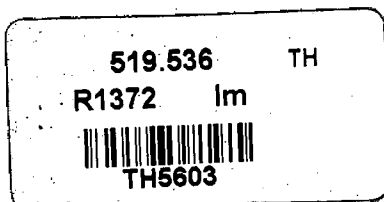


IMPLEMENTATION OF VORONOI DIAGRAMS AND TRIANGULAR BEZIER PATCHES

Dissertation submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements
for the award of the degree of
MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE

SANJAY RAJ

SCHOOL OF COMPUTER AND SYSTEM SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067



1174

Library copy

IMPLEMENTATION OF VORONOI DIAGRAMS AND TRIANGULAR BEZIER PATCHES

JAWAHARLAL

Dissertation submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements
for the award of the degree of
MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE



SANJAY RAJ

SCHOOL OF COMPUTER AND SYSTEM SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067

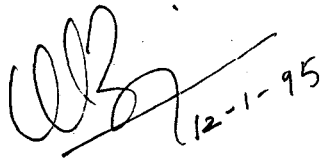
CERTIFICATE

This is to certify that the dissertation entitled

**"Implementation of Voronoi Diagrams and
Triangular Bezier Patches",**

which is being submitted by **Sanjay Raj** to the School of Computer and System Sciences, Jawahar Lal Nehru University, New Delhi for the award of Master of Technology in Computer Sciences, is a record of bonafide project work carried out by him under the supervision and guidance of **Dr. S Balasundaram**.

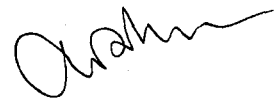
This work is original and has not been submitted in part or full to any other University or Institution for the award of any degree.



12-1-95

Prof K.K. BHARADWAJ

(Dean, SC&SS)



Dr. S. BALASUNDARAM

(Supervisor)

Acknowledgements

I wish to take this opportunity to thank all the persons who have helped and encouraged me throughout my stay here at JNU.

First of all I wish to thank our respected Dean Prof. K.K. Bhardwaj for providing us with a nice and efficient working environment. I also wish to thank all the other hon'ble faculty members for giving me a chance to learn from their vast experience.

Above all I wish to express my gratitude to my supervisor Dr. Balasundaram for his invaluable guidance. He has been a remarkable influence in my life.

Finally I thank Arun for helping me with computations and Vishal for his assistance in typing the project report.

Sanjay Raj

CONTENTS

<i>Chapter One</i>	
Introduction	1
<i>Chapter Two</i>	
Curves and Surfaces	3
<i>Chapter Three</i>	
Voronoi Diagrams and Delaunay's Triangulation	9
<i>Chapter Four</i>	
The Triangular Surface Patches	15
Conclusion	20
References	21
Figure Captions	22

CHAPTER ONE

INTRODUCTION

It can be justifiably be said that we are living in a computer age. From 1940's when computers were just what their name suggested i.e., number crunchers, the computers have evolved a long way. Now they are not only used in arithmetic and logic operations but also in areas such as :

- Simulation
- CAD/CAM
- Medicine
- Image processing
- Communications
- Artificial Intelligence
- Business and Finance
- Modern art and etc.

A cursory look at the above list will show that almost all these fields are based on or utilize extensively the means of pictorial or graphical representation. Thus, it is no coincidence that the field of computer graphics which is primarily concerned with rapid and economical production of pictures has experienced an almost explosive growth.

Computer-generated pictures being a mapping of images from 3D- to 2D-space , it is important to devise methods that would do the mapping in a way so as to preserve the features of the object under study as best as possible. Specially in fields like CAD/CAM, geographical representation and scientific research. It is in this regard that mathematical representation of curves and surfaces comes handy. It is easy to generate and display curves and surfaces using mathematical representation . Moreover even when the exact features of surfaces/curves being represented are

not known, we can make surfaces/curves which would approximate or interpolate to whatever information (points, tangents etc.) we have regards parametric representation for curves and surfaces have thus become very common in graphics. Insofar a representation of surface is concerned, most of the present methods utilize the parametric rectangular surface patches. This is because of their simple mathematical representation and intuitive appeal, the prominent representation in this category are Bezier surface patches, cubic surface patches, Coon surfaces, Splines and Hermite surface patches.

Another method for representation of surfaces is to use Triangular patches and it is this representation that I have endeavoured to understand and implement. Indeed there has not been much research in this formulation as compared to rectangular surface patches. As we shall see the Triangular representation for surface patches is a much more powerful tool for interpolating a surface through a set of datapoints. The outline of my thesis is as follows : Chapter 2 defines the currently used in implementation of curves/surfaces. In Chapter 3 we discuss the concepts of Voronoi diagrams and Delaunay's triangulation.

Finally, in Chapter 4 we discuss in depth the concept of triangular surface patches with special emphasis on the Bezier triangular patches and its implementation.

CHAPTER TWO

CURVES AND SURFACES

2.1 Introduction

In this chapter we shall discuss the currently used methods for implementing curves and surfaces. We define the following representations:

- Algebraic and geometric
 - a) Cubic and Bicubic
 - b) Hermite

- Bezier

We then review the properties of these representations. Finally we introduce Triangle representation for surface patches.

2.2 Curves

Mathematically a curve is defined as a locus of points that satisfies a relation of form

$$f(x, y, z) = c$$

In the parametric representation we also introduced a fourth parameter u into the coordinate system. Any point on the curve can then be represented by a continuous single-valued vector function of the form

$$\mathbf{p}(u) = (x(u), y(u), z(u))$$

The parametric variable u is constrained to the interval $u \in [0, 1]$, and the positive sense on a curve in which u increases. The curve is a point-bounded because it has two definite end-points, one at $u = 0$ and the other at $u = 1$.

Algebraic and Geometric Form

In algebraic form a parametric cubic curve is given by the following equation

$$\mathbf{p}(u) = \mathbf{a}_3 u^3 + \mathbf{a}_2 u^2 + \mathbf{a}_1 u + \mathbf{a}_0 \quad (X)$$

where \mathbf{p} and the coefficients \mathbf{a} are all vectors and $u \in [0, 1]$ and which is equivalent to following set of equations

$$\begin{aligned} x(u) &= a_{3x} u^3 + a_{2x} u^2 + a_{1x} u + a_{0x} \\ y(u) &= a_{3y} u^3 + a_{2y} u^2 + a_{1y} u + a_{0y} \\ z(u) &= a_{3z} u^3 + a_{2z} u^2 + a_{1z} u + a_{0z} \end{aligned} \quad (Y)$$

where a_{3x}, a_{3y}, a_{3z} are components of vector \mathbf{a}_3 and so on.

Using equations (X) and (Y), the end-point condition and the corresponding tangent vectors $\mathbf{p}^u = d\mathbf{p}/du$ we obtain the following

$$\begin{aligned} \mathbf{p}(0) &= \mathbf{a}_0 \\ \mathbf{p}(1) &= \mathbf{a}_0 + \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}_3 \\ \mathbf{p}^u(0) &= \mathbf{a}_1 \\ \mathbf{p}^u(1) &= \mathbf{a}_1 + 2\mathbf{a}_2 + \mathbf{a}_3 \end{aligned}$$

By solving this set of four simultaneous equations in four unknowns we can redefine the algebraic coefficients in terms of the boundary conditions

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{p}(0) \\ \mathbf{a}_1 &= \mathbf{p}^u(0) \\ \mathbf{a}_2 &= -3\mathbf{p}(0) + 3\mathbf{p}(1) - 2\mathbf{p}^u(0) - \mathbf{p}^u(1) \\ \mathbf{a}_3 &= 2\mathbf{p}(0) - 2\mathbf{p}(1) + \mathbf{p}^u(0) + \mathbf{p}^u(1) \end{aligned}$$

Substituting these expressions for algebraic coefficients in equation (X) we get

$$\begin{aligned} \mathbf{p}(u) &= (2u^3 - 3u^2 + 1)\mathbf{p}(0) + (-2u^3 + 3u^2)\mathbf{p}(1) \\ &\quad + (u^3 - 2u^2 + u)\mathbf{p}^u(0) + (u^3 - u^2)\mathbf{p}^u(1) \end{aligned}$$

This equation is simplified by following substitutions:

$$F_1(u) = 2u^3 - 3u^2 + 1$$

$$F_2(u) = -2u^3 + 3u^2$$

$$F_3(u) = u^3 - 2u^2 + u$$

$$F_4(u) = u^3 - u^2$$

It is noteworthy that two curves having exactly the same shape would have different algebraic coefficients depending upon their orientation and size in space; thus user does not have a priori idea regarding the shape of the curves from above representation. In this regard geometric representation is very useful in that it offers user or designer an intuitive idea of the curve. The geometric representation for curves is given as

$$\mathbf{p} = F_1\mathbf{p}_0 + F_2\mathbf{p}_1 + F_3\mathbf{p}_0^u + F_4\mathbf{p}_1^u$$

using subscript to represent end-point value u .

Hermite Curves

Referring to equations (Y) we see that in geometric representation the cubic curves are defined by the coordinates and the tangent vectors at their end-points. Such a representation is called Hermite (cubic) curve.

Obviously in the case of a Hermite cubic curve user has a fairly good idea regards the shape of the curve as he is specifying the end-points as well as the slope at the end-points. These points and the derivatives fix-up the 12-degrees of freedom in the set of equation (Y).

Bezier Curves

A French engineer Bezier devised a parametric representation for curves which is an approximate technique for generating a curve i.e, the curve does not pass through all the control points but rather tries to approximate them where control points is a set

of points input by the designer. This set of points in a specified order defines a control polygon. The shape of the curve being governed by this control polygon. Mathematically, the curve is defined as

$$\mathbf{p}(u) = \sum_{i=0}^n B_{i,n}(u) \quad u \in [0, 1]$$

where $B_{i,n}$ is a Bernstein basis polynomial

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

$B_{i,n}$'s are said to define a Bernstein basis of degree n ; in individual curve coordinates we can write :

$$\begin{aligned} x(u) &= \sum_{k=0}^n x_k B_{k,n}(u) \\ y(u) &= \sum_{k=0}^n y_k B_{k,n}(u) \\ z(u) &= \sum_{k=0}^n z_k B_{k,n}(u) \\ &u \in [0, 1] \end{aligned}$$

The polynomial $B_{k,n}(u)$ are called *blending functions* because they blend the control points to form a composite function describing the curve. This composite function is a polynomial of degree one less than the number of control points used. Three points generate a parabola, four points a cubic curve, and so forth. Figure (2.1) demonstrates the appearance of some Bezier curves for various selections of control points in the xy plane ($z = 0$). An important property of any Bezier curves is that it lies within the *convex hull* (polygon boundary) of the control points (figure 2.2). This ensures that the curve smoothly follows the control points without erratic oscillations.

Bezier also have an added attraction of simple mathematical formulation and good intuitive appeal.

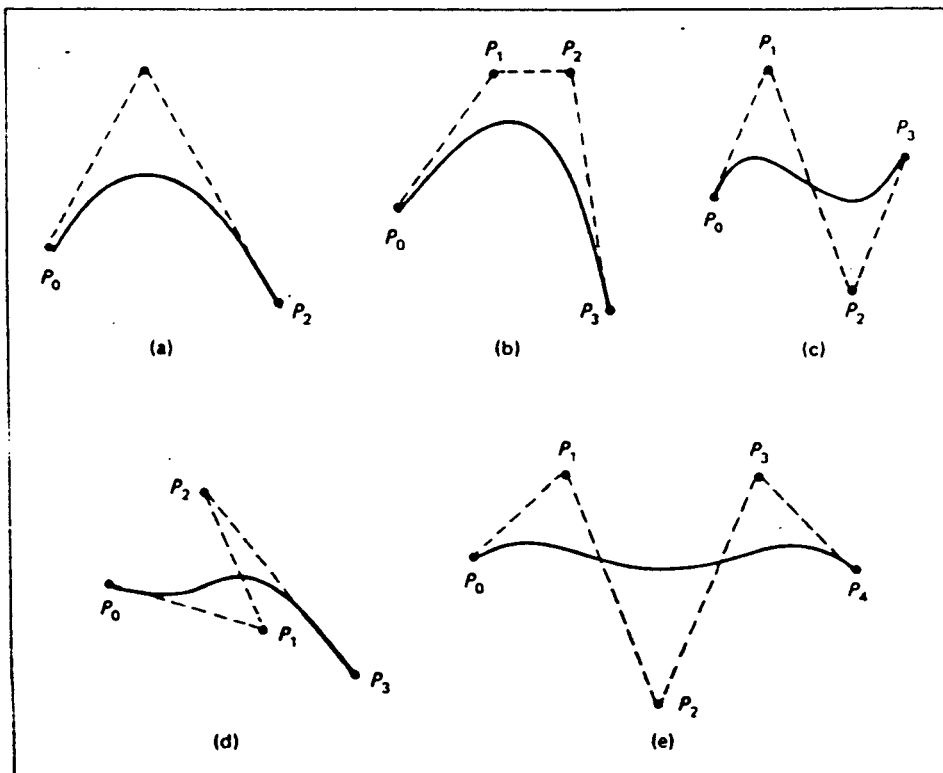


Figure 2.1 Examples of Bezier curves generated from three, four and five control points in the xy plane. Dashed lines show the straight-line connection of the control points.

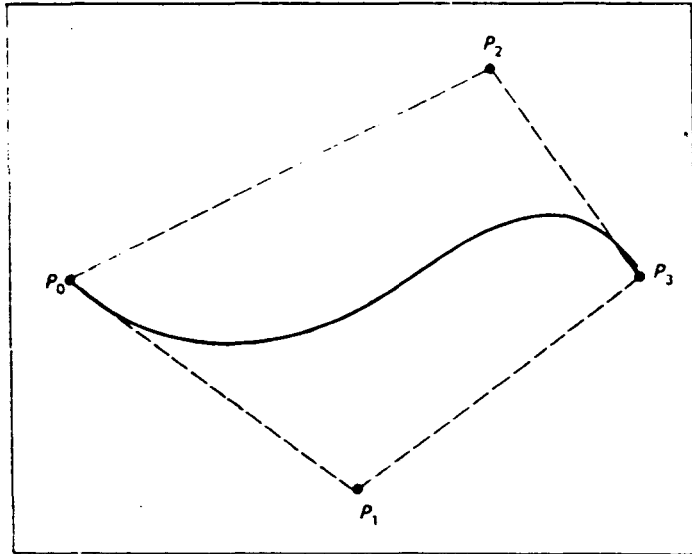


Figure 2.2 Convex hull (dashed line) of the control for a Bezier curve.

2.3 Surfaces

In direct analogy with curves we can define parametric representation for surfaces in different representations by simply introducing a new parameter v . The important thing to note is that both parameters u and v can take values in $[0,1]$; so in parameter space they define a square patch with boundary curves $v = 0$, $v = 1$, $u = 0$ and $u = 1$.

The representations for surfaces are as follows

a) Bicubic Surface

The algebraic form of a bicubic patch is given by

$$\mathbf{p}(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 \mathbf{a}_{ij} u^i v^j$$

The \mathbf{a}_{ij} vectors are called the algebraic coefficients of the surface.

b) Bezier Surface

In this representation the shape of the surface is governed by a grid of $(m + 1) \times (n + 1)$ points which is said to define a control polygon. Points on the surface are given by following equation

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

where the \mathbf{p}_{ij} are the vertices of the characteristic polygon that forms a $(m + 1) \times (n + 1)$ rectangular array of points, and $B_{i,m}$ and $B_{j,n}$ are defined as for curves. The appeal and widespread use of rectangular patches lies in the fact that they are very easy to understand and implement. The user has a good intuitive sense regards the shape of the surface but still they suffer from the handicap that they cannot interpolate to a set of datapoints. Also the shape that is generated for a given set of control points is not always close to the one that is described i.e., given a set of datapoints the user has no way of knowing whether the shape generated is the one that describes the object or not.

CHAPTER THREE

VORONOI DIAGRAMS AND DELAUNAY'S TRIANGULATION

3.1 Introduction

In this chapter we introduce the concept of Voronoi diagram and list briefly the algorithms for obtaining Voronoi diagrams for a set of points in a plane . We shall see that the Voronoi diagrams are actually the basis of Delaunay's triangulation . Finally we explain, in detail, the Delaunay's triangulation .

3.2 Voronoi diagrams

Very often in real life situations we are faced with the problem of partitioning a plane given a set S of N points in the plane such that each partition has exactly one point p_i and all the points in that partition are closer to p_i than to any other p_j , where $i \neq j$. These N regions partition the plane into a convex net which is called a *Voronoi diagram*, denoted as $\text{Vor}(s)$, see Figure 3.1. The vertices of the diagram are *Voronoi vertices*, and its line segments are *Voronoi edges*.

We define each polygonal region as a *Voronoi polygon* $V(i)$. Thus if $(x, y) \in V(i)$ then p_i is the nearest neighbor of (x, y) . The Voronoi diagram contains, in a powerful sense, all of the proximity information defined by the given set.

We state the properties of Voronoi diagrams with a special description of triangulation property which we shall see in Delaunay's triangulation .

An Assumption. *No four points of the original set are cocircular.*

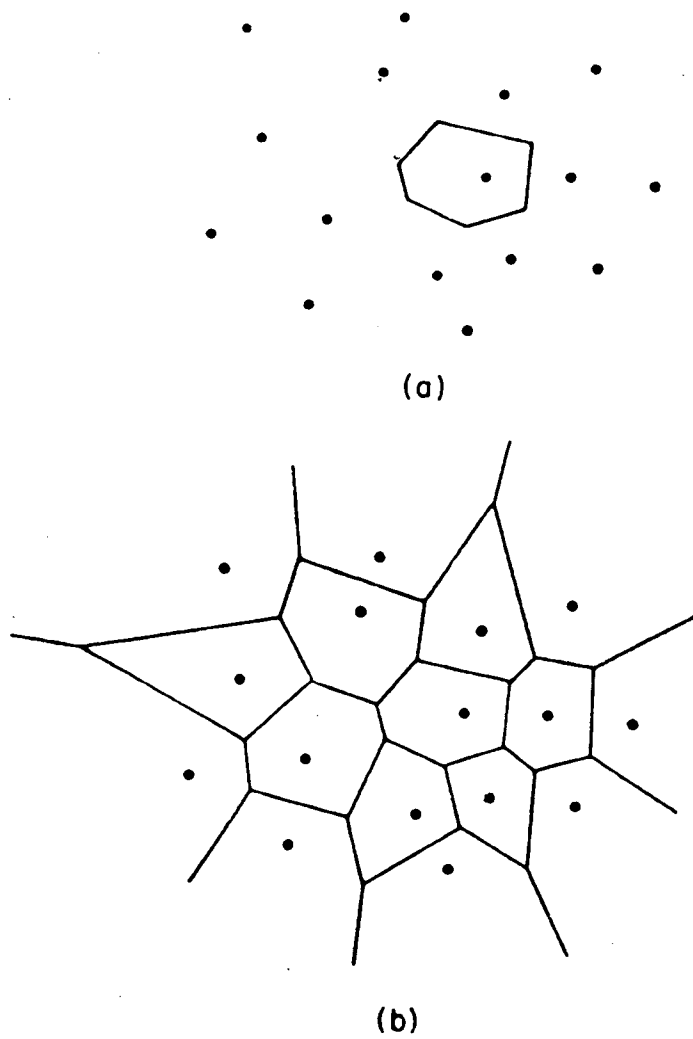


Figure 3.1

(a) Datapoints and Voronoi polygon.

(b) Voronoi diagram for the given datapoints.

Theorem 1 *Every vertex of the Voronoi diagram is the common intersection of exactly three edges of the diagram.*

Theorem 2 *For every vertex v of the Voronoi diagram of S , the circle $C(v)$ contains no other point of S*

Theorem 3 *Every nearest neighbor p_i defines an edge of the Voronoi polygon $V(i)$.*

Theorem 4 *Polygon $V(i)$ is unbounded if and only if p_i is a point on the boundary of the convex hull of the set S .*

Theorem 5 *The straight-line dual of the Voronoi diagram is a triangulation of S .*

The Voronoi diagrams are also used in many fields apart from triangulation . In archaeology, Voronoi polygons are used to map the spread of the use of tools in ancient cultures and for studying the influence of rival centers of commerce [Hodder-Orton(1976)]. In ecology, the survival of an organism depends on the number of neighbors it must compete for food and light, and the Voronoi diagram of forest species and territorial animals is used to investigate the effect of overcrowding [Pielou (1977)]. In physics, Voronoi diagrams are used in the study of disordered systems. The structure of a molecule is determined by the combined influence of electrical and short-range forces, which have been probed by constructing Voronoi diagrams.

3.3 Algorithms for obtaining Voronoi Diagrams

Algorithm 1.0

Let $\{p_1, \dots, p_j\}$ be a set S of points then this algorithm proceeds as follows

Step 1. Partition S into two subsets S_1 and S_2 of approximately equal size.

Step 2. Construct $\text{Vor}(S_1)$ and $\text{Vor}(S_2)$ recursively.

Step 3. "Merge" $\text{Vor}(S_1)$ and $\text{Vor}(S_2)$ to obtain $\text{Vor}(s)$.

A refinement of this algorithm is

Algorithm 1.1

Step 1. Partition S into two subsets S_1 and S_2 of approximately equal size.

Step 2. Construct $\text{Vor}(S_1)$ and $\text{Vor}(S_2)$ recursively.

Step 3. Construct the polygonal chain σ , separating S_1 and S_2 .

Step 4. Discard all edges of $\text{Vor}(S_2)$ that lie to the left of σ and all edges of $\text{Vor}(S_1)$ that lie to the right of σ . The result is $\text{Vor}(S)$, the Voronoi diagram of the entire set.

Where a chain $\sigma = (u_1, \dots, u_p)$ is a planer straightline graph with vertex set $\{u_1, \dots, u_p\}$ and edge set $\{(u_i, u_{i+1}) | i = 1, \dots, p-1\}$

In other words a chain is the planer embedding of a graph-theoretic chain and is also called polygonal line. See [Preparta] for further details.

The above algorithm requires that all points be known and sorted before the user starts constructing the Voronoi diagram.

The order is $(n \log n) \times n$.

Below we present an alternate algorithm for obtaining Voronoi diagrams. This algorithm has been devised by us. It has the advantage that one need not sort the points. Also one can introduce new points at any stage of computation i.e., all the points need not be known *a priori*.

Our Algorithm

- Step 1* Pick one point from the set S and initialize the whole plane as V_1 .
- Step 2* From the points not yet choosed pick one more point (i th) if no more points goto 5.
- Step 3* Rearrange the Voronoi diagram to obtain $V(i + 1)$ from V_i .
- Step 4* Go to Step 2.
- Step 5* Draw all edges and stop.

3.4 Triangulation

Given a set of data points in a plane the process of triangulation consists in dividing the plane into a set of triangles where vertex of each triangle is a data point; and each data point is itself shared by atmost three triangles.

Since, ordinarily the data points are not in a plane we first project each of these data points onto the xy plane and then carry out the triangulation. Most methods for triangulation place a restriction that the input points be in an increasing order of x -coordinate ; this reduces the amount of computation.

We have obtained Delaunay's triangulation using Watson's Algorithm. The Watson's algorithm is as follows:

Watson's Algorithm

1. Sort the N points to be triangulated in ascending sequence of their x -coordinate.
2. Define the vertices of the supertriangle in anticlockwise order. It is convenient to number these vertices as $N + 1$, $N + 2$, and $N + 3$. Set the coordinates of these vertices so that all of the points to be triangulated lie within the supertriangle. Add the

supertriangle to a list of triangles formed and flag it as incomplete.

3. Introduce a new point from the list of sorted points with coordinates (x_{new}, y_{new}) .
4. Examine the list of all triangles formed so far. For each triangle which is flagged as incomplete, do steps 5 to 9.
5. Compute the coordinates of the triangle circumcenter, (x_c, y_c) , and the square of its circumcircle R^2 .
6. Compute the square of the x -distance from the new point to the triangle circumcenter, i.e., the quantity

$$D_x^2 = (x - x_{new})^2$$

7. if $D_x \geq R^2$, then the circumcircle for this triangle cannot be intersected by any of the remaining points. Flag this triangle as complete and do not execute steps 8 and 9.
8. Compute the square of the distance from the new point to the triangle circumcenter, i.e., the quantity

$$D^2 = D_x^2 + (y_c - y_{new})^2$$

9. if $D^2 < R^2$, then the new point intersects the circumcircle of this triangle. Delete this triangle from the list of triangles formed and store the three pair of vertices which define its edges on a list of edges. If $D^2 > R^2$, then the new point lies on or outside the circumcircle for this triangle and the triangle remains unmodified.
10. Loop over the list of edges and delete all edges which are interior to the polygon formed by the intersected triangles. Since the vertices defining the edges of each triangle are always recorded in an anticlockwise sequence, this step may be executed efficiently by searching for ordered pairs.
11. Form the new triangles by matching the new point with each pair of vertices in the list of edges. The new point forms new triangles

with each pair of vertices on the boundary of the polygon formed by the intersected triangles. Define each new triangle such that its vertices are always listed in an anticlockwise sequence and flag it as incomplete.

12. Repeat steps 3 to 11 until the list of points to be triangulated is exhausted.
13. Form the final triangulation by removing all triangles which contain one or more of the supertriangle vertices. This may be achieved by scanning the list of triangles and deleting any of those which have vertex numbers which are greater than N .

We have implemented Watson's algorithm for a set of 7 data points on a sphere, and the triangulation obtained is shown in figures 3.2-5.

In chapter 4 we shall see that the triangular patches can be obtained by mapping the triangular patches in xy plane to the 3-D.

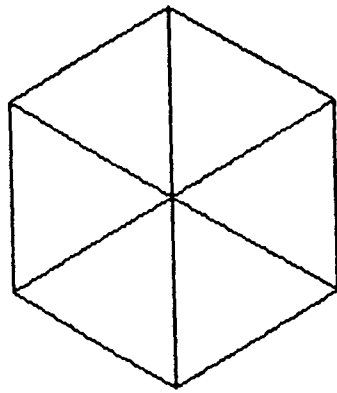


Figure 3.2 Triangulation obtained for 7 datapoints on a sphere.

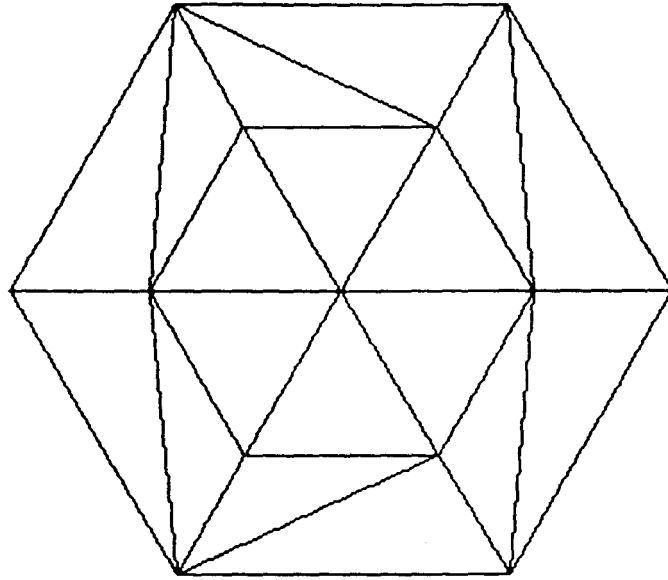


Figure 3.3 Triangulation obtained for 13 datapoints on a sphere.

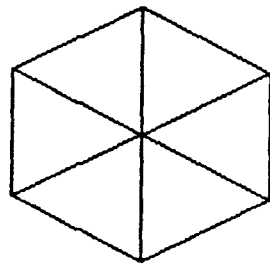


Figure 3.4 Triangulation obtained for n datapoints on a ellipsoid.

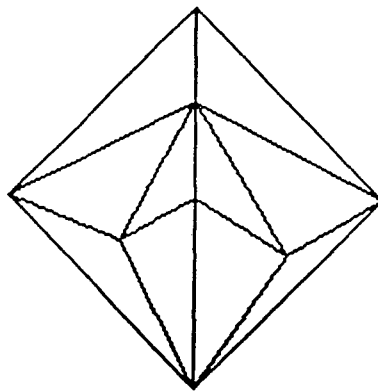


Figure 3.5 Triangulation obtained for scattered datapoints on a sphere.

CHAPTER FOUR

THE TRIANGULAR SURFACE PATCHES

4.1 Introduction

In Chapter two we had briefly introduced the concept of triangular surface patches. In this chapter we discuss them in detail with particular emphasis on Bezier-Bernstein patches.

4.2 Barycentric Coordinates

We here introduce the concept of barycentric coordinates as they form the basis of triangulation technique.

Let T be a triangle with vertices p_1, p_2 and p_3 ; $p_i \in \mathcal{R}^2$. Let p be any point in the same plane as p_1, p_2 and p_3 . It has a unique representation

$$p = u_1 p_1 + u_2 p_2 + u_3 p_3 \quad \text{with } u_1 + u_2 + u_3 = 1$$

This can be seen easily if we consider vectors $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ between the vertices and an arbitrary origin. Then $(\mathbf{p}_1 - \mathbf{p}_2)$ and $(\mathbf{p}_1 - \mathbf{p}_3)$ (see figure 4.1) form an axis for the plane containing p_i 's. Now any position-vector in this plane can be represented as

$$\begin{aligned} \mathbf{p} &= u_1(\mathbf{p}_1 - \mathbf{p}_2) + u_3(\mathbf{p}_3 - \mathbf{p}_2) + \mathbf{p}_2 \\ \Rightarrow \mathbf{p} &= u_1 \mathbf{p}_1 + u_2 \mathbf{p}_2 + u_3 \mathbf{p}_3 \\ &\quad \text{because } u_2 = 1 - u_1 - u_3 \end{aligned}$$

We say that p has barycentric coordinates \mathbf{u} with respect to $T(p_1, p_2, p_3)$. The barycentric coordinates provide a very suitable coordinate system for arbitrary triangles in the plane since they are invariant under affine transformations.

In our discussion we always carry out the triangulation of points after projection in $(x-y)$ plane. Thus any function of the form

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

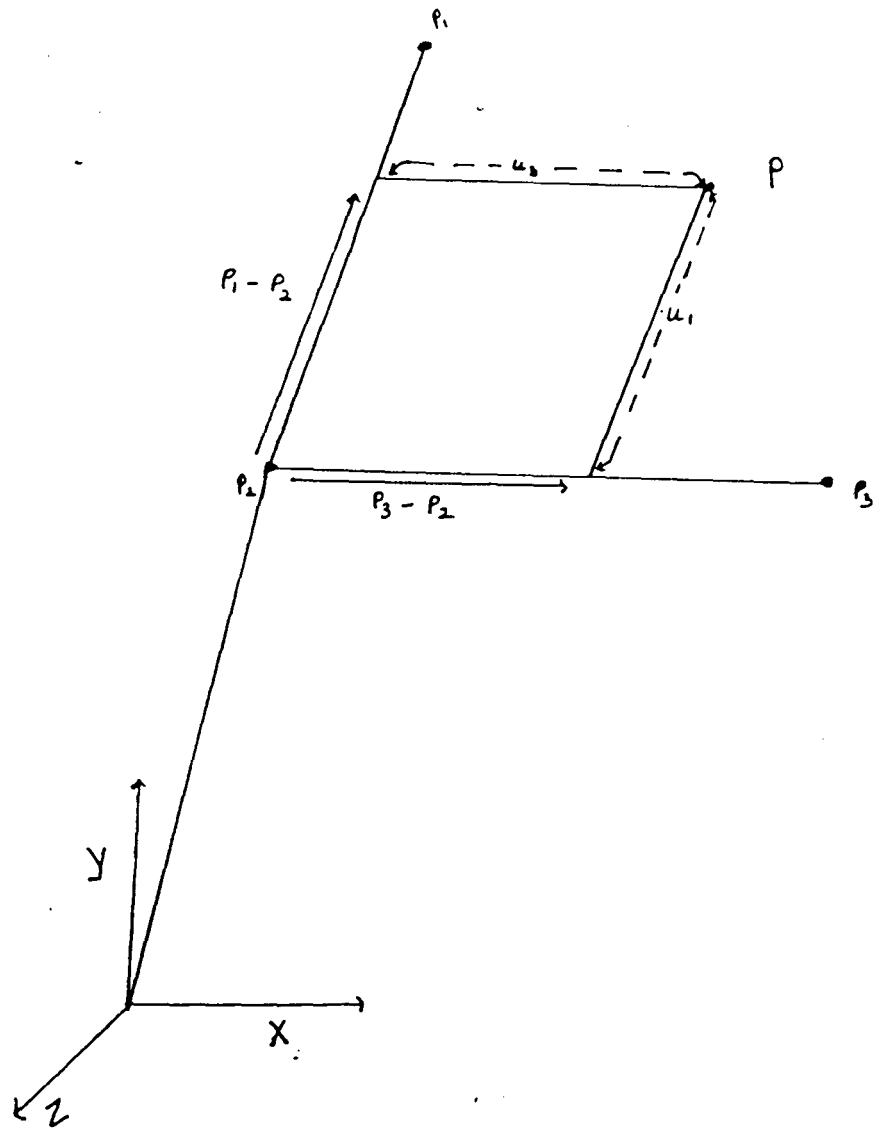


Figure 4.1 Barycentric coordinates, where p_1, p_2, p_3 are centers of the defining triangle. u_1 and u_2 are barycentric parameters. sphere.

can be put down as

$$z = f(\mathbf{u}) := f(u_1, u_2, u_3)$$

where u_1, u_2, u_3 are barycentric coordinates in (x, y) plane.

4.3 Bernstein Polynomials

In terms of barycentric coordinates of a fixed yet arbitrary triangle T the bivariate Bernstein polynomials are given as:

$$B_{\mathbf{i}}^n(u) = \frac{n!}{l_1! l_2! l_3!} u_1^{l_1} u_2^{l_2} u_3^{l_3} \quad (X)$$

$$\begin{aligned} \text{where} \quad & l_1 + l_2 + l_3 = n \\ & u_1 + u_2 + u_3 = 1 \\ & \text{and } l_1, l_2, l_3 \in \mathcal{Z}^+ \end{aligned}$$

Since $B_{\mathbf{i}}^n(u)$ are terms of the binomial expansion of $(u_1 + u_2 + u_3)^n$, and $u_1 + u_2 + u_3 = 1$ we see that

$$\sum_{|\mathbf{i}|=n} B_{\mathbf{i}}^n(u) = \sum_{\substack{i_1, i_2, i_3 \\ i_1 + i_2 + i_3 = n}} B_{i_1, i_2, i_3}^n(\mathbf{u}) = 1$$

Also

$$B_{\mathbf{i}}^n(\mathbf{u}) \geq 0 \quad \text{if } u_i \geq 0 ; i = 1, 2, 3$$

Equation (X) represents a bivariate polynomial of degree n or less. Thus $B_{\mathbf{i}}^n(\mathbf{u})$ form a basis for the space of polynomials of degree less than or equal to n . Infact any polynomial, of degree less than n can be written as

$$p(u) = \sum_{|\mathbf{i}|=n} b_{\mathbf{i}} B_{\mathbf{i}}^n(u)$$

This form is the bivariate equivalent to Bernstein-Bezier curves, and the coefficients $B_{\mathbf{i}}$ possess a geometric interpretation analogous to the univariate case: the points $(\mathbf{i}/n, b_{\mathbf{i}})$ form a piecewise triangular polyhedron—the so-called control polyhedron—which models the shape of the surface patch $(u, p(u))$ for $u_i \geq 0$

models the shape of the surface patch $(u, p(u))$ for $u_i \geq 0$ (see figure 4.1(a)).

For the boundary curve $\mathbf{u} = 0$, we obtain

$$p(u_0) = \sum_{|i|=n} b_i B_i^n(u_0)$$

and analogous expressions for the other boundaries. Note that the above equation describes a relation between univariate polynomials!

4.4 A modified nine parameter cubic

Suppose we wish to interpolate to a function $f(u)$ that is defined over T and suppose that we are given function value and gradient at each vertex. Thus we have nine constraints altogether, and we know that a cubic polynomial has ten coefficients.

The cubic boundary curves of the interpolant Cf are uniquely defined by the given data; their computation is a standard univariate problem. We express the three cubic boundary curves in terms of univariate Bernstein polynomial. Since the control polygons of the boundary curves of Cf constitute the boundaries of the control polyhedron of Cf , we can determine nine coefficients by purely univariate methods, as given under.

In our case we determine b_{ijk} as follows: we are given the endpoints and the derivatives $\frac{dz}{dx}$, $\frac{dz}{dy}$ at three points

Also along a curve (say $u_3 = 0$)

$$\frac{dx}{du_1} = x_2 - x_1 = -\frac{dx}{du_2} \quad (A)$$

as $u_1 = 1 - u_2$

Similarly

$$\frac{dy}{du_1} = y_2 - y_1 = \frac{dy}{du_2} \quad (B)$$

and

$$\frac{dz}{du_1} = \frac{dz}{dx} \frac{dx}{du_1} + \frac{dz}{dy} \frac{dy}{du_1}$$

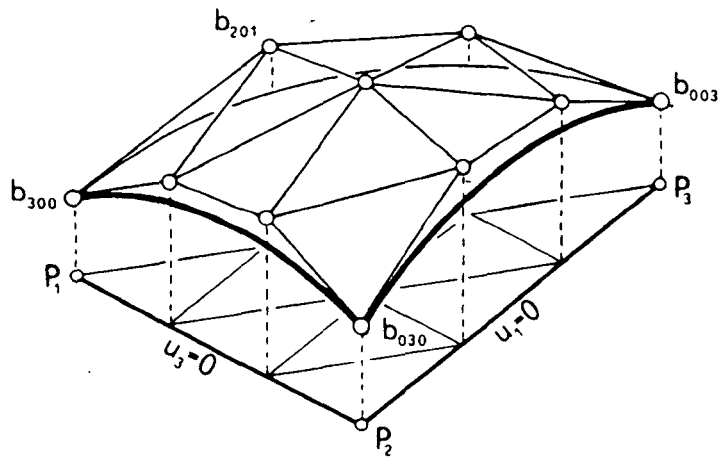


Figure 4.1(a) P_1, P_2, P_3 are the vertices of the triangle and b_{ijk} 's are the vertices of the control polyhedron.

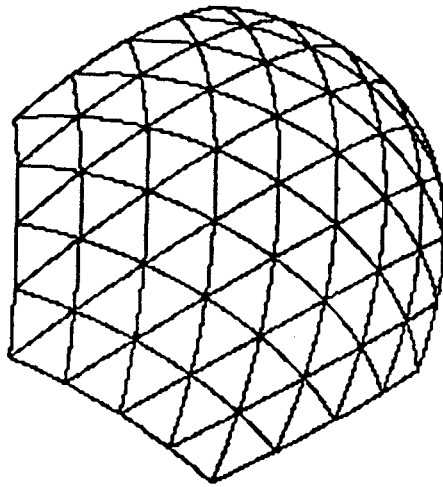


Figure 4.2 Triangular patches for 7 datapoints on a sphere.

from (A) and (B). Since we are given $\frac{dz}{dx}$, $\frac{dz}{dy}$ at end-points we can evaluate (C) at these points.

Let us briefly outline how we can obtain b_{ijk} from this information. In Bezier-Bernstein representation points in surface are given as

$$\begin{aligned} p(\mathbf{u}) &= \sum_{|i|=1} b_i B_i(u_0) \\ &= b_{300}u_1^3 + b_{030}u_2^3 + b_{003}u_3^3 + \frac{3}{2}b_{210}u_1^2u_2 + \frac{3}{2}b_{201}u_1^2u_3 + \frac{3}{2}b_{120}u_1u_2^2 \\ &\quad + \frac{3}{2}b_{021}u_2^2u_3 + \frac{3}{2}b_{102}u_1^2u_3 + \frac{3}{2}b_{012}u_2u_3^2 + \frac{3}{2}b_{111}u_1u_2u_3 \end{aligned}$$

$$\text{At } u_1 = u_2 = 0 \quad p(\mathbf{u}) = \mathbf{p}_1$$

So

$$\begin{aligned} p_1 &= b_{300} \\ \text{and similarly } p_2 &= b_{030} \quad p_3 = b_{003} \end{aligned} \quad (D)$$

Along a particular curve, say $u_3 = 0$

$$u_1 = 1 - u_2$$

Then

$$\begin{aligned} \frac{dp(u)}{du_1} &= 3b_{300}u_1^2 - 3b_{030}(1 - u_1)^2 + 3b_{210}u_1(1 - u_1) - \frac{3}{2}b_{210}u_1^2 \\ &\quad - 3b_{120}u_1(1 - u_1) + \frac{3}{2}(1 - u_1)^2 \end{aligned} \quad (E)$$

Putting $u_1 = 1$ we get

$$\frac{dp(u)}{du_1} = 3b_{300} - \frac{3}{2}b_{210} \quad (F)$$

In eq.(F) the terms $\frac{dp(u)}{du_1}$ and b_{300} are known. Thus from equations (A), (B), (C) and (D) we can determine b_{210}

A similar derivation works for b_{120} , b_{021} , b_{012} , b_{102} , b_{201} . Thus in equation (X) for Bezier-Bernstein patch only one variable is unknown.

It turns out that b_{111} can be given any arbitrary value; it does not affect the interpolation properties of Cf at all. The "standard" value for b_{111} is [Barnhill 1977, Herron 1979]

$$b_{111} = 1/3(b_{300} + b_{030} + b_{003})$$

This choice of b_{111} ensures that Cf has linear precision, i.e., if f is linear, so is Cf .

It is possible, however, to improve the interpolant such that it will have quadratic precision as well. The above equation describes one of many possible relationships between coefficients of a cubic that is actually linear. A similar relation holds for the coefficients of a cubic that is actually a quadratic

$$b_{111} = \begin{array}{l} 1/4 (b_{201} + b_{102} + b_{021} + b_{012} + b_{210} + b_{120}) \\ -1/6 (b_{300} + b_{030} + b_{003}) \end{array}$$

We are now in a position to interpolate surfaces using this method. We are giving here (figure 4.3) a interpolation for datapoints on a sphere, it is seen that the interpolation is fairly accurate as long as the datapoints are not two-sided in space (infact in such cases inversion is possible) the patches obtained are very smooth. The fact that three parameter u_1, u_2, u_3 are involved means the process of computing points can be parallized which is of great utility in real-time modelling and interpretation of data.

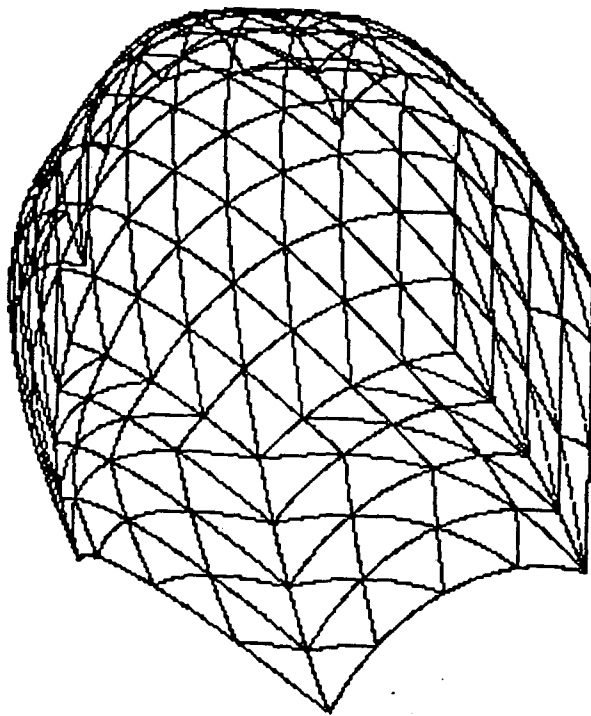


Figure 4.3 Triangular patches for 13 datapoints on a sphere.

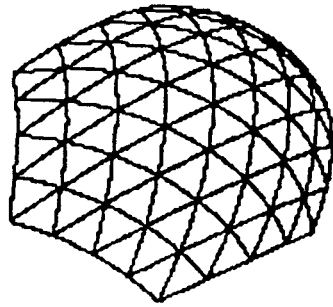


Figure 4.4 Triangular patches for datapoints on a ellipsoid.

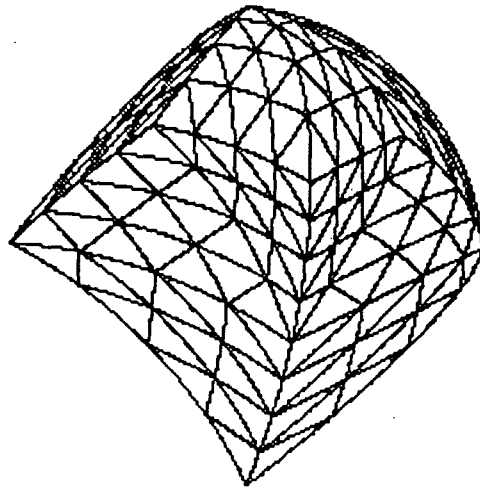


Figure 4.5 Triangular patches for scattered datapoints on a sphere.

CONCLUSION

We see from our results that the triangular patches form an elegant and efficient method for interpolation of a surface through set of datapoints. However triangular patches suffer from the limitation that through scattered datapoints the desired surface is not obtained, as is obvious from the last figure in the previous chapter. We also saw that Voronoi diagrams form an important basis for triangulation techniques. We have partially implemented our algorithm for obtaining Voronoi diagrams. Finally as a continuation of my work I am trying to implement triangular patches using representations other than Bezier-Bernstein which is the one which I have used for my project.

References

1. *Mathematical Elements of Computer Graphics*: Roger and Adams.
2. *Computer Graphics*: Hearn and Baker.
3. *Geometric Modelling*: Micheal E. Mortenson.
4. *Computational Geometry*: F.P.Preparata.
5. *Computing the n -dimensional Delaunay's triangulation with application to Voronoi polytopes*, D.F. Watson, *The Computer Journal* 1981,24(2), 167.
6. *Smooth interpolation to Scattered 3D data*, G. Farin, in *Surfaces in Solids*, R.E. Barnhill and W. Boehm (eds.) 1983.
7. *A survey of curve and surface methods in CAGD*, W. Bohm, G. Farin and J. Kahmann, in *Computer Aided Geometric Design* 1981, 1, 60



TH-5603

Figure Captions

Figure 2.1 Examples of Bezier curves generated from three, four and five control points in the xy plane. Dashed lines show the straight-line connection of the control points.

Figure 2.2 Convex hull (dashed line) of the control for a Bezier curve.

Figure 3.1

(a) Datapoints and Voronoi polygon.

(b) Voronoi diagram for the given datapoints.

Figure 3.2 Triangulation obtained for 7 datapoints on a sphere.

Figure 3.3 Triangulation obtained for 13 datapoints on a sphere.

Figure 3.4 Triangulation obtained for 7 datapoints on an ellipsoid.

Figure 3.5 Triangulation obtained for scattered datapoints on a sphere.

Figure 4.1 Barycentric coordinates, where $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ are centers of the defining triangle. u_1 and u_3 are barycentric parameters. sphere.

Figure 4.1(a) P_1, P_2, P_3 are the vertices of the triangle and b_{ijk} 's are the vertices of the control polyhedron.

Figure 4.2 Triangular patches for 7 datapoints on a sphere.

Figure 4.3 Triangular patches for 13 datapoints on a sphere.

Figure 4.4 Triangular patches for 7 datapoints on an ellipsoid.

Figure 4.5 Triangular patches for scattered datapoints on a sphere.

```

#include<iostream.h>
#include<graphics.h>
#include<math.h>
#include<conio.h>
#include<alloc.h>
#define numpts 13
#define maxtri 8

double X[numpts+3]={125,162.5,162.5,200,237.5,237.5,275};
double Y[numpts+3]={200,265,135,200,265,135,200}; // coord of points to be triangulated
double Z[numpts+3]={130,130,130,150,130,130,130};
double Dz[numpts]={0.577,0.2887,0.2887,0,-0.2887,-0.2887,-0.577};
double Dzy[numpts]={0,-0.5,0,0.5,0,-0.5,0.5,0};

struct point{
    double X,Y,Z;
};

double max(double z[]) //find maximum of z[]
{
    double maximum;
    maximum=z[0];

    for(int i=1;i<numpts;i++){
        if(maximum < z[i]){ maximum=z[i];}
    }
    return maximum;
}

double min(double z[])
{
    double minimum;
    minimum=z[0];

    for(int i=1;i<numpts;i++){
        if(minimum > z[i]){ minimum=z[i];}
    }
    return minimum;
}

struct link{
    int V1,V2,V3;
    link *next;
};

class linklist{
public:
    link *first;

    linklist()
    {first=NULL;}

    void additem(int n1,int n2,int n3)
    {
        link *newlink=new link;
        newlink->V1=n1;newlink->V2=n2;newlink->V3=n3;
        newlink->next = first;
        first = newlink;
    }

    void deleteitem(int a,int b,int c)
    {
        link *previous,*current=first;
        previous=NULL;
        while(current!=NULL){
            if((current->V1==a)&&(current->V2==b)&&(current->V3==c))
                {
                    if(previous==NULL){
                        free(first);
                        first=current;
                    }
                    else
                        {
                            current=first->next;
                            free(current);
                        }
                }
            previous->next=current->next;
            current=current->next;
        }
    }
};

```



```

current=previous->next;
}
else
{
previous=current;
current=current->next;
}
}
}

void remsuper()
{
link *prev,*p2=first;
prev=NULL;
while(p2!=NULL){ //DELETE ALL TRI'S HAVING SUPERTRI
VERTEX
if((p2->V1>=numpts)||!(p2->V2>=numpts)||!(p2->V3>=numpts))
{
if(prev==NULL){
p2=first->next;
free(first);
first=p2;
}
else {
free(p2);
p2=prev;
}
}
else {
prev=p2;
p2=p2->next;
}
}
}

void draw(){
link *current=first;
double x1,x2,x3,y1,y2,y3;
while(current!=NULL)
{
x1=X[current->V1];x2=X[current->V2];
x3=X[current->V3];y1=Y[current->V1];
y2=Y[current->V2];y3=Y[current->V3];

line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x1,y1);

current=current->next;
}
}

void display(){
link *current=first;
cout<<"\n list:";
while(current!=NULL)
{
cout<< current->V1 <<" " << current->V2 <<" " << current->V3<<";
";
current=current->next;
}
}

};

struct edge{
int vertex1,vertex2;
edge *next;
};

```

```

class edgelist{
public:edge *first;

edgelist(){ first = NULL; }

void addedge(int a,int b){
    edge *newedge=new edge;
    newedge->vertex1=a;
    newedge->vertex2=b;
    newedge->next=first;
    first=newedge;
}

/*
void display(){
    edge *current=first;
    cout<<"\n EDGELIST:";
    while(current!=NULL){
        cout<<current->vertex1<<current->vertex2<<" ";
        current=current->next;
    }
}*/

void remove(){
    int c,d,j;
    edge *prev_to_rec=NULL,*recorder=first;
    while(recorder!=NULL)
    {
        j=0;c=recorder->vertex1;d=recorder->vertex2;
        edge *previous=NULL,*current=recorder;
        while(current!=NULL)
        {
            current=current->next;
            if(((c==current->vertex1)&&(d==current->vertex2))||
((d==current->vertex1)&&(c==current->vertex2)))
            {
                j=1;
                if(previous==NULL)
                {recorder->next=current->next;
                 free(current);
                 current=recorder;
                }
                else {
                    previous->next=current->next;
                    free(current);
                    current=previous;
                }
            }
            else {
                previous=current;
            }
        }
        if(j==1)
        {
            if(prev_to_rec==NULL)
            {
                recorder=first->next;
                free(first);
                first=recorder;
            }
            else {
                prev_to_rec->next=recorder->next;
                free(recorder);
                recorder=prev_to_rec->next;
            }
        }
        else {
            //when l=0,there is no duplicate
            prev_to_rec=recorder;
            recorder=recorder->next;
        }
    }
}

void clear()
{
    edge *p1;
    while(first!=NULL)
    {

```

```

        p1=first->next;
        free(first);
        first=p1;
    }
};

int fact(int n)
{
    int x;
    if(n>=0)
    {
        if(n==0)
        {
            return 1;
        }
        else
        {
            x=n*fact(n-1);
            return x;
        }
    }
    else
    {
        cout<<"\n error: -ve int;";
        return 0;
    }
}

double mypow(double u,int i)
{
    if( ( u==0 )&&( i==0 ) )
        { return 1.0; }
    else { return pow(u,i); }
}

double B(int i1,int i2,int i3,double u1,double u2,double u3)
{
    double x;
    x= 6*( mypow(u1,i1)*mypow(u2,i2)*mypow(u3,i3) )/( fact(i1)*fact(i2)*fact(i3) );
    return x;
}

void main()
{
    int N1,N2,N3,data_,v1_,v2_,v3_;
    N1=numpts;
    N2=numpts+1;
    N3=numpts+2;

    double ymax,ymin,xmax,xmin,xcen,ycen,Dmax,xnew,ynew,x1,x2,x3,y1,y2,y3,x21,x31,y21,y31,DET;
    double R21,R31,Xcentr,Ycentr,Radiusq,Distsq,radius;
    ymax=max(Y); //find the maxmium y-coord value from all the given points
    ymin=min(Y);

    xmin=X[0];
    xmax=X[numpts-1];

    xcen=(xmax+xmin)/2;
    ycen=(ymax+ymin)/2;

    if((xmax-xmin)>(ymax-ymin)){ Dmax=(xmax-xmin)*3;} //define Dmax
    else{ Dmax=3*(ymax-ymin);}

    X[N1]=xcen-0.866*Dmax; //FIND COORD VERTICES OF SUPERTRIANGLE
    X[N2]=xcen+0.866*Dmax;
    X[N3]=xcen;
    Y[N1]=ycen-(Dmax/2);
    Y[N2]=ycen-(Dmax/2);
    Y[N3]=ycen+Dmax;

    linklist triincomplete,tricomplete; //ADD THE SUPERTRIANGLE AS THE '0'TRIANGLE IN THE
    edgelist edges;
    triincomplete.additem(N1,N2,N3); //LIST OF TRIANGLE VERTICES

```

```

triincomplete.display();
struct link *p1,p2,*present;

int gd=DETECT,gm;
initgraph(&gd,&gm,"");

for(int i=0;i<numpts;i++) //LOOP FOR ALL GIVEN POINTS
(
xnew=X[i]; //INTRODUCE A NEW POINT
ynew=Y[i];

p1=triincomplete.first; //LOOP THRU ALL TRIANGLES FORMED
while(p1!=NULL)
(
int l=1;
x1=X[p1->V1];x2=X[p1->V2];x3=X[p1->V3];
y1=Y[p1->V1];y2=Y[p1->V2];y3=Y[p1->V3];

x21=x2-x1;x31=x3-x1; //FIND THE ORTHOCENTER OF TRIANGLE
y21=y2-y1;y31=y3-y1;
DET=(x21*y31)-(x31*y21);
DET=1/(2*DET);
R21=(x21*x21)+(y21*y21);
R31=(x31*x31)+(y31*y31);
Xcentr=DET*((R21*y31)-(R31*y21)); //COORD OF ORTHOCENTER
Ycentr=DET*((R31*x21)-(R21*x31)); //XCENR & YCENTR
Radisq=(Xcentr*Xcentr)+(Ycentr*Ycentr); //SQUARE OF CIRCUMCIRCLE RADIUS
radius=sqrt(Radisq);
line(x1,y1,x2,y2);line(x2,y2,x3,y3);line(x3,y3,x1,y1);
Xcentr+=x1;
Ycentr+=y1;
circle(Xcentr,Ycentr,radius);putpixel(xnew,ynew,15);

Distsq=(Xcentr-xnew)*(Xcentr-xnew);

if(Distsq>Radisq)( //triangle is complete
v1_=p1->V1;v2_=p1->V2;
v3_=p1->V3;
triincomplete.additem(v1_,v2_,v3_);
p1=p1->next;l=0;
triincomplete.deleteitem(v1_,v2_,v3_);
)
else ( //compute the DISTANCE OF NEW POINT FROM THE ORTHOCENTER
Distsq=Distsq+(Ycentr-ynew)*(Ycentr-ynew);
if(Distsq<Radisq)( //POINT IS INSIDE THE TRIANGLE
v1_=p1->V1;v2_=p1->V2;v3_=p1->V3;p1=p1->next;l=0; //DELETE THIS TRIANGLE FROM THE
triincomplete.draw();
triincomplete.deleteitem(v1_,v2_,v3_); //LIST OF
INCOMPLETE TRIANGLES

cleardevice();
triincomplete.draw();
edges.addedge(v1_,v2_); //AND STORE THE THREE
EDGES OF TRIANGLE IN A LIST

edges.addedge(v2_,v3_);
edges.addedge(v3_,v1_);
)
)
if(l==1){p1=p1->next;}
} //ALL TRI'S HAV BEEN CHECKED
edges.remove(); //FORM TRI'S WITH EACH EDGE IN THE LIST
edge *current;
current=edges.first; //FROM THE i'th POINT
while(current!=NULL){
v1_=current->vertex1;v2_=current->vertex2;
if(i==(numpts-1)){
triincomplete.additem(i,v1_,v2_);
}
else (
triincomplete.additem(i,v1_,v2_);
)
current=current->next;
}
edges.clear();

```

```

cleardevice();
tricomplete.display();
tricomplete.draw();
}

cleardevice();
//tricomplete.display();
//tricomplete.draw();

getch();

cleardevice();

tricomplete.remsuper();
tricomplete.draw();

getch();
setviewport(0,0,640,480,1);

double u1,u2,u3,dummyX,dummyY,dummyZ,x,y;
float Xe,Ye,Ze,l,a;
cout<<"\n GIVE EYE COORD:(Xe,Ye,Ze)";cin>>Xe>>Ye>>Ze;
cout<<"\n GIVE Z-PLANE OF PROJECTION:";cin>>a;
clearviewport();

present=tricomplete.first;
while(present!=NULL) // TRAVERSE EACH TRIANGLE IN THE LIST
{
int v1,v2,v3;
v1=present->V1;v2=present->V2;
v3=present->V3;
struct point b300,b210,b120,b030,b021,b012,b003,b102,b201,b111;

b300.X=X[v1];
b300.Y=Y[v1];
b300.Z=Z[v1];

b210.X=( 2*X[v1] + X[v2] )/3;
b210.Y=( 2*Y[v1] + Y[v2] )/3;
b210.Z=( X[v2] - X[v1])*Dzx[v1] + (Y[v2] - Y[v1])*Dzy[v1] +3*Z[v1])/3;

b120.X=( X[v1] + 2*X[v2] )/3;
b120.Y=( Y[v1] + 2*Y[v2] )/3;
b120.Z=( 3*Z[v2] - (X[v2] - X[v1])*Dzx[v2] - (Y[v2] - Y[v1])*Dzy[v2])/3;

b030.X=X[v2];
b030.Y=Y[v2];
b030.Z=Z[v2];

b021.X=( 2*X[v2] + X[v3] )/3;
b021.Y=( 2*Y[v2] + Y[v3] )/3;
b021.Z=( X[v3] - X[v2])*Dzx[v2] + (Y[v3] - Y[v2])*Dzy[v2] +3*Z[v2])/3;

b012.X=( X[v2] + 2*X[v3] )/3;
b012.Y=( Y[v2] + 2*Y[v3] )/3;
b012.Z=( 3*Z[v3] - (X[v3] - X[v2])*Dzx[v3] - (Y[v3] - Y[v2])*Dzy[v3])/3;

b003.X=X[v3];
b003.Y=Y[v3];
b003.Z=Z[v3];

b102.X=( 2*X[v3] + X[v1] )/3;
b102.Y=( 2*Y[v3] + Y[v1] )/3;
b102.Z=( X[v1] - X[v3])*Dzx[v3] + (Y[v1] - Y[v3])*Dzy[v3] +3*Z[v3])/3;

b201.X=( X[v3] + 2*X[v1] )/3;
b201.Y=( Y[v3] + 2*Y[v1] )/3;
b201.Z=( 3*Z[v1] - (X[v1] - X[v3])*Dzx[v1] - (Y[v1] - Y[v3])*Dzy[v1])/3;

b111.X=( X[v1] +X[v2] + X[v3] )/3;
b111.Y=( Y[v1] +Y[v2] + Y[v3] )/3;
b111.Z=( Z[v1] +Z[v2] + Z[v3] )/3;

b111.X=( b300.X + b030.X + b003.X )/3;
b111.Y=( b300.Y + b030.Y + b003.Y )/3;
b111.Z=( b300.Z + b030.Z + b003.Z )/3;

```

```

b111.X=( b201.X + b102.X + b021.X + b012.X + b210.X + b120.X )/4 - ( b300.X + b030.X + b003.X )/6;
b111.Y=( b201.Y + b102.Y + b021.Y + b012.Y + b210.Y + b120.Y )/4 - ( b300.Y + b030.Y + b003.Y )/6;
b111.Z=( b201.Z + b102.Z + b021.Z + b012.Z + b210.Z + b120.Z )/4 - ( b300.Z + b030.Z + b003.Z )/6;

```

```

for(u3=0;u3<=1.01;u3+=0.25)

```

```

{
  for(u2=0;u2<=1.001-u3;u2+=.005)

```

```

{
  u1=1-u2-u3;

```

```

  dummyX= b300.X*B(3,0,0,u1,u2,u3) + b210.X*B(2,1,0,u1,u2,u3) +
b120.X*B(1,2,0,u1,u2,u3) + b030.X*B(0,3,0,u1,u2,u3) + b021.X*B(0,2,1,u1,u2,u3) +
b012.X*B(0,1,2,u1,u2,u3) + b003.X*B(0,0,3,u1,u2,u3) + b102.X*B(1,0,2,u1,u2,u3) +
b201.X*B(2,0,1,u1,u2,u3) + b111.X*B(1,1,1,u1,u2,u3);

```

```

  dummyY= b300.Y*B(3,0,0,u1,u2,u3) + b210.Y*B(2,1,0,u1,u2,u3) +
b120.Y*B(1,2,0,u1,u2,u3) + b030.Y*B(0,3,0,u1,u2,u3) + b021.Y*B(0,2,1,u1,u2,u3) +
b012.Y*B(0,1,2,u1,u2,u3) + b003.Y*B(0,0,3,u1,u2,u3) + b102.Y*B(1,0,2,u1,u2,u3) +
b201.Y*B(2,0,1,u1,u2,u3) + b111.Y*B(1,1,1,u1,u2,u3);

```

```

  dummyZ= b300.Z*B(3,0,0,u1,u2,u3) + b210.Z*B(2,1,0,u1,u2,u3) +
b120.Z*B(1,2,0,u1,u2,u3) + b030.Z*B(0,3,0,u1,u2,u3) + b021.Z*B(0,2,1,u1,u2,u3) +
b012.Z*B(0,1,2,u1,u2,u3) + b003.Z*B(0,0,3,u1,u2,u3) + b102.Z*B(1,0,2,u1,u2,u3) +
b201.Z*B(2,0,1,u1,u2,u3) + b111.Z*B(1,1,1,u1,u2,u3);

```

```

  l= ( a - Ze )/( dummyZ - Ze );
  x= Xe + ( dummyX - Xe )*l;
  y= Ye + ( dummyY - Ye )*l;
  putpixel(x,y,15);
}

```

```

}

```

```

for(u2=0;u2<=1.01;u2+=0.25)

```

```

{
  for(u1=0;u1<=1.001-u2;u1+=.005)

```

```

{
  u3=1-u2-u1;

```

```

  dummyX= b300.X*B(3,0,0,u1,u2,u3) + b210.X*B(2,1,0,u1,u2,u3) +
b120.X*B(1,2,0,u1,u2,u3) + b030.X*B(0,3,0,u1,u2,u3) + b021.X*B(0,2,1,u1,u2,u3) +
b012.X*B(0,1,2,u1,u2,u3) + b003.X*B(0,0,3,u1,u2,u3) + b102.X*B(1,0,2,u1,u2,u3) +
b201.X*B(2,0,1,u1,u2,u3) + b111.X*B(1,1,1,u1,u2,u3);

```

```

  dummyY= b300.Y*B(3,0,0,u1,u2,u3) + b210.Y*B(2,1,0,u1,u2,u3) +
b120.Y*B(1,2,0,u1,u2,u3) + b030.Y*B(0,3,0,u1,u2,u3) + b021.Y*B(0,2,1,u1,u2,u3) +
b012.Y*B(0,1,2,u1,u2,u3) + b003.Y*B(0,0,3,u1,u2,u3) + b102.Y*B(1,0,2,u1,u2,u3) +
b201.Y*B(2,0,1,u1,u2,u3) + b111.Y*B(1,1,1,u1,u2,u3);

```

```

  dummyZ= b300.Z*B(3,0,0,u1,u2,u3) + b210.Z*B(2,1,0,u1,u2,u3) +
b120.Z*B(1,2,0,u1,u2,u3) + b030.Z*B(0,3,0,u1,u2,u3) + b021.Z*B(0,2,1,u1,u2,u3) +
b012.Z*B(0,1,2,u1,u2,u3) + b003.Z*B(0,0,3,u1,u2,u3) + b102.Z*B(1,0,2,u1,u2,u3) +
b201.Z*B(2,0,1,u1,u2,u3) + b111.Z*B(1,1,1,u1,u2,u3);

```

```

  l= ( a - Ze )/( dummyZ - Ze );
  x= Xe + ( dummyX - Xe )*l;
  y= Ye + ( dummyY - Ye )*l;
  putpixel(x,y,15);
}

```

```

}

```

```

for(u1=0;u1<=1.01;u1+=0.25)
{
  for(u3=0;u3<=1.001-u1;u3+=.005)
  {
    u2=1-u1-u3;
    dummyX= b300.X*B(3,0,0,u1,u2,u3) + b210.X*B(2,1,0,u1,u2,u3) +
    b120.X*B(1,2,0,u1,u2,u3) + b030.X*B(0,3,0,u1,u2,u3) + b021.X*B(0,2,1,u1,u2,u3) +
    b012.X*B(0,1,2,u1,u2,u3) + b003.X*B(0,0,3,u1,u2,u3) + b102.X*B(1,0,2,u1,u2,u3) +
    b201.X*B(2,0,1,u1,u2,u3) + b111.X*B(1,1,1,u1,u2,u3);
    dummyY= b300.Y*B(3,0,0,u1,u2,u3) + b210.Y*B(2,1,0,u1,u2,u3) +
    b120.Y*B(1,2,0,u1,u2,u3) + b030.Y*B(0,3,0,u1,u2,u3) + b021.Y*B(0,2,1,u1,u2,u3) +
    b012.Y*B(0,1,2,u1,u2,u3) + b003.Y*B(0,0,3,u1,u2,u3) + b102.Y*B(1,0,2,u1,u2,u3) +
    b201.Y*B(2,0,1,u1,u2,u3) + b111.Y*B(1,1,1,u1,u2,u3);
    dummyZ= b300.Z*B(3,0,0,u1,u2,u3) + b210.Z*B(2,1,0,u1,u2,u3) +
    b120.Z*B(1,2,0,u1,u2,u3) + b030.Z*B(0,3,0,u1,u2,u3) + b021.Z*B(0,2,1,u1,u2,u3) +
    b012.Z*B(0,1,2,u1,u2,u3) + b003.Z*B(0,0,3,u1,u2,u3) + b102.Z*B(1,0,2,u1,u2,u3) +
    b201.Z*B(2,0,1,u1,u2,u3) + b111.Z*B(1,1,1,u1,u2,u3);
    l= ( a - Ze )/( dummyZ -Ze );
    x= Xe + ( dummyX -Xe )*l;
    y= Ye + ( dummyY -Ye )*l;
    putpixel(x,y,15);
  }
  getch();
  present=present->next;
}
getch();
}

```