ARISTOTLE UNIVERSITY OF THESSALONIKI

Efstratia Lampridou

A Web-based Design Framework for Shapes Outline

SCHOOL OF INFORMATICS

Supervisor: Associate Professor Athena Vakali
A Web-based Design Framework for Shapes Outline
Acknowledgements

This project was implemented as a part of the requirements for obtaining the MSc in Informational Systems at Informatics Department of Aristotle University of Thessaloniki.

Firstly, I would like to thank my supervisor, Athena Vakali, Associate Professor of my department, for giving me the opportunity to work on the project and her valuable advices during the academic year. Many thanks would also go to Georgios Andreadis, Assistant Professor at the department of Mechanical Engineering of Aristotle University of Thessaloniki, for his guidance throughout the completion of the project.

Finally, I thank my family and friends for their support during this year.

Efstratia Lampridou

22/09/2010
Abstract

This thesis concerns the development of a Web-based framework for shapes outlines. In particular, it provides the framework that enables access to a CAD embroidery font digitizing software that is specialized for East-Asian language character sets. The application is accessible through the Web. The need for such an application is greater in the East-Asian languages whose characters are much more complex than the Latin ones and a slight change in the way that they are embroidered can result in misconception or illegibility of the character. The application can process any True Type font of the Windows Operating System. The end user is provided with many operations, including editing the font’s characters by setting break lines and defining the stroke curve order for each character. Following, Bézier curves are used in order to describe the shape of the character which the user can further edit. The final outcome of the software is an accurate contour of the selected characters that can be exported in various formats in order to be fed to an Embroidery Designing System. The area of study of the thesis lies in the Computer-Aided Design field and specifically in the Computer-Aided Design for Embroidery.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>V</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>VII</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>IX</td>
</tr>
<tr>
<td>FIGS</td>
<td>XI</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. EMBROIDERY</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Machine Embroidery</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Computer-Aided Design</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1 Computer-Aided Embroidery</td>
<td>4</td>
</tr>
<tr>
<td>2.2.2 File Formats</td>
<td>5</td>
</tr>
<tr>
<td>3. DESCRIBING SHAPES</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Research Work</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Curves</td>
<td>8</td>
</tr>
<tr>
<td>3.2.1 Bézier Curves</td>
<td>9</td>
</tr>
<tr>
<td>3.2.2 Continuity</td>
<td>12</td>
</tr>
<tr>
<td>3.3 The Branches</td>
<td>15</td>
</tr>
<tr>
<td>4. EAST ASIAN LANGUAGES</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Writing Rules</td>
<td>19</td>
</tr>
<tr>
<td>5. THE DESIGN FRAMEWORK AND THE APPLICATION</td>
<td>23</td>
</tr>
<tr>
<td>5.1 The purpose</td>
<td>23</td>
</tr>
<tr>
<td>5.2 The Web-based Framework</td>
<td>23</td>
</tr>
<tr>
<td>5.3 The Designing Application</td>
<td>24</td>
</tr>
<tr>
<td>5.3.1 The features</td>
<td>26</td>
</tr>
<tr>
<td>5.4 The modes</td>
<td>34</td>
</tr>
<tr>
<td>5.4.1 Break Lines</td>
<td>34</td>
</tr>
</tbody>
</table>
Figs

Fig 1 – Embroidery Design ................................................................. - 5 -
Fig 2 – Bézier Curve ........................................................................ - 9 -
Fig 3 – Bernstein Polynomials ............................................................... - 10 -
Fig 4 – No continuity ........................................................................ - 13 -
Fig 5 – Continuity C⁰ ......................................................................... - 13 -
Fig 6 – Continuity C¹ ......................................................................... - 14 -
Fig 7 – Continuity C² ......................................................................... - 14 -
Fig 8 – A branch ................................................................................ - 15 -
Fig 9 – Non-corner, automatic knot ......................................................... - 16 -
Fig 10 – Non-corner, non-automatic knot .................................................. - 16 -
Fig 11 – Corner, automatic knot ................................................................. - 17 -
Fig 12 – Corner, non-automatic knot ......................................................... - 17 -
Fig 13 – East-Asian Characters ................................................................. - 19 -
Fig 14 – The Design Framework ............................................................... - 24 -
Fig 15 – Main Window ........................................................................ - 25 -
Fig 16 – Main Window and Character-Window .......................................... - 25 -
Fig 17 – Font Selection ........................................................................ - 27 -
Fig 18 – Destination Folder Selection for Preview Images ........................ - 27 -
Fig 19 – Character Selection ................................................................ - 30 -
Fig 20 – Multiple Character-Windows ....................................................... - 31 -
Fig 21 – Information about the FFI file ...................................................... - 33 -
Fig 22 – Right-click Menu ................................................................... - 33 -
Fig 23 – Reordering of the Line ................................................................. - 34 -
Fig 24 – The Break Lines ..................................................................... - 35 -
Fig 25 – The Stroke Order Lines .............................................................. - 37 -
1 Introduction

Computer-Aided Design (CAD) is a field with constantly increasing growth during the last few decades. Embroidery is one of the areas that utilize the advance technology of Computer-Aided-Design, so as to improve the quality of the resulting designs, to speed up the designing procedure and to reduce the total cost of the embroidery manufacturing.

This thesis concerns the development of a Web-based design Framework for CAD Embroidery that is used for the digitization of True Type Fonts. The application that was developed can process and edit the characters of any True Type Font existing in the Windows Operating System. Regarding the editing of the character, three different modes can be distinguished, namely the break lines, the stroke order lines and the outlines. Therefore, the user is able to set a number of break lines that divide the character into separate areas, then set the flow of the embroidery with the stroke order lines and finally edit the Bézier Curves that correspond to the outline of each area. The Bézier Curves can be further processed so as to best fit the shape of the area. So, the result of the editing is the character consisting of separate areas each of which has a defined direction for the embroidery and its shape is accurately described using Bézier Curves. The result is exported in various formats containing information concerning the characters along with the break lines, the stroke order lines and the outlines. The user interface of the application is friendly and direct. Finally, the Framework that was developed provides explanatory information about the design application and gives access to it through the Web.

The first three chapters of this thesis cover the background of the topics needed to accomplish the project. In chapter 2 the primitives of Embroidery are described and in chapter 3 the fundamental theory for the Curves is explained as Curves are crucial for the description of the shape of the characters of the fonts. Chapter 4 describes the writing system used in East-Asian Languages.

In the following two chapters the technical work concerning the project is described in detail. In chapter 5 the commands and utilities that the Web-based framework and the
font tool application offer are explicitly specified, while in chapter 6 are explained the issues regarding the programming code as well as the organization of the classes. Finally, chapter 7 reviews the work carried out and achievements of the project and proposes possible directions for further developments.
2 Embroidery

Embroidery can be defined as “the art or handicraft of decorating fabric or other materials with needle and thread or yarn”. The word ‘embroidery’ is a Middle English word derived from the old French ‘broder’ meaning edge or border. The art of embroidery goes back into history and its origins can be traced down to the Iron Age. Samples of embroidery are found in Ancient Egypt, Northern Europe, Persia, India and China. [1]

2.1 Machine Embroidery

Although traditional hand embroidery exists for thousands of years, ‘machine embroidery’ is only aged at about 200 years. Josue Heilmann created the first hand-embroidery machine in 1828. This machine was able to utilize up to 4 hand-embroiderers and signalled the start of the revolution in embroidery, with many other machines soon to follow. [2]

Today, embroidery machines can be single-head or multi-head, which can fit up to 56 heads. Each head can fit one thread colour, which means that the number of heads determine the amount of colours they can be included into the embroidery without interrupting the machine function. They also have multiple needles and the speed of the machine can be up to 1000 stitches per minute. The machines are accompanied with many extra features. Some of the most common features include LCD touch screen, user interface, network capability and USB connection, and internal memory that can store stitches and locations. Along with the embroidery machines, design software specifically to serve the embroidery requirements was developed. This break-through boosted the embroidery production and a new revolution began in the embroidery industry.
2.2 Computer-Aided Design

Computer-Aided design –CAD– can be defined as the use of computer technology for the design of objects, real or virtual to achieve precise drawing. The resulting drawing contains symbolic information which can be the materials, the processes, the dimensions, and the tolerances, according to application-specific conventions. Computer-Aided design may be used to design curves and figures in two-dimensional space –2D; or curves, surfaces, and solids as three-dimensional objects –3D.

CAD is an important industrial tool used extensively in various applications. Some of the applications are automotive, shipbuilding, and aerospace industries, industrial and architectural design, prosthetics, and many more. CAD is also used widely to develop computer animation for special effects in movies, advertising and technical manuals. Computer-Aided is also utilized by the embroidery industry so as to improve the embroidery designs.

2.2.1 Computer-Aided Embroidery

The embroidery design software that has been developed fulfils the needs of the home, commercial and the industrial embroidery.

Nowadays, there exist various commercial and free software products to choose from, that enable the designing and editing of embroidery patterns and images. The purpose of the design embroidery software is to translate the drawing made in the computer into stitches executed by the embroidery machine.

The software products usually have advanced user interfaces with sophisticated functions. The user can create a drawing, set up the number of stitches, define the type of the stitch (satin, run fill stitches, etc), select colours and edit the design. More complicated functions include the import of images to the software and their digitalization offering all the main attributes of the image processing.

Furthermore, some of the embroidery design software solutions offer lettering functions. These functions aim to convert automatically a true type font into stitches. In this way, a font character can be embroidered.

In figure 1, the Latin character ‘a’ is displayed in its final form after its processing through the CAD embroidery software EOS [3].
The final embroidery result can be saved into formats that are then capable of being imported into an embroidery machine.

### 2.2.2 File Formats

Embroidery file formats are various. Two main categories can be distinguished; the source formats and the machine formats.

Source formats files are files supported by the design software whereas machine formats files are oriented to a particular brand of embroidery machine. These files contain primarily stitch data as well as machine functions. The main drawback of the machine files is that they cannot be easily scaled or edited.

Each different embroidery machine manufacturer has developed its own embroidery machine formats specifically designed to service its brand or a specific machine. However, although machine formats where initially designed to service only their own machine brand, some formats have dominated and became so prevalent that gradually they are becoming regarded as standard even by companies in competition.
3 Describing Shapes

Shape description is a field with significant research activity. The purpose is to achieve an approximation of a given shape using segments of defined curves. Most of the algorithms developed use the Bézier curve format to perform the shape description.

3.1 Research Work

The research in this field is conveyed on the identification of the outline of a given shape and its description with a certain model of curves – eg. Bézier curves.

Cinque et al [5] in their work create a model of the object with approximate segments of the shape by using the Bézier cubic curves which interpolate most accurately the endpoints of the outline of the shape. Sohel et al [6] developed a shape descriptor using a Bézier Curves algorithm. The algorithm divides the shape into segments and identifies the control points.

In this project shape description is needed for the representation of the font characters. Each character is described by a set of Bézier Curves as it will be explained thoroughly below. There exists research activity on the shape description of character letters in particular.

Plass and Stone [10] developed a method for fitting shapes defined by a discrete set of data points with parametric piecewise cubic polynomial curves. Their work focuses on letter-form shapes and converts the bitmap representation of the character into curve outlines.

Sarfraz and Khan [7] developed an algorithm for capturing automatically non-Latin characters such as the Arabic ones. They also use parametric cubic Bézier curves for this task. The character is divided into a number of segments and the control points along with other significant points are detected.

Furthermore, Itoh and Ohno [8] track the outline of fonts from scanned characters’ images. The algorithm extracts the contour points from the character and divides the points into a number of segments at the corner points. Following it fits a piecewise cubic Bézier curve to each segment.
Finally, Yang et al [9] used Bézier curves to describe the outline of Chinese letters. Firstly, the contour segment is extracted and points with high curvature are identified as corner points, and afterwards the contour segment is described by a straight line or by a Bézier curve.

In the following section the fundamental theory of curves is explained focusing on the Bézier curves as these are used by the tool developed in order to describe the outline of the font characters.

3.2 Curves

Graphic systems and Computer-Aided Design software use certain basic primitives in order to represent the lines. Therefore, complex shapes can arise by using these primitives. However, as the shapes become more complex, more detailed data is required to represent them accurately.

A case of such a shape is a curve. A curve is a kind of shape that can be difficult to describe. A curve can be viewed as the union of short segments of straight lines. The representation of a curve can be explicit, implicit or parametric.

An explicit curve is one defined by an equation. In general, an explicit curve in 2 dimensions is defined by an equation of the form \( y = f(x) \). The order of the polynomial \( f(x) \) defines the order of the curve; therefore there are quadratic, cubic, quartic curves, and so on.

Implicit curves are those described by an implicit function, such as in 2D, \( f(x,y)=0 \).

Concerning parametric representation of curves, two main categories can be distinguished; approximation curves and interpolating curves. Approximating curves are those passing ‘close’ to the points that were used to define it whereas interpolating curves are those curves passing through the points used to define it.

Another specific kind of curve that is used extensively in computer-aided design and in computer graphics is the splines. Splines are curves with certain shape defining attributes and are especially useful as they can be utilised for modelling arbitrary functions.

Well-known and widely used splines in the field of computer-aided design are the Bézier curves, B-splines –which are the generalization of Bézier curves, and non-uniform rational B-splines (NURBS).
3.2.1 Bézier Curves

Bézier curves are used extensively in computer graphics and are constructed as a sequence of Bézier segments. [4]

A single segment cubic Bézier curve consists of 4 points, 2 end control points and 2 middle control points as depicted in Fig 2. The end points are \( p_0 \) and \( p_3 \) while the middle control points are the \( p_1 \) and \( p_2 \). As it can be seen, the line connecting the end control point with the corresponding middle control point is tangent to the curve. The control points affect the shape of the curve. Should a control point be moved, the shape of the Bézier curve will change accordingly.

\[
\text{Fig 2 – Bézier Curve}
\]

Generally, given a set of \( n+1 \) control points \( P_0, P_1, \ldots, P_n \), the resulting Bézier curve of degree \( n \) is given by the equation (1),

\[
C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t)
\]

where \( t \in [0,1] \) and \( B_{i,n} \) is a Bernstein polynomial. Bernstein polynomials are defined by the formula (2),

\[
B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}
\]
The degree of the curve is equal to the number of control points minus 1 (n). The degree of the Bernstein polynomials is also n. The Bernstein polynomials of degree n form a \textit{basis} for the power polynomials of degree n.

All basis functions are non-negative. Moreover, the sum of the basis functions is 1, and since they are nonnegative, it can be seen that the value of any basis function is in the range of 0 and 1.

In Fig 3 the plot of the Bernstein polynomials of degree 9 is presented.

![Bernstein Polynomials](image)

\textbf{Fig 3 – Bernstein Polynomials}

\textbf{Properties}

The Bézier Curves have some significant properties that differentiate them from the other types of curves. A property of the Bézier curve –that has been already mentioned, is that the curve always passes through the first and last control points. However, Bézier curves do not interpolate the interior points but only approximate them. So unlike other types of curve, the interpolation is made for only some control points.

The variation diminishing property of the Bézier curves is that no straight line intersects a Bézier curve more times than it intersects its control polygon. A control polygon is the join of the line segments that pass through the control points of the curve. The importance of this property lies to the fact that the complexity of the curve (by means of turning and twisting) is not greater than the complexity of the control polygon.
Therefore, the control polygon twists and turns more frequently than the Bézier curve does.

The Bézier curves also satisfy *the convex hull* property. This property declares that every point of the curve is inside the convex hull that corresponds to the curve, meaning that the curve lies completely in the convex hull. The property ensures that the curve will never pass outside of the convex hull formed by the control vertices and so it lends a measure of predictability to the curve. This property is important because it secures that any Bézier curve will be in a well-defined and computable region.

An undesirable property of Bézier curves is their *numerical instability* for large numbers of control points. This can be avoided by smoothly joining together low-order Bézier curves, as it will be explained below. Furthermore, the fact that by moving a single control point the shape of the curve is affected in whole is another negative aspect.

**Piecewise Bézier Curves**

More than one Bézier segment can be joined together forming a larger connected curve; also known as *piecewise Bézier curve*. Therefore, in order to create a larger curve, instead of increasing the degree of the curve –which will make the curve much more complex to calculate, cubic Bézier curves can be joined together. As already explained, a cubic Bézier curve has 4 points. The one end point of a curve can be connected with an end point of another curve forming a joint. This point, which is shared by the two curves, is called knot. The result is a curve, which if divided into pieces, cubic curves arise. Moreover, each control point only affects a specific part of the curve and not the whole curve.

Hence, rather than use a high degree curve to interpolate a large number of points, it is more common to break the curve up into several simple curves. For instance, a large complex curve could be broken into cubic curves, resulting in a piecewise cubic curve.

For the entire curve to be smooth and continuous, it is necessary to maintain $C^1$ parametric continuity across segments. Therefore the position and tangents should match at the end points. However, it is advised to maintain the $C^2$ continuity too. The way that continuity is determined is explained in the following section.
3.2.2 Continuity

Two types of continuity can be distinguished when two curves are joined together; geometric continuity and parametric continuity. The continuity of a curve at a knot describes how the curves meet at this point and determines the smoothness of the curve. A curve or surface can be described as having $G^n$ continuity, with $n$ being the increasing measure of smoothness. Considering the segments at either side of a point on a curve the following options emerge:

- $G^0$: The curves touch at the join point.
- $G^1$: The curves share a common tangent direction at the join point. The first derivatives at the common point are proportional at this point.
- $G^2$: The curves share a common centre of curvature at the join point. Hence, the first and second derivatives are proportional at the point.

Parametric continuity is applied to parametric curves. It describes the smoothness of the parameter's value with distance along the curve. The types of continuity are explained below:

- $C^0$: The curves are joined.
- $C^1$: The first derivatives at the join point are equal.
- $C^2$: The first and second derivatives at the join point are equal.
- $C^n$: The first $n$ derivatives at the join point are equal.

As can be seen, geometric continuity requires the geometry to be continuous, whereas the parametric continuity requires the parameterization to be continuous too. Having parametric continuity of a given order implies geometric continuity of the same order, yet the reverse is not valid.

Below is explained the nature of the joint –continuity– that can arise from the connection of two curves. [11]

Fig 4 shows the case where no continuity exists between the curves. The two curves don't share any common point. This is declared as no continuity.
In Fig 4, the two curve segments share a single common point. In this case, the curves are connected, but with an obvious break in direction. The derivatives (for right and left segment) are different both in direction and size in this point. This is an example of \textit{continuity zero}, \(C^0\).

In Fig 5, the two curve segments share a single common point. In this case, the curves are connected, but with an obvious break in direction. The derivatives (for right and left segment) are different both in direction and size in this point. This is an example of \textit{continuity zero}, \(C^0\).

In Fig 6 are presented two curve segments that have a point in common and the joint is smooth. The derivatives in this point have the same direction. The first derivatives are equal. This is defined as tangential continuity or \textit{continuity one} – \(C1\).
Finally in Fig 7, the two curve segments have a common point, and the joint is smooth. The derivatives in the common point have both the same direction and size. In this point the curves have identical curvature. This is declared as curvature continuity or \textit{continuity two} – $C^2$. Here, the second derivatives are equal.

In the above example it is obvious that the corresponding geometric continuity exists as well.

It should also be noted that as tangential continuity implies positional continuity, whereas curvature continuity implies both tangential and positional continuity as well. Moreover, the order of a curve determines the maximum continuity possible. Thus, it might be needed a higher order of curve if more continuity is necessary. The maximum continuity that can be achieved for a given curve is order – 2. For instance, for the cubic curves, the maximum continuity is $C^2$. 
3.3 The Branches

Most of the software solutions use Bézier Curves in order to depict the resulting areas for embroidery.

A customised model that was used in the design software EOS is the combination of two piecewise Bézier curves. From now on, this will be referred to as *branch*. A branch is defined as two piecewise cubic Bézier curves connected together. Both Bézier curves have the same number of control and end points. Each end point of the Bézier curve is connected with the corresponding end point of the second Bézier curve with a straight line named the *reference line*. Hence, each branch defines an area which will be embroidered on the same way. Moreover, the reference lines are used as a delimiter for the path that the stitches will be created on. In particular, the stitches will go perpendicular through the reference lines. An example of a branch is pictured in Fig 8.

![Fig 8 – A branch](image)

The continuity on the curves of the branch can differ. The different types of continuity are distinguished by the nature of the point that the two curves have in common. In particular a knot can be a corner or a non-corner. Defining a point as a corner, it means that the joint doesn’t have to be smooth. In this case the resulting continuity is $G^0$. Otherwise, a non-corner knot creates a smoother nature of join. The tangents of the
control points have the same direction but not the same size. This results in continuity $G^1$. A corner knot is displayed with a square, whereas a non-corner knot is displayed as a circle.

Furthermore, the calculation of the knot can be automatic or non-automatic. An automatic calculation assumes that the calculation was made by internal rules whereas the non-automatic type arises by the movement of a control point by the user.

The filled circle, as depicted in Fig 9, is used to declare that the point that joins the two curves is not a corner and is created automatically.

![Fig 9 – Non-corner, automatic knot](image)

In Fig 10 is shown a blank circle which defines a non-corner knot. A modification to the control points was made resulting in a non-automatic point.

![Fig 10 – Non-corner, non-automatic knot](image)
Fig 11 shows the corner point. It is obvious that there is a break in the direction. The geometric continuity is $G^0$ as the derivatives are different in size and in direction. The direction of the derivatives is in the direction of the adjacent user points.

![Fig 11 – Corner, automatic knot](image)

Fig 12 shows the type of join pictured with a black square. This also declares a corner point but this one isn’t created automatically. The movement of the control points is more flexible and the direction of the tangents isn’t pointing to the adjacent knots.

![Fig 12 – Corner, non-automatic knot](image)
4 East Asian Languages

The East Asian languages are the languages of East and Southeast Asia which have been influenced by Classical Chinese and its writing system. These are the Chinese, the Japanese, the Korean and the Vietnamese.

![East-Asian Characters](image)

Fig 13 – East-Asian Characters

4.1 Writing Rules

Characters of the East Asian Languages are written with very precise rules. The three most important rules are the strokes employed, stroke placement, and the order in which they are written (stroke order). Most words can be written with just one stroke order, though some words also have variant stroke orders, which may occasionally result in different stroke counts; certain characters are also written with different stroke orders in different languages.

The stroke order refers to the correct order in which the strokes of a character are written. A stroke can be defined as the movement of a writing instrument where its writing edge is touching the page.

The simplest characters are written as one stroke, yet the most complicated characters require more than 30 strokes. [12]
Writing characters in the correct order is essential for the character to look correct and to be legible. The direction when drawing a stroke is from left to write and from top to down. When a character that involves several different strokes needs to be written, strict rules that declare the order must be applied. The two basic rules that are followed when writing East Asian language characters are that the top lines are drawn before the bottom ones and the left lines before the right ones. [13]

- Top before bottom

```
二 二 二
```

- Left before right

```
八 八
```

The conflicts that may arise with the combination of the above rules are resolved with five more additional rules. The left vertical stroke is drawn before top horizontal stroke. Moreover, the bottom horizontal stroke should be the last one to be drawn. Centre strokes are also to be drawn before the ‘wings’. Furthermore, the horizontal strokes should be drawn before any intersecting vertical stroke. Left-falling strokes are drawn before the corresponding right-falling ones. Finally, a rule that contradicts the others is that the minor strokes are the last ones to be drawn.

- Left vertical before top horizontal

```
田 田 田
```

- Bottom horizontal last

```
王 王 王 王
```

- Centre before wings

```
小 小 小 小 小
```
• Horizontal before intersecting vertical


• Left-falling before right-falling


• Minor strokes at the end


When these characters are to be embroidered, the correct direction of the stroke should be followed. Thus, the flow of the stitches has to be from left to right and from top to bottom.
5 The Design Framework and the Application

5.1 The purpose

As it is mentioned above, there are various software solutions that enable the user to create or edit drawings in order to export them in the appropriate format and import them to an embroidery machine. Most of them have the utility to edit language sets that will be given for embroidery.

These automated functions seem to perform well for Latin character sets. However, in East Asian Language character sets, such an automated tool fails due to the high complexity of the characters. As it was noted in the previous section, the stroke order of the character is also crucial in order for it to be legible.

The purpose of this project is to develop an application which edits a true type font by saving the required information for the character in order to embroider it in a correct way. An expert in writing the language characters can edit the character providing the information needed. The result can be saved in a format that will be read again from the application so the user can store the fonts that have been edited.

Moreover, the World Wide Web offers a number of advantages and therefore the design Framework can be utilized so as to give access to the application through the Web.

5.2 The Web-based Framework

The Web-based Framework that was developed facilitates to the creation of the outlines though the Web. In particular, a website was developed, which provides information about the designing application and also gives access to it. It enlists the operations that can be executed in the editing process of the shape-character, as well as the resulted files that are exported by the system. By pressing the button "Font Tool" the application runs on the client-machine. In Fig 14 a screenshot of the Web Framework is presented.
5.3 The Designing Application

The designing application is the application that processes the True Type Fonts and serves to the editing of the characters. In Fig 15 a screenshot from the main application— that opens via the website— is shown, while in Fig 16 a screenshot of the application with a certain character opened is displayed.
Fig 15 – Main Window

Fig 16 – Main Window and Character-Window
5.3.1 The features

The application enables the user to select a true type font and then to select a certain character and the character of the corresponding font appears in a window. The application has three basic functions. The setting of the break lines, the definition of the stroke order and the editing of the resulting outline.

The break lines are straight lines which split the character into separate areas. The definition of the stroke order includes the creation of curves which correspond to the direction of the stroke in the embroidery. Furthermore, the outlines of the character can be displayed depending on the branches that result from the character, taking into account the break lines. Consequently, the result is a number of branches according to the number of areas generated with the break lines. Also, each branch has a distinct stoke order curve. Finally, the branches are ordered in the order that the embroidery will be done.

The different functions of the application are enabled by selecting the corresponding mode from the toolbox or main menu.

Menu – Toolbar Buttons

The application has a toolbar which contains a subset of the commands available by the application’s menu. A sort description of the commands in the menu follows:

- File->New

A new FFI file is created. The system’s “Choose Font” dialog is displayed as shown in Fig 17. The user can select the font he wants to work with.
Following, the user is asked to select an output folder where a *.JPG preview is automatically saved for every character that will be opened, as shown in figure 18.
File->Open…
An existing FFI file is opened. The system’s “Choose Font” dialog is displayed in this case too. It is initialized with the font it was selected when the FFI file was created, only if this Font is available in the current Windows System. Again, the user selects a destination folder for the preview output.

File->Save
This command enables the saving of the Stoke Order, the Break Line and the Outline information for all processed characters in the .FFI and .OUT files.

File->Save As
The FFI and OUT file are saved with a new name.

File->Print…
The active Character-Window is printed.

File->Print Preview
A print preview is displayed.

File->Print Setup…
The printer and printing options can be changed.

File->Exit
Quit the application; prompts to save changes.

View->Toolbar
Show or hide the toolbar.

View->Status Bar
Show or hide the status bar.

**Window->Cascade**

Arrange windows so they overlap.

**Window->Tile**

Arrange windows as non-overlapping tiles.

**Window->Arrange Icons**

Arrange icons at the bottom of the window.

**Window->Close**

Close the active Character-Window.

**Window->Close All**

Close All Character-Windows. It is useful when a lot of Character-Windows are displayed simultaneously.

**Character->Start**

Open the character ‘!’ –ASCII 33 Dec, so that the user can start the font processing. This is the first printable character in the ASCII table.

**Character-> Goto Specific...**

Select a specific character to process. A dialog is displayed in which the user can select any character. The user can do so either by writing the character directly from the keyboard either by giving the ASCII code as presented in Fig 19.
Fig 19 – Character Selection

- **Character->Previous**
  Process the Previous character, e.g. “B”-> “A”.

- **Character->Next**
  Process the Next character, e.g. “A”-> “B”.

- **Character->New Char Mode->Close Current Window on**
  When a new Character-Window is created, the Current Character-Window closes.

- **Character->New Char Mode->Close Current Window off**
  When a new Character-Window is created, the Current Character-Window does not close. In this way a lot of Character-Windows are displayed simultaneously. In Fig 20, a screenshot of multiple windows view is shown.
Fig 20 – Multiple Character-Windows

**Character->Lines Mode->Break Lines**
Set the Break Lines mode. Then, the mouse is used for the input of Break Lines.

**Character->Lines Mode->Stroke Order**
Set the Stroke Order Lines mode. Then, the mouse is used to insert Stroke Order Lines.

**Character->Lines Mode->Outlines**
Set the Outlines mode. Then, the mouse is used to edit the branches.

**Character->Previous Line**
It activates the previous Line (break line, stroke order line or outline).

**Character->Next Line**
It activates the next Line (break line, stroke order line or outline).
Character->Delete Line

It deletes the active Line (break line, stroke order line or outline).

Character->Store lines

It stores all the Lines information (break lines, stroke order lines and outlines) for the current character in the memory buffer.

Character->Auto Store Lines->On

It turns on the Auto-Store mode to store the Lines after the Character-Window closes.

Character->Auto Store Lines->Off

It turns off the Auto-Store. The Lines are not stored after the Character-Window Closes.

Character->Zoom In

The Zoom factor is increased. The zoom range is [200…1500] with step 50. The zoom factor for the current Character-Window is displayed on the Status Bar of the application. The new Character Windows uses this Zoom factor. Initially the character is displayed with zoom factor 450.

Character->Zoom Out

The Zoom factor is decreased.

Info->About FONT TOOL…

The program information, version number and copyright is displayed.

Info->Information…

It displays information about the FFI file and the Font. In Fig 21 is a screenshot of the information of a FFI file, showing the file name, the character set, the font type, the destination folder. Furthermore is shown the number of the characters that are saved on
the file along with the information on whether further modifications have been made to the font set currently.

Fig 21 – Information about the FFI file

The most commonly used menu commands of the application can be invoked by using the associated Acceleration Keys. For example the plus key “+” corresponds to Zoom In, the delete key Del to the deletion of the active Line.

Furthermore, the right mouse click appears a menu on the point the user clicked and makes two commands available which are shown in Fig 22. These are the Delete and the Reorder. Depending on the mode that is selected, the user can delete the active break line, the active stroke order line, or the selected line/point from the active branch. Similarly, with the reorder command the user can reorder the active break line, stroke line or branch.

Fig 22 – Right-click Menu

Once the reorder command is selected, a dialog box appears in the screen as shown in Fig 23. It takes as an input a positive integer with ranging between 0 and the max number of lines.
The user can manually or automatically store the current stroke line/break line/outline information for the current character in a memory buffer. However, in order to save these changes for all the processed characters stored in the memory buffer to a disk file, the user must explicitly execute the Save command.

The file extension associated with this application is *.FFI (Font File Info). The file includes the information about the characters that are stored along with the break lines and the stroke order lines. One more file is created when the save command is executed with the extension *.OUT. This file contains the necessary information to describe the branches that each character consists of.

5.4 The modes

As explained above there are three available modes; the break lines, the stroke order lines and the outlines.

5.4.1 Break Lines

The break lines are straight lines which are drawn by pressing the left mouse button and moving the mouse. The break lines must always be drawn having their square ends outside the active area of the character (i.e. outside black areas and on white) while at the same time cover at least one active pixel of it. If this condition is not met, the break line will be discarded. It should be noted that even when having what seems to be a valid input, there is a possibility that the resulting embroidery will have more splits than the user declared. The reason for that is that internal rules can be forced in order to ensure embroidery validity.
The break lines are drawn with blue colour. Yet, the active break line is displayed with red colour. A break line can be made active once the user clicks the left mouse button on it. Moreover, the user can edit the active break line by moving its end points or by moving the whole line. The user can also delete the active break line by executing the delete command – from the keyboard (Del button) or from the menu.

In Fig 24 is pictured the setting of break lines in a Japanese character. As it can be seen 6 break lines were created in order to divide the character in separate sections each of which forms a shape that could be easily described. In general, the resulting divided areas are desired to be well-defined shapes such as rectangles.

Fig 24 – The Break Lines
5.4.2 Stroke Order Lines

The stroke order lines are drawn in a similar way as the break lines, when the Stroke Order Line Mode is selected. The Stroke Order Lines should be as many as the distinct areas resulting by the requested divisions, which is usually the number of Break lines plus one. If there are more than one stroke lines per area, the first one of them will be used for the ordering process. If the strokes are less, the program will use them and order the remaining areas using internal rules. Each stroke should be assigned to a distinct area. In order to do so, the stroke’s starting point (the end near which the numbered label is placed) is used as a test point for inclusion. The first area that will be found to contain that point will be ordered first.

The stroke order lines are displayed with yellow colour and the active stroke order line is with red. The user can delete the active stroke order line by executing the delete command.

In Fig 25 are shown the stroke order lines that were inserted in the same character of the previous Fig after the break lines were set. As it can be seen, the number of the stroke order lines is equal to the number of the areas that arose from the division of the character by the break lines. Finally, it can be seen that since the stroke order lines mode is selected, the break lines appear blue without their end points. This means that the break lines cannot be modified when the break line mode is not the one selected.
5.4.3 Outlines

By selecting the Outlines mode, the branches that correspond to each area are calculated and are depicted with green color. Actually, each branch declares the outline of the divided areas. When the Outlines mode is on, the break lines as well as the stroke order curves are hidden.

The user can select a specific branch by clicking inside of it in order to make it active. Once a Branch is selected, it is depicted with red color instead of green. Moreover, when a branch is active, the knots along with the control points are pictured in the screen. In Fig 26 the branches of the same Japanese character are depicted. There are 7 resulting branches. The active branch presented with red color is the branch with index
number 1. It can be also seen that since the Outlines mode is selected, neither the break lines nor the stroke order curves are shown.

Should the user go back to Break Lines mode and apply changes to the break lines by adding new ones or by deleting or modifying existing ones, the branches will be recalculated according to the new structure of break lines.

Since the branches are displayed in the character window, the user can edit the active branch by moving the control points, the handles, the connecting lines, or the whole branch. The user can also delete control points or lines, or insert new control points to the curves.
The delete command is executed when a certain point or line is selected and the delete button (del) is pressed from the keyboard. Alternatively, the right mouse click displays a menu which enables the user to select the delete command.

The insertion is executed when the Insert button is pressed from keyboard and the user clicks with the mouse in a point along the Bézier curve. Then, by dragging the mouse, the reference line is created until the point in the second Bézier curve, where a corresponding knot is created as well.

When the Control (Ctrl) button from keyboard is pressed, while a knot is selected, the type of connectivity in the knot is changing.

In Fig 27 is shown the branch with index number 7 being active. Also, it can be seen that the second reference line on the left has a square in it, showing that the user has clicked the mouse on it to select it. This square operates as a ‘focus’ object to declare the selection of the user which can be a knot, a reference line or the centre point of the branch –which is pointed by the four-direction arrow. Each Bézier curve of the branch has 4 points which are connected with the corresponding reference lines.
In Fig 28 are depicted the changes that were made to the current active branch. In particular, one knot was deleted and the centered reference line was moved. Moreover, the type of the knot changed and became ‘circle’ instead of ‘square’ to show that the point is not a corner. The control points were also moved so as to make more descriptive the curves that represent the outline. So, in this branch each Bézier curve has 3 user points (knots) two of which are corners.
5.5 Reordering

The reordering command can be applied in each of the available modes, in order to reorder the break lines, the stroke order curves and the outlines. So, depending on the mode that is selected, the corresponding item can be reordered. It should be noted that the item that is to be reordered should be first made active.

Break Lines Reordering
The break line reordering enables the break lines to be in the desired order. It is not significant though in the embroidery order and it is not taken into account at the calculation of the branches.
**Stroke Lines Reordering**
The stroke curve ordering is important as it determines the ordering that the embroidery will be made and the order that the embroidery machine will function. However, the order is not final as it can be changed –by the user– on the outlines mode. Nevertheless, this order affects the calculation and ordering of the branches.

**Outline Reordering**
The ordering of the branches is the final ordering that will be taken into account when the embroidery is to be made. Fig 29 shows a screenshot showing the same character after reordering the branches in the proper order.
5.6 Storing and Saving

When the user is making changes on a character, there is the option in saving the results in a memory buffer. This can be done either automatically or manually by the user. When the auto-store is on, the changes that the user is making to the character are saved in the memory buffer. Once a break line, a stroke curve or a branch is added, deleted or modified, it will be stored in the buffer. The auto store is on by default when the application starts. However it can be set to off, resulting in a non-automatic storing. In this case, if the user wants to store the changes of the character in the memory buffer, he
should explicitly press the *Store the Lines* button. When this button is pressed, the processes that have taken place until that time to the character’s set are stored. So, in the memory buffer can be stored all the informational lines for the given font. When the auto-store is off, and the user closes the current character-window a message box appears, which asks the user whether he wishes to store the line information to the memory. A screenshot from this message box is shown in Fig 30.

![Message Box for Storing the Characters](image)

Fig 30 – Message Box for Storing the Characters

If the user wants to create a new font set, open an existing one or exit the application, the data will be lost if not saved. The lines data can be saved to files. In particular the saving is made into two separate files. The first one, with the extension of the application *FFI*, is the main file that the user creates. The second file is created automatically by the application in order to store the additional information of the outlines. The second file is named after the name of the main file with the extension *OUT*. For instance if the first file is named by the user ‘example.ffi’, the additional file will be named by the application ‘example.out’. Only the main files with the FFI extension can be opened directly from the application. However, both files should exist in the same directory in order for the font set to be loaded by the application. In any other case the appropriate error message is displayed in the screen.

The result is that each pair of files contains all the information for the characters of a specific font set. Additional modifications can be made to the existing saved files in the characters of the font set. These modifications can be overwritten in the files after the desired changes.

### 5.7 Testing and Validation

The application was tested extensively and was found to be well-functioning, robust and error-free. Furthermore, cases where incorrect input can be given by the user have been foreseen, and the application handles the invalid input without crashing.
In particular, when the open command is executed, the application tests whether the file that is to be opened is valid as only FFI files with certain format can be opened. In addition an OUT file with the same name should also exist in the same directory. In any other case the proper error message is displayed.

Furthermore, another input of the user is the selection of the character that will be edited. In the selection dialog box the user can set the ASCII value of the character. Yet, there are only permitted ASCII values that correspond to printable characters – depending on the selected font. Otherwise an error message appears to the screen.

Moreover, in the reordering operation the application allows the user to enter an index value for the reordering that lies in the range of the existing index numbers. If the user gives a value that is out of range, the application automatically sets the minimum/maximum index number in this case.

Finally, it is not allowed the deletion of a whole branch through consequent deletion of the control points of a branch. When the minimum number of control points is reached, that is 4, the deletion command cannot be further executed.

Finally, the project was verified and is in accordance with the initial requirements as those were set at the development phase – and are described in the Introduction of this document. Moreover, the user interface meets the specifications as it is user friendly and does not require further knowledge on CAD software. It can be deduced that the application is suitable for its intended purpose and it runs as expected.
6 The Programming Code

For the development of the Web-based framework the HyperText Markup Language - HTML was used in combination with the CSS style sheet language and JavaScript scripting language.

For the development of the font designing tool the integrated development environment of Microsoft Visual Studio 6.0 was used. In particular, the tool was implemented in C++ with the use of the Microsoft Foundation Class Library (MFC) for the graphical user interface.

6.1 Code Analysis

UML class diagrams and use cases were used for the analysis of the designing font tool.

The use case of the project is shown in the following figure.
Following, are presented two class diagrams of the program. In the first one the relationships between the classes of that are used for the representation of the outlines are shown. In the second one, are depicted the relationships between the classes regarding the interface.
6.2 The Design Framework

The Web-based design framework was developed with the use of HTML. In particular, an index web page was constructed that displays relative information regarding the designing tool. The CSS style sheet language was used for the style of appearance of the web page.

Furthermore, JavaScript was used in order to run the designing application on the client browser. The method that runs the application is the `RunExe()` which is shown below.

```javascript
<script language="JavaScript" type="text/javascript">
  var oShell = new ActiveXObject( "Shell.Application" );
  function RunExe()
  {
    var commandtoRun = "C:\out\dFontTool.exe" ;
    oShell.ShellExecute(commandtoRun);
  }
</script>
```
6.3 The Application Classes

The basic classes that are used in order to store the outlines are the CF_Bézier and CF_Branches that hold the required information for the Bézier Curves and the Branches respectively.

6.3.1 CF_Bézier

CF_Bézier is a class that is used for the representation of Bézier Curves. Below, the private members of the class are presented. As it can be seen the input Bézier Curve points are stored into a vector called m_vPoints. The analytical points of the curve are calculated and stored in a vector of CF_POINT objects. Moreover there are bool variables that declare whether a Curve is open, or it is a circle curve, or the elastic mode of the curve is on.

**Private Attributes**

<table>
<thead>
<tr>
<th>Vector&lt;BEZIER_PT&gt;</th>
<th>m_vPoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector&lt;CF_Point&gt;</td>
<td>m_vAnaPoints</td>
</tr>
<tr>
<td>Vector&lt;bool&gt;</td>
<td>ConPointsChangeMap</td>
</tr>
<tr>
<td>double</td>
<td>m_fPrecision</td>
</tr>
<tr>
<td>bool</td>
<td>m_bCurveOpen</td>
</tr>
<tr>
<td>bool</td>
<td>m_bCircleCurve</td>
</tr>
<tr>
<td>bool</td>
<td>mbFastBezier</td>
</tr>
<tr>
<td>bool</td>
<td>m_bEnableUpdate</td>
</tr>
<tr>
<td>CF_BezierStyle</td>
<td>m_BStyle</td>
</tr>
<tr>
<td>CF_Rect</td>
<td>m_BoundRect</td>
</tr>
<tr>
<td>CF_Rect</td>
<td>m_BoundRectExact</td>
</tr>
<tr>
<td>CF_Rect</td>
<td>m_ChangesRect</td>
</tr>
<tr>
<td>bool</td>
<td>m_bElasticModeOn</td>
</tr>
<tr>
<td>CF_Point</td>
<td>m_oldEIPt</td>
</tr>
<tr>
<td>BEZIER_PT</td>
<td>m_oldPt1</td>
</tr>
<tr>
<td>BEZIER_PT</td>
<td>m_oldPt2</td>
</tr>
<tr>
<td>int</td>
<td>m_ELCurvIndex</td>
</tr>
<tr>
<td>double</td>
<td>m_EL_ParamT</td>
</tr>
</tbody>
</table>

CF_Bézier has numerous of member functions that are used for the calculation and modification of the Bézier Curves as well as the setting and the retrieval of information concerning them.

6.3.2 CF_Branches

CF_Branches is a class that is used to represent a branch object. The private members are the ones shown below. There are used two CF_Bézier objects that store the two
Bézier Curves of the branch and a vector of pairs of points (objects of \textit{CF\_PointEx}) that define the connecting lines of the branch.

\textbf{Private Attributes}

<table>
<thead>
<tr>
<th>std::vector&lt; std::pair&lt; CF_PointEx, CF_PointEx &gt; &gt; m_vLines</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF_Bezier = m_Bez1</td>
</tr>
<tr>
<td>CF_Bezier = m_Bez2</td>
</tr>
<tr>
<td>double m_fPrecision</td>
</tr>
<tr>
<td>bool m_bCircleCurve</td>
</tr>
<tr>
<td>bool m_bEnableUpdate</td>
</tr>
<tr>
<td>CF_BezierStyle m_BStyle</td>
</tr>
<tr>
<td>bool m_bAutoInvertLines</td>
</tr>
</tbody>
</table>

The class has the public member functions that are shown below, which provide all the functionality regarding the branches.

The \textit{SetCurvesUserAndControlPoints} is the function that calculates the branch by passing to it as parameters the user and control points of the curve.
Public Member Functions

virtual ~CF_Branches ()

void SetPrecision (double precision)

void SetOpenCurve (bool b)

void SetCircleCurve (bool b=true)

void SetStyle (CF_BezierStyle bs)

bool AddLine (const CF_PointEx &p1, const CF_PointEx &p2)

bool ModifyLine (unsigned int index, CF_PointEx &p1, CF_PointEx &p2)

bool InsertLine (CF_PointEx &p, int &index, double dist=CF_BEZIER_PRECISION)

bool RemoveLine (unsigned int index)

void AddExtraAnaPointsToFirstBezier (vector< CF_Point > &p, CF_Point po)

void AddExtraAnaPointsToSecondBezier (vector< CF_Point > &p, CF_Point po)

void SetCurvesUserAndControlPoints (const vector< CF_PointEx > &UserPts1, const vector< CF_Point > &CtrlPts1, const vector< CF_PointEx > &UserPts2, const vector< CF_Point > &CtrlPts2)

bool ReverseCurve ()

double GetPrecision () const

bool IsCircleCurve () const

CF_BezierStyle GetStyle () const

unsigned int NumLines () const

bool GetLine (unsigned int index, CF_PointEx &p1, CF_PointEx &p2) const

void GetFirstCurveAnaPoints (vector< CF_Point > &v_p) const

const std::vector< CF_Point > & GetFirstCurveAnaPoints () const

void GetSecondCurveAnaPoints (vector< CF_Point > &v_p) const

const std::vector< CF_Point > & GetSecondCurveAnaPoints () const

bool GetAnaPointsFromIndex (unsigned int from_index, unsigned int to_index, int side, vector< CF_Point > &v_p) const

void GetFirstCurvePoints (vector< CF_PointEx > &v_p) const

void GetSecondCurvePoints (vector< CF_PointEx > &v_p) const

void GetFirstCurveConPoints (vector< CF_Point > &v_p) const

void GetSecondCurveConPoints (vector< CF_Point > &v_p) const

int GetClosestAnaPoint (const CF_Point &TestPt, CF_Point &NearestPt, int &index, double dist=CF_NUMBER_MIN_DIFF)

CF_Rect BoundRect ()

CF_Rect BoundRectExact ()

void EnableUpdate (bool b=true)

bool IsValid () const

bool IsInvalid () const

bool InvalidateConPoints ()

bool IsClosed () const

void Clear ()

void EnableAutoInvertLines (bool b=true)

bool IsInsideCurve (const CF_Point &p)

const CF_Bezier * GetFirstBezier () const

const CF_Bezier * GetSecondBezier () const

CF_Bezier * GetFirstBezier ()

CF_Bezier * GetSecondBezier ()
6.3.3  CF_BranchesEditTool

The class responsible for the editing of the branches is the *CF_BranchesEditTool* class. The public member functions are presented below.

### Public Member Functions

```cpp
CF_BranchesEditTool ()
virtual ~CF_BranchesEditTool ()
void SetView (CDibView *p_view)
void SetPresenter (Presenter *p_pres)
bool AttachObject (CF_Branches *pbranches)
void ActivateInputMode (bool b)
bool IsInputModeActive () const
void ActivateControlPointsEditing (bool b)
void ActivateEditAtCreation (bool b)
void RemoveCurrentLine ()
void SetBranchesOpen (bool b=true)
bool IsBranchesOpen () const
bool InsertPoint ()
bool DeletePoint ()
bool HasEnoughLinesToDelete () const
bool HasEnoughLinesToInsert () const
BOOL OutsideBranch ()
void OnStart ()
void OnFinish ()
void ProtectedDraw ()
void Reset ()
bool InterestedInMousePos (CF_Point &pt)
void LButtonDown (CF_Point &pt)
void MouseMove (CF_Point &pt)
void LButtonUp ()
void CtrlDown ()
void DelDown ()
void InsertDown ()
void KeyUp ()
void Update (unsigned int action, CF_Point &p="new CF_Point")
```

This class is connected with the class Presenter as the latter provides the interface for the drawing of the branches to the screen. Therefore, an object of Presenter and an object of CDibView (which is described in the next section) are set.

The function that holds the event functionality is the *Update*. This class takes as an argument the action of the user which can be a mouse or a keyboard event. The second argument is optional and is needed for the mouse events as it declares the point of the mouse at the moment of the event. The *Update* function calls different functions depending on the action of the user, that are the *LButtonDown*, the *LButtonUp*, the *MouseMove*, the *DelDown*, the *CtrlDown*, the *InsertDown* and the *KeyUp*. 
6.3.4 Presenter

The class Presenter provides an interface for drawing on the screen. Below are shown the public member functions and the private attributes of the class.

**Public Types**

```
enum PenStyle { PEN_SOLID, PEN_DOT, PEN_NULL }
enum BrushStyle { BRUSH_SOLID, BRUSH_DOT, BRUSH_NULL }
```

**Public Member Functions**

```
Presenter ()
~Presenter ()
HDC GetDrawingDC ()
void SetDrawingDC (HDC hdc)
void AttachView (CDibView *pview)
void SetPenMode_XOR ()
void SetPenMode_NOT ()
void SetPenMode_Normal ()
void SetPenBkMode_Transparent ()
void SetPenBkMode_Opaque ()
void SetPenInfo (PenStyle ps, unsigned int width, int red, int green, int blue)
void SetBrushInfo (BrushStyle bs, unsigned int red, unsigned int green, unsigned int blue)
void SetControlPointMode (bool bMode)
bool GetControlPointMode () const
void DrawMoveCenterHandle (const CF_Point &ln)
void DrawFocusRectangle (const CF_Point &point, unsigned int size=10)
void DrawArea (const CF_Banches &br, bool bedit, unsigned short handle_size, bool buserpoints)
void DrawArea (const CF_Bezier &bez, bool bedit, unsigned short handle_size, bool buserpoints)
```

**Private Attributes**

```
CDibView = _m_pView
HDC  _m_hdc
COLORREF  _m_PenColor
HPEN  _m_hPen
HPEN  _m_hOldPen
PenStyle = _m_PenStyle
unsigned int  _m_nPenWidth
COLORREF  _m_BrushColor
HBRUSH  _m_hBrush
HBRUSH  _m_hOldBrush
BrushStyle = _m_BrushStyle
bool _m_bControlPointMode
```

In particular the drawing device context can be set. Then, different pen modes can be applied to it. The drawing mode can be \( R_2\_XOR \), \( R_2\_NOT \), or \( R_2\_COPYPEN \). Furthermore the essential info for the brush and pen can be set concerning the style, the colour and the width of the pen. Moreover, the class enables the drawing of the branch and the Bézier curve (through the overloaded function DrawArea), the drawing of the centre handle of the branch and the drawing of the focus rectangle.
For the pen and brush style two enumerators are used to declare them, which enable them to be *SOLID*, *DOT*, and *NULL*.

The **AttachView** is the function that connects the current view with the Presenter. As shown in the following code, this is where the information about the pen and brush are declared.

```cpp
void Presenter::AttachView( CDibView* pview )
{
    m_pView = pview;
    SetPenInfo( PEN_SOLID, 0, 255,255,255 );
    SetBrushInfo( BRUSH_NULL, 255,255,255 );
    SetPenMode_Normal();
}
```

The **DrawArea** is another significant function of the class. It draws a certain branch on the screen. The branch is passed as a parameter to it. Furthermore, two bool variables are passed to the function to declare whether the control and user points should be also drawn. As it was noted in the previous section, when the branch is not active, only the outline and the reference lines are Pictured, while if it is the active one the user points with the corresponding control points are depicted. Also, the handle size –in pixels, is also given to the function as a parameter.

### 6.4 The MFC Classes

The MFC document/view architecture is followed in the project. The main MFC classes are the CView, CDocument and CMainFrame. In the project the CView is inherited by CDibView and CDocument by CDibDoc. Furthermore, CMainFrame, which inherits the CMDIFrameWnd class, is also created automatically by the MFC project creation wizard.

#### 6.4.1 CMainFrame

Below are displayed the public and protected member functions of the CMainFrame class.
In Mainfrm.cpp there are variables that are defined globally and are used by other files and classes of the project.

Two of them are buffers that are used to hold all the essential information about the font set that is being processed and store the characters that are edited and their outlines. These are the `CHARACTERS` and the `ALLBRANCHES`. The first one is a vector of pointers to char variables and the second one is a vector of vectors of `CF_Branches` objects.

```cpp
std::vector<unsigned char *> CHARACTERS;
std::vector< std::vector< CF_Branches > > ALLBRANCHES;
```

In the `CHARACTERS` vector the information about the characters along with the break lines and the stroke order lines are stored. Actually, each element of the `CHARACTERS` vector is essentially a buffer that stores bytes of information serially. This information is organized in a certain way. Firstly, the total space needed in memory, for a certain branch, is calculated. This space is allocated dynamically and a pointer points to it.
Then the required information is copied to it. In particular, the actual character is the first information that is stored to the buffer. Then is the number of polylines (stroke order lines) followed by the points that define each polyline. Similarly, the number of breaklines is the next information that is stored, followed by the points defining each breakline. The result is that each element stores the character with its break lines and stroke order lines.

The \textit{ALLBRANCHES} is a vector that stores the branches of each character. In particular, each element of the vector is a vector of branches that corresponds to a certain character. The \textit{CHARACTERS} and the \textit{ALLBRANCHES} correlate with each other. Thus, the \textit{n$^{\text{th}}$} element of the \textit{ALLBRANCHES} stores the outline of the \textit{n$^{\text{th}}$} element of the \textit{CHARACTERS}.

\texttt{Mainfrm.cpp} also has functions that are declared in it. Some worth addressing are the following:

\begin{verbatim}
void CreateDocumentChar(int character, int prev_next);
BOOL LoadCHARACTERS (LPCTSTR lpszPathName);
BOOL SaveCHARACTERS (LPCTSTR lpszPathName);
\end{verbatim}

The function \textit{CreateDocumentChar} creates the document in which the character will appear and also triggers the method that creates the bitmap of the character which is the \textit{CreateInMemChar} and has the following syntax:

\begin{verbatim}
HBITMAP CreateInMemChar (int character, LOGFONT* p_lf, int ppm,
        int& BBMaxInMem, CRect& rc, int& offsetX, int& offsetY);
\end{verbatim}

The \textit{SaveCHARACTERS} is a function that saves the information stored in the two vectors to the disk files. Two files with the name given by the user are created with the Windows function \textit{CreateFile}. In the case that the files already exist, they are overwritten as \textit{CREATE ALWAYS} parameter is passed to \textit{CreateFile}.

Firstly general information about the application is saved to the .ffi file. This is a header includes the extension name of the file, the version of the program, the data offset and the number of characters that will be saved. The three first parameters are constant with usemark to be defined as “FFI”, the version is declared as 1 and the data offset is 1024. Following the font type is written to the file (eg. Arial). Then, the digit 0 is written repeatedly to the file until reaching the value of data offset. The data offset operates as a
delimiter for the beginning of the actual data. Finally, the contents of the vector *CHARACTERS* are written to the file.

Apart from the .FFI file, a file with the extension .OUT is created. This file does not include any additional information as a header. The first information that is saved is the number of branches, which is the number of characters that have outline information which has been edited and is not the default outline. Then for each branch the number of the first Bézier Curve user points is saved. Then for each point is saved the x coordinate, the y coordinate and the bool flag (0 or 1) whether the point is a corner. The same information with the same structure is saved for the second Bézier Curve. Following is the required information for the control points. In particular, the number of control points of the first Bézier Curve is written and for each control point the x and y coordinates are saved. Finally, the control points of the second Bézier Curve are saved in a similar way.

The *LoadCHARACTERS* function is the one that loads the characters from a given file. As it has been already mentioned, the allowed files that the application opens are those with extention .FFI—provided that there is also a *.OUT file in the same directory. So in this function the two files are read and the information is stored into the vectors *CHARACTERS* and *ALLBRANCHES*. The reading procedure is in accordance with the way that the information was written in *SaveCHARACTERS*.

### 6.4.2 CDibDoc

The document class stores and controls the data of the program. Each document represents the collection of data that the user usually opens and creates with the Open command and saves with the Save command.

The functions and attributes of CDibDoc are shown below.
A certain character with its font size is stored in the document class. The `BlackBoxMaxXY` integer variable represents the maximum bounding rectangle of the character. The values of the bounding rectangles are returned by the `GLYPHMETRICS` structure at Mainfrm.cpp. This information is important as the real dimensions of the character are extracted.

Finally the integer `m_OffsetXY` stores the bitmap offset of the character on the window. This value is defined at the Mainfrm.cpp. The offset is useful as the extra space is required when the user creates the break lines, whose ends should usually be placed outside of the character.
In Fig 31 is a painted example of the black box and the offset at a screenshot of a character. The bounding rectangle of the character ‘a’ of Arial font is the one painted with the green border. Moreover the red lines show the offset of the character up, down left and right from the window.

![Fig 31 – Bounding Rectangle and Offset](image)

### 6.4.3 CDibView

The view class is one of the most important functions of the program. It is responsible for the presentation of the data and also provides the functionality for the user interaction; it handles the mouse and keyboard events.

As described above there are 3 basic modes supported by the application. These modes are the break-lines mode, the stroke order lines mode and the outlines mode. Three structures are used to store the information needed for each mode. Below are shown the types that were created.
The type **POLYLINES** is defined as a vector of pointers to a vector of objects of type **POINT**. The **POINT** is a structure of MFC Library that stores two long numbers that correspond to x and y coordinates. The **POLYLINES** is a structure that is used for the storing of the stroke order lines.

The type **BREAKLINES** is defined as a vector of pointers to a vector of objects of type **POINT** and is used to store the break-lines of the character.

Finally, the type **OUTLINE** is defined as a vector of objects of **CF_Branches**.

The public attributes of the class are shown below.

**Public Attributes**

```
typedef std::vector< POINT > POLYLINE;
typedef std::vector< POLYLINE * > POLYLINES;
typedef std::vector< POINT > BREAKLINE;
typedef std::vector< BREAKLINE * > BREAKLINES;
typedef std::vector< CF_Branches > OUTLINE;
```

The m_polylines, m_breaklines and m_outlines are used to store the polylines, breaklines and branches of the character that is being processed on the present view. The m_ppolyline and m_pbreakline are used to store the current polyline and breakline that are processed by the user. Finally, the m_outlines_order is a vector that stores the order that corresponds to the branches stored in the m_outlines.

The private attributes of the class are the following.
Private Attributes

```c
enum MouseMode { MMD_NORMAL, MMD_LDN };
enum LineMode { LMD_STROKES, LMD_BREAKS, LMD_CURVES };
```

The `m_mousedown` declares the state and can be either normal or with the left button down. It takes values from the enumerator `enum MouseMode`.

```c
enum enum_MouseMode { MMD_NORMAL=0, MMD_LDN };
```

The `m_linenode` defines the mode that has been selected. It also takes values from an enumerator.

```c
enum enum_LineMode { LMD_STROKES=0, LMD_BREAKS, LMD_CURVES };
```

Moreover the `m_activepolyline`, the `m_activebreakline` and the `m_activeoutline` store the index of the active stroke line, the active break line and the active outline respectively.

Two objects, related to the editing of the outlines and their appearance on the screen, are the `editBr` and the `mPresenter`.

The constructor of the class is the following:

```c
CDibView::CDibView()
{
    m_ppolyline=NULL;
    m_pbreakline=NULL;
    m_mousemode=MMD_NORMAL;
    m_linemode=LMD_BREAKS;
    m_activepolyline=-1;
    m_activebreakline=-1;
    m_activeoutline=-1;
    m_haschanged=0;
    m_brk_handle_clicked=-1;
    mvoffs.x=mvoffs.y=0;
    branch_calculated = false;
    CF_BranchesEditTool editBr;
    Presenter mPresenter;
    IsPrint = 0;
}
```
Also, the bool variable *branch_calculated* declares whether the calculation of the branch has been made so as not to recalculate it every time the user selects the outline mode – unless changes have been applied to the break lines.

The bool variable *m_haschanged* declares if any kind of editing has been made to the character since the last storing. When the storing is made the variable is set to 0 again.

The *IsPrint* variable states whether the device-context is the printer, as different scaling should be applied when the drawing is not on screen and pixel coordinates are not used.

The private and public functions are the ones Pictured below.

**Private Member Functions**

```cpp
void DrawBreakline (BREAKLINE *abreakline, CDC *pDC)
void ClearLineBuffers ()
void RedisplayChar ()
void UpdateTitle ()
BOOL IsClickingBreakline (const CPoint &point)
double DistancePointFrom2PointsLineInt (int x1, int y1, int x2, int y2, int x3, int y3)
bool RestrictLineInRect (CIProBreakLine &brk_line, const CRect &rc)
bool MoveLineInsideRect (CIProBreakLine &brk_line, const CRect &rc)
void CalculateBranches ()
void OrderOutlines ()
void PrintPreview ()
```

**Public Member Functions**

```cpp
CDibDoc * GetDocument ()
void SetLinesData ()
void GetLinesData ()
void DoCharStore ()
virtual ~CDibView ()
virtual void OnDraw (CDC *pDC)
virtual void OnInitUpdate ()
virtual void OnActivateView (BOOL bActivate, CView *pActivateView, CView *pDeactiveView)
virtual void OnUpdate (CView *pSender, LPARAM lHint, CObject *pHint)
void TenMM2Pix (CF_Point &point)
void Pix2TenMM (CF_Point &point)
void Pix2TenMM (unsigned int &p)
bool GetPrint ()
```
The OnDraw function is responsible for the drawing of the characters along with the 3 different kinds of lines on the screen.

The structure of the method is based on the mode that is selected and it is presented with the following pseudo-code:

```c++
if (m_linemode<>LMD_CURVES)
{
    //set colour info

    //setup display properties for arrows

    for each m_polyline
    {
        // set the colour to be the general one
        if (current=m_ppolyline =m_activepolyline & m_linemode=LMD_STROKES)
            //set the colour to be the active one

        //set pen and brush info
        //transform the points to pixel coordinates

        if(m_linemode=LMD_STROKES )
            //draw the polyline

            if( this=m_activepolyline && m_linemode==LMD_STROKES )
                //draw the middle and end rectangle
            else
                //draw the end rectangle

            if( m_linemode=LMD_STROKES )
                //draw the index number of the line
    }

if (m_linemode<>LMD_CURVES)
{
    for each m_breakline
        //make the colour the general one
    if( this = m_activebreakline & m_linemode=LMD_BREAKS )
        //make the colour the active one

    //set pen and brush info

    if( m_linemode==LMD_BREAKS )
        draw rectangles at the two ends

    if( this=m_activebreakline & m_linemode=LMD_BREAKS )
        draw the central rectangle
```
The way that the OnDraw is structured is to result in having different views depending on the mode that has been selected by the user. Therefore, if the mode is the break lines the break lines are displayed on the screen with the end point rectangles and having index numbers and the stroke order lines appear as simple arrows. The reverse outcome is when the mode that is on is the stroke order curves. Yet, the outlines are shown only if the curves mode is selected and in this case both the break lines and the stroke order curves are hidden.

The procedure to show the branches on the screen is more complex and thus is explained in more detail. The first thing to do is to calculate the branches if they are not already calculated. This is done through the CalculateBranches function. Then the style of the pen is set to be R2_COPYPEN and the colour of the outline is defined to be green, whereas the active outline is red.

As it was noted above, the outlines are stored into the m_outlines vector. Each member of the vector is a branch object. Then for each branch, the settings of the pen and branch are defined and the drawing device context is set to the mPresenter. Then the branch is drawn and if it is the active one, the control points along with the end points are drawn too. Moreover, for the active branch the ProtectedDraw function is called which sets the pen and brush information and draws the branch, through the DrawArea function, with enabled the drawing of the user and control points. This is done, as only the active branch can be edited and therefore it is needed for the points to be displayed.

Finally the centre point of the branch is found and there is printed the index number of the branch starting from 1.

The code that follows implements the above procedure:
int oldmode = pDC->SetROP2(R2_COPYPEN);
DWORD RGB_general = RGB(0,255,0); // GREEN
DWORD RGB_active = RGB(255,0,0); // RED
DWORD RGB_value;
char buffer[128];

int NP = m_outlines.size();
int active = NP - 1;
RGB_value = RGB_general;
unsigned short handle_size = 3;

for(int i = 0; i < NP; i++)
{
    RGB_value = RGB_general;
    if( i == m_activeoutline) RGB_value = RGB_active;
    CPen MyPen(PS_SOLID,0,RGB_value);
    CPen* pOldPen = pDC->SelectObject(&MyPen);
    CBrush MyBrush(RGB_value);
    CBrush* pOldBrush = pDC->SelectObject(&MyBrush);

    mPresenter.SetDrawingDC(pDC->GetSafeHdc());
    if (i != m_activeoutline)
        mPresenter.DrawArea(m_outlines[i],0,handle_size,0);
    else
        {
            mPresenter.DrawArea(m_outlines[i],1,handle_size,1);
            editBr.ProtectedDraw();
        }

    CF_Point p = m_outlines[i].BoundRectExact().GetCenter();
    TenMM2Pix(p);
    sprintf(buffer,"%d",m_outlines_order[i]+1);
    pDC->TextOut( p.m_x,p.m_y, buffer, strlen(buffer));
    pDC->SelectObject(pOldBrush);
    pDC->SelectObject(pOldPen);
}
pDC->SetROP2(oldmode);

TenMM2Pix and Pix2TenMM are two important functions related to the primitives of the embroidery functionality. The smallest measurable unit that an embroidery machine can identify is 0.1 mm; hence the embroidery designs in the application are measured in 1/10 mm. However, should we wish to print such a draw on the screen, the appropriate scaling must be made to convert this quantity to pixels. Similarly, when the user
modifies the outlines on the screen, the pixel coordinates should be transformed on the real ones with the reverse procedure.

Two more functions of high importance are the \texttt{GetLinesData} and \texttt{SetLinesData}.

\textbf{SetLinesData} is called in \texttt{CreateDocumentChar} on \texttt{Mainfrm.cpp} and it is used to copy the data from the global vectors \texttt{CHARACTERS} and \texttt{ALLBRANCHES} to the vectors \texttt{m\_polylines}, \texttt{m\_breaklines} and \texttt{m\_outlines}. So, when a new document is created, the \texttt{SetLinesData} is called –which checks whether there exists stored information of the character in question– and if so, it presents it. Also the \textit{branch\_calculated} variable is set to be true if outline information exists and is shown.

\textbf{GetLinesData} is called when the storing command is executed and it’s doing the reverse procedure that \texttt{SetLinesData} does; the information from the local vectors is copied to the global vectors. The result is to keep in the vectors \texttt{CHARACTERS} and \texttt{ALLBRANCHES} the changes that have been made to the different kind of lines. Furthermore, it calls the function \texttt{OrderOutlines} which orders the branches according to the given order that was set by the user and is stored in the vector \texttt{m\_outlines\_order}.

The \texttt{OrderOutlines} is changing the position of the branches that are stored in \texttt{m\_outlines} and reorders them inside the vector according to the values of the \texttt{m\_outlines\_order}. Hence if the first value in the \texttt{m\_outlines\_order} is 5, the first branch of the \texttt{m\_outlines} vector will result in the 5\textsuperscript{th} position in the vector.

The function responsible for the event handling of the mouse clicks is the \texttt{OnLButton\_Down} and \texttt{OnLButton\_Up}. The first one is activated when the user clicks the right mouse button and the second is activated when the user releases the button. On the \texttt{LButton\_Down}, function the appropriate commands are executed depending on the mode that is selected. If the break lines mode is on, unless the user is clicking on an existing breakline, a \texttt{BREAKLINE} –\texttt{m\_breakline}– is created. It also stores in the vector the point of the mouse when the user clicked. Similarly, if the mode is the stroke order lines, a new \texttt{POLYLINE} –\texttt{m\_polyline}– is created, storing again the mouse point. Finally, if the mode is the outlines one, the handling proceeds to the \texttt{BranchesEditTool} class through the \texttt{editBr} object.

The \texttt{OnLButton\_Up} creates the appropriate lines according to the mode. In the break lines and stroke order lines, the line is calculated by taking into account the first point that is already stored in the corresponding vector, and the current point that is the point
that the user released the mouse button. The resulting line is Pictured on the screen. Again if the mode is the outlines one, the the handling proceeds to the BranchesEditTool.

The way that the class CF_BranchesEditTool and the class Presenter operate can be made clearer with the following segments of code:

To start with, the mPresenter is updated in OnInitialUpdate as the current view is attached to it.

```cpp
void CDibView::OnInitialUpdate()
{
    ....
    ....
    mPresenter.AttachView(this);
}
```

Also, when the outlines mode is selected, the branches of the character are shown, and the first one is made active – meaning that its knots are presented and it can be edited. The following segment of code shows how the editBr object is related to a certain branch and how is connected with the Presenter.

```cpp
......
editBr.AttachObject(&m_outlines[0]);
editBr.OnStart();
editBr.SetPresenter(&mPresenter);
......
```

So, generally, when the outlines mode is selected and the user clicks inside of a branch, this branch is attached to the editBr and the OnStart function is called, which initializes the required variables. When the user selects another branch, then this one is made active and the OnFinish function is called before the new branch is attached to the editBr.
7 Conclusions and Further Work

During my MSc thesis a Web-based design Framework for Shapes Outline which is used for the digitization of true type font characters has been developed. This application is especially useful for East-Asian language characters as the way those characters are embroidered is crucial in order for them to be legible.

The user can access through the Web a tool that creates shapes’ outlines. In particular it gives access to a design tool application which processes any True Type Font available on the Windows Operating System and following, edit the characters so that the resulting outline best fits the contour of the character in hand. The application is easy to use by users with the requirement of knowing the writing rules of the characters that are processed and the fundamental rules of the embroidery procedure.

The application has the desired functionality and the user interface is friendly and direct. Furthermore, complex character sets such as those of the East-Asian Languages can be embroidered with improved quality.

In the future the application could be further extended in several aspects. Firstly, the incorporation of additional information regarding the embroidery design is an important extension that would make the application a stand-alone one, as the resulting outcome would be directly imported into the embroidery machine with no further processing. Such information is for instance the type, the position and the number of stitches. The result would be a complete font system providing all the required functionality for the embroidery process of a font.

An extension regarding the Web-based framework is the implementation of applications that would enable the creation of outlines of different shapes and not exclusively of font characters, which could be accessed through the World Wide Web. Moreover, it could also be supported the deployment of a Web Service that would communicate with the applications in a client-based architecture (eg. SOA).
References

Appendix

Reference Code
Below is part of the code that was commented in the above sections.

```cpp
void CDibView::OrderOutlines ()
{
    std::vector <CF_Branches> temp;

    int pos;
    int size = m_outlines_order.size();
    for (int i=0; i<size; i++)
    {
        for(int j=0; j<size ; j++)
        {
            if (i==m_outlines_order[j])
            {
                pos = j;
                break;
            }
        }
        temp.push_back(m_outlines[pos]);
    }

    m_outlines.clear();
    m_outlines = temp;

    for( i=0; i<size; i++)
        m_outlines_order[i]=i;
}

void CDibView::OnLButtonDown(UINT nFlags, CPoint point)
{
    if(m_mousemode!=MMD_NORMAL) return;

    POINT p;
    p.x=point.x; p.y=point.y;

    if(m_linemode==LMD_STROKES)
    {
```
//adds a new polyline to m_ppolyline vector

else if(m_linemode==LMD_BREAKS)
{
    //if user clicked on an existing Breakline, it is made active
    //else adds a new breakline to m_pbreakline vector
}
else //LMD_CURVES mode is selected
{
    CF_Point cp(point.x, point.y);
    editBr.SetView(this);
    editBr.Update(1,cp);

    if ( editBr.OutsideBranch() )
    {
        Pix2TenMM (cp);
        for (int i=0; i<m_outlines.size(); i++)
        {
            if (m_outlines[i].IsInsideCurve(cp))
            {
                editBr.OnFinish();
                m_activeoutline = i;
                editBr.AttachObject(&m_outlines[i]);
                editBr.OnStart();
                break;
            }
        }
    }
}

m_mousemode=MMD_LDN;

void CDibView::Pix2TenMM(CF_Point &point)
{
    int BBMax=GetDocument()–>GetBlackBoxMaxXY();
    int offs=GetDocument()–>GetOffsetXY();
    int Fontsize = GetDocument()–>GetFontSize(); //
    double ratio = (double)Fontsize / (double)TT_EMB_FONT_HEIGHT;

    point.m_x -= offs;
    point.m_y = BBMax - point.m_y -offs*(ratio-1);
    point.m_x = point.m_x/ratio;
    point.m_y = point.m_y/ratio;
}
void CF_BranchesEditTool::ProtectedDraw()
{
    m_pPresenter->SetPenMode_Normal();
    m_pPresenter->SetPenInfo( Presenter::PEN_SOLID, 0, 255,0,0 );
    m_pPresenter->SetBrushInfo( Presenter::BRUSH_SOLID, 255,0,0 );

    if( m_pBranches )
        m_pPresenter->DrawArea(*m_pBranches, true, mnHandleSize, true);

    if( mbActivatedMoveCenterMode && HasEnoughLinesToInsert() )
        m_pPresenter->DrawMoveCenterHandle( m_pBranches->BoundRectExact().GetCenter() );

    m_pPresenter->SetBrushInfo( Presenter::BRUSH_NULL, 255,0,0 );
    m_pPresenter->SetPenMode_Normal();

    DrawCurrentPointFocusRect();
}

void CDibView::TenMM2Pix(CF_Point &point)
{
    int BBMax=GetDocument()->GetBlackBoxMaxXY();
    int offs=GetDocument()->GetOffsetXY();
    int Fontsize = GetDocument()->GetFontSize();
    double ratio = (double)Fontsize / (double)TT_EMB_FONT_HEIGHT;

    point *= ratio;
    point.m_x += offs;
    point.m_y = BBMax -point.m_y -offs*(ratio-1) ;

    if (IsPrint)
{
        point.m_x = point.m_x * PrintScale;
        point.m_y = point.m_y * PrintScale;
    }
}