrm{a} + v\mathrm{b} + w\mathrm{c}$, where and $u + v + w = 1$.

Note that the first equation is under-constrained; it has many solutions because only two points (actually, linearly independent vectors) are needed to define a point in a plane (here, the plane defined by the triangle (a, b, c)). The condition that the coefficients sum to one guarantees a unique solution. If, in addition, the following holds

$0 \le u \le 1,\ 0 \le v \le 1,\ 0 \le w \le 1,$

then P is within the triangle defined by (u, v, w).

Figure 9 shows the relationship between point *P* and three points (a, b and c) and their coefficients (*u*, *v*, and *w*), which can be expressed as below:

$$u = \frac{area(P,b,c)}{area(a,b,c)} \qquad v = \frac{area(a,P,c)}{area(a,b,c)} \qquad w = \frac{area(a,b,P)}{area(a,b,c)}$$

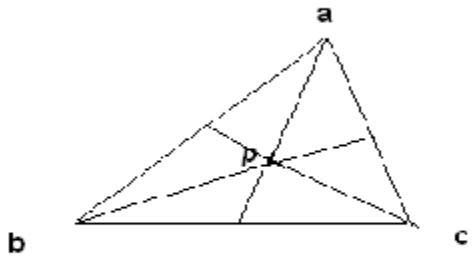

Figure 9. The Triangle and Plane *T* Defined by Points (a, b, and c).

From the above theory, for any point in the plane, the barycentric coordinates can be found by solving the equations for (*u*, *v*, and *w*). (Farin 1986).

Bernstein Polynomials and The de Casteljau Algorithm

Bernstein Polynomials

A triangular patch is defined in terms of Bernstein polynomials, using a form similar to that of rectangular patches. The general formula is

$$P = \sum_{|\mathrm{I}|=n} b_I B_I^n$$

where $B_I^n = \binom{n}{\mathrm{I}} u^i v^j w^k = \frac{n!}{i!\,j!\,k!} u^i v^j w^k ; |\,\mathrm{I}\,| = n = i + j + k$.

In Figure 3.2 below, the point *P* on a cubic triangular patch is obtained as

$$P(u,v,w) = P_{003}.w^3 + P_{012}.3vw^2 + P_{021}.3v^2w + P_{030}.v^3 + P_{102}.3uw^2 + P_{111}.6uvw + P_{120}.3uv^2 + P_{201}.3u^2w + P_{210}.3u^2v + P_{300}; u_3$$

(calculated from Farin 2002). Note that the total degree of triangular patches is n, unlike bivariate rectangular patches, whose total degree is 2n.

The de Casteljau Algorithm

The de Casteljau Algorithm for curves uses repeated linear interpolation to find a point on that curve, given a single parameter value. The de Casteljau Algorithm for triangular patches is similar to that for curves. Three parameters are used – those of the Barycentric coordinates of a point *(u, v, w)* inside the standard triangle (0,0), (1,0), (0,1) in parameter space. Since $u + v + w = 1$, and $0 \le u \le 1$, $0 \le v \le 1$, $0 \le w \le 1$, these coefficients are not independent each other, as discussed in the previous section (Farin 2002).

As with rectangular patches, the triangular patch has its own set of control points. The number of control points is related to the degree of the triangular patch. If the degree of a triangular patch is *n*, the number of the control points is *½(n+1)(n+2)*. Any control point can be defined as b*ijk*, where $|i| = i + j + k$, $i, j, k \ge 0$, $|i| = n$ and $i + j + k = n$. The abbreviations e1 = (1,0,0), e2 = (0,1,0), and e3 = (0,0,1) are used (Farin 2002).

The de Casteljiau Algorithm for triangular patches is defined below:

For a triangular patch of degree n with control points bi $\in E^3$ and |i| = n, use repeated interpolation:

$$b_i^r = ub_{i+e1}^{r-1} + vb_{i+e2}^{r-1} + wb_{i+e3}^{r-1},$$

where $r = 1, \ldots, n$, and |i| = n-r and $b_i^0 = b_i$. The point *P* will be b^n.

Figure 10 illustrates finding a point *P* on a cubic Bezier triangular patch using the de Casteljiau algorithm (Farin 2002).

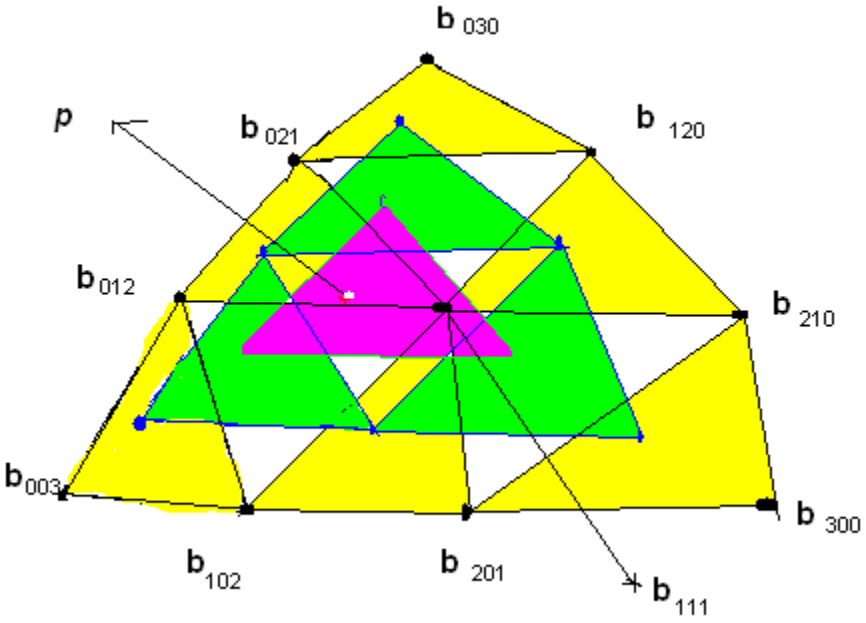

Figure 10. The Triangular de Casteljau Algorithm: a point *P* can be obtained by repeated linear interpolation (Farin 2002).

Implementation of Triangular Patch in OpenGL

Because OpenGL lacks any functions for implementing triangular patches, a set of functions is described here that allow an OpenGL application program to define and render triangular patches. The proposed functions follow the format of the existing Bezier rectangular patch functions in OpenGL.

Prototype of Evaluators of Triangular Patch in OpenGL

The first function for defining triangular patches, `myTriangleMap2f( )`, lets an application define an evaluator for a set of control points for a triangular patch. This function only needs to be called once, usually in an initialization function. These control points must be specified in a certain order, as described below.

```
myTriangleMap2f(GLenum target, TYPE u1, TYPE u2, GLint stride,
GLint order,  TYPE v1, TYPE v2, TYPE *points);
```

The Glenum *target* parameter tells what the control points represent from the choices vertices, RGBA color data, normal vectors, or texture coordinates. For example, the choice for vertices is the constant GL_MAP2_VERTEX_3. Parameters u_1 and u_2 indicate the range of the variable u . Parameters v_1 and v_2 indicate the range of the variable v . For u and v, $0 \le u + v \le 1$, which guarantees that the convex hull property holds. The formula for a triangular patch is $P(t) = \sum_{|\mathbf{I}|=n} p_i B_i^n(t)$, where, as discussed earlier, $t = u(1,0,0) + v(0,1,0) + w(0,0,1)$ and $u + v + w = 1$. The *stride* and *order* have the same meaning as in one-dimensional evaluators: the stride tells the distance between consecutive control points, and the order is the polynomial's total degree plus one. The number of control points is ½(degree+1)(degree+2); for example, for a cubic triangular patch, degree = 3, and 10 control points are required. There is only one polynomial $B_i^n(t)$ in trivariate form of total degree $n = i + j + k$, and $i, j, k \ge 0$, where $B_i^n(t) = \frac{n!}{i!\,j!\,k!} u^i v^j w^k$. This is the reason why there is only one *order*, unlike rectangular patches that have two orders, one for each bivariate polynomial The *order* is the degree plus one, and it should agree with the number of control points. The **points* parameter is pointer to the first coordinate of

the first control point. Here, it points to the one-dimensional array of the control points (instead of two-dimensional array of the control points in two-dimensional evaluators). Parameter **points* is an array of points input row-wise – that is, the application must be careful to order the control points as shown in Figure 11. This is the reason why there is only one parameter *stride*.

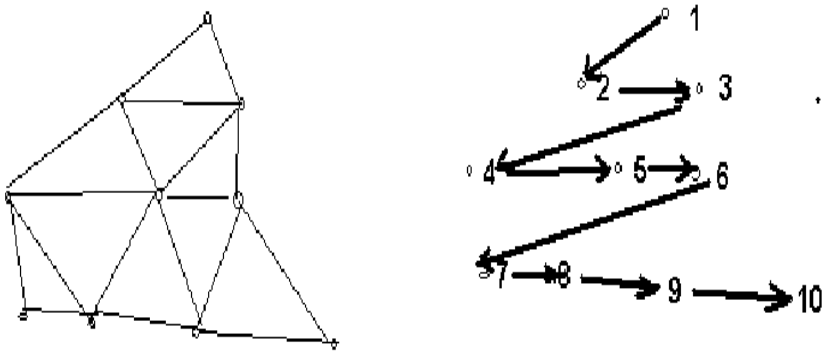

Figure 11. The Triangular Patch and Its Control Points' Order

The second function, `myTriangleEvalCoord2f( )`, evaluates the triangular patch previously defined by `myTriangleMap2f( )` and renders the patch.

```
myTriangleEvalCoord2f(TYPE u, TYPE v);
```

The variables u and v are the values (or a pointer to the **values*) of the domain, and $0 \leq u + v \leq 1$. The example code fragment in Figure 3.x shows the use of this function. Note that because each call to `myTriangleEvalCoord2f( )` resolves to a single point on the triangular patch, the application program must decide how to use these points. In the example, the points are used to draw line segment approximations to curves on the surface.

The next function, `myTriangleMapGrid1f( ),` renders the triangular patch as a set of (planar) triangles, using the equidistant steps in the three parameter directions.

```
MyTriangleMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv,  TYPE
v1, TYPE v2)
```

A triangle grid will be drawn from computing with two pairs of boundary curves u_1 to u_2 in *nu* steps and from v_1 to v_2 in *nv* steps, both with even spacing, $0 \le u, v \le 1$ and $0 \le u + v \le 1$.

The function `myTriangleEvalMesh2f( )` evaluates a previously defined triangular mesh.

`MyTriangleEvalMesh2(Glenum` *mode*`, Glint` i_1`, Glint` i_2`, Glint` j_1`, Glint` j_2`)`

The parameter *mode* can be GL_POINT, GL_FILL, or GL_LINE, and $0 \le i_1 \le i_2 \le nu$, $0 \le j_1 \le j_2 \le nv$. The mesh function applies the currently defined two-dimensional map grid.

Functions `myTriangleMapGrid2f()` and `myTriangleEvalMesh2()` are used together to efficiently generate and evaluate a series of evenly-spaced map domain values. They play the same role as `myTriangleEvalCoord2()` but take the place of the loops that generate the patch approximation. The triangular patch is formed by use of `myTriangleEvalMesh2()` to step through the integer domain of a triangle grid, whose range is the domain of the evaluation maps specified `glTriangleMap2()`.

Implementation of Evaluators of Triangular Patch in OpenGL

The algorithm used to implement an evaluator of a triangular patch is de Casteljau Algorithm. According to this algorithm, every point in this patch is calculated from the given control points, as described earlier. The normal of each point is calculated by is original three points, which decides a plate, and this point is in this plate, so the normal of plate is the normal of this point also.

A small extension library of OpenGL named myOpenGL (myOpenGL.h, myOpenGL.cpp) was created with the above functions' implementations. The code was implemented in standard C; it is shown in Appendix A. The picture of triangular patch rendered with above new functions is shown in Figure 12. Figure 12a is triangular patch in point mode, 12b is in frame mode, and 12c is in surface mode.

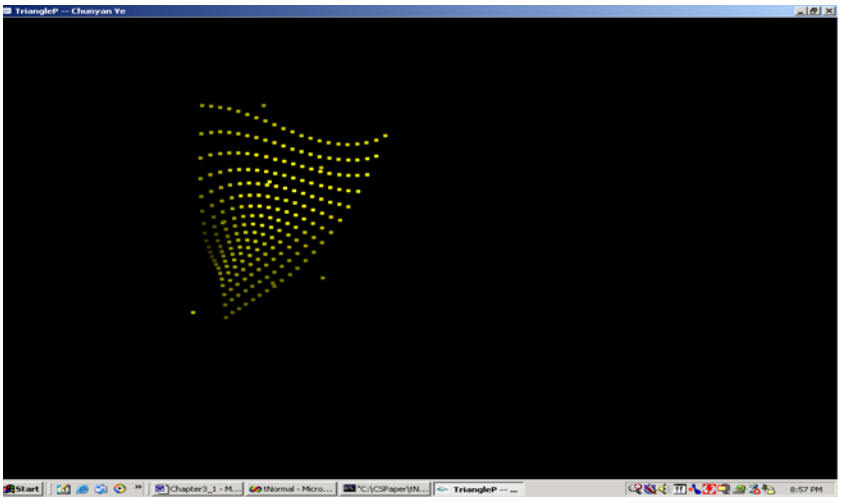

Figure 12a. Triangular Patch in Point Mode

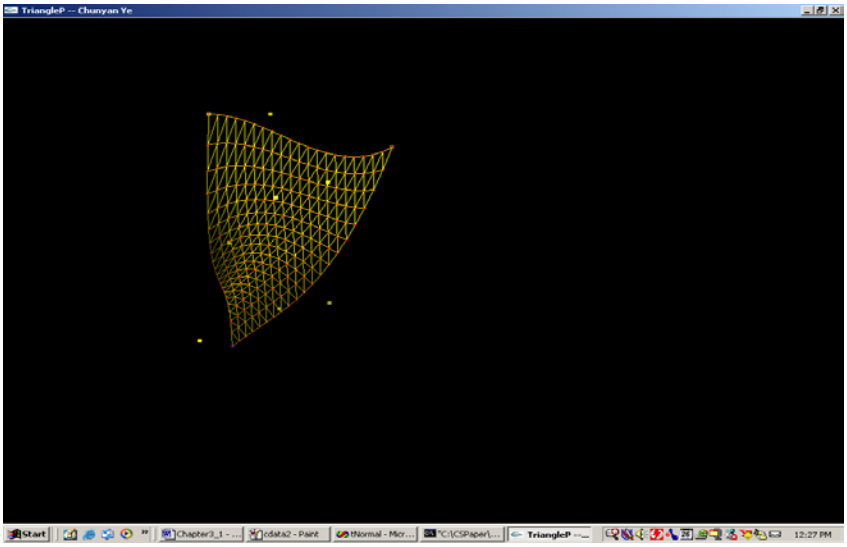

Figure 12b. Triangular Patch in Frame Mode

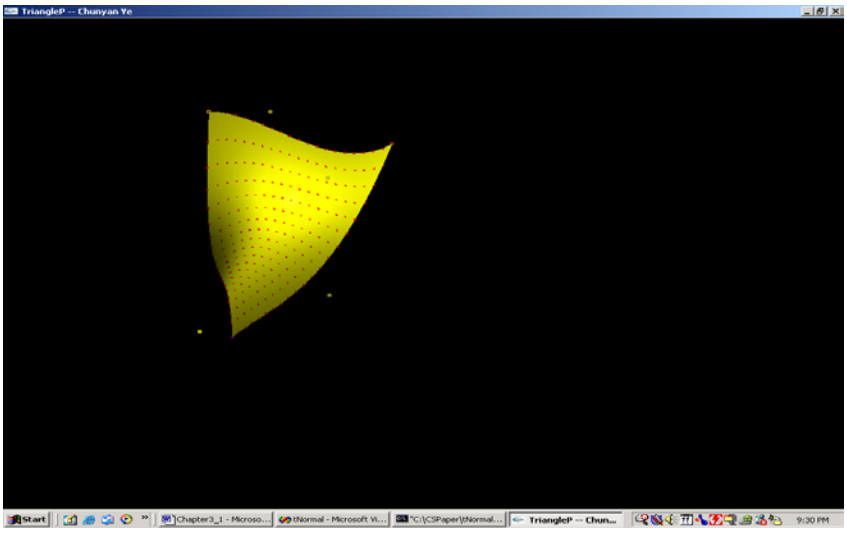

Figure 12c. Triangular Patch in Surface Mode

Comparisons of Triangular Patch and Degenerated Cubic Patch

To compare triangular patches with rectangular patches whose control points are set to make the patch degenerate into a three-sided patch, two sets of control points were created from the same data set. One is 16 control points and is used to define a degenerate triangular patch using a rectangular cubic Bezier patch. The other is 10 control points, used to define a triangular Bezier patch. Figure 13a shows degenerated rectangular patch control point data set. Note that point 0 is repeated four times across the bottom row, point 5 is repeated three times across the next row, and point 10 is repeated twice in the next row. This data set gives the normally rectangular patch a triangular shape. Figure 13b shows the triangular patch control point data set. The repeated points were taken from the Figure 13a data set to get 10 control points data set for triangular patch in Figure 13b.

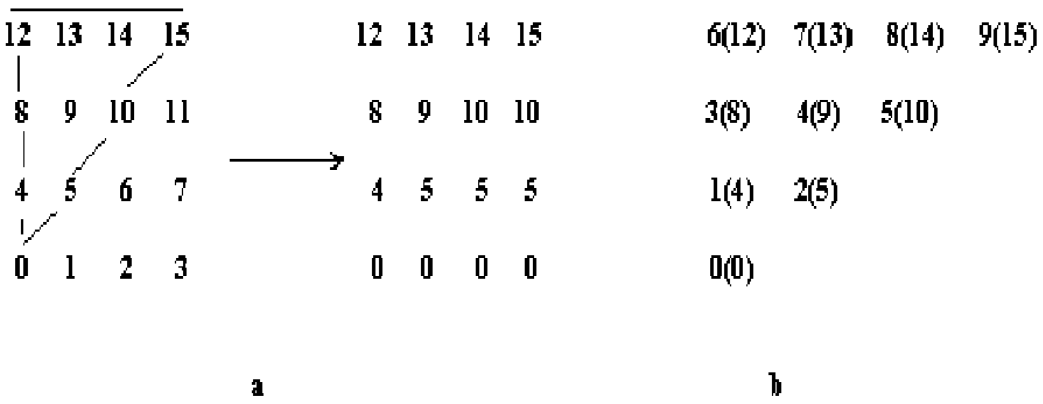

Figure 13. Degenerated Rectangular Patch Control Points (a) vs Triangular Patch Control Points (b)

The results were showed in Figure 14a, 14b, 15a, and 15b. In Figure 14a and 14b the right triangular patch is the degenerate rectangular patch and the left one is the triangular patch in surface mode and frame mode respectively. Figure 15a shows three rectangular patches that come together in a triangular corner; the corner is filled with a triangular patch. Figure 15b shows a degenerate rectangular patch covering the corner instead. Comparison of the two related patches shows that the degenerate rectangular patch has some shortcomings (Figure 16). First, the edges of the patch were not smooth; obvious jagged edges appeared. This will influence the smooth connection with other patches to form an ideal designed feature; it was showed in Figure 15a and 15b. Second, it is difficult to determine the control points. Continuity across patch boundaries must be calculated. Third, it takes more calculations to display a degenerate triangular patch, since there are more points, and thus more work in each deCastlejau step; this will incur more CPU time. Finally, storage for 16 points obviously takes more room than 10 points. If a scene requires thousands of patches, the degenerate rectangular patches will result in a large amount of extra storage space.

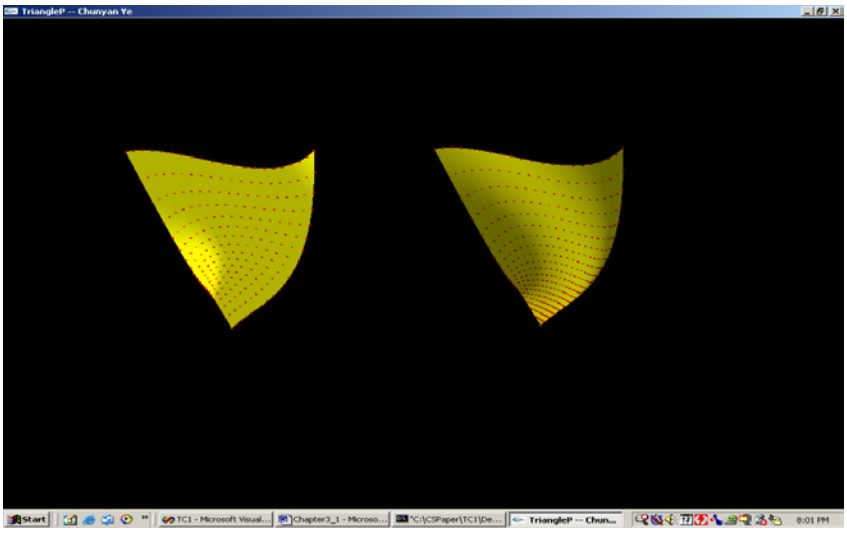

Figure 14a. The Right One is Degenerated Triangular, and the Left One is Regular Triangular Patch in Surface Mode

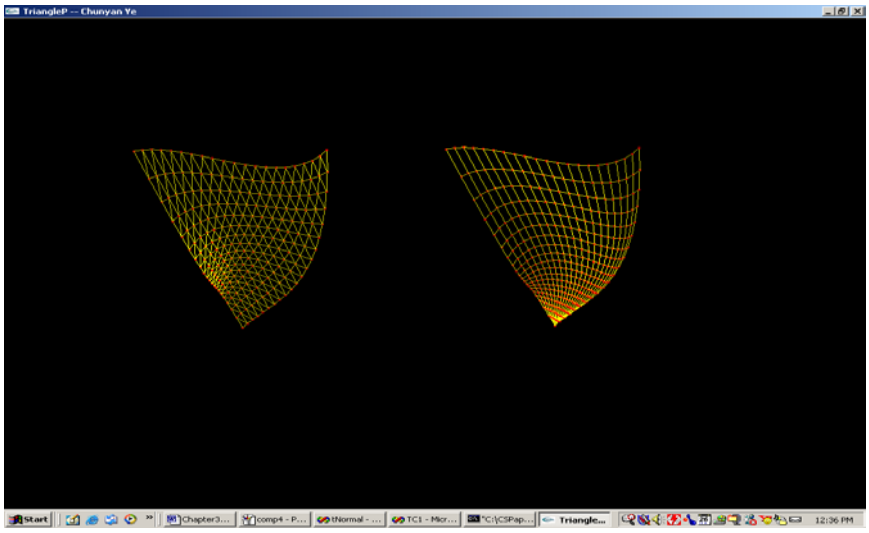

Figure 14b. The Right One is Degenerated Triangular, and the Left One is Regular Triangular Patch in Frame Mode

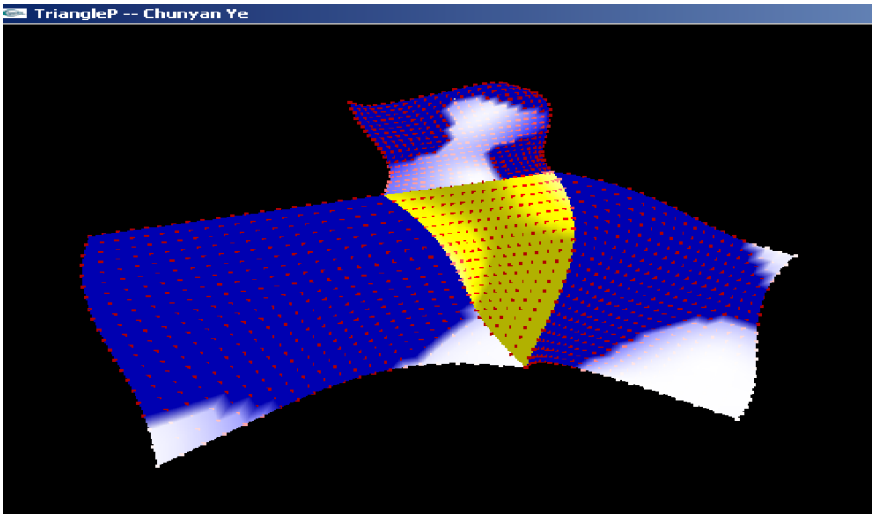

Figure 15a. New Designed Triangular Patch in Yellow.

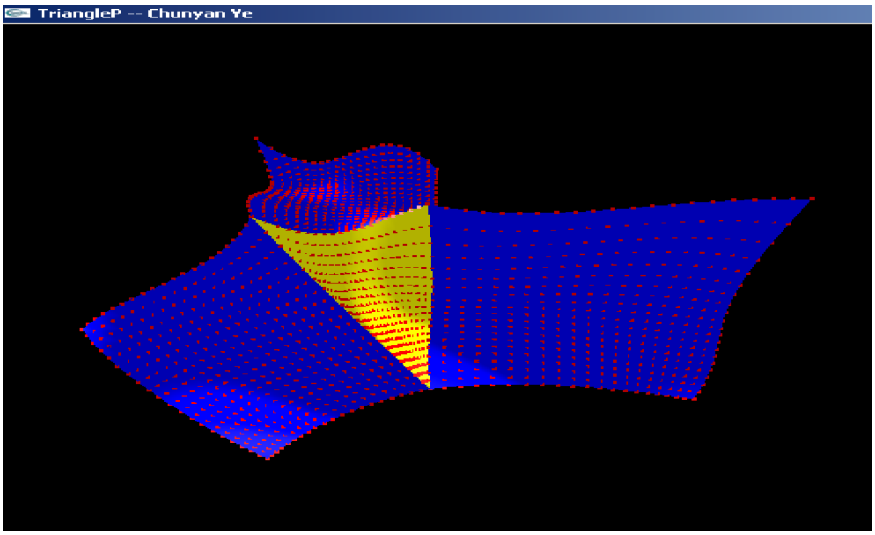

Figure 15b. Degenerated Triangular Patch in Yellow

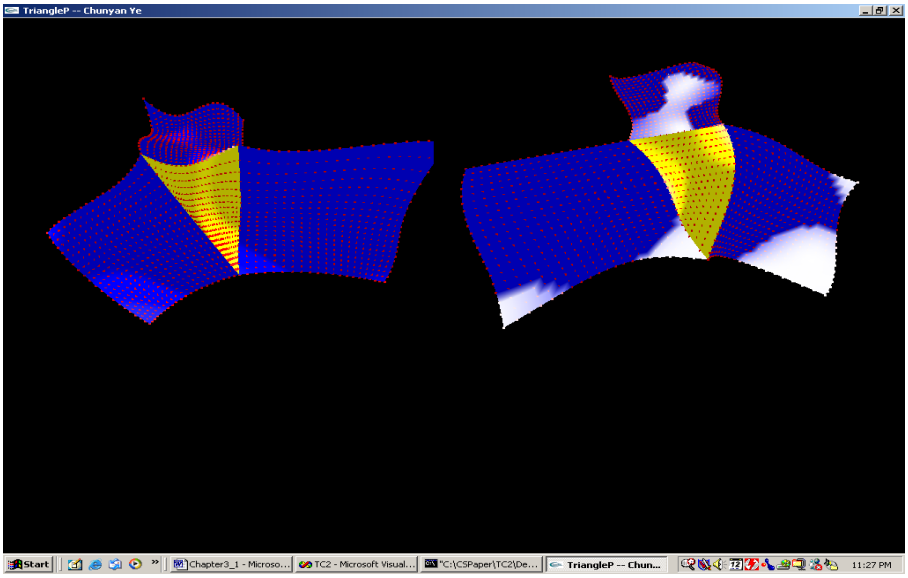

Figure 16. Comparison of Triangular Patch with Degenerated Triangular Patch in a Designed Feature.

Conclusion

Triangular Bezier patches can be useful to designers in certain situations. Some modeling requires that triangular holes be filled in between rectangular patches, for example. The triangular patch created from a degenerate rectangular Bezier patch and rendered with the existing functions provided by OpenGL was shown to have several deficiencies. Triangular Bezier patches, and the new functions developed here that are compatible with OpenGL, allow designers to achieve high-quality results in applications suited to their usage.

CHAPTER 4

COONS PATCHES

Rectangular patches are often used in design problems as solutions to interpolation problems. Usually, the rectangular patch interpolates the given data points at the four patch corners. In some situations, though, space curves, rather than individual points, must be interpolated. If the curves are laid out on a rectangular grid, then a Coons patch can be used for this purpose. A Coons patch's four edges are derived from the given curves at each boundary of the grid. This chapter describes the motivation to add Coons patches to the OpenGL library. Four new prototype functions for rendering Coons patches are described.

Motivation

In CAGD, data points typically need to be organized in some way before patches can be fit to the data. Data can be (or already may be, based on the type of modeling techniques used) organized as points over a two-dimensional grid, though the grid may not be regular. The rectangular shapes based on Bezier patches or B-spline patches are used to interpolate four data points in one grid area. Alternatively, the data points can be used to design control nets, where the points on a grid line are first fit with space curves. However, inside the control nets, data are missing, so a method to naturally "fill in" control nets is needed to get a "natural" feature patch. This leads to the creation of the Coons patch, which was invented by S. Coons (Farin 2002).

Bilinearly Blended Coons Patches

The bilinearly Coons' patch interpolates two boundary curves at a time. It is defined as follows (Farin 2002): given four curves C1(u), C2(u) on one pair of opposite sides of a grid segment and D1(v), D2(v) on the other pair, where $u, v \in [0,1]$, a surface X must be determined with these four curves as boundary curves:

$$X(u,0) = C1(u),\ \ X(u, 1) = C2(u),\ \ X(0,v) = D1(v),\ \ \ X(1, v) = D2(v).$$

A ruled surface (also known as a lofted surface) uses a linear connection between two boundary curves. Such a surface has the effect of connecting the curves with line segments between corresponding points (in parameter space) on each curve. A ruled surface, by definition, interpolates the boundary curves. For four curves in a grid area, two ruled surfaces can be derived from the above: surface Rc, the linear combination of curves C1 and C2, and surface Rd, the linear combination of curves D1 and D2:

$$Rc(u,v) = (1-v)X(u,0) + vX(u,1) \text{ and } Rd(u,v) = (1-u)X(0,v) + uX(1,v)$$

Adding the curves, Rc + Rd, does not give the desired patch because each boundary curve, correctly interpolated by one of the surfaces, is spoiled by the addition of the linear interpolant component of the other surface. To solve this problem, consider the bilinear interpolant Rcd to the four corners (see Figure 17):

$$Rcd(u,v) = [1-u \quad u]\begin{bmatrix} X(0,0) & X(0,1) \\ X(1,0) & X(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

Surface Rcd contains the linear component along each boundary, that is, it contains the linear combination of a pair of corner points. This is exactly the excess contained in the sum Rc + Rd, so subtracting out Rcd provides the desired answer.

The bilinear interpolant makes Rcd to the four corners, see Figure 4.1:

$$Rcd(u,v) = [1-u \quad u]\begin{bmatrix} X(0,0) & X(0,1) \\ X(1,0) & X(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

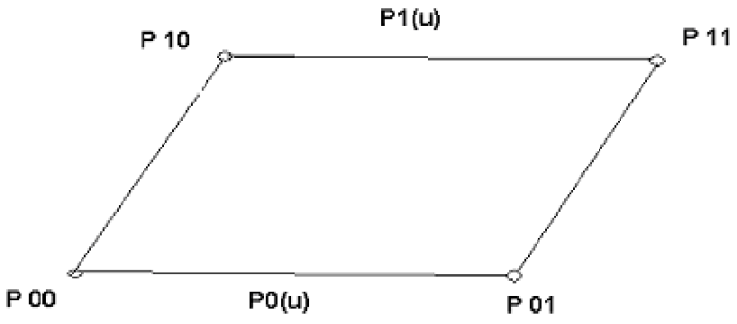

Figure 17. The Bilinear Interpolant for Rcd

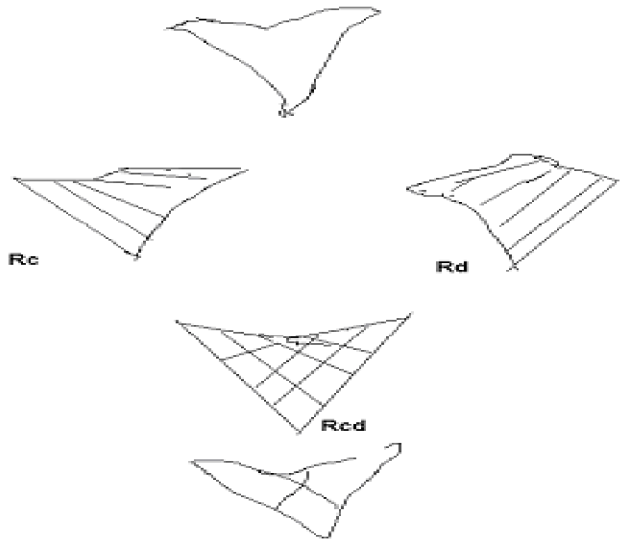

Figure 18. Coons Patches: A Bilinearly Blended Coons Patch (Farin 2002)

These two ruled surfaces built up a bilinearly Coons' patch governed by the bilinear interpolant, described as: X = Rc + Rd – Rcd (Figure 18), and

$$X(u,v) = [1-u \quad u]\begin{bmatrix} X(0,v) \\ X(1,v) \end{bmatrix} + [X(u,0) \quad X(u,1)]\begin{bmatrix} 1-v \\ v \end{bmatrix} - [1-u \quad u]\begin{bmatrix} X(0,0) & X(0,1) \\ X(1,0) & X(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

Above is the algorithm to calculate a coons patch from a set of given points.

Implementation of Coons Patch in OpenGL

Prototype of Evaluators of Coons Patch in OpenGL

An OpenGL function to implement a Coons patch, `myCoonsMap2f( )`, is given below. The boundary curves are defined as four Bezier curves, using a similar format as the Bezier patch functions in OpenGL.

```
myCoonsMap2f(GLenum target, TYPE u1, TYPE u2, GLint stride, GLint
             order1, GLint order2, TYPE v1, TYPE v2, GLint order3,
             GLint order4,TYPE *points);
```

The Glenum *target* parameter tells what the control points represent among the choices vertices, RGBA color data, normal vectors, or texture coordinates. Parameters u_1, u_2 indicate the range of the variable u . Parameters v_1, v_2 indicate the range of the variable v . For u and v, $0 \le u, v \le 1$, which guarantees that the convex hull property holds. The *order* parameters define the degree +1 of each curve but can be different for each boundary curve. The *order* must agree with the number of control points. The *stride* is the number of floating-point values to advance in the data between one control point and the next one, which is same for every boundary curve. The **points* parameter is pointer to the first coordinate of the first control point, a one-dimensional array of control points. The data input is described in Figure 19, which shows a 4x4x4x4 Coons patch data set. The set contains 12, not 16, data points, because the corner points are shared.

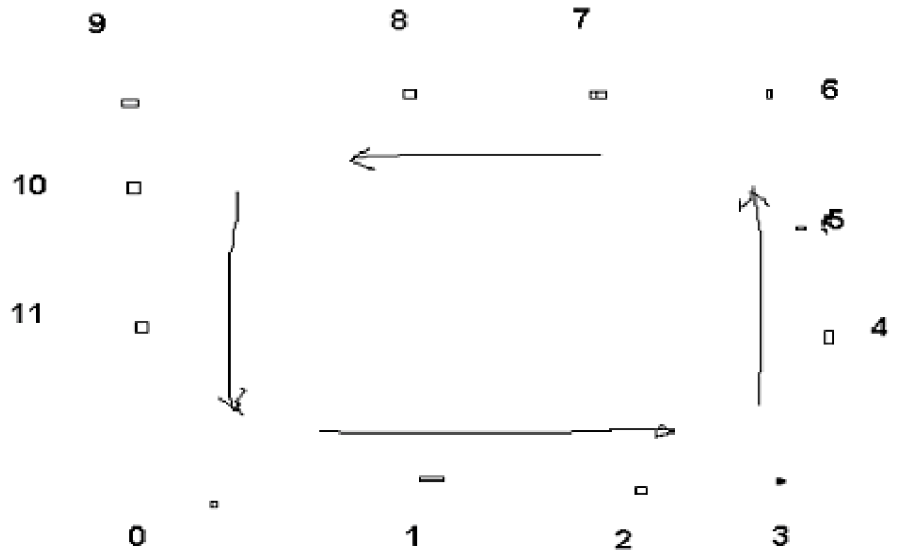

Figure 19. Data Input for 4x4x4x4 Coons Patch

```
myCoonsEvalCoord2f(TYPE u, TYPE v);
```

This function evaluates the previously-defined Coons patch at a domain point The variables u and v are the values (or a pointer to **values*) of the domain point, where $0 \le u, v \le 1$.

```
MyCoonsMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1,
                 TYPE v2)
```

This function computes a two-dimensional grid of surface points on the previously-defined Coons patch. The parameters u_1 and u_2 are the endpoints in *u* space, subdivided into *nu* steps; the parameters v_1 and v_2 are the endpoints in *v* space, subdivided into *nv* steps, both with even spacing, and $0 \le u, v \le 1$.

`MyCoonsEvalMesh2(Glenum` *mode*`, Glint` i_1`, Glint` i_2`, Glint` j_1`, Glint` j_2 `)`

This function draws a Coons patch in a form that depends on *mode*. The parameter *mode* can be GL_POINT, GL_FILL, or GL_LINE, and $0 \le i_1 \le i_2 \le nu$, $0 \le j_1 \le j_2 \le nv$. The mesh function applies the currently defined two-dimensional map grid.

Functions `MyCoonsMapGrid2` and `MyCoonsEvalMesh2` are used to efficiently generate and evaluate a series of evenly-spaced map domain values. They play the same role as a double loop using `MyCoonsEvalCoord2` to evaluate the patch and using drawing commands "by hand".

Implementation of Evaluators of Coons Patch in OpenGL

The algorithm used here is the bilinearly blended patch algorithm described above. There are 12 control points for this patch, and each adjacent edge shares a common point. The normal of the Coons patch will be obtained by calculating the normal of the triangle formed by the adjacent three points in this patch. For example, if values of parameters *u* and *v* are given, then *u + 0.1* and *v + 0.1* will be calculated to get the normal of point obtained from *u* and *v*. If *u* or *v* is 1.0, then *u-0.1* and *v-0.1* will be applied.

All of the above prototype functions' implementations have been added to a small extension library of OpenGL named myOpenGL, which is comprised of myOpenGL.h and myOpenGL.cpp.

Figure 20, 21, and 22 show examples of Coons patches designed with the above functions in myOpenGL library in surface, frame, and point mode.

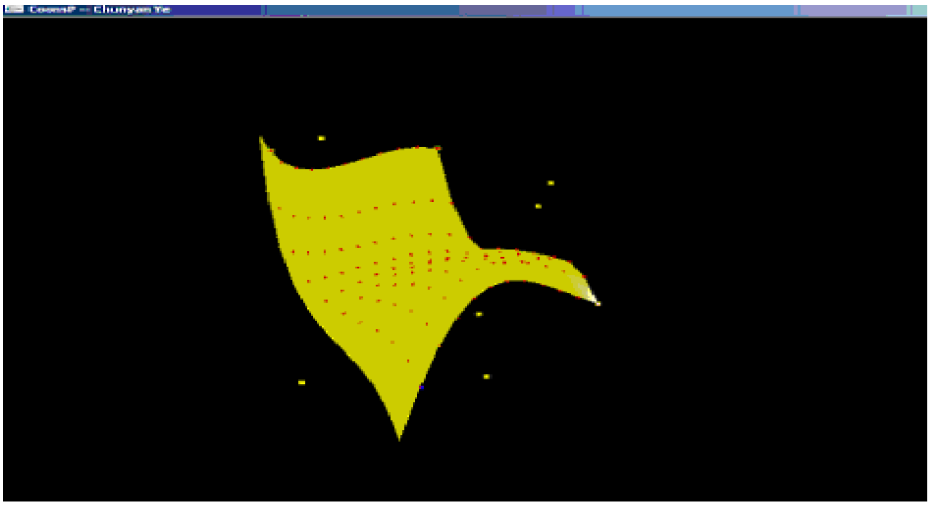

Figure 20. Coons Patch in Surface Mode.

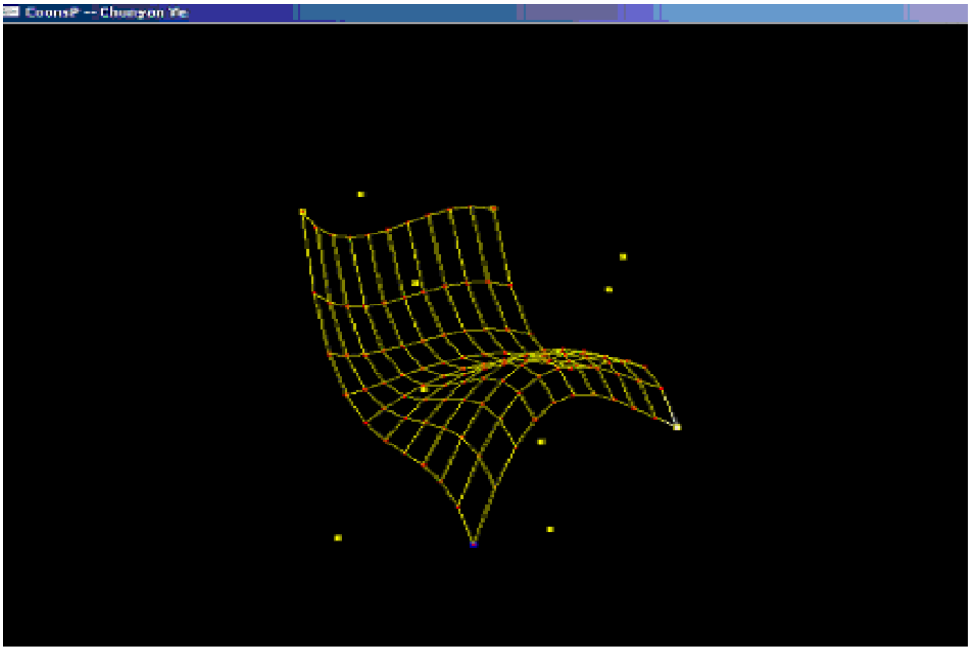

Figure 21. Coons Patch in Frame Mode.

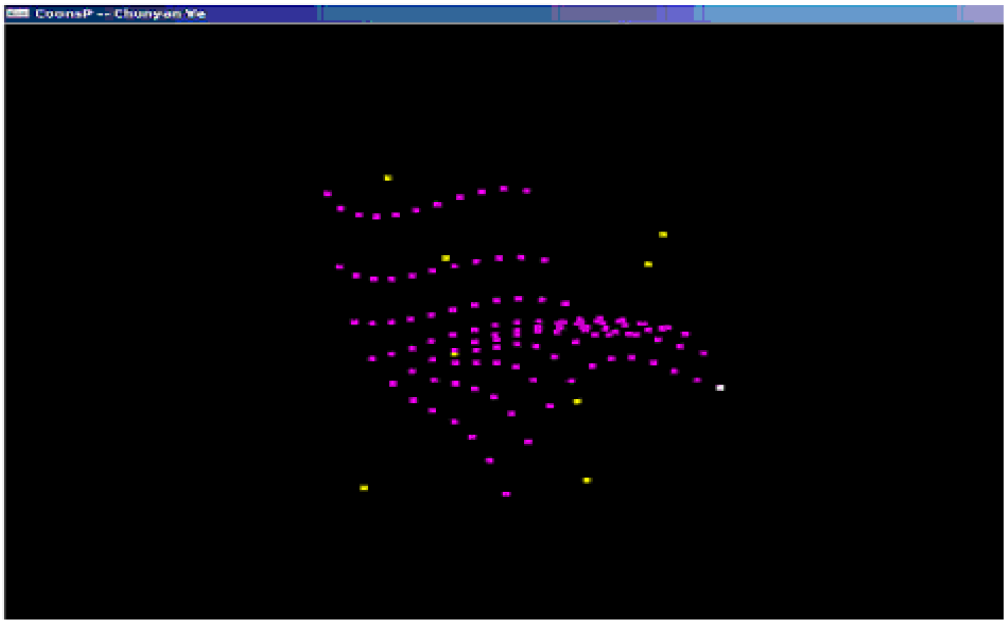

Figure 22. Coons Patch in Points Mode.

Comparisons of Coons Patch and Rectangular Patch with Same Control Points

The 12 control points for a Coons patch were obtained from the 16 rectangular patch control points set. Figure 23 shows the two data sets. In Figure 23a, a rectangular data set is shown using one ordering method, and in Figure 23b, the reordering of the data to fit the Coons patch data set is shown.

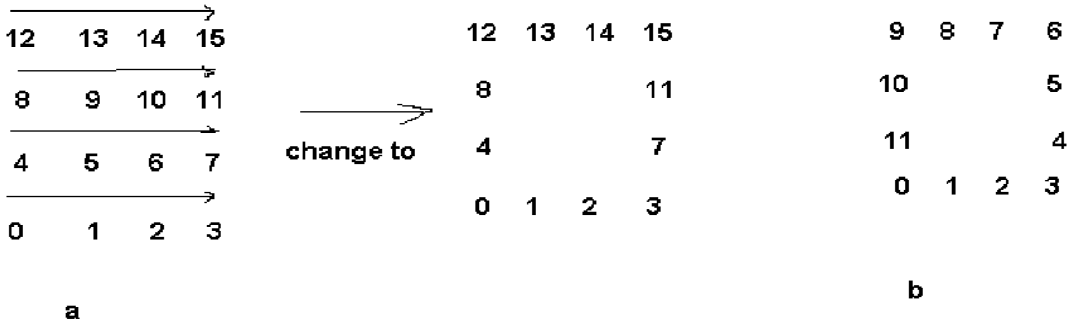

Figure 23. Data Sets for (a) a Rectangular Patch and (b) a Coons Patch.

The results were showed in Figure 24, Figure 25, and Figure 26.

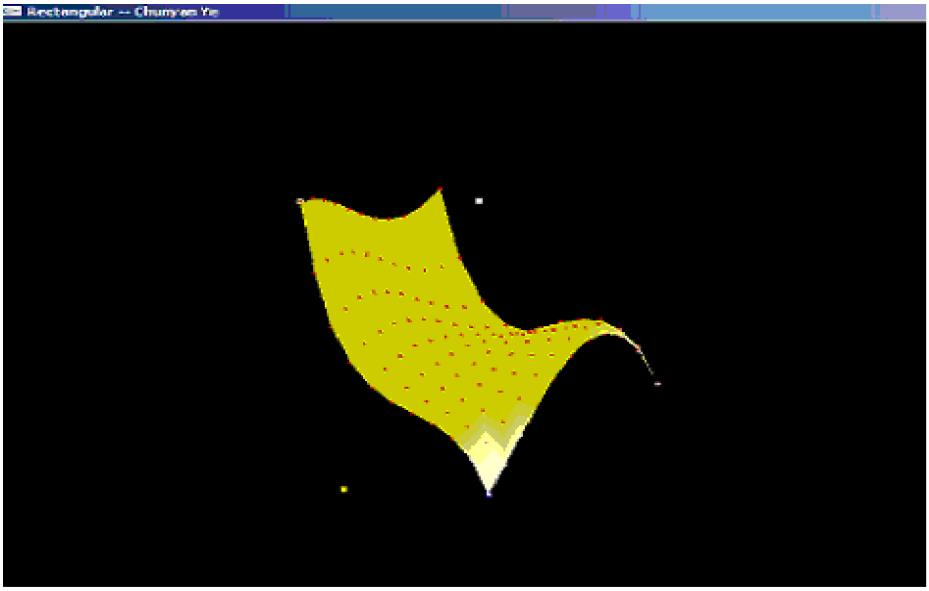

Figure 24. Rectangular Patch

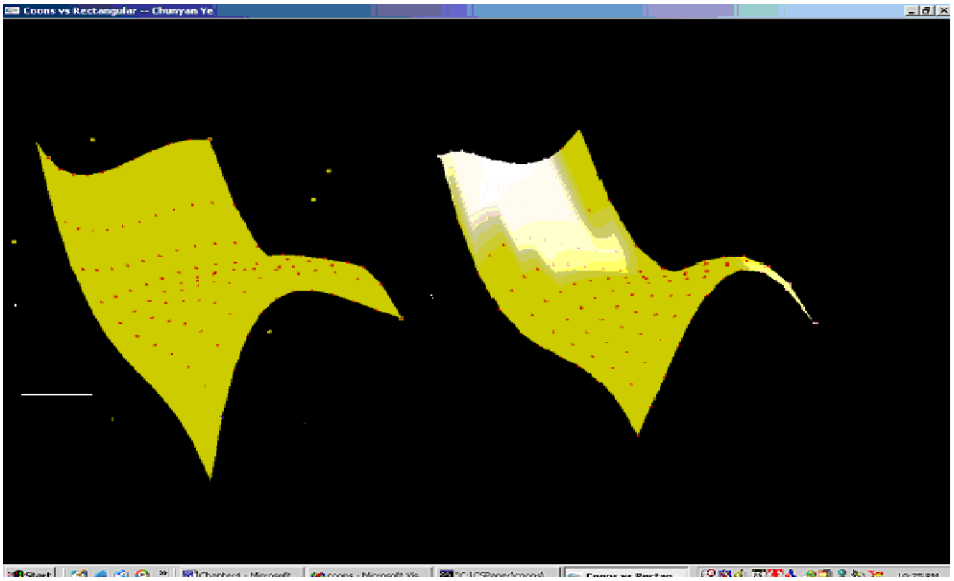

Figure 25. Comparison of Rectangular and Coons Patch in Surface Mode.

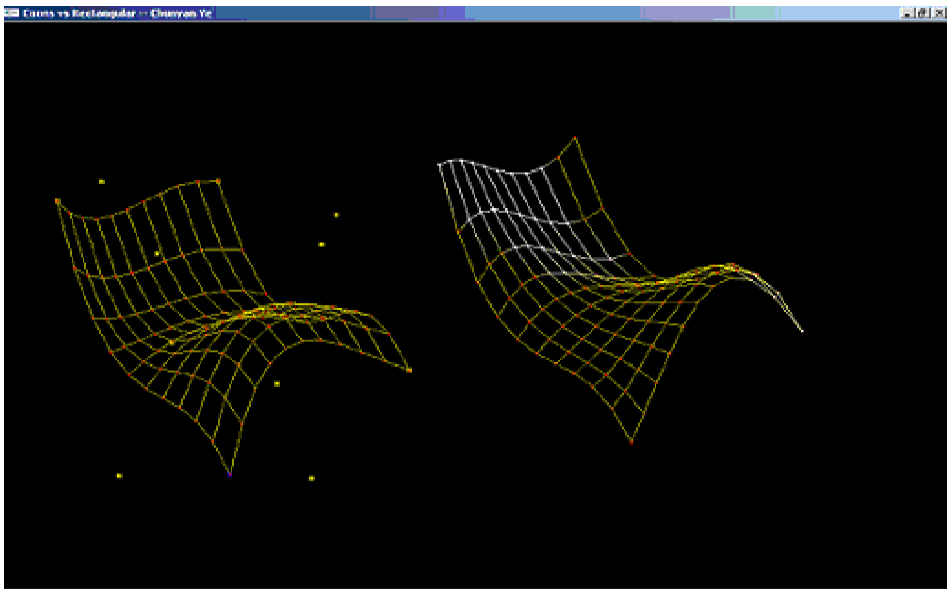

Figure 26. Comparison of Rectangular and Coons Patch in Frame Mode

The results show that the Coons patch handles the interior of the grid area differently from the rectangular patch. Recall that the Coons patch was obtained by interpolating the four edge curves. Both patches interpolate the four corner points. Both patches are similar in shape. There is no reasonable way to compare the patches, other than on general aesthetics, because they are derived by different algorithms and give different results. Problems associated with Coons patches and their variants (such as bicubically blended curves and the more general Gordon patches) are discussed in (Farin 2002).

Conclusion

Coons patches will be needed in some situations. In some real life modeling, four edges' data sets are found and used to design suitable patches, or sometimes the inside data is difficult to determine or approximate. The Coons patch will give a satisfactory answer for these situations.

The advantages of Coons patch are more freedom to design features, fewer points to store in memory, and potentially a different degree for each boundary edge. The addition of Coons patch functions to the OpenGL library allow for more flexibility in modeling and visualization.

CHAPTER 5

BOX SPLINES

Box splines are an extension of B-splines. This chapter describes the motivation to add box spline functions to the OpenGL library. Prototype of evaluators of box splines will be given. Implementation of these functions for rendering box splines surface is described.

Motivation

Box splines are multivariate splines introduced by de Boor and DeVore as a generalization of B-splines with equidistant knots. Box splines have desirable approximation properties, and they typically require a lower degree polynomial for results equivalent to higher degree tensor product splines (rectangular spline patches) for a given continuity. Thus, they can reduce the amount of CPU usage compared with B spline patches (Asahi et al. 2001). Also, a simple subdivision algorithm is available for displaying surfaces defined by box splines, as described below (Höllig 1986).

The addition of box splines as a new tool to OpenGL for rendering surfaces gives application programmers another tool for surface modeling.

Box Spline Definition

The basis function B_V is formally defined as an integral that describes the volume of a solid over some area. An informal definition is that B_V is the "shadow" of a translucent solid box, where the shadow is computed over a given area on a 2-dimensional plane using a defined direction for the light rays. Using a set of vectors V that contains, at minimum, the subset {(1,0), (0,1)} (the area over which the shadow is computed), the basis functions B_V are piecewise polynomials defined over the grid defined by V. This definition is similar to those used to derive univariate B splines and was, in fact, the inspiration for the multivariate box splines. The set V is typically larger than the minimum – it usually additionally contains (1,1) – and may contain multiple copies of vectors; that is, V is a multiset of integer-coordinate vectors.

For example, using the minimum V above, the B_V's are constant functions over the square defined by V – the shadow of a cube over a square. Adding (1,1) to V generates piecewise linear "hat" functions (i.e., six-sided pyramids) over the hexagon defined by V, since the shadow must be stretched over the hexagonal target defined by V. Adding one more vector, (-1,1) generates piecewise quadratic functions over an octagon, and so on.

Translates of the B_V's are used to define surfaces, using a set of control points $a_j \in \mathrm{R}^3$, that, when connected, form a control polygon for the surface p:

$$x \to p(x) = \sum_j a_j B_V(x - j), \qquad x \in \mathrm{R}^2,$$

where the j's form a grid in 2-D space, thus evaluating translations of the basis functions B_V. Recall that the set V contains at least the vectors (1,0) and (0,1); we will assume it also contains (1,1), so that the basis functions are at least piecewise linear.

Using the subdivision algorithm described below, a continuous surface can be approximated by the above discrete version – the equivalent of the B spline control polygon – using some "reasonable" number of steps, i.e., until the resulting object meets some visual criterion of goodness.

Algorithm

A subdivision algorithm, described in (Höllig 1986), was implemented to render box spline surfaces. This algorithm, which is similar to subdivision algorithms used to render one-dimensional B spline curves, repeatedly refines the control polygon until a fine-enough mesh is created.

At each stage, a refined polygon (a piecewise linear surface) {p'} with vertices a_j is computed according to the following two steps. Step 1 computes the averages of neighboring control points using neighbors in each axis-aligned direction and the upper-right diagonal direction, and also copying the original points. Step 2 uses the vector set U to create, by looping over the members of U to define the direction in which to average, a new set of control points. The initial set

(1). $b_{U,2j+v} := (a_j + a_{j+v})/2, \qquad v = (0,0),(1,0),(0,1),(1,1);$

This is done for each control point. Note: using v = (0,0) simply means that the original a's are copied into b's.

(2). Set W = U := {(1,0), (1,1), (0,1)}; V is the vector multiset that contains U as a subset.

for each $v \in$ V\U do: $b_{W \cup v, j} := (b_{W,j} + b_{W, j+v})/2;$

W := W $\cup$ v; stop when W = = V

$a'_{,} := b_{U,j.}$.

The vectors $b_{U,2j+(v,\mu)} \neq (0, 0)$, defined in step (1) of the algorithm, are the midpoints of the edges of the triangles that form the surface {p}; see Figure 27. Thus, the number of control points increases from n^2 to $(n+2)^2$, where n is the number of points on one side of the mesh. Each substep of step 2 typically decreases the number of control points from n2 to (n-1)2. This is because control points on two of the edges of the mesh do not have neighbors in the indicated direction. For example, for v = (-1,1), which indicates "left one, up one" as the nearest neighbor, the leftmost column of the mesh (no points to its left) and the topmost row of the mesh (no points above it) have no neighbors – although these are used as neighbors for the next-to-leftmost column and next-to-topmost row – decreasing the size of the new mesh by one column and one row. For this reason, the larger the set V\U, fewer control points are left to use for triangulation, although, because of convergence of the mesh, the closer the points are to the actual surface. It is up to the modeler to define a large enough grid of control points, which can be easily done by adding extra, redundant points at the edges or by using a grid that wraps back on itself (useful for closed surfaces, for example).

The combination of the triangles with vertices a_j, $a_{j+(1,0)}$, $a_{j+(1,1)}$ and the triangles with vertices a_j, $a_{j+(0,1)}$, $a_{j+(1,1)}$ forms the surface {p}. Repeating this algorithm yields a sequence of piecewise linear surfaces {p}, {p'}, {p"}, … that converge to the box spline surface {s} (Höllig 1986).

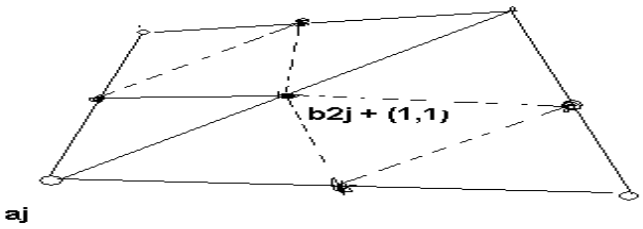

Figure 27. The Subdivision Algorithm for Box Splines (Höllig 1986)

Implementation of Box Splines in OpenGL

Two functions are described below that add box spline functionality to OpenGL. They allow an application program to render a box spline surface with a set of control points.

Prototype of Evaluators of Box Splines in OpenGL

The first function for box spline surface rendering is `myBoxMap2f( )`, which initializes the box spline data for a surface. This function will be called only once, at the start of an application.

```
myBoxMap2f(GLenum target, GLint stride, GLint order1, GLint
order2, TYPE *points);
```

The Glenum *target* parameter defines the usage for the control points, which is the same meaning as in the other surface evaluator in OpenGL. Choices are vertices, RGBA color data, normal vectors, or texture coordinates. The *stride* parameter represents the distance between consecutive control points. The parameters *order1* and *order2* stand for the two sides of arbitrary size of input control points, and is each side degree plus 1. *Order1* follows the x axis and *order2* follows the y axis, see Figure 28. The **points* parameter is the pointer that points to the first element of the control points. Here, it points the two-dimensional array of the control points.

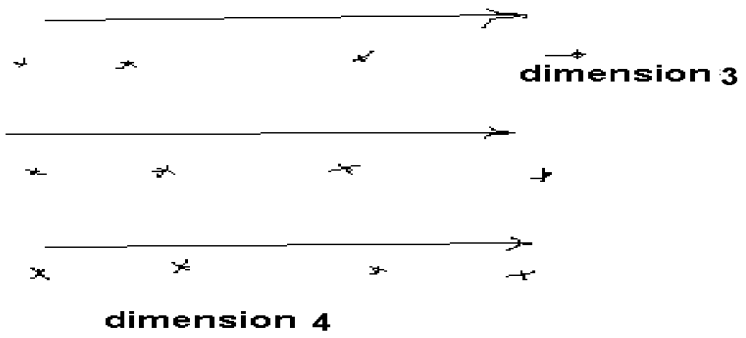

Figure 28. Control Points Input with Dimension 4 and Dimension 3

The second function `myBoxEvalMesh2( )`, evaluates the Box splines surface according to the above function `myBoxMap2f( )`.

```
myBoxEvalMesh2(GLenum mode, GLint *v, GLint vCount);
```

The Glemum mode represents GL_POINT, GL_FILL, or GL_LINE for Box splines surface. The parameter *v is int vector, which represents (1, 0), (0,1), (1,1) and (1,-1). The vCount parameter the subdivision times applied to get this Box splines surface.

Implementation of Evaluators of Box Splines in OpenGL

The algorithm used to implement evaluators of Box splines surface is recursion algorithm. According to this algorithm, subdivision was applied to obtain the Box splines surface with a set of given control points.

These two functions were added to the small extension library of OpenGL created before. The name of the library is myOpenGL (myOpenGL.h, myOpenGL.cpp).

The results were shown in Figure 28, 29, and 30. The coding part is in standard C; it is shown in Appendix C.

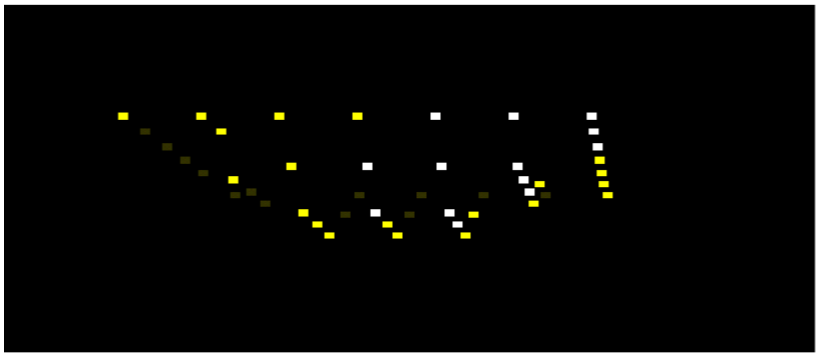

Figure 29. Box spline surface in point mode

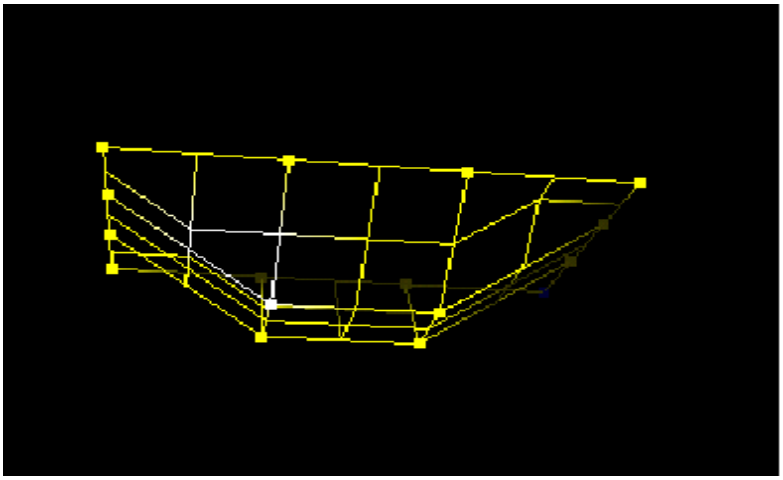

Figure 30. Box spline surface in frame mode

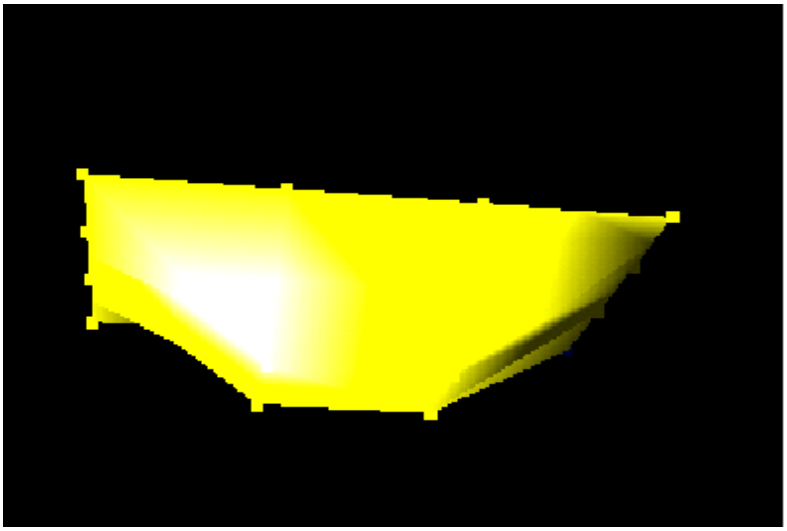

Figure 31. Box spline surface in surface mode

Conclusion

Box spline surfaces use fewer points and less storage to create a rich patch, which is more flexible. The Box splines surface rendering will bring OpenGL another powerful design tool in CAGD. The problem here is the ray tracing is not as good as the other modeling.

CHAPTER 6

CONCLUSION

This paper described three features added to the OpenGL library: triangular patches, Coons patches, and box spline surfaces. These additions make OpenGL more attractive, competitive, and practical in CAGD applications. This chapter describes the contributions of this study and potential future work.

Summary of Work

Three new types of surface types were added in a library auxiliary to OpenGl. These were triangular patches, Coons patches, and box spline surfaces. Functions were developed to support computing and rendering these surface types, implemented in C++ and tested with application programs using the industry standard OpenGL graphics library.

Each new surface type was evaluated for strengths and weaknesses compared to the standard rectangular patches that are already part of OpenGL.

Conclusions

The theory of triangular patches has been successfully applied as a development tool for general surface rendering with OpenGL applications. Compared with degenerate rectangular patches, using the rectangular patches provided by OpenGL, the new triangular patch functionality achieves better output—more smooth edges, potentially easier to design in continuity with neighboring patches, fewer points to compute with and thus reduced CPU time, and less storage space needed to keep the data. The addition of triangular patches makes it easier for designers to fit a triangular feature.

Coons patch rendering allows a surface designer to fit a surface between existing curves. Boundary edge curves, possibly with different degrees, forming a four-sided grid space, are used to define a linearly-blended surface to fill in that grid space. This situation can occur, for example, when a model is designed using surface curves – such as curves on a clay model. Knowing what interior control points to use for a fill-in patch can be difficult to asses; Coons patches simplify this process and yield a reasonable

fit. A Coons patch needs less storage space for data because it has fewer control points compared with a rectangular patch.

Box splines are an extension of B splines. The theory of box splines is more complicated than, for example, for the tensor product Bezier patches which define OpenGL's rectangular patches. Rendering of box splines was done using a subdivision algorithm, simplifying the control point mesh, rather than generating explicit points on the box spline surface. This reduces CPU time yet yields good results. Box splines can create complicated surfaces with fewer control points and, again, reduced storage space.

Future work

The continuity of triangular and Coons patches across patch boundaries was not considered in this thesis. OpenGL does not address this issue, either. However, in the modeling of complicated surfaces, continuity is a vital aspect that should be addressed. OpenGL's avoidance of continuity is based on its intended usage – as a rendering library, not as a modeling library. Still, building in automatic generation of cross-boundary control points for assured continuity would be a useful feature to add, both for OpenGL's built-in patch types and for the ones discussed here.

The application of texture to the new patches should be studied. Texture mapping is a built-in feature of OpenGL. Because the triangular and Coons patch implementations discussed here generate surface normals and compute points on the patch surfaces, applying textures to these surfaces should be trivial. However, this was not tested. Adding texture mapping to box spline surfaces, as implemented by the subdivision algorithm discussed in Chapter 5, would be more problematic. No points are generated on the surface itself because the control point mesh is used to approximate the surface. Additionally, texture mapping onto rectangular, triangular, or Coons patches is done over a relatively small area – one patch – then repeated across patches. For a box spline surface, the whole surface would be used, thus negating some of the usual effects possible with texture maps, such as checkerboarding or wood grains.

Only cubic triangular patches were tested. The functions for triangular patches, however, do allow for other degree polynomials. Cubics were used because they are widely used in the modeling world.

Only bilinearly blended Coons patches were implemented. Other kinds of Coons patches, such as bicubically blended, would be a useful addition to OpenGL.

Box spline theory is, for the non-mathematician, somewhat opaque. While modeling with box splines is relatively straightforward, there are currently few, if any, tools for modeling with them – unlike the situation for triangular and Coons patches, for which tools do exist. A modeling tool that supported box splines would provide a test bed for usage of this surface type.

BIBLIOGRAPHY

BIBLIOGRAPHY

Angel, Edward. *Interactive Computer Graphics, A Top-Down Approach with OpenGL.* 2nd edition. Addison-Wesley Press, 2000.

Asahi, Takeshi, Koichi Ichige, and Rokuya Ishii. "A New Formulation for Discrete Box Splines Reducing Computational Cost and Its Evaluation." *IEICE Trans. Fundamentals* E84-A (March 2001): 884-892.

Cohen, Elaine. "Discrete box splines and refinement algorithms." *Computer Aided Geometric Design* 1 (November 1984): 131-148

de Boor. C, K. Hollig, and A. Riemmenschneider. *Box Splines.* New York: Springer-Verlag, 1994.

Farin, Gerald. *Curves and Surfaces for Computer Aided Geometric Design, A Practical Guide.* 3rd edition. London: Academic Press, 1993.

Farin, Gerald. *Curves and Surfaces for Computer Aided Geometric Design, A Practical Guide.* 5th edition. San Francisco : Mogan Kaufmann Publishers, 2002.

Farin, Gerald. "Shape." In *Mathematics Unlimited-2001 and Beyond*, ed. B.Engquist and W.Schmid, 463-466. Springer-Verlag., 2001.

Farin, Gerals. "Triangular Bernstern-Bezier Patches." *Computer Aided Geometric Design* 3 (August 1986): 83-127.

Farin, G and Dianne Hansford. "Discrete Coons Patches." *Computer Aided Geometric Design* 16 (August 1999): 691-700.

Faux, I.D and M.J.Pratt. *Computational Geometry for Design and Manufacture.* John Willey & Sons Press, 1979.

Höllig, Klaus. Box Splines. Computer Science Technical Report #640. Computer Science Department University of Wisconsin-Madison. 1986.

Lai, Ming-Jun. "Geometric interpretation of smoothness conditions of triangular polynomial patches." *Computer Aided Geometric Design* 14 (February 1997): 191-199.

Watt, Alan. *3D Computer Graphics.* 3rd edition. Addison-Wesley Press, 2000.

Woo, Mason Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide. The Official Guide to learning OpenGL, Version 1.2.* 3rd edition. Addison-Wesley Press, 1999.

APPENDICES

APPENDIX A

MYOPENGL.H

```
/*****************************myOpenGL.h ************************************

/* Purpose : This is a small library to extend OpenGL for Computer Aided
/*                      Geometric Design (CAGD). This is a research activity, and
/*                      OpenGL has no knowlege about it.
/*Copy Right: All the code contained in this library is protected by copy
/*           right. Permission to use, copy, modify, and distribution for
/*           free. This software has no implied warranty, and no respossible
/*           for any misuseof it, or any damage arising out of its use. The
/*           entire risk of using this software lies with the party doing so.
/*Author:    Chunyan Ye
/*Date  :    03/31/2003
/*****************************************************************************/

#include <GL/glut.h>
#include <stdlib.h>

#ifndef __myopengl_h_
#ifndef __myOpenGL_h_
#ifndef __MYOPENGL_H__

typedef GLfloat TYPE;

/* data structure for triagnular patch */
typedef struct{
       GLenum m_target;
       TYPE   m_u1;
       TYPE   m_u2;
       GLint  m_stride;
       GLint  m_order;
       GLint  m_nu;
       GLint  m_nv;
       TYPE   m_v1;
       TYPE   m_v2;} triangleData;

/* data structure for coons patch */
typedef struct{
       GLenum m_target;
       TYPE   m_u1;
       TYPE   m_u2;
       GLint  m_stride;
       GLint  m_order1;
       GLint  m_order2;
       GLint  m_nu;
       GLint  m_nv;
       TYPE   m_v1;
       TYPE   m_v2;
       GLint  m_order3;
       GLint  m_order4;} coonData;

/* data structure for box spline patch */
typedef struct{
       GLenum m_target;
       GLint  m_stride;
       GLint  m_order1;
       GLint  m_order2;
       GLenum m_mode;} boxData;

/* functions for triangular patch */
extern void myTriangleMap2f(GLenum target, TYPE u1, TYPE u2,
                                    GLint stride, GLint order, TYPE v1, TYPE v2, TYPE
*points);
```

```
extern void myTriangleEvalCoord2f(TYPE u, TYPE v);

extern void myTriangleMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2);

extern void myTriangleEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);

/* functions for coons patch */
extern void myCoonsMap2f(GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order1,
                                        GLint order2, TYPE v1, TYPE v2, GLint
order3, GLint order4, TYPE *points);
extern void myCoonsEvalCoord2f(TYPE u, TYPE v);
extern void myCoonsMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2);
extern void myCoonsEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);

/* functions for box splines patch */
extern void myBoxMap2f(GLenum target, GLint stride, GLint order1, GLint order2, TYPE
*points);

extern void myBoxEvalMesh2(GLenum mode, GLint *v, GLint vCount);


/* other functions */

extern void normal(TYPE *p);

extern void crossproduct(TYPE *p1, TYPE *p2, TYPE *n);

extern void myEnable(GLenum nor);

extern void getPoint(GLint aOrder, GLint aStride, TYPE *ps, TYPE m);


#endif
#endif
#endif
```

APPENDIX B

MYOPENGL.CPP

```
/*****************************myOpenGL.cpp ************************************

/* Purpose : This is a small library to extend OpenGL for Computer Aided
/*                      Geometric Design (CAGD). This is a research activity, and
/*                      OpenGL has no knowlege about it.
/*Copy Right: All the code contained in this library is protected by copy
/*           right. Permission to use, copy, modify, and distribution for
/*           free. This software has no implied warranty, and no respossible
/*           for any misuseof it, or any damage arising out of its use. The
/*           entire risk of using this software lies with the party doing so.
/*Author:    Chunyan Ye
/*Date  :    03/31/2003
/*****************************************************************************/

#include "myOpenGL.h"
#include <iostream.h>
#include <math.h>

TYPE *ctrlpoints;
TYPE *rpoints;
TYPE *cpoints;
triangleData myData;
coonData rData, cData;

boxdata bData;
bool isNormal = false;

/***************** myEnable ************************
* Purpose : Define a condition
* Argument: GLenum
* Return  : None.
* Note    : First edition
*****************************************************/
void myEnable(GLenum nor)
{
       if(nor == GL_AUTO_NORMAL)
              isNormal = true;
}

/***************** myTriangleMap2f*********************
* Purpose : Define a object, get it registered.
* Argument: GLenum, TYPE, TYPE, GLint, GLint, TYPE, TYPE, TYPE
* Return  : None.
* Note    : First edition
*****************************************************/
void myTriangleMap2f(GLenum target, TYPE u1, TYPE u2,
                                       GLint stride, GLint order, TYPE v1, TYPE v2, TYPE
*points)
{
       myData.m_order = order;
       myData.m_stride = stride;
       ctrlpoints = points;
}

/***************** myTriangleEvalCoord2f*********************
* Purpose : Define a point in the object.
* Argument: TYPE, TYPE
* Return  : None.
* Note    : First edition
*****************************************************/
void myTriangleEvalCoord2f(TYPE u, TYPE v)
{
```

```
	int i, j,k, h, numPoints, numTriangle, newOrder;
	TYPE w; /* define the third parameter */

	/* Calculate number of control points */
	numPoints = (myData.m_order)*(myData.m_order + 1)/2;

	/* Get number of triangles for degration */
	numTriangle = numPoints - myData.m_order;

	/* Get new order of Bezier thriangle */
	newOrder = myData.m_order;

	/* Set stride to a simple letter */
	int s = myData.m_stride;

	/* Three parameters */
	w = 1 - u - v; /* u+v+w=1*/

	/* New number of control points after degration */
	int newPoints = numTriangle*3;

	/* Allocate memory for all pointers */
	TYPE *temp = (TYPE *)malloc(newPoints * sizeof(TYPE));
	TYPE *point, *p1[3], *p2, *p3;
	point = (TYPE *)malloc(s * sizeof(TYPE));
	for(i = 0; i < 3; i++)
		p1[i] = (TYPE *)malloc(s * sizeof(TYPE));

	p2 = (TYPE *)malloc(s * sizeof(TYPE));
	p3 = (TYPE *)malloc(s * sizeof(TYPE));

	/* First iteration */
	h=0;
	for (k=1; k<newOrder; k++)
	{
		for(j=0; j<k*s; j++)
		{
			temp[h]= u * ctrlpoints[h] + v * ctrlpoints[h+k*s] + w *
ctrlpoints[h+(1+k)*s];
			h++;
		}
	}
	numPoints = numPoints - newOrder;
	newOrder--;
	numTriangle = numPoints - newOrder;
	newPoints = numTriangle*3;

	/* Continue iteration */
	while(numTriangle>1)
	{
		h=0;
		for (k=1; k<newOrder; k++)
		{
			for(j=0; j<k*s; j++)
			{
				temp[h]= u * temp[h] + v * temp[h+k*s] + w *
temp[h+(1+k)*s];
				h++;
			}
		}
		numPoints = numPoints - newOrder;
		newOrder--;
		numTriangle = numPoints - newOrder;
		newPoints = numTriangle*3;
	}

	/* Get normal for each point */
	if(isNormal)
	{
		for (i=0; i<3; i++)
		{
```

```
                        for (j=0; j<s; j++)
                                p1[i][j]=temp[i*s+j];
                }
            for(j=0; j<s; j++) p2[j]=p1[2][j]-p1[1][j];
         for(j=0; j<s; j++) p3[j]=p1[0][j]-p1[1][j];
                crossproduct(p2, p3, point);
                normal(point);
            glNormal3fv(point);
        }

        /* Get the point */
        for(j=0; j<s; j++)
                temp[j]= u * temp[j] + v * temp[j+s] + w * temp[j+2*s];
        glVertex3fv(temp);

        /* Return memory to system */
        free (temp);
        free (point);
        for(i = 0; i < 3; i++)
                free(p1[i]);
        free (p2);
        free (p3);

}

/***************** myTriangleMapGrid2f*********************
* Purpose : Design a object.
* Argument: GLint, TYPE, TYPE, GLint, TYPE, TYPE
* Return  : None.
* Note    : First edition
*****************************************************/
void myTriangleMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2)
{
        myData.m_nu = nu;
        myData.m_nv = nv;
        myData.m_u1 = u1;
        myData.m_u2 = u2;
        myData.m_v1 = v1;
        myData.m_v2 = v2;
}


/***************** myTriangleEvalMesh2********************
* Purpose : Make a mesh object.
* Argument: GLenum, GLint, GLint, GLint, GLint
* Return  : None.
* Note    : First edition
*****************************************************/
void myTriangleEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2)
{
                int i, j, h;
        TYPE u0, v0;
        switch (mode)
     {
                case GL_POINT:
            // Execute these statements if expression is equal to GL_POINT
                glBegin(GL_POINTS);
                for (i = i1; i <= (i2 + j2); i++)
                {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                        for (j = j1; j <= (i2+j2)-i; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                           u0 = myData.m_u1 + i*(myData.m_u2-
myData.m_u1)/(myData.m_nu+myData.m_nv);
                           v0 = myData.m_v1 + j*(myData.m_v2-
myData.m_v1)/(myData.m_nu+myData.m_nv);
                           myTriangleEvalCoord2f((GLfloat)u0, v0);
                        }

                }
```

```
                glEnd();

        break;

                case GL_LINE:
           // Execute these statements if expression is equal to GL_LINE

                for(i=i1; i<(i2+j2); i++)
                {

                  for(h=j1; h<(i2+j2)-i; h++)
                  {
                        glBegin(GL_LINE_STRIP);
                   for (j = 0; j <= (i2+j2)-i-h; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = myData.m_u1 + i*(myData.m_u2-
myData.m_u1)/(myData.m_nu+myData.m_nv);
                                v0 = myData.m_v1 + j*(myData.m_v2-
myData.m_v1)/(myData.m_nu+myData.m_nv);
                                myTriangleEvalCoord2f((GLfloat)u0, v0);
                        }
                 glEnd();
                        glBegin(GL_LINE_STRIP);
                   for (j = 0; j <= (i2+j2)-i-h; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = myData.m_u1 + j*(myData.m_u2-
myData.m_u1)/(myData.m_nu+myData.m_nv);
                                v0 = myData.m_v1 + i*(myData.m_v2-
myData.m_v1)/(myData.m_nu+myData.m_nv);
                                myTriangleEvalCoord2f((GLfloat)u0, v0);
                        }
                 glEnd();
                        glBegin(GL_LINE_STRIP);
                   for (j = 0; j <= (i2+j2)-i-h; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = myData.m_u1 + j*(myData.m_u2-
myData.m_u1)/(myData.m_nu+myData.m_nv);
                                v0 = myData.m_v1 + (i2+j2-j-h)*(myData.m_v2-
myData.m_v1)/(myData.m_nu+myData.m_nv);
                                myTriangleEvalCoord2f((GLfloat)u0, v0);
                        }
                 glEnd();

                  }

                }

        break;

                case GL_FILL:
           // Execute these statements if expression is equal to GL_FILL
                for (i = i1; i <=(i2+j2); i++)
                {
                glBegin(GL_TRIANGLE_STRIP);
            for (j = j1; j <(i2+j2)-i; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = myData.m_u1 + i*(myData.m_u2-myData.m_u1)/(myData.m_nu +
myData.m_nv);
                                v0 = myData.m_v1 + j*(myData.m_v2-myData.m_v1)/(myData.m_nu +
myData.m_nv);
                                myTriangleEvalCoord2f(u0, v0);

                                     // myTriangleEvalCoord2f((GLfloat)i/nLines,
(GLfloat)j/nLines);
                                u0 = myData.m_u1 + (i+1)*(myData.m_u2-myData.m_u1)/(myData.m_nu
+ myData.m_nv);
```

```
                        v0 = myData.m_v1 + j*(myData.m_v2-myData.m_v1)/(myData.m_nu +
myData.m_nv);
                        myTriangleEvalCoord2f(u0, v0);
                             // myTriangleEvalCoord2f((GLfloat)i/nLines,
(GLfloat)j/nLines);
                        u0 = myData.m_u1 + (i)*(myData.m_u2-myData.m_u1)/(myData.m_nu +
myData.m_nv);
                        v0 = myData.m_v1 + (j+1)*(myData.m_v2-myData.m_v1)/(myData.m_nu
+ myData.m_nv);
                        myTriangleEvalCoord2f(u0, v0);
                        u0 = myData.m_u1 + (i+1)*(myData.m_u2-myData.m_u1)/(myData.m_nu
+ myData.m_nv);
                        v0 = myData.m_v1 + (j)*(myData.m_v2-myData.m_v1)/(myData.m_nu +
myData.m_nv);
                        myTriangleEvalCoord2f(u0, v0);
                    }
            glEnd();
               }
            // ...
               default:
         // These statements executed if none of the others are
               break;
     }
}

/***************** normal ***************************
* Purpose : Make a normal vector for that plate.
* Argument: Point p
* Return  : None.
* Note    : From Dr. Barrettm note.
*****************************************************/
void normal(TYPE *p)
{

/* normalize a vector */

    float d =0.0;
    int i;
    for(i=0; i<3; i++) d+=p[i]*p[i];
    d=sqrt(d);
    if(d>0.0) for(i=0; i<3; i++) p[i]/=d;
}
/***************** crossproduct ***************************
* Purpose : Multiple two vectors get cross product.
* Argument: Point p1, point p2, point p3, point product
* Return  : None.
*****************************************************/
void crossproduct(TYPE *p1, TYPE *p2, TYPE *n)
{
   /* two vector multiple */
    n[0]=p1[1]*p2[2]-p1[2]*p2[1];
    n[1]=p1[2]*p2[0]-p1[0]*p2[2];
    n[2]=p1[0]*p2[1]-p1[1]*p2[0];
}
void myRuleMap2f(GLenum target, TYPE u1, TYPE u2,
                                        GLint stride, GLint order, TYPE v1, TYPE v2, TYPE
*points)
{
       rData.m_order1 = order;
       rData.m_stride = stride;
       rpoints = points;
}

void myRuleEvalCoord2f(TYPE u, TYPE v)
{

       int i;
       TYPE w = 1-u;
       TYPE temp[3], temp1[3], temp2[3];
       for (i=0; i<rData.m_stride; i++)
       {
```

```
        temp1[i] =
w*w*w*rpoints[i]+3*u*w*w*rpoints[i+3]+3*u*u*w*rpoints[i+6]+u*u*u*rpoints[i+9];
        temp2[i] =
w*w*w*rpoints[i+12]+3*u*w*w*rpoints[i+15]+3*u*u*w*rpoints[i+18]+u*u*u*rpoints[i+21];
                temp[i]  = (1-v)*temp1[i] + v*temp2[i];
        }
        glVertex3fv(temp);
}
/***************** myCoonsMap2f ***************************
* Purpose : Define a Coons patch (evaluator)
* Argument: GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order1,
                      GLint order2,TYPE v1, TYPE v2, GLint order3, GLint order4, TYPE
*points
* Return  : None.
* Notice  : First edition
*******************************************************/
void myCoonsMap2f(GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order1,
                                                GLint order2,TYPE v1, TYPE v2, GLint
order3, GLint order4, TYPE *points)
{
        cData.m_order1 = order1;
        cData.m_stride = stride;
        cData.m_order2 = order2;
        cData.m_order3 = order3;
        cData.m_order4 = order4;

        cpoints = points;
}

/***************** getPoint ***************************
* Purpose : Calculate a point
* Argument: Lint aOrder, GLint aStride, TYPE *ps, TYPE m
* Return  : None.
*******************************************************/
void getPoint(GLint aOrder, GLint aStride, TYPE *ps, TYPE m)
{
        int i;
        while(aOrder > 1)
        {
                for (i=0; i<aStride*aOrder; i++)
                {
                        ps[i] = (1-m)*ps[i] + m*ps[i+aStride];
                }
                aOrder--;
        }

}
/***************** myCoonsEvalCoord2f*********************
* Purpose : Define a point in a Coons patch
* Argument: TYPE u, TYPE v
* Return  : None.
* Note    : First edition
*******************************************************/
void myCoonsEvalCoord2f(TYPE u, TYPE v)
{
        /* declare local variables */
        int i, j;
        int numP1, numP2, numP3, numP4;
        int numC1, numC2, numC3;
        TYPE w, h;
        w = (1.0-u);
        h = (1.0-v);
        /* get number of points in one vector */
        numP1 = cData.m_order1*cData.m_stride;
        numP2 = cData.m_order2*cData.m_stride;
        numP3 = cData.m_order3*cData.m_stride;
        numP4 = cData.m_order4*cData.m_stride;

        /* Allocate memory for all pointers */
        TYPE *p1 = (TYPE *)malloc(numP1 * sizeof(TYPE));
        TYPE *p2 = (TYPE *)malloc(numP2 * sizeof(TYPE));
```

```
	TYPE *p3 = (TYPE *)malloc(numP3 * sizeof(TYPE));
	TYPE *p4 = (TYPE *)malloc(numP4 * sizeof(TYPE));
	TYPE *p5 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *p6 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *p7 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *p8 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *n1 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *n2 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *n3 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *n4 = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));
	TYPE *n = (TYPE *)malloc(cData.m_stride * sizeof(TYPE));

	numC1 = numP1+numP2+numP3-(3*cData.m_stride);
	numC2 = numP1+numP2-(2*cData.m_stride);
	numC3 = numP1+numP2+numP3+numP4-(5*cData.m_stride);
	/* get each vector points */
	for(i=0; i<numP1; i++)
		p1[i]=cpoints[i];
	for(i=0; i<numP2; i++)
		p2[i]=cpoints[i+(numP1-cData.m_stride)];
	j =0;
	while(j<numP3)
	{
		for(i=0; i<cData.m_stride; i++)
		{
			p3[j]=cpoints[numC1+i];
			j++;
		}
		numC1 = numC1-cData.m_stride;
	}
	for(i=0; i<cData.m_stride;i++)
		p4[i]=cpoints[i];

	j=cData.m_stride;
	while (j<numP4)
	{
		for(i=0; i<cData.m_stride; i++)
		{
			p4[j]=cpoints[numC3+i];
			j++;
		}
		numC3 = numC3-cData.m_stride;
	}
	/* get plot point from each vector */
	getPoint(cData.m_order1, cData.m_stride, p1, u);
	getPoint(cData.m_order2, cData.m_stride, p2, v);
	getPoint(cData.m_order3, cData.m_stride, p3, u);
	getPoint(cData.m_order4, cData.m_stride, p4, v);

	numC1 = numP1+numP2+numP3-(3*cData.m_stride);
	for (j=0; j<cData.m_stride; j++)
	{
		p5[j]=h*p1[j] + v*p3[j];
		p6[j]= w*p2[j] + u*p4[j];
		p7[j]=h*w*cpoints[j]+h*u*cpoints[numP1+j-
cData.m_stride]+v*w*cpoints[numC1+j]+u*v*cpoints[numC2+j];

//	p7[j]=h*w*cpoints[j]+h*u*cpoints[27+j]+v*w*cpoints[9+j]+u*v*cpoints[18+j];
		p8[j]=p5[j]+p6[j]-p7[j];
	//	p8[j]=p7[j];
	}
	if((isNormal)&&(cData.m_stride>2))
	{
		if(u==1.0)
			u = u-0.1;
		else
			u = u+0.1;
		if(v==1.0)
			v=v-0.1;
		else
			v = v+0.1;
```

```
                getPoint(cData.m_order1, cData.m_stride, p1, u);
                getPoint(cData.m_order2, cData.m_stride, p2, v);
                getPoint(cData.m_order3, cData.m_stride, p3, u);
                getPoint(cData.m_order4, cData.m_stride, p4, v);
                for (j=0; j<cData.m_stride; j++)
                {
                        n1[j]=h*p1[j] + v*p3[j] + p6[j] - p7[j];
                        n2[j]= w*p2[j] + u*p4[j] + p5[j] - p7[j];
                }
                for(j=0; j<cData.m_stride; j++) n3[j]=n2[j]-n1[j];
         for(j=0; j<cData.m_stride; j++) n4[j]=p8[j]-n1[j];
                crossproduct(n3, n4, n);
                normal(n);
            glNormal3fv(n);
        }

        glVertex3fv(p8);
        /* get back memory */
        free(p1);
        free(p2);
        free(p3);
        free(p4);
        free(p5);
        free(p6);
        free(p7);
        free(p8);
        free(n1);
        free(n2);
        free(n3);
        free(n4);
        free(n);

}

/***************** myCoonsMapGrid2f*********************
* Purpose : Define a Coons patch grid.
* Argument: GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2
* Return  : None.
* Note    : First edition
*******************************************************/
void myCoonsMapGrid2f(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2)
{
        cData.m_nu = nu;
        cData.m_nv = nv;
        cData.m_u1 = u1;
        cData.m_u2 = u2;
        cData.m_v1 = v1;
        cData.m_v2 = v2;
}
/***************** myCoonsEvalMesh2 *********************
* Purpose : Draw Coons patch mesh
* Argument: GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2
* Return  : None.
* Note    : First edition
*******************************************************/
void myCoonsEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2)
{
        int i, j;
        TYPE u0, v0;
        switch (mode)
     {
                case GL_POINT:
            // Execute these statements if expression is equal to GL_POINT
                glBegin(GL_POINTS);
                for (i = i1; i <= i2; i++)
                {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                        for (j = j1; j <= j2; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                           u0 = cData.m_u1 + i*(cData.m_u2-cData.m_u1)/cData.m_nu;
```

```
                        v0 = cData.m_v1 + j*(cData.m_v2-cData.m_v1)/cData.m_nv;
                        myCoonsEvalCoord2f((GLfloat)u0, v0);
                    }

                }
                glEnd();

        break;

                case GL_LINE:
           // Execute these statements if expression is equal to GL_LINE

                for(i=i1; i<=i2; i++)
                {

                        glBegin(GL_LINE_STRIP);
                   for (j = j1; j <= j2; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = cData.m_u1 + i*(cData.m_u2-cData.m_u1)/cData.m_nu;
                                v0 = cData.m_v1 + j*(cData.m_v2-cData.m_v1)/cData.m_nv;
                                myCoonsEvalCoord2f((GLfloat)u0, v0);
                        }
                 glEnd();
                        glBegin(GL_LINE_STRIP);
                    for (j = j1; j <= j2; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = cData.m_u1 + j*(cData.m_u2-rData.m_u1)/cData.m_nu;
                                v0 = cData.m_v1 + i*(cData.m_v2-rData.m_v1)/cData.m_nv;
                                myCoonsEvalCoord2f((GLfloat)u0, v0);
                        }
                 glEnd();

                }

        break;

                case GL_FILL:
           // Execute these statements if expression is equal to GL_FILL
                for (i = i1; i <= i2; i++)
                {
                glBegin(GL_QUAD_STRIP);
             for (j = j1; j <=j2; j++)
                        {
                // myTriangleEvalCoord2f((GLfloat)i/nLines, (GLfloat)j/nLines);
                                u0 = cData.m_u1 + i*(cData.m_u2-cData.m_u1)/cData.m_nu;
                                v0 = cData.m_v1 + j*(cData.m_v2-cData.m_v1)/cData.m_nv;
                                myCoonsEvalCoord2f(u0, v0);

                                    // myTriangleEvalCoord2f((GLfloat)i/nLines,
(GLfloat)j/nLines);
                                u0 = cData.m_u1 + (i+1)*(cData.m_u2-cData.m_u1)/cData.m_nu;
                                v0 = cData.m_v1 + j*(cData.m_v2-cData.m_v1)/ cData.m_nv;
                                myCoonsEvalCoord2f(u0, v0);
                                    // myTriangleEvalCoord2f((GLfloat)i/nLines,
(GLfloat)j/nLines);
                        }
             glEnd();
                }
           // ...
                default:
         // These statements executed if none of the others are
                break;
    }
}

/***************************** myBoxMap2f ************************************
* Purpose : Box Splines evaluator
* Argument: GLenum target, GLint stride, GLint order1, GLint order2, TYPE *points
* Return  : None.
```

```
* Note    : New design function
*******************************************************************************/
void myBoxMap2f(GLenum target, GLint stride, GLint order1, GLint order2, TYPE *points)
{
	bData.m_order1 = order1;
	bData.m_order2 = order2;
	bData.m_stride = stride;
	bpoints = points;
}

/****************************** myBoxEvalMesh2*************************************
* Purpose : Box Splines evaluator
* Argument: GLenum mode, GLint *v, GLint vCount
* Return  : None.
* Note    : New design function
*******************************************************************************/
void myBoxEvalMesh2(GLenum mode, GLint *v, GLint vCount)
{
	/* declare variables  */
	int i, j, h, ii, pos, lev;
	int newPoints, numPoints, numPoint;
	int newOrder1, newOrder2;
	int temp1, temp2, temp3;
	TYPE *p[12];
	TYPE *bpoints2, *bpoints3;
	GLint *vec[2];

	/* Calculate number of input control points */
	numPoint = (bData.m_order1-1)*(bData.m_order2-1);

	/* get new contol points after subdivision */
	temp1 = (bData.m_order1*2) -2;
	temp2 = (bData.m_order2*2) -2;
	temp3 = temp1+1;
	newPoints = temp1*temp2;
	bData.m_mode = mode;
	numPoints = (temp1+2)*(temp2+2);

	/* allocate memory */
	bpoints2 = (TYPE *)malloc((bData.m_stride*numPoints) * sizeof(TYPE));
	bpoints3 = (TYPE *)malloc((bData.m_stride*numPoints) * sizeof(TYPE));

	for(i=0; i<2; i++)
		vec[i] = (GLint*)malloc(vCount*sizeof(GLint));

	for(i = 0; i < 12; i++)
		p[i] = (TYPE *)malloc(bData.m_stride * sizeof(TYPE));

	/* get vector from user */
	h=0;
	for(i=0; i<vCount; i++)
	{
		for(j=0; j<2; j++)
		{
			vec[i][j]=v[h];
			h++;
		}
		h++;
	}

	/* Basic subdivision from vector{(1,0), (0,1), (1,1)} */
	pos=0; lev=0;
	for (i=0; i<bData.m_order2-1; i++)		/* row data  */
	{
		lev=i*bData.m_order1*bData.m_stride;
		for(j=0; j<bData.m_order1-1; j++)		/* column data */
		{
			/* average neighbors */
			for(h=0; h<bData.m_stride; h++)
			{
				bpoints2[h+pos]=bpoints[lev+h+j*bData.m_stride];
```

```
	bpoints2[h+pos+bData.m_stride]=(bpoints[lev+h+j*bData.m_stride]+bpoints[lev+h+(j+1
)*bData.m_stride])/2;

	bpoints2[h+pos+2*bData.m_stride]=bpoints[lev+h+(j+1)*bData.m_stride];

	bpoints2[h+pos+(temp3)*bData.m_stride]=(bpoints[lev+h+j*bData.m_stride]+bpoints[le
v+h+(j+bData.m_order1)*bData.m_stride])/2;

	bpoints2[h+pos+(temp3+1)*bData.m_stride]=(bpoints[lev+h+j*bData.m_stride]+bpoints[
lev+h+(j+bData.m_order1+1)*bData.m_stride])/2;

	bpoints2[h+pos+(temp3+2)*bData.m_stride]=(bpoints[lev+h+(j+1)*bData.m_stride]+bpoi
nts[lev+h+(j+bData.m_order1+1)*bData.m_stride])/2;

			}
			pos=pos+2*bData.m_stride;
		}
		pos = (i+1)*temp3*2*bData.m_stride;
	}
	/* last row process */
	for(j=0; j<bData.m_order1-1; j++)
	{
		for(h=0; h<bData.m_stride; h++)
		{
			lev=(bData.m_order2-1)*bData.m_order1*bData.m_stride;
			bpoints2[h+pos]=bpoints[lev+h+j*bData.m_stride];

	bpoints2[h+pos+bData.m_stride]=(bpoints[lev+h+j*bData.m_stride]+bpoints[lev+h+(j+1
)*bData.m_stride])/2;

	bpoints2[h+pos+2*bData.m_stride]=bpoints[lev+h+(j+1)*bData.m_stride];
		}
		pos=pos+2*bData.m_stride;
	}

	/* get new set of control points */

	newOrder1 = bData.m_order1*2 -1;
	newOrder2 = bData.m_order2*2 -1;

	/* process vector from user */
	for(i=0; i<vCount; i++)
	{
		int x, y;
		x=vec[i][0];
		y=vec[i][1];
		if(x==0)
		{
			if(y>0)			/* (0, +y) */
			{
				pos=0;
				for(j=0; j<newOrder2-y; j++)
				{
					lev=j*newOrder1*bData.m_stride;
					for(h=0; h<newOrder1; h++)
					{
						for(ii=0; ii<bData.m_stride; ii++)
						{

	bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+y*newOrde
r1)*bData.m_stride])/2;
						}
						pos=pos+bData.m_stride;
					}
				}
				newOrder2=newOrder2-y;
			}
			else if(y <0)		/* (0, -y) */
			{
				pos=0;
```

```
                                for(j=0-y; j<newOrder2; j++)
                                {
                                        lev=j*newOrder1*bData.m_stride;
                                        for(h=0; h<newOrder1; h++)
                                        {
                                                for(ii=0; ii<bData.m_stride; ii++)
                                                {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+y*newOrde
r1)*bData.m_stride])/2;
                                                }
                                                pos=pos+bData.m_stride;
                                        }
                                }
                                newOrder2=newOrder2+y;
                        }
                }
                else if(x >0)
                {
                        if(y >0)                                /* (+x, +y) */
                        {
                                pos=0;
                                for(j=0; j<newOrder2-y; j++)
                                {
                                        lev=j*newOrder1*bData.m_stride;
                                        for(h=0; h<newOrder1-x; h++)
                                        {
                                                for(ii=0; ii<bData.m_stride; ii++)
                                                {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x+y*newOr
der1)*bData.m_stride])/2;
                                                }
                                                pos=pos+bData.m_stride;
                                        }
                                }
                                newOrder1=newOrder1-x;
                                newOrder2=newOrder2-y;
                        }
                        else if(y<0)                            /* (+x, -y) */
                        {
                                pos=0;
                                for(j=0-y; j<newOrder2; j++)
                                {
                                        lev=j*newOrder1*bData.m_stride;
                                        for(h=0; h<newOrder1-x; h++)
                                        {
                                                for(ii=0; ii<bData.m_stride; ii++)
                                                {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x+y*newOr
der1)*bData.m_stride])/2;
                                                }
                                                pos=pos+bData.m_stride;
                                        }
                                }
                                newOrder2=newOrder2+y;
                                newOrder1=newOrder1-x;
                        }
                        else                                    /* (+x, 0) */
                        {
                                pos=0;
                                for(j=0; j<newOrder2; j++)
                                {
                                        lev=j*newOrder1*bData.m_stride;
                                        for(h=0; h<newOrder1-x; h++)
                                        {
                                                for(ii=0; ii<bData.m_stride; ii++)
                                                {
```

```
        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x)*bData.
m_stride])/2;
                                        }
                                        pos=pos+bData.m_stride;
                                }
                        }
                        newOrder1=newOrder1-x;
                }
        }
        else
        {
                if (y>0)                            /* (-x, y) */
                {
                        pos=0;
                        for(j=0; j<newOrder2-y; j++)
                        {
                                lev=j*newOrder1*bData.m_stride;
                                for(h=0-x; h<newOrder1; h++)
                                {
                                        for(ii=0; ii<bData.m_stride; ii++)
                                        {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x+y*newOr
der1)*bData.m_stride])/2;
                                        }
                                        pos=pos+bData.m_stride;
                                }
                        }
                        newOrder1=newOrder1+x;
                        newOrder2=newOrder2-y;
                }
                else if(y <0)                       /* (-x, -y) */
                {
                        pos=0;
                        for(j=0-y; j<newOrder2; j++)
                        {
                                lev=j*newOrder1*bData.m_stride;
                                for(h=0-x; h<newOrder1; h++)
                                {
                                        for(ii=0; ii<bData.m_stride; ii++)
                                        {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x+y*newOr
der1)*bData.m_stride])/2;
                                        }
                                        pos=pos+bData.m_stride;
                                }
                        }
                        newOrder1=newOrder2+x;
                        newOrder2=newOrder2+y;
                }
                else                            /* (-x, 0) */
                {
                        pos=0;
                        for(j=0; j<newOrder2; j++)
                        {
                                lev=j*newOrder1*bData.m_stride;
                                for(h=0-x; h<newOrder1; h++)
                                {
                                        for(ii=0; ii<bData.m_stride; ii++)
                                        {

        bpoints3[pos+ii]=(bpoints2[lev+ii+(h*bData.m_stride)]+bpoints2[ii+lev+(h+x)*bData.
m_stride])/2;
                                        }
                                        pos=pos+bData.m_stride;
                                }
                        }
                        newOrder1=newOrder1+x;
                }
```

```
                }
        }

        /* if no vector input by user */
        if(vCount==0)
        {
                for(i=0; i<bData.m_order2*2-2; i++)
                {
                        for(j=0; j<bData.m_order1*2-2; j++)
                        {
                                temp3=(bData.m_order1*2-1);
                                lev=i*temp3*bData.m_stride;
                                for(h=0; h<bData.m_stride; h++)
                                {
                                        p[0][h]=bpoints2[lev+h+j*bData.m_stride];
                                        p[1][h]=bpoints2[lev+h+(1+j)*bData.m_stride];
                                        p[2][h]=bpoints2[lev+h+(j+temp3)*bData.m_stride];
                                        p[3][h]=bpoints2[lev+h+(j+1+temp3)*bData.m_stride];
                                }
                                draw(p[0], p[1], p[3],p[2]);
                        }
                }
        }
        else
        {
                if((newOrder1>1)&&(newOrder2>1))
                {
                        for(i=0; i<newOrder2-1; i++)
                        {
                                for(j=0;j<newOrder1-1; j++)
                                {
                                        lev=i*newOrder1*bData.m_stride;
                                        for(h=0; h<bData.m_stride; h++)
                                        {
                                                p[0][h]=bpoints3[lev+h+j*bData.m_stride];

        p[1][h]=bpoints3[lev+h+(1+j)*bData.m_stride];

        p[2][h]=bpoints3[lev+h+(j+temp3)*bData.m_stride];

        p[3][h]=bpoints3[lev+h+(j+1+temp3)*bData.m_stride];
                                        }
                                        draw(p[0], p[1], p[3],p[2]);
                                }
                        }
                }
        }

        /* return memeory to system */
        for(i = 0; i < 12; i++)
                free(p[i]);
        for(i=0; i<2; i++)
                free(vec[i]);
        free(bpoints2);
}

/***************** draw ***************************
* Purpose : Draw grid from given four points
* Argument: TYPE *p1, TYPE *p2, TYPE *p3, TYPE *p4
* Return  : None.
*****************************************************/
void draw(TYPE *p1, TYPE *p2, TYPE *p3, TYPE *p4)
{
        TYPE *pp[6];
        int i, j;

        /* allocate memory */
        for(i = 0; i < 6; i++)
                pp[i] = (TYPE *)malloc(bData.m_stride * sizeof(TYPE));

        /* get normal */
```

```
        if((isNormal)&&(bData.m_stride>2))
        {
                for(i=0; i<bData.m_stride; i++)
                {
                        pp[0][i]=p2[i]-p1[i];
                        pp[1][i]=p4[i]-p1[i];
                        pp[2][i]=p3[i]-p2[i];
                        pp[3][i]=p4[i]-p3[i];
                }
                crossproduct(pp[0], pp[1], pp[4]);
                crossproduct(pp[2], pp[3], pp[5]);
                normal(pp[4]);
                normal(pp[5]);
        }

        /* draw feature in a given mode */
        switch (bData.m_mode)
    {
                case GL_POINT:
                        glBegin(GL_POINTS);
                        break;
                case GL_LINE:
                        glBegin(GL_LINE_STRIP);
                        break;
                case GL_FILL:
                        glBegin(GL_TRIANGLE_STRIP);
                        break;
                default:
                        break;
        }
        glNormal3fv(pp[4]);
        glVertex3fv(p1);
        glVertex3fv(p2);
        glNormal3fv(pp[4]);
        glVertex3fv(p3);
        glNormal3fv(pp[4]);
        glVertex3fv(p4);
        glNormal3fv(pp[4]);
        glVertex3fv(p1);
        glEnd();

        /* return memory to system */
        for(i = 0; i < 6; i++)
                free(pp[i]);
}
```

APPENDIX C

A EXAMPLE OF MAIN PROGRAM

```
/****************** Triangular patch main*********************

/* Purpose: A main program to test tiangular fuctions in myOpenGL
/*          library.
/* Functionalities: ESC for exit, x for eyex++, c for eyex--
/*           y fo reyey++, t for eyey--, z for eyez++, a for eyez--
/*           l for increasing x value of a point
/*           r for decresing  x value of a point
/*           u for increasing y value of a point
/*           d for decresing  y value of a point
/*           m for more defined point
/*           p for less defined point
/*           h for rotate ++
/*           g for rotate --
/*           Right mouse have a list of menu options
/* Reference: A programmer Guide for OpenGL
/* Author   : Chunyan Ye
/* Date for last revision     : 3/31/2003
/*****************************************************************/

#include <GL/glut.h>
#include <stdlib.h>
#include "myOpenGL.h"
#include <math.h>
/* Set up the initial control points.*/


typedef GLfloat TYPE;

TYPE ctrlpoints[10][3] = {{-1.5, -1.5, 4.0},
       {-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
       {-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0}, {0.5, 0.5, 3.0},
       {-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
       {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}};


GLint whichPoint = 0;    /* which control point */
GLint nLines = 10;       /* number of points generated */

/* declare variables */
static GLfloat theta[] = {0.0,0.0,0.0};   /* for rotate */
static GLint axis = 2;                    /* for rotate axis id */

GLfloat xvalue =  2.0;    /* default eye x  */
GLfloat yvalue =  12.0;   /* default eye y  */
GLfloat zvalue =  5.0;    /* default eye z  */
GLfloat lightx =  4.0;    /* default lightx */
GLfloat lighty =  0.0;    /* default lighty */
GLfloat lightz = -0.5;    /* default lightz */

int isColor = 0;          /* default for non colorful*/
GLfloat r = 1.0, g = 1.0, b = 1.0; /* drawing color  */
int color = 0;            /* for color    */
int FuncMode = GL_FILL;   /* default first function  */
typedef float point[4];   /* define point */


/***************** key ***************************
* Purpose : keyboard input to change eye, light position
* Argument: unsigned char key,  int x, int y
* Return  : None.
*****************************************************/
```

```
void key(unsigned char k, int x, int y)
{
         GLfloat delta = 0.2;
    switch (k) {
      case 27:                      /* ESC for exit    */
       exit(0);
       break;
      case 'y':                     /* y for ++eye y v */
       yvalue += 0.2;
       break;
      case 't':                     /* t for --eye y v */
       yvalue -= 0.2;
      case 'x':                     /* x for ++eye x v */
       xvalue += 0.2;
       break;
      case 'c':                     /* c for --eye x v */
       xvalue -= 0.2;
       break;
      case 'z':                     /* z for ++eye z v */
       zvalue += 0.2;
       break;
      case 'a':                     /* a for --eye z v */
       zvalue -= 0.2;
       break;
         case 'l':                      /* increase point x value */
    ctrlpoints[whichPoint][0] = ctrlpoints[whichPoint][0] + delta;
    break;
         case 'r':                      /* decrease point x value */
    ctrlpoints[whichPoint][0] = ctrlpoints[whichPoint][0] - delta;
    break;
         case 'u':                      /* increase point y value */
    ctrlpoints[whichPoint][1] = ctrlpoints[whichPoint][1] + delta;
    break;
         case 'd':                      /* decrease point y value */
    ctrlpoints[whichPoint][1] = ctrlpoints[whichPoint][1] - delta;
    break;
         case 'p':                      /* decrease points */
    nLines = nLines + 1;
    break;
         case 'm':                      /* increase points */
    nLines = nLines - 1;
    break;
         case 32:                      /* choose control point */
    whichPoint = whichPoint + 1;
    if (whichPoint > 10)
      whichPoint = 0;
    break;
         case 'h':                      /* ++rotate points */
    theta[axis] += 2.0;
       if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
    break;
         case 'g':                      /* __rotate points */
    theta[axis] -= 2.0;
       if( theta[axis] < 0.0 ) theta[axis] += 360.0;
    break;
      } /* end switch*/
    glutPostRedisplay( );           /* recall display() */
}
/*void key1( unsigned char k, int x, int y)
{
       if(k==38)
ctrlpoints[whichPoint][0] = ctrlpoints[whichPoint][0] + 0.2;
}*/
/***************** display ****************************
* Purpose : Display responsed call back
* Argument: None.
* Return  : None.
*****************************************************/
void display(void)
{
       int i;
```

```
    point light;         /* point variable to hold light*/
    light[0]=lightx;     /* light x value */
    light[1]=lighty;     /* light y value */
    light[2]=lightz;     /* light z value */

/* Displays all grid and 3d-draw */

     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();
     gluLookAt(xvalue,yvalue,zvalue,2.0,0.0,2.0,0.0,1.0,0.0);
     glLightfv(GL_LIGHT0,GL_POSITION, light);
        glRotatef(theta[0], 1.0, 0.0, 0.0);
        glRotatef(theta[1], 0.0, 1.0, 0.0);
        glRotatef(theta[2], 0.0, 0.0, 1.0);
         /* set color for 3D draw from menu option*/
     if(color == 1) {glColor3f(1.0, 0.0, 0.0);}
     else if(color == 2) {glColor3f(0.0, 1.0, 0.0);}
     else if(color == 3) {glColor3f(0.0, 0.0, 1.0);}
     else if(color == 4) {glColor3f(0.0, 1.0, 1.0);}
     else if(color == 5) {glColor3f(1.0, 0.0, 1.0);}
     else if(color == 6) {glColor3f(1.0, 1.0, 0.0);}
     else if(color == 7) {glColor3f(1.0, 1.0, 1.0);}

     myTriangleMapGrid2f(nLines, 0.0, 1.0, nLines, 0.0, 1.0);
     myTriangleEvalMesh2(FuncMode, 0, nLines, 0, nLines);

  /* Draw the points definng the line. */

        glColor3f(1.0, 0.0, 0.0);
     glPointSize(3.0);
        myTriangleEvalMesh2(GL_POINT, 0, nLines, 0, nLines);

  /* Display the control points as dots, the selected one in blue.*/
     glPointSize(5.0);
     glColor3f(1.0, 1.0, 0.0);
     glBegin(GL_POINTS);
     for (i = 0; i < 10; i++)
        {
       if (i == whichPoint)
           {
          glColor3f(0.0, 0.0, 1.0);
          glVertex3fv(&ctrlpoints[i][0]);
          glColor3f(1.0, 1.0, 0.0);
           }
       else
          glVertex3fv(&ctrlpoints[i][0]);
        }
     glEnd();
     glFlush();
        glutSwapBuffers();
}
/***************** myReshape ***********************
 *  Purpose : Reshape the window if resize the window
 ************************************************/
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    display();
}

/***************** color_menu ***************************
* Purpose : Display color menu
* Argument: Int id.
* Return  : None.
*****************************************************/
void color_menu(int id)
{
```

```
    if(id == 1) {color = 1; isColor=0;}       /* Red     */
    else if(id == 2) {color = 2; isColor=0;}  /* Green   */
    else if(id == 3) {color = 3; isColor=0;}  /* Blue    */
    else if(id == 4) {color = 4; isColor=0;}  /* Cyan    */
    else if(id == 5) {color = 5; isColor=0;}  /* Magenta */
    else if(id == 6) {color = 6; isColor=0;}  /* Yellow  */
    else if(id == 7) {color = 7; isColor=0;}  /* White   */

    glutPostRedisplay( );
}

/***************** func_menu ***************************
* Purpose : Display function menu                      *
* Argument: Int id.                                    *
* Return  : None.                                      *
*******************************************************/
void func_menu(int id)
{
    if(id == 1) { FuncMode = GL_POINT; }
    else if(id == 2) { FuncMode = GL_LINE; }
    else if(id == 3) { FuncMode = GL_FILL; }
    else if(id == 4) { exit(0); }
    glutPostRedisplay( );   /* Calls display function */
}
/***************** point_menu ***************************
* Purpose : Display point increase or decrease menu
* Argument: Int id.
* Return  : None.
*******************************************************/
void point_menu(int id)
{
    if(id == 1)         /* increase point */
    {
       nLines++;
    }
    else if(id == 2)    /* decrease point */
       nLines--;

    glutPostRedisplay( );
}

/***************** whichp_menu ***************************
* Purpose : Choose a control point
* Argument: Int id.
* Return  : None.
*******************************************************/
void whichp_menu(int id)
{
       whichPoint = id -1;
    glutPostRedisplay( );
}

/***************** idle ***************************
* Purpose : Non-stop moving around
* Argument: None.
* Return  : None.
*******************************************************/
void myidle()
{
/* Idle callback, spin toy 2 degrees about selected axis */

       theta[axis] += 2.0;
       if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
       /* display(); */
       glutPostRedisplay();
}

/***************** rotate_menu ***************************
* Purpose : Display Rotate direction                    *
* Argument: Int id.                                     *
* Return  : None.                                       *
```

```
*****************************************************/
void rotate_menu(int id)
{
       if(id == 1) { axis = 0; }
    else if(id == 2) { axis = 1; }
    else if(id == 3) { axis = 2; }
//    glutIdleFunc(myidle);
    glutPostRedisplay( );   /* Calls display function */
}
/*****************************************************************
*  Function Name: mouse                                          *
*  Purpose: If user click on left mouse button, they will stop   *
*           the motion. And click on right mouse will stop motion*
*  Parameters: btn, state, x, y                                  *
*  Returns:  none                                                *
*****************************************************************/
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)  glutIdleFunc(NULL);
}


/***************** myinit ***************************
* Purpose : Initialize window, set up view, atributs
* Argument: None.
* Return  : NOne.
*****************************************************/
void myinit()
{
    GLfloat mat_specular[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_diffuse[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_ambient[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_shininess={1.0};
    GLfloat light_ambient[]={0.2, 0.2, 0.2, 1.0};
    GLfloat light_diffuse[]={1.0, 1.0, 1.0, 1.0};
    GLfloat light_specular[]={1.0, 1.0, 1.0, 1.0};

/* set up ambient, diffuse, and specular components for light 0 */

    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

/* define material proerties for front face of all polygons */

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

    glShadeModel(GL_SMOOTH); /*enable smooth shading */
    glEnable(GL_LIGHTING);   /* enable lighting      */
    glEnable(GL_LIGHT0);     /* enable light 0       */
    glEnable(GL_DEPTH_TEST); /* enable z buffer      */
    glEnable(GL_COLOR_MATERIAL);/*enable color material mode*/

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f (1.0, 1.0, 0.0);

    myTriangleMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
       0.0, 1.0, &ctrlpoints[0][0]);
    /* enable above evaluator */
    glEnable(GL_MAP2_VERTEX_3);
        myEnable(GL_AUTO_NORMAL);
}

/***************** Main Loop ***********************
 *  Purpose :Open window with initial window size,
 *           title bar, RGBA display mode,
 *           and handle input events.
 ************************************************/
```

```
void main(int argc, char **argv)
{
    int c_menu, o_menu, p_menu, r_menu, n_menu;  /* to hold two submenu */

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(700, 700);/* sets window size     */
    glutInitWindowPosition(0,0); /* sets window position */
    glutCreateWindow(" TriangleP -- Chunyan Ye ");   /* create window name */
    glutDisplayFunc(display);    /* sets the call back   */

 /* make function submenu  */
    c_menu = glutCreateMenu(func_menu);
    glutAddMenuEntry("Point",1);
    glutAddMenuEntry("Frame",2);
    glutAddMenuEntry("Surface",3);
    glutAddMenuEntry("Exit",4);

 /* make choose point submenu   */
    n_menu = glutCreateMenu(whichp_menu);
    glutAddMenuEntry("Point1",1);
    glutAddMenuEntry("Point2",2);
    glutAddMenuEntry("Point3",3);
    glutAddMenuEntry("Point4",4);
    glutAddMenuEntry("Point5",5);
    glutAddMenuEntry("Point6",6);
    glutAddMenuEntry("Point7",7);
    glutAddMenuEntry("Point8",8);
    glutAddMenuEntry("Point9",9);
    glutAddMenuEntry("Point10",10);

 /* make color submenu */
    o_menu = glutCreateMenu(color_menu);
    glutAddMenuEntry("Red",1);
    glutAddMenuEntry("Green",2);
    glutAddMenuEntry("Blue",3);
    glutAddMenuEntry("Cyan",4);
    glutAddMenuEntry("Magenta",5);
    glutAddMenuEntry("Yellow",6);
    glutAddMenuEntry("White",7);

 /* make grid submenu */
    p_menu = glutCreateMenu(point_menu);
    glutAddMenuEntry("increase points", 1);
    glutAddMenuEntry("decrease points", 2);

 /* make function submenu  */
    r_menu = glutCreateMenu(rotate_menu);
    glutAddMenuEntry("X axis",1);
    glutAddMenuEntry("Y axis",2);
    glutAddMenuEntry("Z axix",3);

 /* make right button   */
    glutCreateMenu(color_menu);
    glutAddSubMenu("Change_Design", c_menu);
    glutAddSubMenu("ChangeColor", o_menu);
        glutAddSubMenu("ChangePoints", p_menu);
        glutAddSubMenu("ChoosePoint", n_menu);
        glutAddSubMenu("RotateAxis", r_menu);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    myinit();                    /* call initial func     */
    glutReshapeFunc(myReshape);  /* resize window         */
//    glutMouseFunc (mouse);       /* Enable mouse function */
    glutKeyboardFunc(key);       /* call back on key call */
//    glutIdleFunc(myidle);        /* call idle function */
//     glutSpecialFunc( key1 );
    glutMainLoop();
}
```

GLOSSARY

Affine map

Any map that is composed of translations, rotations, scalings, and shears is an affine map.

Affine space

A vector space that adds a third element: the point.

API

Application programming interface

Approximation

Fitting a curve or surface to given data. As opposed to interpolation, the curve or surface approximation only has to be close to data.

Barycentric combination

A weighted average where the sum if the weights equals one.

Barycentric coordinates

A point in E^3 may be written as a unique barycentric combination of three points. The coefficients in this combination are its barycentric coordinates.

Bernstein Functions

The Bernstein functions were originally devised by Bernstein to prove the Weierstrass theorem in 1912.

They are formally given by $B_i^n(t) = \frac{n!}{i!(n-i)!}(1-t)^{n-1}t^i$

Bernstein polynomial

Bernstein function.

Beta-spline curve

B-spline curve. A G^2 piecewise cubic curve is defined over uniform knot sequence.

Bézier Patches

A Bézier patch is a three-dimensional extension of a Bézier curve.

Bézier curve

A polynomial curve that is expressed in terms of Bernstein polynomials

Bézier polygon

Connecting the control points in the correct order, from which a Bezier curve is made.

Blossom

A multivariate polynomial that is associated with a given polynomial.

Blossoming

The process of applying de Casteljau algorithm steps or n de Boor steps to a polynomial.

B-Spline

A piecewise polynomial function

B-Spline Curve

A B-spline curve is a set of piecewise (usually cubic) polynomial segments that pass close to a set of control points.

C2 Continuity

Recall C2 continuity: This means that the second derivatives of the curves are continuous.

CAD

Computer Aided Design.

CAGD

Computer Aided Geometric Design

CAM

Computer Aided Manufacture

Collinear

Points being on a straight line

Continuity

Continuity implies a notion of smoothness: that is, curves are not jagged and do not break.

Control Points

Control points are points in two or more dimensions that define the behavior of the resulting curve.

Control Polygon

A Bézier polygon

Convex

A polygon is convex if no straight line in the plane of the polygon intersects the polygon more than twice.

Convex hull

The smallest convex region encloses a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

Coons patch

A patch is fitted between four arbitrary boundary curves.

Euclidean space

An affine space that adds the concept of distance.

Explicit Equation

A planar curve is given by $y = f(x)$, where $f(x)$ is a prescribed function of x.

GAGD

Graphic Aided Geometry Design

Geometric continuity

Smoothness of a curve or a surface formed by several segments of curves or patches.

Hermite interpolation

Generating a curve or surface from derivatives.

Homogeneous coordinates

A coordinate system with the fourth parameter added to the three-dimensional coordinate that is used to represent rational curves and surfaces.

Implicit Equation

A curve is given by function $f(x, y) = 0$.

Interpolation

Calculation for values at the boundaries such as at the vertices for a polygon or a line.

Knot

A spine curve is defined over a partition of an interval of the real line. The points that define the partition are called knots.

Non-parametric

Curves Explicitly and implicitly defined curves.

Nonuniform Curves

A curve whose knots are unevenly spaced.

NURBS Non-uniform rational B-spline curves.

Parameterization

Parameterization uses an independent parameter or variable to compute points on a curve. It gives the "motion" of a point on the curve.

Parametric Form

In describing curves, using an auxiliary parameter to represent the position of a point.

Patch

Complicated surfaces are usually broken into smaller units, called patches.

Point

An exact location in space, donated as a finite-diameter dot.

Polygon

A near-planar surface bounded by edges specified by vertices.

Polynomials

A polynomial is a function of the form $P(t) = a_0 + a_1t + a_2t^2 + \ldots + a_nt^n$, where the a_i are scalars or vectors.

Polynomial Curve

A parametric curve is a curve that has a polynomial parameterization.

Primitive

A point, a line, a polygon

Rational curve and surface

The curve and surface are ones that are represented in homogeneous coordinates.

Ruled surface

A surface obtained by linear interpolation between two given curves.

Segment

A polynomial (or rational polynomial) curve pieces can form a big curve.

Space

A three-dimension

Spline curve

A continuous curve made from several polynomial segments.

Surface

A continuous map in which there is the locus of all points of a moving and deforming curve.

Surface Patches

Surface patches are three-dimensional surface sections that may be combined to form solid objects.

Tangent Line

The tangent line to a curve is the straight line that gives the curve's slope at a point. This is deduced from the derivative of the curve at the point.

Tensor product

Rectangular surfaces are generated by curve methods.

Texture mapping

The process of applying and image (the texture) to a primitive is called texture mapping.

Triangular patch

The shape of the patch is triangle

Twist vector

The mixed second partial of a parametrization dependent surface

Vector

It is a direction from the difference of two points.

Vertex

A point in three-dimensional space

Weight

The component of the homogeneous coordinate

Weight point

The weights of the control points

VITA

CHUNYAN YE

Personal Data:	Place of Birth: Heilongjiang, China Marital Status: Married, Daughter: Kelly Liu, Husband: Zhiping Liu
Education	Heilongjiang 'August 1st' Land Reclamation University, Mishan, China; Animal Science, B.S., 1985 Nanjing Agricultural University, Nanjing, China and Northeast Forestry University, Harbin, China; Animal Physiology and Biochemistry, M.S., 1989
Professional Expenrience:	Assistant Professor, Northeast Forestry University Harbin, China 1987-1995 Graduate Assistant, East Tennessee State University Computer Science Department 2000-2003
Selected Publications:	Effects of panax ginseng, panax pseudoginseng , eleutercoceus senticosus and schizadra chinesis on protein biosynthsis in mouse brain. Chinese Traditional Patent Medicine. 1993,15(6):30-31 Investigation on the histology of digestive tract in yellow-throated buntings Wildlife, 1994, 1:34 Study on the influence of panax ginseng, panax pseudoginseng , eleutercoceus senticosus and schizadra chinesis on mouse memory. China Forestry By-products, 1994, 3:10-13 Influence of panax ginseng, panax pseudoginseng , eleutercoceus senticosus and schizadra chinesis on the weight of mouse testes. Special Wild Economic Animal and Plants Research , 1995, 3:14-16 A study on the amount of inosinic acid in the muscle of songhuajiang river carps and their meat keeping time. Aqatic Science, 1995, 14(5):15-17