Development of CAD Algorithms for Bezier Curves/Surfaces Independent of Operating System

Yogesh Kumar[1*], S.K. Srivastava[2], A.K. Bajpai[3] Neeraj Kumar[4]
[1]Mechanical Engineering, Indraprastha Institute of Technology, J.P. Nagar (U.P.) – 244 221, India
[2]Mechanical Engineering, M.M.M. Engineering College, Gorakhpur (U.P.) – 273 010, India
[3]Mechanical Engineering, M.M.M. Engineering College, Gorakhpur (U.P.) – 273 010, India
[4]Electronics & Comm. Engineering, Indraprastha Institute of Technology, J.P. Nagar (U.P.) – 244 221, India

Email id: yogesh22jan85@gmail.com.

Abstract:-Most of the CAD software, which are available currently, works only on the operating system (generally windows) for which they are designed. Alternatively, the commercial CAD software is dependent upon the operating system. If CAD software is designed such that it is Independent of the operating system, then such CAD software will be much beneficial for the present scenario of the CAD softwares and it will be independent of the operating system.

Now-a-days most of the commercial software use application programming interfaces (APIs) which provide libraries of common graphics operations that allow developers to incorporate many more realistic effects into their applications. But the CAD software is dependent on the Operating System, which is the major drawback of the software. There is a need to develop the CAD algorithms independent of operating system so that the same can be used for the development of any CAD software.

Keeping this in view, the present work is devoted to the development of CAD algorithm for the Bezier curves and Bezier surfaces. The algorithms are independent of the operating system. The operating system independent graphics library *OpenGL* has been used for the development of these CAD algorithms.

Keywords: Operating System Independent, CAD Algorithms, Bezier Curves/Surfaces etc.

1 Introduction and Literature Review

The arena of computer-aided design in mechanical engineering has shown tremendous growth with advent of latest hardware and software, the web, 3D modeling, rendering, virtual realism and rapid prototype technologies. The latest tools in CAD are changing. The process of design is also changing and guiding us how to work together. The exchange of CAD model across the various disciplines requires sharing of computer-generated information. New data management tools to manage computer-shared information or e-data (electronic-data) enable engineers to work together as team.

Most of the CAD softwares, which are available currently, work only on the platform (i.e. operating system) for which they are designed. Means the available CAD software are dependent on the platform. If CAD software is designed such that it is Independent of the platform, than such a CAD software will be much beneficial for the present scenario of the CAD softwares and it will also remove the dependency of CAD software on the operating system for which they are designed.

The field of engineering design has undergone a rapid evolution in the past three decades due to advent of the low cost digital computer. Later, computers improved the drafting process by eliminating the need to work with paper and pencil. The part can be electronically created through a representation of bit-mapped images. This is similar to drawing on paper but electronic dots (bits) are drawn (mapped) on the screen/monitor. While this facilitated storage (no need for cumbersome racks for standard sized prints/tracings/blueprints/) and copying, making changes to the drawing was just as tedious involving the dots that construct a line and reconstructing new ones. Images of the part were still drawn in 2-dimensional representations.

Now-a-days, parameterized software developed the first true 3-dimensional model. Parameterized models are not based on bit-mapped images. These are mathematically generated entities based on special computer a 3-dimensional image of the part that can be modified by changing the parameters which govern that dimension. This 3-dimensional image can quickly be viewed from any angle and laid out creating a technical drawing. Most of the commercially available CAD softwares use this technique.

In the mathematical field of numerical analysis, a Bezier Curve in parametric form is very important in Computer Graphics and related fields. These curves were first developed in 1959 by Paul de Castlejau using the Castlejau's algorithm, a numerically stable method to evaluate these Curves [13]. Bezier curves have been widely publicized in 1962 by a French Engineer Pierre Bezier. He used these curves to design automobile bodies.

In Vector graphics, Bezier Curves are an important tool which is used to model smooth curves that can be scaled indefinitely. "Paths", as they are commonly referred to in image manipulation programs such as InkScape, Adobe Illustrator, Adobe Photoshop, and GIMP are combinations of Bezier Curves patched together. Paths are not bound by the limits of rasterized images and are intuitive to modify [13].

Bezier curves have some interesting properties, unlike other classes of curves; they can fold over on themselves. They can also be joined together to form smooth, continuous shapes. Fig. 1 shows an example of a cubic Bezier Curve with a smooth curvature [13].

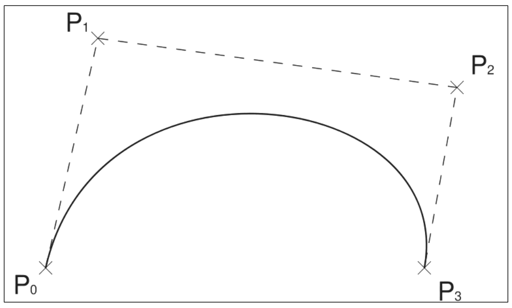


Fig. 1: Cubic Bezier Curve [16].

Before going through the technical details of how to write a program for the creation of Bezier Curve, It is necessary to describe how to construct a Bezier Curve graphically. To construct a cubic Bezier curve four control points. Depending on the alignment of these points the curve gets constructed as shown in Fig. 2 [13].

Bezier surfaces can be of any degree, but bicubic Bezier surfaces generally provide enough degrees of freedom for most applications. Similar to interpolation in many respects, a key difference is that the surface does not, in general, pass through the central control points; rather, it is "stretched" toward them as though each were an attractive force. They are visually intuitive, and for many applications, mathematically convenient [21].

The Bezier curves have the following properties:

- Basis functions are real.
- Degree of polynomial is one less than the number of points.
- Curve generally follows the shape of the defining polygon.
- First and last points on the curve are coincident with the first and last points of the polygon.
- Tangent vectors at the ends of the curve have the same directions as the respective spans.
- The curve is contained within the convex hull of the defining polygon.
- Curve is invariant under any affine transformation.

The Bezier surfaces have the following properties:

- A Bezier surface will transform in the same way as its control points under all linear transformations and translations.
- All u = constant and v = constant lines in the (u, v) space, and, in particular, all four edges of the deformed (u, v) unit square are Bezier curves.
- A Bezier surface will lie completely within the convex hull of its control points, and therefore also completely within the bounding box of its control points in any given Cartesian coordinate system.
- The points in the patch corresponding to the corners of the deformed unit square coincide with four of the control points.
- However, a Bezier surface does not generally pass through its other control points.

The Bezier curves and Bezier Surfaces have greater applications in designing an aircraft wing. To even complicate the design further, the wing has to look nice on the rest of the jet so as to promote more military funding and generate recruits into the Air Force. There are many different possible designs for a wing, some that are more optimal than others, and some that are more aesthetically pleasing those others as well. To find a balance between optimizing the air flow around the wing and how the shape looks is quite a task.

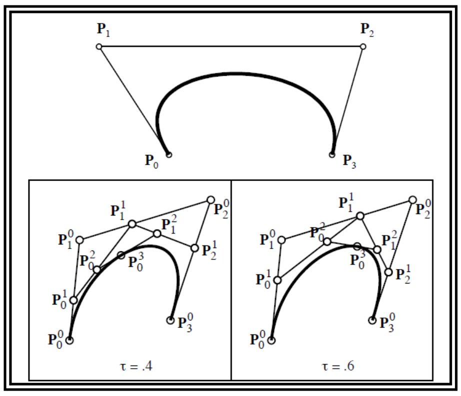


Fig. 2: Subdivision of a Cubic Bezier Curve [13].

Assume for a moment that user's job as a visualization specialist is to make use of the computer system that utilizes recorded data from an aircraft testing facility. Their system is able to relate the flow of turbulence around the wing of an aircraft to the shape of the aircrafts wing. User is asked to create small software using the model to allow an efficient way for a designer to specify an optimal and aesthetically pleasing shape for the wing. The relation between shape and turbulence is already completed, so it is user's job to give global control to the designer. Mainly, a way of specifying smooth curves on a computer screen is required, and splines are the natural way of completing the task.

In order to effectively represent a smooth curve on a computer screen, it is need to somehow approximate it. It can be realized that a computer can only draw pixels, which have a predefined width and height. If user gets really close to an LCD screen and observes the tiny squares making up the outline of an image, it's easy to understand that everything represented in computer graphics is just an approximation.

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that can be used to specify the objects and operations needed to produce interactive three-dimensional applications [14].

OpenGL is a software interface that allows the programmer to create 2D and 3D graphics images. *OpenGL* is both a standard API and the implementation of that API. Using *OpenGL* any program can be called from a program to see the same results no matter where the program is running [15].

OpenGL is independent of the hardware, operating, and windowing systems in use. The fact that it is windowing-system independent, makes it portable. *OpenGL* program must interface with the windowing system of the platform where the graphics are to be displayed. There are a number of windowing toolkits, which have been developed for use with *OpenGL*. *OpenGL* functions in a client/server environment. That is, the application program producing the graphics may run on a machine other than the one on which the graphics are displayed. The server part of *OpenGL*, which runs on the workstation where the graphics are displayed, can access whatever physical graphics device or frame buffer is available on that machine [15].

OpenGL is hardware-independent. Many different vendors have written implementations that run on different hardware. These implementations are all written to the same *OpenGL* standard and are required to pass strict conformance tests. Vendors with licenses include SGI, AT&T, DEC, Evans & Sutherland, Hitachi, IBM, Intel, Intergraph, Kendall Square Research, Kubota Pacific, Microsoft, NEC, and Raster Ops. The RS/6000 version comes with X and Motif extensions. However X is not required to run *OpenGL* since *OpenGL* also runs with other windowing systems [14].

The *OpenGL* Evaluator function allows us to use a polynomial mapping to produce vertices, normals, texture coordinates, and colors. These calculated values are then passed on to the processing pipeline as if they had been directly specified. The Evaluators functions are also the basis for the NURBS (Non-Uniform Rational BSpline) functions which allows us to define curves and surfaces. These NURBS function can be used to generate non uniform spacing of points. Any polynomial form can be converted to Bezier form by proper generation of control points. Thus NURBS function allows finer control of the space and rendering of the surface. [15]

2 Problem Formulation

The literature reveals that all commercially available CAD softwares are dependent of operating systems such as Windows, means they have been developed for the specific Operating System. There is a need to develop the CAD algorithms independent of

operating system so that the same can be used for the development of any CAD software.
Bezier Curves and Bezier Surfaces are being frequently used in CAD applications. The objective of the present work is to develop CAD algorithms for generating Bezier Curves and Bezier Surfaces independent of the operating system. The operating system independent graphics library *OpenGL* has been used for the development of these CAD algorithms.

3 CAD Algorithms Independent of Operating Systems

In the present work, CAD algorithms have been developed using with *OpenGL* using keyboard and mouse. The CAD algorithms are implemented by making use of extensive use of library functions offered by graphic package of *OpenGL*. The most important of all is the *OpenGL* Evaluators, without which it wouldn't be possible to implement these algorithms. A list of standard library functions that are used is also discussed. At first, the standard library functions are described which is followed by the user defined functions. In the present work the CAD algorithms independent of operating system have been developed for the following primitives:
(A) Bezier Curves.
(B) Bezier Surfaces.

(A) Bezier Curves

CASE 1: Generating Bezier Curves for known Control Points.

A Bezier curve is a vector-valued function of one variable;

$$C(u) = \sum_{i=0}^{n} B_{i,n}(u) P_i$$

$$0 \le u \le 1,$$

Where, u varies in some domain (0-1).
In the present case, a bezier curve has to be generated for which control points are already known using one-dimensional evaluators. It then describes the commands and equations that control evaluators.

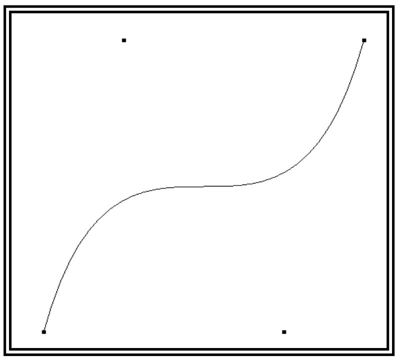


Fig. 3: Cubic Bezier Curve using four Control Points.

A cubic Bezier curve is described by four control points, which appear in this example in the *ctrlpoints[][]* array. This array is one of the arguments to *glMap1f()*. The curve is drawn in the routine *display()* between the *glBegin()* and *glEnd()* calls. Since the evaluator is enabled, the command *glEvalCoord1f()* is just like issuing a *glVertex()* command with the coordinates of a vertex on the curve corresponding to the input parameter u.

(a) The Bernstein polynomial of degree n (or order n+1) is given by,

(b) $B_{i,n}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i}$

If P_i represents a set of control points (one-, two-, three-, or even four dimensional), then the equation

$$C(u) = \sum_{i=0}^{n} B_{i,n}(u) P_i$$

(c) represents a Bezier curve as u varies from 0.0 to 1.0. To represent the same curve but allowing u to vary between u_1 and u_2 instead of 0.0 and 1.0, evaluate

(d) $C(\frac{u - u_1}{u_2 - u_1})$

The command *glMap1()* defines a one-dimensional evaluator that uses these equations.
*Void glMap1{fd}(Glenum target, TYPEu1, TYPEu2, Glint stride, Glint order, const TYPE*points);*
Defines a one-dimensional evaluator. The *target* parameter specifies what the control points represent. The points can represent vertices, RGBA color data, *normal vectors, or texture coordinates. For example, with GL_MAP1_COLOR_4,* the evaluator generates color data along a curve in four-dimensional (RGBA) color space. The *target* parameter values are used to enable each defined evaluator before user invokes it. The appropriate value are passed to *glEnable()* or *glDisable()* to enable or disable the evaluator.

The second two parameters for *glMap1*()*, u_1 and u_2, indicate the range for the variable u. The variable *stride* is the number of single- or double-precision values (as appropriate) in each block of storage. Thus, it's an offset value between the beginning of one control point and the beginning of the next.
The *order* is the degree plus one, and it should agree with the number of control points. The *points* parameter points to the first coordinate of the first control point. Using the example data structure for

glMap1(),* use the following for *points*: *(Glfloat *)(&ctlpoints[0].x).*

More than one evaluator can be evaluated at a time. If both a *GL_MAP1_VERTEX_3* and a *GL_MAP1_COLOR_4* evaluator defined and enabled, for example, then calls to *glEvalCoord1()* generate both a position and a color. Only one of the vertex evaluators can be enabled at a time, although user might have defined both of them. Similarly, only one of the texture evaluators can be active. Other than that, however, evaluators can be used to generate any combination of vertex, normal, color, and texture-coordinate data. If more than one evaluator of the same type is defined and enabled, the one of highest dimension is used.

Use *glEvalCoord1*()* to evaluate a defined and enabled one-dimensional map.
*Void glEvalCoord1{fd}(TYPE u); void glEvalCoord1{fd}v(TYPE *u);* Causes evaluation of the enabled one-dimensional maps. The argument *u* is the value (or a pointer to the value, in the vector version of the command) of the domain coordinate. For evaluated vertices, values for color, color index, normal vectors, and texture coordinates are generated by evaluation. Calls to *glEvalCoord*()* do not use the current values for color, color index, normal vectors, and texture coordinates. *glEvalCoord*()* also leaves those values unchanged.

CASE 2: Generating Bezier Curves through Mouse. The *void mouse(int button, int state, int x, int y)* function has been used in the source code for entering the control points through mouse click. The following source code has been developed for generating Bezier Curves and Entering Points through mouse.

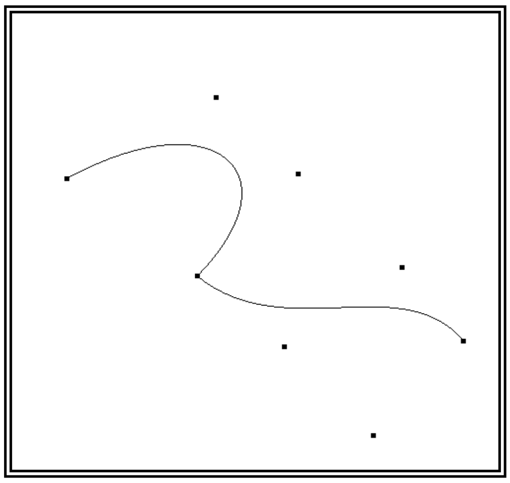


Fig. 4: Generating Bezier Curves through Mouse

(B) Bezier Surfaces

CASE 1: Generating Wireframe Bezier Surface.
A Bezier surface patch is a vector-valued function of two variables
S(u,v) = [X(u,v) Y(u,v) Z(u,v)]
Where, u and v can both vary in some domain. The range isn't necessarily three-dimensional for getting two-dimensional output for curves on a plane or texture coordinate or for getting four-dimensional output to specify RGBA information. Even one-dimensional output may make sense for gray levels. For each u (or u and v, in the case of a surface), the formula for *C()* (or *S()*) calculates a point on the curve (or surface). To use an evaluator, first define the function *C()* or *S(),* enable it, and then use the *glEvalCoord1()* or *glEvalCoord2()* command instead of *glVertex*().* This way, the curve or surface vertices can be used like any other vertices - to form points or lines, for example. In addition, other commands automatically generate series of vertices that produce a regular mesh uniformly spaced in u (or in u& v).

In two dimensions, everything is similar to the one-dimensional case, except that all the commands must take two parameters, u and v, into account. Points, colors, normals, or texture coordinates must be supplied over a surface instead of a curve. Mathematically, the definition of a Bezier surface patch is given by

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

Where, P_{ij} are a set of $m*n$ control points and the B_i are the same Bernstein polynomials for one dimension. As before, the *P*ij can represent vertices, normals, colors, or texture coordinates.

The procedure to use two-dimensional evaluators is

- Define the evaluator(s) with *glMap2*()*.
- Enable them by passing the appropriate value to *glEnable()*.
- Invoke them either by calling *glEvalCoord2()* between a *glBegin()* and *glEnd()* pair or by specifying and then applying a mesh with *glMapGrid2()* and *glEvalMesh2()*.

glMap2()* and *glEvalCoord2*()* are used to define and then invoke a two-dimensional evaluator.
void glMap2{fd}(GLenum target, TYPEu1, TYPEu2, GLint ustride, GLint uorder, TYPEv1, TYPEv2, GLint vstride, GLint vorder, TYPE points); The target parameter can have any of the values in except that the string *MAP1* is replaced with *MAP2*.

As before, these values are also used with *glEnable()* to enable the corresponding evaluator. Minimum and maximum values for both *u* and *v* are provided as u_1, u_2, v_1, and v_2. The parameters ustride and vstride indicate the number of single- or double-precision values (as appropriate) between independent settings for these values, allowing users to select a subrectangle of control points out of a much larger array. If the data appears in the form GLfloat *ctlpoints[100][100][3]*; and to use the 4x4 subset beginning at *ctlpoints[20][30]*, ustride is choosen to be 100*3 and vstride to be 3. The starting point, points, should be set to *&ctlpoints[20][30][0]*. Finally, the order parameters, uorder and vorder, can be different, allowing patches that are cubic in one direction and quadratic in the other.

*void glEvalCoord2{fd}(TYPE u, TYPE v); void glEvalCoord2{fd}***v***(TYPE *values);* Causes evaluation of the enabled two-dimensional maps. The arguments u and v are the values (or a pointer to the values *u* and *v*, in the vector version of the command) for the domain coordinates. If either of the vertex evaluators is enabled (*GL_MAP2_VERTEX_3* or *GL_MAP2_VERTEX_4*), then the normal to the surface is computed analytically. This normal is associated with the generated vertex if automatic normal generation has been enabled by passing *GL_AUTO_NORMAL* to *glEnable***()**. If it's disabled, the corresponding enabled normal map is used to produce a normal. If no such map exists, the current normal is used.

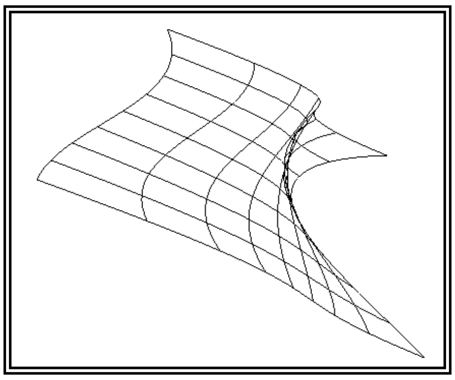


Fig. 5: Wireframe Bezier Surface.

A wireframe Bezier surface using evaluators, as shown in Fig. 5, is drawn with nine curved lines in each direction. Each curve is drawn as 30 segments.

CASE 2: Generating Meshed Bezier Surface

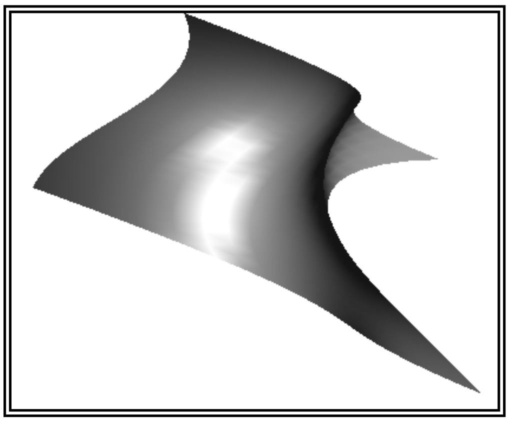


Fig. 6: Meshed Bezier Surface

Bezier Surface with mesh is shown in Fig. 6.

CASE 3: Generating Textured Bezier Surface

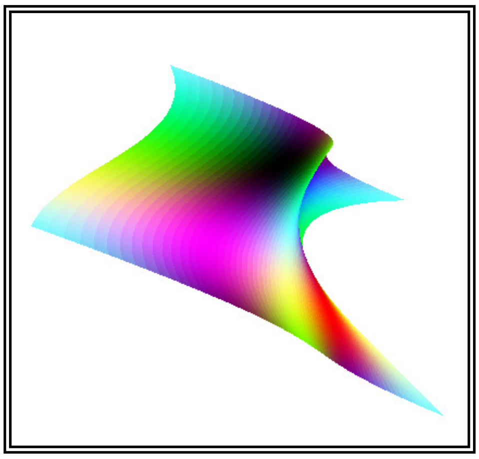


Fig. 7: Textured Bezier Surface

Bezier Surface with texture is shown in Fig.7, which enables two evaluators at the same time. The first generates three-dimensional points on the same Bezier surface as Fig. 6, and the second generates texture coordinates. In this case, the texture coordinates are the same as the *u* and *v* coordinates of the surface, but a special flat Bezier patch must be created to do this.

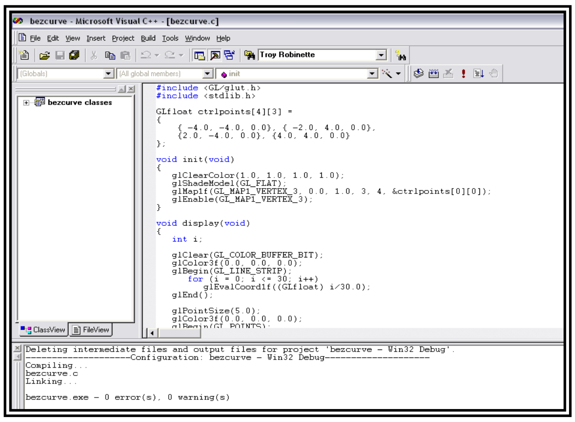
The flat patch is defined over a square with corners at (*0, 0*), (*0, 1*), (*1, 0*), and (*1, 1*); it generates (0, 0) at corner (*0, 0*), (*0, 1*) at corner (*0, 1*), and so on. Since it's of order two (linear degree plus one), evaluating this texture at the point (*u, v*) generates texture coordinates (*s, t*). It's enabled at the same time as the vertex evaluator, so both take effect when the surface is drawn. If user want the texture to repeat three times in each direction, change every 1.0 in the array *texpts[][][]* to 3.0. Since the texture wraps in this example, the surface is rendered with nine copies of the texture map.

4 Results and Discussion

The CAD algorithms designed has been tested for its working, and is found to be working properly to meet all its requirements. The algorithms have been found to be giving correct outputs to the inputs that were given, like the appropriate displaying of the output, and following appropriate conditions for termination of the program etc. All the platform independent algorithms developed for the present work have been successfully implemented using visual basic 6.0 under the Windows XP Professional Operating System. In the present work the CAD algorithms have been tested and results have been found as follows:

(A) Bezier Curves

CASE 1: *Generating Bezier Curve for known Control Points.*

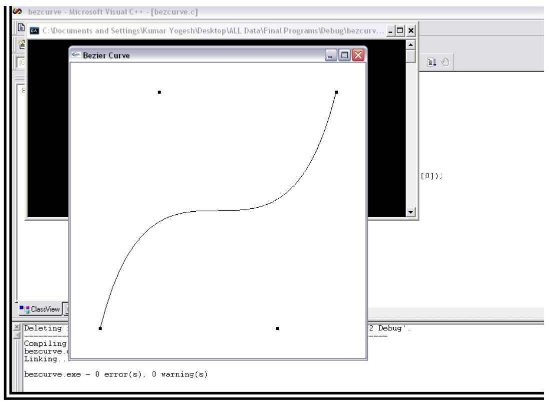

bezcurve - Microsoft Visual C++ - [bezcurve.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
(All global members)
init
bezcurve classes
ClassView
FileView

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

GLfloat ctrlpoints[4][3] =
{
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}
};

void init(void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord1f((GLfloat) i/30.0);
   glEnd();

   glPointSize(5.0);
   glColor3f(0.0, 0.0, 0.0);
   glBegin(GL_POINTS);
```

```
Deleting intermediate files and output files for project 'bezcurve - Win32 Debug'.
--------------------Configuration: bezcurve - Win32 Debug--------------------
Compiling...
bezcurve.c
Linking...

bezcurve.exe - 0 error(s), 0 warning(s)
```

Fig. 8: Compiling the Algorithm for Generating Bezier Curve

The Fig. 8 shows that the platform independent algorithm for generating bezier curve has been successfully compiled and results are shown in Fig. 9.

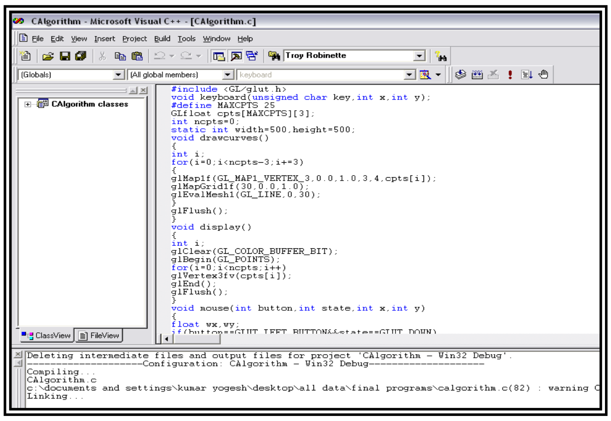

bezcurve - Microsoft Visual C++ - [bezcurve.c]
C:\Documents and Settings\Kumar Yogesh\Desktop\ALL Data\Final Programs\Debug\bezcurv...
Bezier Curve
ClassView

```
bezcurve.exe - 0 error(s), 0 warning(s)
```

Fig. 9: Generation of Bezier Curve using *OpenGL*

CASE 2: Generating Bezier Curve through Mouse

The Fig. 10 shows that the platform independent algorithm for generating bezier curve has been successfully compiled and results are shown in Fig. 11, 12 and 13.

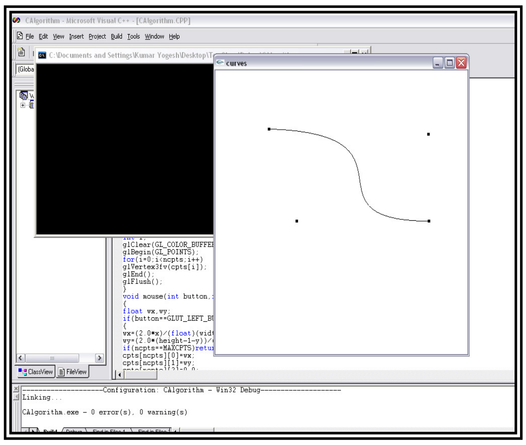

CAlgorithm - Microsoft Visual C++ - [CAlgorithm.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
(All global members)
keyboard
CAlgorithm classes
ClassView
FileView

```
#include <GL/glut.h>
void keyboard(unsigned char key,int x,int y);
#define MAXCPTS 25
GLfloat cpts[MAXCPTS][3];
int ncpts=0;
static int width=500,height=500;
void drawcurves()
{
int i;
for(i=0;i<ncpts-3;i+=3)
{
glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,4,cpts[i]);
glMapGrid1f(30,0.0,1.0);
glEvalMesh1(GL_LINE,0,30);
}
glFlush();
}
void display()
{
int i;
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
for(i=0;i<ncpts;i++)
glVertex3fv(cpts[i]);
glEnd();
glFlush();
}
void mouse(int button,int state,int x,int y)
{
float wx,wy;
if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
```

```
Deleting intermediate files and output files for project 'CAlgorithm - Win32 Debug'.
--------------------Configuration: CAlgorithm - Win32 Debug--------------------
Compiling...
CAlgorithm.c
c:\documents and settings\kumar yogesh\desktop\all data\final programs\calgorithm.c(82) : warning C
Linking...
```

Fig. 10: Compiling the Algorithm for Generating Bezier Curve.

After executing the CAlgorithm.exe, 'b' is pressed and curve as shown in the Fig. 11 the curve gets created. The curve can be erased by pressing 'e'. The screen can be made clear by pressing 'c'. The output window is quit by pressing 'q'.

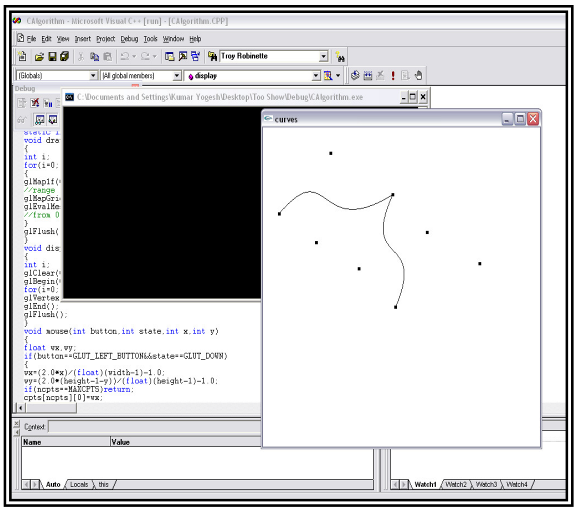


```
glClear(GL_COLOR_BUFFE
glBegin(GL_POINTS);
for(i=0;i<ncpts;i++)
glVertex3fv(cpts[i]);
glEnd();
glFlush();
}
void mouse(int button,
{
float wx,wy;
if(button==GLUT_LEFT_BU
{
wx=(2.0*x)/(float)(wid
wy=(2.0*(height-1-y))/
if(ncpts==MAXCPTS)retur
cpts[ncpts][0]=wx;
cpts[ncpts][1]=wy;
```

```
--------------------Configuration: CAlgorithm - Win32 Debug--------------------
Linking...

CAlgorithm.exe - 0 error(s), 0 warning(s)
```

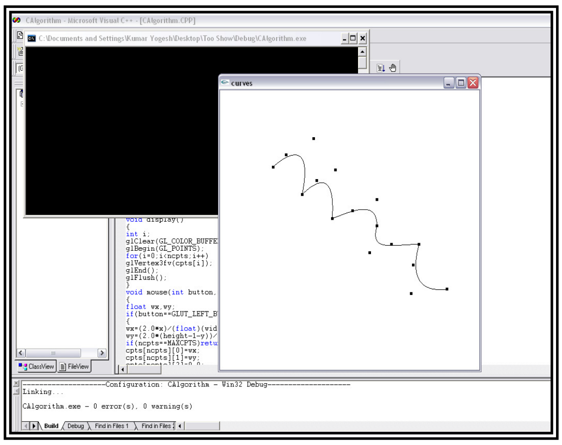
Fig. 11: Generation of 4 points Bezier Curve using *OpenGL*

Fig. 11 shows 4 points Bezier Curve, Fig. 12 shows 8 points Bezier Curve and Fig. 13 shows 16 points Bezier Curve drawn using *OpenGL* which are independent of operating systems.



```
void dra
{
int i;
for(i=0;
{
glMap1f(
//range
glMapGri
glEvalMe
//from 0
}
glFlush(
}
void dis
{
int i;
glClear(
glBegin(
for(i=0;
glVertex
glEnd();
glFlush();
}
void mouse(int button,int state,int x,int y)
{
float wx,wy;
if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
{
wx=(2.0*x)/(float)(width-1)-1.0;
wy=(2.0*(height-1-y))/(float)(height-1)-1.0;
if(ncpts==MAXCPTS)return;
cpts[ncpts][0]=wx;
```

Fig. 12: Generation of 8 points Bezier Curve using *OpenGL*.



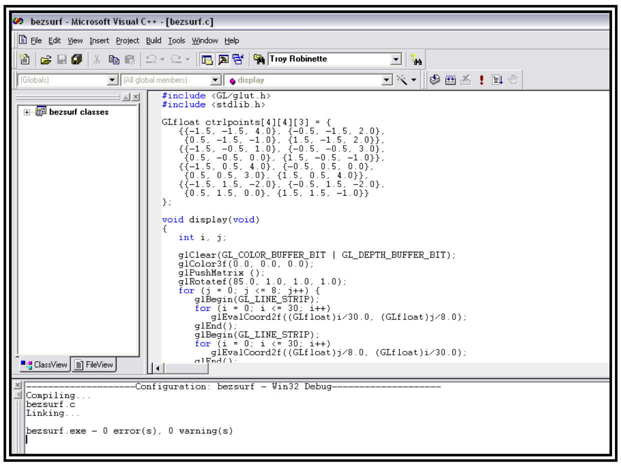
```
void display()
{
int i;
glClear(GL_COLOR_BUFFE
glBegin(GL_POINTS);
for(i=0;i<ncpts;i++)
glVertex3fv(cpts[i]);
glEnd();
glFlush();
}
void mouse(int button,
{
float wx,wy;
if(button==GLUT_LEFT_B
{
wx=(2.0*x)/(float)(wid
wy=(2.0*(height-1-y))/
if(ncpts==MAXCPTS)retu
cpts[ncpts][0]=wx;
cpts[ncpts][1]=wy;
```

```
--------------------Configuration: CAlgorithm - Win32 Debug--------------------
Linking...

CAlgorithm.exe - 0 error(s), 0 warning(s)
```

Fig. 13: Generation of 16 points Bezier Curve using *OpenGL*

(B) Bezier Surfaces

CASE 1: Generating Wireframe Bezier Surface



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

GLfloat ctrlpoints[4][4][3] = {
   {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
   {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
    {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
   {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
    {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
   {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
    {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void display(void)
{
   int i, j;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(0.0, 0.0, 0.0);
   glPushMatrix ();
   glRotatef(85.0, 1.0, 1.0, 1.0);
   for (j = 0; j <= 8; j++) {
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
      glEnd();
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
```

```
--------------------Configuration: bezsurf - Win32 Debug--------------------
Compiling...
bezsurf.c
Linking...

bezsurf.exe - 0 error(s), 0 warning(s)
```

Fig. 14: Compiling the Algorithm for Generating Wireframe Bezier Surface

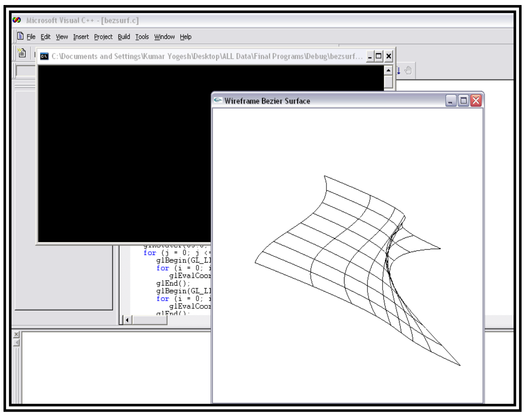
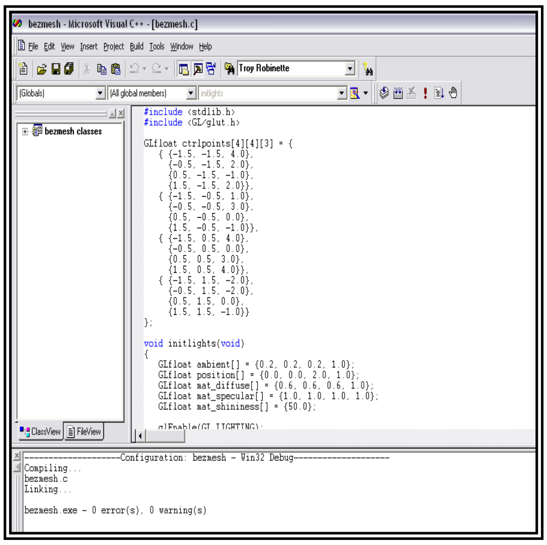

Microsoft Visual C++ - [bezsurf.c]
File Edit View Insert Project Build Tools Window Help
C:\Documents and Settings\Kumar Yogesh\Desktop\ALL Data\Final Programs\Debug\bezsurf...
Wireframe Bezier Surface

Fig. 15: Generation of Wireframe Bezier Surface using *OpenGL*

The Fig. 14 shows that the platform independent algorithm for generating Wireframe Bezier Surface has been successfully compiled and results are shown in Fig. 15.

CASE 2: Generating Meshed Bezier Surface

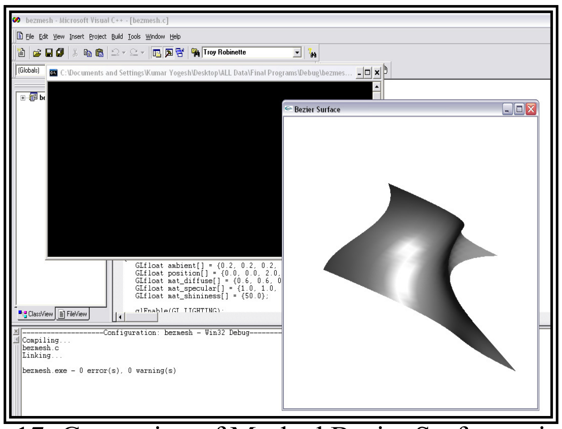

bezmesh - Microsoft Visual C++ - [bezmesh.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
(All global members)
initlights
bezmesh classes
ClassView
FileView

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

GLfloat ctrlpoints[4][4][3] = {
   { {-1.5, -1.5, 4.0},
     {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0},
     {1.5, -1.5, 2.0}},
   { {-1.5, -0.5, 1.0},
     {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0},
     {1.5, -0.5, -1.0}},
   { {-1.5, 0.5, 4.0},
     {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0},
     {1.5, 0.5, 4.0}},
   { {-1.5, 1.5, -2.0},
     {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0},
     {1.5, 1.5, -1.0}}
};

void initlights(void)
{
   GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
   GLfloat position[] = {0.0, 0.0, 2.0, 1.0};
   GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
   GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat mat_shininess[] = {50.0};

   glEnable(GL_LIGHTING);
```

```
--------------------Configuration: bezmesh - Win32 Debug--------------------
Compiling...
bezmesh.c
Linking...

bezmesh.exe - 0 error(s), 0 warning(s)
```

Fig. 16: Compiling the Algorithm for Generating Meshed Bezier Surface

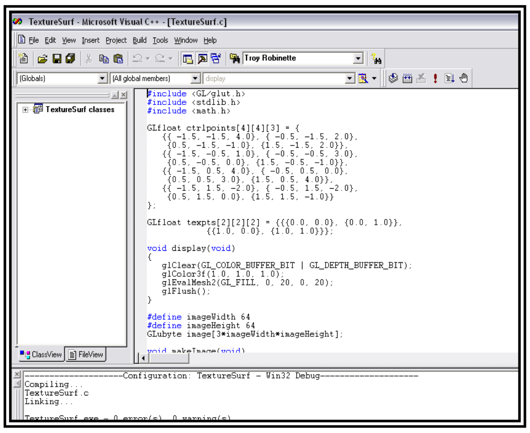
The Fig. 16 shows that the platform independent algorithm for generating Meshed Bezier Surface has been successfully compiled and results are shown in Fig. 17.


bezmesh - Microsoft Visual C++ - [bezmesh.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
C:\Documents and Settings\Kumar Yogesh\Desktop\ALL Data\Final Programs\Debug\bezmes...
Bezier Surface
ClassView
FileView

```
--------------------Configuration: bezmesh - Win32 Debug--------
Compiling...
bezmesh.c
Linking...

bezmesh.exe - 0 error(s), 0 warning(s)
```

Fig. 17: Generation of Meshed Bezier Surface using *OpenGL*

CASE 3: Generating Textured Bezier Surface

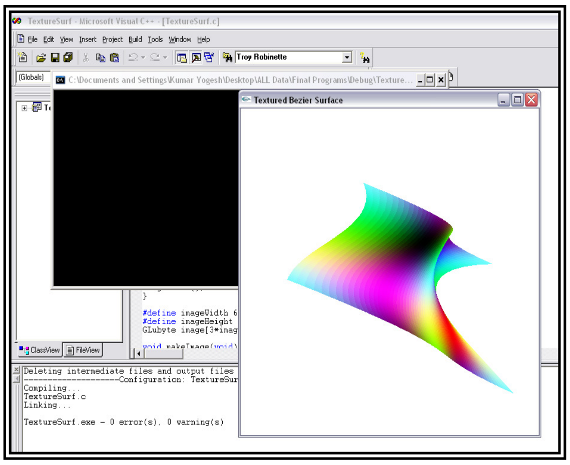

TextureSurf - Microsoft Visual C++ - [TextureSurf.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
(All global members)
display
TextureSurf classes
ClassView
FileView

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

GLfloat ctrlpoints[4][4][3] = {
   {{ -1.5, -1.5, 4.0}, { -0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
   {{ -1.5, -0.5, 1.0}, { -0.5, -0.5, 3.0},
    {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
   {{ -1.5, 0.5, 4.0}, { -0.5, 0.5, 0.0},
    {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
   {{ -1.5, 1.5, -2.0}, { -0.5, 1.5, -2.0},
    {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}},
            {{1.0, 0.0}, {1.0, 1.0}}};

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glEvalMesh2(GL_FILL, 0, 20, 0, 20);
   glFlush();
}

#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];

void makeImage(void)
```

```
--------------------Configuration: TextureSurf - Win32 Debug--------------------
Compiling...
TextureSurf.c
Linking...

TextureSurf.exe - 0 error(s), 0 warning(s)
```

Fig. 18: Compiling the Algorithm for Generation of Textured Bezier Surface

The Fig. 18 shows that the platform independent algorithm for generating Meshed Bezier Surface has been successfully compiled and results are shown in Fig. 19.


TextureSurf - Microsoft Visual C++ - [TextureSurf.c]
File Edit View Insert Project Build Tools Window Help
Troy Robinette
(Globals)
C:\Documents and Settings\Kumar Yogesh\Desktop\ALL Data\Final Programs\Debug\Texture...
Textured Bezier Surface
ClassView
FileView

```
}

#define imageWidth 6
#define imageHeight
GLubyte image[3*imag

void makeImage(void)
```

```
Deleting intermediate files and output files
--------------------Configuration: TextureSur
Compiling...
TextureSurf.c
Linking...

TextureSurf.exe - 0 error(s), 0 warning(s)
```

Fig. 19: Generation of Textured Bezier Surface using *OpenGL*

5 Conclusions

In the present work, an attempt has been made to develop CAD algorithms for the generation of Bezier curves and Bezier surfaces using *OpenGL*. This satisfies all the necessary requirements for the development of CAD software which will be independent of operating systems. The main idea behind the current work is to bring together the ideas of Mathematical curves and Graphics Programming into single complex unit which can be implemented for development of CAD software. The developed algorithms are not only user friendly and interactive but also gives worthy information about the construction of Bezier Curves and Bezier Surfaces with minimum of technical complexities.

6 Scope of the Future Work

The present work is mainly concerned with the design and implementation of Cubic Bezier Curves. However implementation of higher order Bezier Curves is also possible. One more enhancement for the future is that the increase in the number of control points. With higher order curves the complex drawings can be created.

Although this might get a little complicated but not at all impossible. It will just require only doing some minor adjustment in the present algorithms for writing a whole new source code for a higher order Bezier curve and a new CAD software can be developed which will be independent of operating system.

References:

[1] Bettig, B., Shah, J. (1999), "An object-oriented program shell for integrating CAD software tools", *Advances in Engineering Software,* 30 (8): pp. 529-541.

[2] Bhankdarkar M.P., Downie B., Hardwick M., Nagi R. (2000), "Migrating from IGES to STEP: one to one translation of IGES drawing to STEP drafting data", *Computers in Industry,* 41 (3): pp.261-277.

[3] Butdee, S. (2002), "Hybrid feature modeling for sport shoe sole design", *Computers & Industrial Engineering,* vol.42, no.2-4, pp.271-279.

[4] Conference.et.byu.edu/~paracad/theses/Travis%20L.%20Astle.doc

[5] Gordon, W.J., Riesenfeld, R.F. (1974), "B-spline curves and surfaces", *Computer Aided Geometric Design,* New York: Academic Press.

[6] Hope, A. (1985), "Room for Improvement", *Engineering (London)*, vol. 225, no. 3, p. 158.

[7] Mack II, R.G., Lee, C.-H., Letcher Jr., J.S., Newman, J.N. Shook, D.M., Stanley, E. (2002), "Integration of geometry definition and wave analysis software", *Proceedings of the International Conference on Offshore Mechanics and Arctic Engineering - OMAE*, vol.1, pp. 721-733.

[8] Monies, F., Redonnet, J.M., Lagarrigue, P. (2000), "Improved positioning of a conical mill for machining ruled surfaces: application to turbine blades", *Proceedings of the Institution of Mechanical Engineers. Part B, Journal of Engineering Manufacture,* vol.214, no.7, pp.625-634.

[9] Rogers, David F. (2001), "An Introduction to NURBS: With Historical Perspective", copyright by Academic Press.

[10] Rohm, T., Jones, C.L., Tucker, S.S., Jensen, C.G. (2000), "Parametric Engineering Design Tools and Applications", Proceedings of DETC2000: ASME Design Automation Conference, September 10-13, 2000, Baltimore, Maryland.

[11] Tonshoff, H.K., Rackow, N., Gey, C. (2000), "Advanced Tool Path Generation For Flank Milling Turbo machinery Components", *The Third World Congress on Intelligent Manufacturing Processes and Systems, Cambridge, MA June 28-30*, pp.446-452.

[12] Versprille, K.J. (1975), "Computer-Aided Design Applications of the Rational B-spline Approximation Form," Ph.D. dissertation, Syracuse University.

[13] http://www.fei.edu.br/~psergio/CG_arquivos/IntroSplines.pdf

[14] http://www.glprogramming.com.

[15] http://www.opengl.org.

[16] http://en.wikipedia.org/wiki/B%C3%A9zier_curve.

[17] http://en.wikipedia.org/wiki/CAD.

[18] http://en.wikipedia.org/wiki/Computer-aided_design.

[19] http://mathworld.wolfram.com/BezierCurve.html

[20] http://www.ibiblio.org/e-notes/Splines/Inter.htm

[21] http://en.wikipedia.org/wiki/B%C3%A9zier_surface.