Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics

Application of Bézier curves in Computer-Aided Design

Verslag ten behoeve van het
Delft Institute for Applied Mathematics
als onderdeel ter verkrijging

van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE

door

Jeroen Dekkers

Delft, Nederland
November 2010

BSc verslag TECHNISCHE WISKUNDE

"Application of Bézier curves in Computer-Aided Design"

Jeroen Dekkers

Technische Universiteit Delft

Begeleider

Dr. F. Vallentin

Overige commissieleden

Prof.dr.ir. K.I. Aardal

Prof.dr.ir. A.W. Heemink

Dr. G.F. Ridderbos

November, 2010

Delft

Contents

1 Introduction **1**

2 Bézier curves **2**
2.1 Integral Bézier curves 2
2.2 Rational Bézier curves 4

3 Bézier curve algorithms **6**
3.1 De Casteljau algorithm 6
3.2 Splitting a curve into two curves 7
3.3 Rendering a curve 9
3.4 Intersecting two curves 10

4 Areas **12**
4.1 Determining All Areas 12
4.2 Finding the right area 15
4.3 Computing area size 15

5 Implementation **17**
5.1 Python and QT toolkit 17
5.2 The program 17
5.3 Classes 18
5.4 Screenshots 20

6 Conclusion **23**

Chapter 1

Introduction

This bachelor thesis was written because a software company asked the Delft University of Technology for advice. The company is the developer of a professional computer aided design program made for housing developers and carpenters to design windows, frames, and doors. It allows to plan the complete process of making windows, frames and doors: starting from the first design and ending with the prize calculation of the final product.

The current version of their program has only the possibility to model with rectangular shapes. Now the company wants to be able to model more complicated shapes, such as diagonal lines and curves. For example their program is able to model a window like the one the left of Figure 1.1, but not the one the right of Figure 1.1. In this project we look at whether Bézier curves can be used as the mathematical basis for a new version of their program.

During this project we have developed and implemented algorithms for drawing Bézier curves, for finding their intersection points, for recognising connected regions and for calculating their area. Furthermore we created a user-friendly interface so we could test and show that our algorithms work. Using this interface it is possible to create standard objects such as straight lines and circular arcs and it will automatically detect intersection points and areas. With this is possible to draw complex figures, where the points where frames connect are automatically calculated. The program also knows where the areas are, so user would be able select an area with the mouse and set certain properties of that area.

Figure 1.1: Example windows

Chapter 2

Bézier curves

Bézier curves were invented by Pierre Bézier and Paul de Casteljau around 1960. The primary application was in the automobile industry back then, but today they are widely used in computer programs to draw curves. More information about Bézier curves can be found in [1].

2.1 Integral Bézier curves

Integral Bézier curves are parametric polynomial curves. So the formula of the curve is a polynomial and that they are parametric means that a Bézier curve is given as a graph of a function depending on a parameter $t \in [0, 1]$. The simplest version of a Bézier curve is a linear Bézier curve. One can see an example of such a curve in the left of Figure 2.1. The linear Bézier curve P(t) defined by the points $P_0, P_1 \in \mathbb{R}^2$ is the line segment connecting P_0 and P_1

$$\{P(t) = (1-t)P_0 + tP_1 \quad : \quad t \in [0,1]\}.$$

A quadratic curve is defined by three points P_0, P_1, P_2. We first create two lines between P_0 and P_1 and between P_1 and P_2 in the same way as the linear curve:

$$\begin{aligned} C(t) &= (1-t)P_0 + tP_1 \\ D(t) &= (1-t)P_1 + tP_2. \end{aligned}$$

Now we take the points $C(t)$ and $D(t)$ and calculate the point $P(t)$ which lies on the line from $C(t)$ to $D(t)$ for every $t \in [0, 1]$:

$$P(t) = (1-t)C(t) + tD(t).$$

You can see an example of such a curve in the middle of Figure 2.1. If we spell this out we get the following formula:

$$\begin{aligned} P(t) &= (1-t)C(t) + tD(t) \\ &= (1-t)((1-t)P_0 + tP_1) + t((1-t)P_1 + tP_2) \\ &= (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \end{aligned}$$

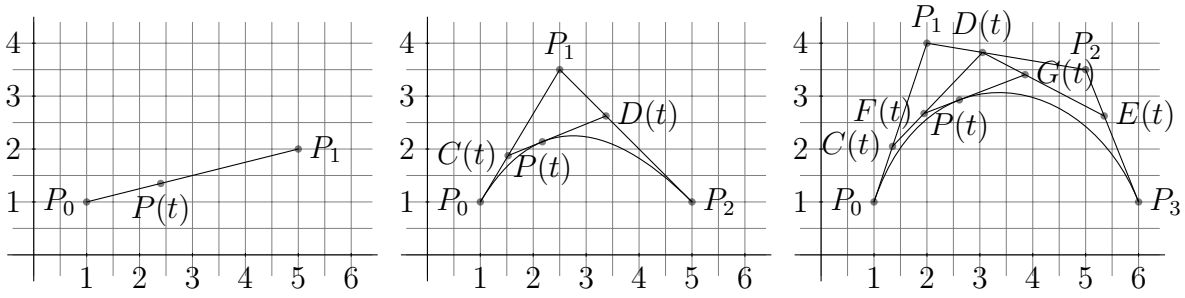

Figure 2.1: A Linear, Quadratic and Cubic Bézier curve

Defining a cubic Bézier curve is now easy, we just follow the same procedure, but with four points. An example of this is in the right of Figure 2.1. Here $F(t) = (1-t)C(t)+tD(t)$ and $G(t) = (1-t)D(t) + tE(t)$. Writing out the whole formula gives:

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

This way we can create a Bézier curve of any order n. Such a curve will be given by $n+1$ points, called *control points*.

Definition (Bézier curve)**.** *The general Bézier curve is defined by $n+1$ control points $P_0, P_1, \ldots, P_n$ and equals*

$$B(t) = \sum_{i=0}^{n} b_{i,n}(t) P_i.$$

Here $b_{i,n}(t)$ are the Bernstein polynomials or Bernstein basis functions of degree n defined by

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

And $\binom{n}{i}$ is the Binomial coefficient:

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}.$$

This formula is easy to remember: the polynomial $b_{i,n}(t)$ is the ith summand of the sum of the binomial theorem applied to $(1-t)$ and t:

$$((1-t)+t)^n = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i = \sum_{i=0}^{n} b_{i,n}(t).$$

Some examples of Bernstein polynomials are:

$$b_{0,2}(t) \;=\; \binom{2}{0} (1-t)^{2-0} t^0 = (1-t)^2$$

$$
\begin{aligned}
b_{1,2}(t) &= \binom{2}{1}(1-t)^{2-1}t^1 = 2(1-t)t \\
b_{2,2}(t) &= \binom{2}{2}(1-t)^{2-2}t^2 = t^2 \\
b_{1,3}(t) &= \binom{3}{1}(1-t)^{3-1}t^1 = 3(1-t)^2t.
\end{aligned}
$$

These kind of Bézier curves are also called *integral* Bézier curves to distinguish them from the rational Bézier curves, which we will discuss next. The polygon we get by joining the control points with line segments is called the *control polygon*. In the case of a quadratic Bézier curve we have only three control points and we get the triangle of those control points. One of the important properties of Bézier curves is that the whole curve always lies in the *convex hull* of the control points.

Definition (Convex hull)**.** *The convex hull of the set of points $X = \{x_0, x_1, \ldots, x_n\}$ is defined to be the set of points*

$$
\mathrm{CH}(X) = \left\{ a_0x_0 + a_1x_1 + \ldots + a_nx_n : \sum_{i=0}^{n} a_i = 1,\ a_i \geq 0 \right\}
$$

Theorem (Convex hull property)**.** *Every point of a Bézier curve lies inside the convex hull of its defining control points. Thus for all $t \in [0,1]$, $B(t) \in \mathrm{CH}(P_0, P_1, \ldots, P_n)$*

Proof. We need to show that every point $B(t)$ has the form $a_0P_0 + a_1P_1 + \ldots + a_nP_n$ for some a_i, with $\sum_{i=0}^{n} a_i = 1$ and $a_i \geq 0$. We can simply take $a_i = b_{i,n}(t)$. We have $b_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i \geq 0$ when $t \in [0,1]$. Applying the binomial theorem to $((1-t)+t)^n = 1$ gives

$$
((1-t)+t)^n = \sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i = \sum_{i=0}^{n} b_{i,n}(t) = 1.
$$

□

2.2 Rational Bézier curves

There are three types of quadratic curves: parabolas, ellipses and hyperbolas. The first type of curve, parabolas, can be parametrized only by polynomial functions, but ellipses and hyperbolas are parametrized by rational functions. Because integral Bézier curves have polynomial parametrizations, it is not possible to represent ellipses and hyperbolas. For this we need *rational* Bézier curves. With rational Bézier curves, each control point has an additional weight. We can see this weight has how strong the curve goes in the direction of a point compared to another. The formula for a rational Bézier curve is

$$
B(t) = \frac{\sum_{i=0}^{n} w_i b_{i,n}(t) P_i}{\sum_{i=0}^{n} w_i b_{i,n}(t)}
$$

Here $b_{i,n}$ are the Bernstein polynomials as before and w_i is the weight of the point P_i.

In the rest of the thesis, we only use quadratic rational Bézier curves. The reason for this is that we can create any quadratic curve (parabola, hyperbola, ellipse) we want to draw in our CAD program. It also simplifies the mathematics and the programming and is good enough for the application.

Circular arcs

Rational Bézier curves are needed for drawing circular arcs. The circular arc with radius r, centered at the origin with end points $(r, 0)$ and $(r\cos\theta, r\sin\theta)$, $\theta \in [-\pi, \pi]$, is given by the following control points and weights:

$$\begin{aligned} P_0 &= (r, 0) \\ P_1 &= (r, r\tan\frac{\theta}{2}) \\ P_2 &= (r\cos\theta, r\sin\theta) \\ w_0 &= 1 \\ w_1 &= \cos\frac{\theta}{2} \\ w_2 &= 1 \end{aligned}$$

To see that this is true, filling in the equation of the Bézier curve we get

$$\begin{aligned} x(t) &= \frac{r(1-t)^2 + 2rt(1-t)\cos\frac{\theta}{2} + rt^2\cos\theta}{(1-t)^2 + 2t(1-t)\cos\frac{\theta}{2} + t^2} \\ y(t) &= \frac{2rt(1-t)\sin\frac{\theta}{2} + rt^2\sin\theta}{(1-t)^2 + 2t(1-t)\cos\frac{\theta}{2} + t^2} \end{aligned}$$

Now we can show that $x(t)^2 + y(t)^2 = r^2$, but we skip the details here. From this we can conclude that the arc is circular.

We have to be careful with the arc $\theta = \pi$, because in that case P_1 is $(r, r\tan\frac{\pi}{2})$ and $\tan\frac{\pi}{2}$ is ∞. What happens in practice is that in our formulas P_1 is multiplied by w_1. If we do this calculation without first calculating the tangent, we will get $r\tan\frac{\theta}{2}\cdot\cos\frac{\theta}{2} = r\sin\frac{\theta}{2}$. For $\theta = \pi$ this means $r\sin\frac{\pi}{2} = r$.

Using this formula we can draw every circular arc. If we want to start at a different angle we can rotate the resulting figure around its centre and if we want to have a different centre we can use a translation.

Chapter 3

Bézier curve algorithms

There are many algorithms for Bézier curves known in the literature, such as those found in [1]. We will briefly discuss the algorithms used.

3.1 De Casteljau algorithm

To evaluate a Bézier curve at a specific $t \in [0,1]$ we can use the de Casteljau algorithm. A pseudocode implementation of the algorithm can be found is given in the box "Algorithm 1". The basic algorithm for an integral Bézier curve uses the following recursive formula:

$$\begin{aligned} P_i^0 &= P_i, \qquad i = 0, \ldots, n \\ P_i^j &= (1-t)P_i^{j-1} + tP_{i+1}^{j-1}, \quad j = 0, \ldots, n, \quad i = 0, \ldots, n-j \end{aligned} \tag{3.1}$$

Then P_0^n is the point of the curve at t, i.e. $B(t) = P_0^n$. If you compare the formula with our introduction of Bézier curves in the previous section, you'll see that we are actually doing the same thing, we only write it in a different way. Formula 3.1 gives a triangular set of points. In the case of a quadratic we have Bézier curve we have $B(t) = P_0^2$ and we get the following triangle:

$$\begin{array}{ccccc} P_0^0 & & P_1^0 & & P_2^0 \\ & P_0^1 & & P_1^1 & \\ & & P_0^2 & & \end{array}$$

We can compute rational curves in the same way by adding the weights to the formula:

$$\begin{aligned} P_i^j &= (1-t)\frac{w_i^{j-1}}{w_i^j}P_i^{j-1} + t\frac{w_{i+1}^{j-1}}{w_i^j}P_{i+1}^{j-1} \\ w_i^j &= (1-t)w_i^{j-1} + tw_{i+1}^{j-1} \end{aligned}$$

Algorithm 1 Calculating the point of a rational Bézier curve at parameter t

Input: Bézier curve C with control points P_0^0, P_1^0, P_2^0 and weights w_0^0, w_1^0, w_2^0, parameter $t \in [0, 1]$

Output: The point P of the curve C at parameter t

$w_0^1 \leftarrow (1-t)w_0^0 + tw_1^0$
$P_0^1 \leftarrow (1-t)(w_0^0/w_0^1)P_0^0 + t(w_1^0/w_0^1)P_1^0$
$w_1^1 \leftarrow (1-t)w_1^0 + tw_2^0$
$P_1^1 \leftarrow (1-t)(w_1^0/w_1^1)P_1^0 + t(w_2^0/w_1^1)P_2^0$
$w_0^2 \leftarrow (1-t)w_0^1 + tw_1^1$
$P_0^2 \leftarrow (1-t)(w_0^1/w_0^2)P_0^1 + t(w_1^1/w_0^2)P_1^1$
return P_0^2

3.2 Splitting a curve into two curves

The de Casteljau algorithm can also be used for subdividing a Bézier curve into two. The box "Algorithm 2" gives the pseudocode of the subdivision algorithm. We have the general integral curve $B(t)$ and want to split it into two Bézier curves at the value t to get two curves: $B_{\text{left}}(t)$ and $B_{\text{right}}(t)$. The left curve corresponds to the part of the original curve between 0 and t, the right curve corresponds to the curve between t and 1. We can do this by using the de Casteljau algorithm with parameter t, the control points for B_{left} are then P_0^0, P_0^1, ..., P_0^{n-1}, P_0^n and for B_{right} they are P_0^n, P_1^{n-1}, ..., P_{n-1}^1, P_n^0. One can see an example for a quadratic curve in Figure 3.1.

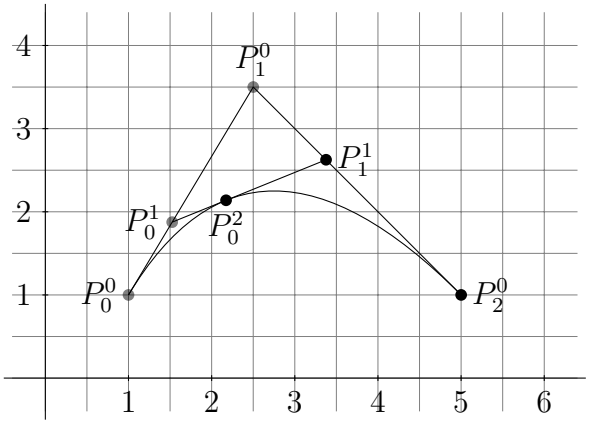

Figure 3.1: Splitting a quadratic Bézier curve

We will only prove that this algorithm is correct in the case of quadratic curves, but it is obvious that we can generalise this for any degree n. Writing out the de Casteljau formula for $t = \alpha$ gives us the following:

$$
\begin{aligned}
P_0^0 &= P_0 \\
P_1^0 &= P_1 \\
P_2^0 &= P_2 \\
P_0^1 &= (1-\alpha)P_0^0 + \alpha P_1^0 \\
P_1^1 &= (1-\alpha)P_1^0 + \alpha P_2^0 \\
P_0^2 &= (1-\alpha)P_0^1 + \alpha P_1^1 \\
&= (1-\alpha)((1-\alpha)P_0^0 + \alpha P_1^0) + \alpha((1-\alpha)P_1^0 + \alpha P_2^0)
\end{aligned}
$$

We will now show that P_0^0, P_0^1 and P_0^2 are indeed the control points for $B_{\text{left}}(t)$ when we split at $t = \alpha$. We do this by showing that the $B_{\text{left}}(t)$ is the same as $B(\alpha t)$ with $t \in [0, 1]$, i.e. $B(t)$ going from 0 to α.

$$
\begin{aligned}
B_{\text{left}}(t) &= (1-t)^2 P_0^0 + 2(1-t)tP_0^1 + t^2 P_0^2 \\
&= (1-t)^2 P_0^0 + 2(1-t)t((1-\alpha)P_0^0 + \alpha P_1^0) \\
&\quad + t^2((1-\alpha)((1-\alpha)P_0^0 + \alpha P_1^0) + \alpha((1-\alpha)P_1^0 + \alpha P_2^0)) \\
&= ((1-t)^2 + 2(1-t)t(1-\alpha) + \ t^2((1-\alpha)(1-\alpha))P_0^0 \\
&\quad + (2(1-t)t\alpha + t^2((1-\alpha)\alpha) + t^2(\alpha(1-\alpha)))P_1^0 \\
&\quad + \alpha^2 t^2 P_2^0 \\
&= (1-\alpha t)^2 P_0^0 + 2(1-\alpha t)\alpha t P_1^0 + (\alpha t)^2 P_2^0 \\
&= B(\alpha t)
\end{aligned}
$$

We can prove that P_0^2, P_1^1 and P_2^0 are the control points for $B_{\text{right}}(t)$ in the same way.

Algorithm 2 Splitting a curve at parameter t

Input: Bézier curve C with control points P_0^0, P_1^0, P_2^0 and weights w_0^0, w_1^0, w_2^0, parameter $t \in [0, 1]$

Output: Two curves C_1 and C_2 that result from splitting C at t

procedure SPLIT(C,t)

$w_0^1 \leftarrow (1-t)w_0^0 + tw_1^0$
$P_0^1 \leftarrow (1-t)(w_0^0/w_0^1)P_0^0 + t(w_1^0/w_0^1)P_1^0$
$w_1^1 \leftarrow (1-t)w_1^0 + tw_2^0$
$P_1^1 \leftarrow (1-t)(w_1^0/w_1^1)P_1^0 + t(w_2^0/w_1^1)P_2^0$
$w_0^2 \leftarrow (1-t)w_0^1 + tw_1^1$
$P_0^2 \leftarrow (1-t)(w_0^1/w_0^2)P_0^1 + t(w_1^1/w_0^2)P_1^1$
$C_1 \leftarrow (P_0^0, P_0^1, P_0^2; w_0^0, w_0^1, w_0^2)$
$C_2 \leftarrow (P_0^2, P_1^1, P_2^0; w_0^2, w_1^1, w_2^0)$
return C_1, C_2

end procedure

3.3 Rendering a curve

For rendering a Bézier curve we use the de Casteljau algorithm to subdivide the curve. The algorithm for rendering is the following:

Step 1: Split the curve at $t = 0.5$ using the de Casteljau algorithm to get B_{left} and B_{right}

Step 2: Check whether B_{left} is "near linear" (see below what we mean with that). If this is the case, go to step 3. Else go to step 1 and apply the algorithm to B_{left} and to B_{right}.

Step 3: Because the segment of the curve is near linear, we can approximate it using a straight line, so we draw a straight line between the starting point and end point of the segment.

A pseudocode for this algorithm is given in the box "Algorithm 3". The algorithm will recursively subdivide the curve into pieces that are near linear and draw the curve by approximating those segments with straight lines.

Algorithm 3 Render a curve

Input: Bézier curve C that needs to be rendered.
Output: No output, but side effect is that the curve C is rendered

```
procedure RENDER(C)
    C_1, C_2 ← SPLIT(C, 0.5)
    if NEARLINEAR(C_1) then
        Draw straight line from the starting point to end point of C_1
    else
        RENDER(C_1)
    end if
    if NEARLINEAR(C_2) then
        Draw straight line from the starting point to end point of C_2
    else
        RENDER(C_2)
    end if
end procedure
```

There are a lot of different ways to decide whether a given curve is close to being linear or not. We use the height of the control triangle to decide whether the curve is near linear. If height of the triangle is small, the distance between the edge of the triangle and the curve is also very small and we can conclude that the curve is near linear. We can very easily calculate the length using trigonometry, the height of a triangle is $a \sin \gamma$, where γ is the angle at the starting point of the Bézier curve and the a is the length of the line of the starting point to the control point. See Figure 3.2 for an illustration. Pseudocode

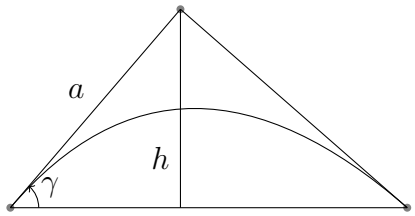

Figure 3.2: Height of the control triangle

of this algorithm can be found in the box "Algorithm 4". ATAN2 is the standard two-argument arctangent function as defined in [3] and LENGTH returns the distance between two points. $P.x$ means the x coordinate of P.

Algorithm 4 Calculate if a Bézier curve is near linear

Input: Bézier curve C with control points P_0^0, P_1^0, P_2^0
Output: True if the curve is near linear, False otherwise

```
procedure NEARLINEAR(C)
    angle ← ATAN2(P0.x − P1.x, P0.y − P1.y) - ATAN2(P0.x − P2.x, P0.y − P2.y)
    if angle > 180 then
        angle ← 360 - angle
    end if
    if angle > 90 then
        angle ← 180 - angle
    end if
    height ← LENGTH(P0, P1)(sin γ)
    if height < precision then
        return True
    else
        return False
    end if
end procedure
```

3.4 Intersecting two curves

The algorithm for calculating intersections of two curves is similar to the rendering algorithm:

Step 1: First we check whether the control triangles of the curves intersect. If the control triangles do not intersect, then also the curve do not intersect and we can stop here. This is so because a curve always lies inside its control triangle.

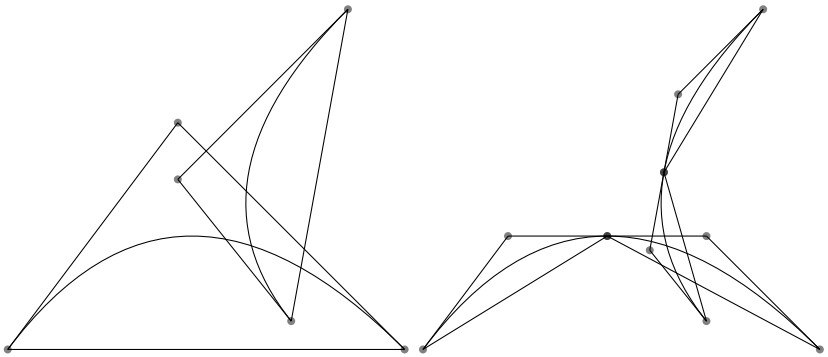

Figure 3.3: One iteration of intersection algorithm

Step 2: We check whether the curves are near linear. If this is the case, we go to step 3. Else we split both curves using the De Casteljau algorithm and start again with step 1 for each pair of resulting curves.

Step 3: If both curves are near linear, we can approximate them with a straight line. The intersection point is the point where the straight lines intersect, if they intersect.

Although this algorithm is recursive in nature, this does not mean that it is slow. The reason for this is that we can immediately remove the non-intersecting segments after we have split in step 1, as illustrated in Figure 3.3. This means that the algorithm approximates the intersection point (or points) quite fast in practice.

Algorithm 5 Calculating the intersection points of two curves

Input: Two Bézier C_1 and C_2
Output: The list of intersection points of the curves C_1 and C_2

```
procedure INTERSECT(C1, C2)
    if Control polygons of C1 and C2 don't overlap then
        return
    else if NEARLINEAR(C1) AND NEARLINEAR(C2) then
        return The intersection point of the straight line approximations of C1 and C2,
if it exists
    else
        C3, C4 ← SPLIT(C1, 0.5)
        C5, C6 ← SPLIT(C2, 0.5)
        return INTERSECT(C3,C5), INTERSECT(C3,C6), INTERSECT(C4,C5), INTER-
SECT(C4,C6)
    end if
end procedure
```

Chapter 4

Areas

There are a few different reasons why we need algorithms for areas presented in this chapter. Areas are one of the basic structures of the CAD program. A user needs to be able to select an area and tell whether the area is a piece of wood, a windows, etc. For this the program needs to know in which area a given coordinate lies in. For example for ordering materials and calculating the price of it we need to know the size of the area.

4.1 Determining All Areas

Algorithm 7 is used to identify every area of the drawing. This uses Algorithm 6 to find the next point while looping over points and curves. This works by precalculating the polar angles of all curves and having in each point a list of curves sorted by angle. We calculate the polar angle using the ATAN2 function (see giving x and y values of the distance between the point and the point on the other end of the curve as arguments. The procedure NEXTCURVE gets the current curve and point as arguments, looks up this curve in the sorted list and returns the next curve from the list.

Algorithm 6 Get next curve in a point when going clockwise around that point

Input: The previous curve C_p and point P that curves point to
Output: The next curve C_n when going clockwise around P starting from C_p

```
procedure NextCurve(C_p, P)
    Look up C_p in the sorted list of curves of P
    return The next curve in the list, or the first one if it is the last
end procedure
```

The main algorithm works by looping over all the points in the drawing. Each point is then used as the starting point of a potential new area. For each such starting point, we loop over all curves starting in that point. This way we will correctly find all the available areas in complex drawings. After selecting the first point and curve, we follow the curve to the other end point and select that point. Now we have to find the next curve of the

Algorithm 7 List all the available areas

Input: All points
Output: List of all areas
  for each point P **do**
    $P_{\text{start}} \leftarrow P$
    for each curve C in starting in P_{start} **do**
      $C_{\text{current}} \leftarrow C$
      $P_{\text{current}} \leftarrow$ the other end point of C
      $L \leftarrow$ list of P_{start}, C_{current} and P_{current}
      repeat
        $C_{\text{current}} \leftarrow \text{NextCurve}(C_{\text{current}}, P_{\text{current}})$
        $P_{\text{current}} \leftarrow$ the other end point of C_{current}
        Append C_{current} and P_{current} to the list L
      until $P_{\text{current}} = P_{\text{start}}$
      The points and curves in L are an area.
      Add this area to the list of areas if it is not already there. (This will be explained in the text)
    end for
  end for
  return The list of areas

area. We can do this by starting at the previous curve, going clockwise around the select point, and choosing the first curve we find. See Figure 4.1 for an example. We then follow that curve, do the same in the next point, etc. until we are back in the starting point.

It is obvious that with this procedure we will find the same area multiple times, so we need a method to compare the area with the areas we have found before. We do this by sorting the list of points and curves of every area in a such a way that it always starts with the leftmost point, and in the case there are multiple leftmost points, the topmost one. In this way we have a unique representation of the area and can quickly compare areas. One problem with the method is that we also find the area surrounding the whole drawing, because if we start at a certain point and curve we will loop over all the curves

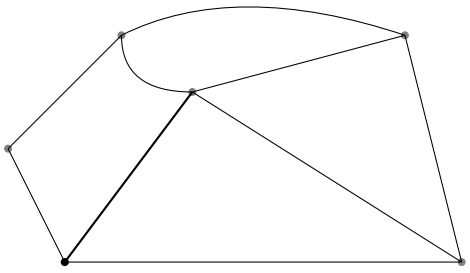

Figure 4.1: We start with a point and curve

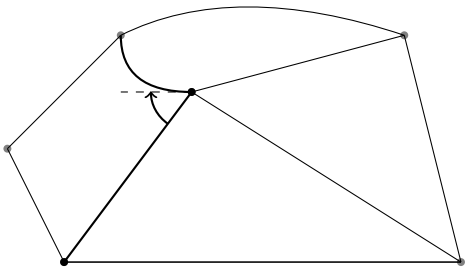

Figure 4.2: Decide which curve to take next

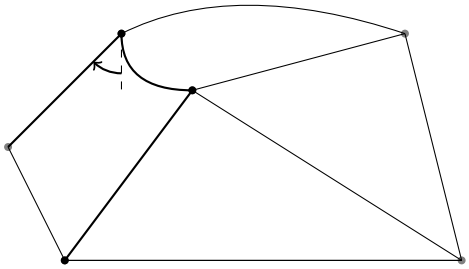

Figure 4.3: Do the same at the next points

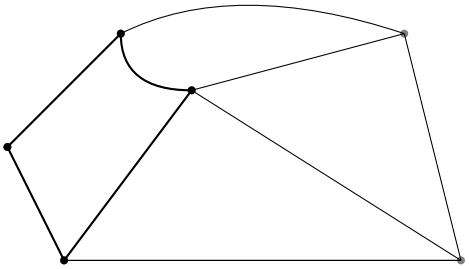

Figure 4.4: Found the area

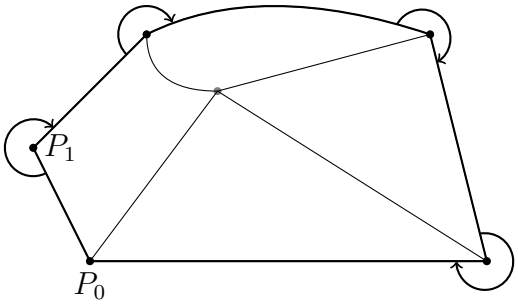

Figure 4.5: Outside area

on the outside boundary of our drawing. See Figure 4.5 for an example. If we here start at point P_0 and with the line P_0P_1 and run our algorithm, then we will get the area marked by thick lines. We can easily detect this area however: *the order of the points of the area is clockwise instead of counterclockwise.* If we take a look at the leftmost point, then in the case of a clockwise area (at the left of Figure 4.6) we will find that the first curve is the top one and the last curve the bottom one. If we look at the counterclockwise area (at the right of Figure 4.6, the first curve is actually the bottom one and the top one is the last. By comparing the angles between the curves as shown in the figure, we can distinguish between clockwise and counterclockwise areas.

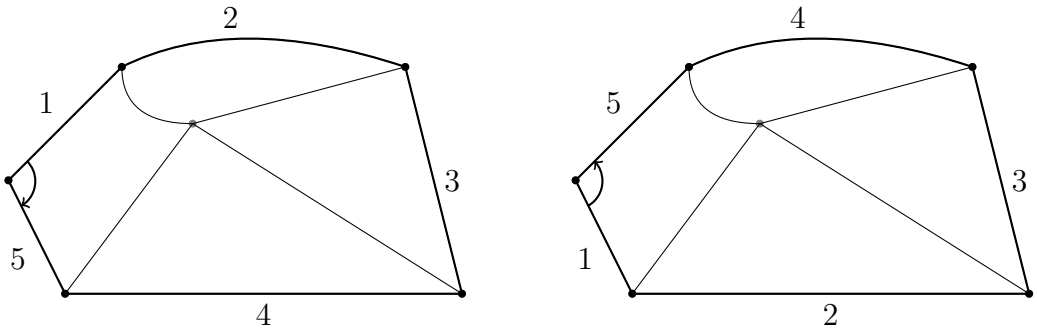

Figure 4.6: Clockwise area (left) and counterclockwise area (right)

4.2 Finding the right area

For a lot of different tasks we need to know in which area a given point is. For example if the user wants to do something with the area under the cursor. With the areas found in the previous section this is not really difficult to do. We first make a polygon approximation of every area. We do this by subdividing every curve of the area such that all resulting curves are near linear. Then we approximate every curve with a straight line from the starting point to the end point. After that we only need to check for each point whether a given point is in the polygon approximation. We can do this by counting how many times a line from the point to a point outside the figure intersects the boundary of the polygon. When the lines crosses the boundary of the polygon, it alternately goes from the outside to inside, then from the inside to the outside, etc. The point is outside the polygon when we get an even number of crossings and inside the polygon if we have an odd number of crossings.

4.3 Computing area size

To compute the size of an area, we again use the polygon approximation. To compute the size of the polygon, we use the Surveyor's formula.

Theorem (Surveyor's formula)**.** *Let $(x_i, y_i), i = 0, \ldots, n$ be the coordinates of the end points of the polygon, such that each line from (x_i, y_i) to (x_{i+1}, y_{i+1}) is an edge of the polygon and $(x_0, y_0) = (x_n, y_n)$. Then the following formula gives the area of the polygon:*

$$\left| \frac{1}{2} \sum_{i=0}^{n-1} \det \begin{pmatrix} x_i & x_{i+1} \\ y_i & y_{i+1} \end{pmatrix} \right| = \left| \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|$$

Proof outline. From linear algebra we know that the absolute value of the determinant of two vectors gives the area of a parallelogram with the two vectors as its side. If we only take half of it, we get the area of the triangle with both vectors as its side. So the formula

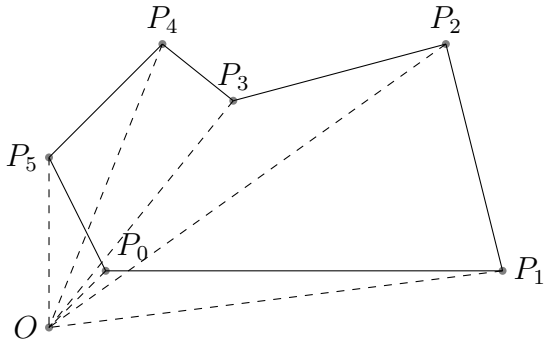

Figure 4.7: Surveyor's formula

is actually a summation of the areas of triangles, see Figure 4.7 for an example. Here we got the triangles P_0P_1O, P_1P_2O, etc. The key is that the order of the vectors determine the sign of the determinant. If for $\det(x\ y)$ rotating x towards y is a clockwise movement, we get a negative determinant and if rotating x towards y is counterclockwise, $\det(x\ y) > 0$. In our example this means that $\det(OP_4\ OP_5) > 0$, because going from OP_4 to OP_5 is a counterclockwise rotation around O and $\det(OP_5\ OP_0) < 0$, because going from OP_5 to OP_0 is a clockwise rotation around O. In total this results in that we add the areas of OP_1P_2, OP_2P_3, OP_3P_4, OP_4P_5 and subtract the areas OP_5P_0 and OP_0P_1, which results in the area of the polygon. The order of the points in the polygon does not really matter, because if we add OP_5P_0 and OP_0P_1 and subtract OP_1P_2, OP_2P_3, OP_3P_4, OP_4P_5 we will get the same absolute value. □

Chapter 5

Implementation

To test our algorithms and show that they work we created a small graphical program. In this chapter we will discuss how we created this program.

5.1 Python and QT toolkit

The program we created written in the language Python[1] language and uses the Qt toolkit[2] for the graphical user interface. Using these technologies it is very easy to create a working program without spending a lot of time on things like memory management, user interface parts such as menus, etc. Especially Qt's Graphic View Framework was very useful. The framework provides a way to easily work with two-dimensional graphical items, including handling events like mouse presses, moving of items, etc. Also standard objects like lines and polygons are already available in Qt, with easy methods for calculating angles between lines, the intersection of lines and polygons, etc. This made it possible to implement everything described in the previous chapters in less than 1000 lines.

5.2 The program

The program can draw straight lines, integral Bézier curves and circular arcs. When curves intersect, the program detects this and split the curves at the intersection point. In the program we actually did not implement different types of lines, all lines used in the program are Bézier curves. A straight line is simply a Bézier curve with its controlpoint in the middle of the start and end point and a circular arc is a rational Bézier curve with specific weights. So internally every line has the same class. It can also work with areas when the figure is closed, highlight the area where the cursor is on and compute the size of an area.

[1] `http://www.python.org`
[2] `http://qt.nokia.com/`

5.3 Classes

In this section the class diagrams of the main classes are shown, together with a small description of their methods. Not all methods and attributes are listed, only the important ones. The BezierCurve class is the class with most of the discussed mathematical logic. The algorithms for intersection, splitting and approximating Bézier curves are implemented there. The QBezierCurve class handles all the communication with the Qt framework and implements methods for drawing curves, handling mouse events, etc. and calls BezierCurve for splitting, intersecting etc. Point is a very simple class that just draws a point and keeps tracks of which curves it is a starting or end point. Those curves are in a list sorted by angle, so we can easily get the right curve when calculating the areas. And as last we have the Area class, which is basically a list of points and curves that represent the area. It implements a test whether a given point in the area and can calculate the size of the area.

Class Point

This class represents a point on the screen and is a starting or end point of at least one curve.

Point
- curves[]
+ calculateAngles() + addCurve(curve) + removeCurve(curve) + getNextCurve(previouscurve) + move()

calculateAngles()
Calculate all the angles of the curves ending in this point and sort the curves list accordingly.

addCurve(curve)
Add *curve* to the list of curves. Calls the method calculateAngles() after that.

removeCurve(curve)
Remove *curve* from the list of curves.

curve getNextCurve(previouscurve)
Return the next curve of the area according to the algorithm described in Section 4.1.

move()
This method is called when the point is moved. For each curve ending in this point, the method checkintersections() is called. Then it calls the method calculateAngles() on itself.

Class BezierCurve

BezierCurve
- startpoint: Coordinate - endpoint: Coordinate - controlpoint: Coordinate - weights: Coordinate[3]
+ isNearlinear(precision): boolean + inverse(point): float + split(point): curve1, curve2 + intersect(curve): Coordinate[] + approximate(): Coordinate[]

This class represents a Bézier curve and implements all the mathematical logic.

boolean isNearLinear(precision)
Return True if the curve is near linear. *precision* is the maximum height of the triangle.

float inverse(point)
Return the t corresponding to *point*.

curve1, curve2 split(point)
Split the curve at *point* using the De Casteljau algorithm, return two new curves.

Coordinate[] intersect(curve)
Return the list of intersection points with *curve*. Return an empty list if the curves do not intersect.

Approximate()
Return a list of points that is a good approximation of the curve when they are connected with straight lines.

Class QBezierCurve

QBezierCurve
- startpoint: Point - endpoint: Point - controlpoint: Coordinate - curve: BezierCurve
+ split(point): curve1, curve2 + checkintersections()

This class implements the Bézier curve as seen by the GUI toolkit. It deals with all the interaction of the GUI. It uses *curve* as underlying mathematical object. Here *startpoint* and *endpoint* are a Point as described earlier, but controlpoint is just a coordinate because it is not visible and no curves ends there.

curve1, curve2 split(point)
Split the curve at *point*, return two new curves.

checkintersections()
Checks whether the curves intersects with any other curve. In the case it intersects, it splits the curve at the intersection point, destroying the old curve. Then it calls checkintersections() on the two new curves.

Class Area

Areas
- List of points and curves
+ containsPoint(point): boolean + calculateArea(): float

The class represents a closed area.

containsPoint(point)
Return True when *point* is in the area, False otherwise.

calculateArea()
Return the size of the area.

5.4 Screenshots

In this section we show several screenshots of the program.

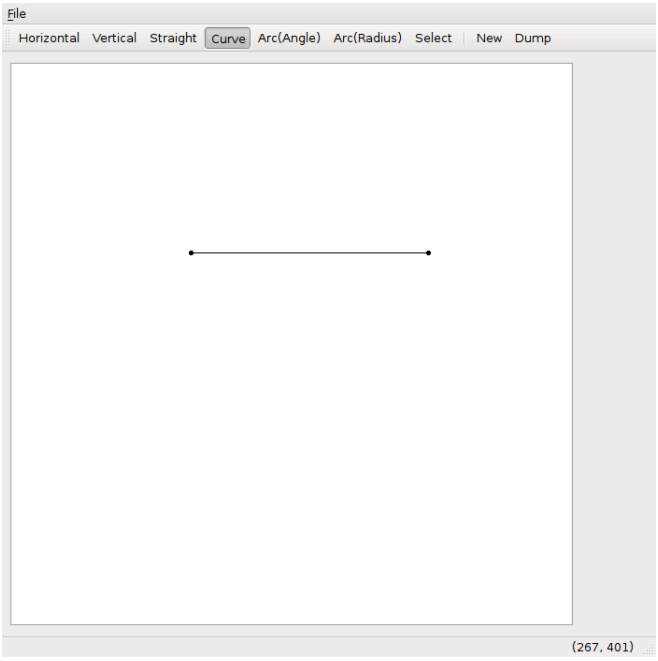

Figure 5.1: Draw a straight line

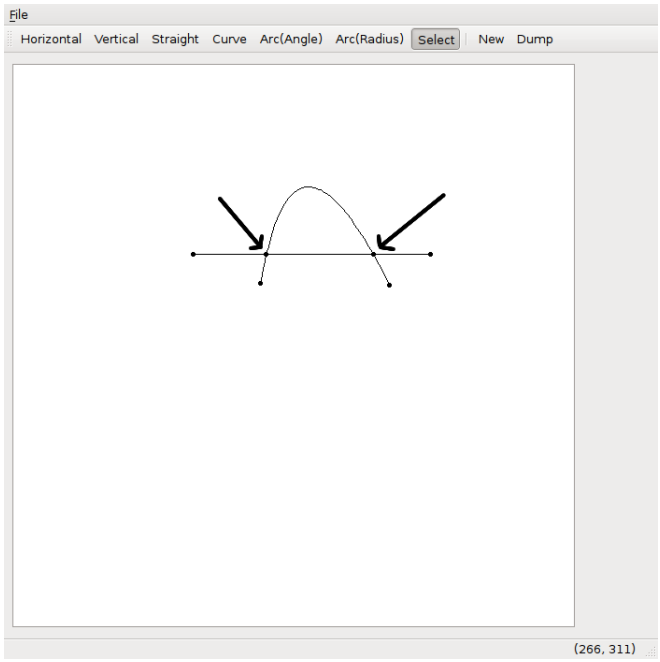

Figure 5.2: Draw a curve, intersection points are calculated

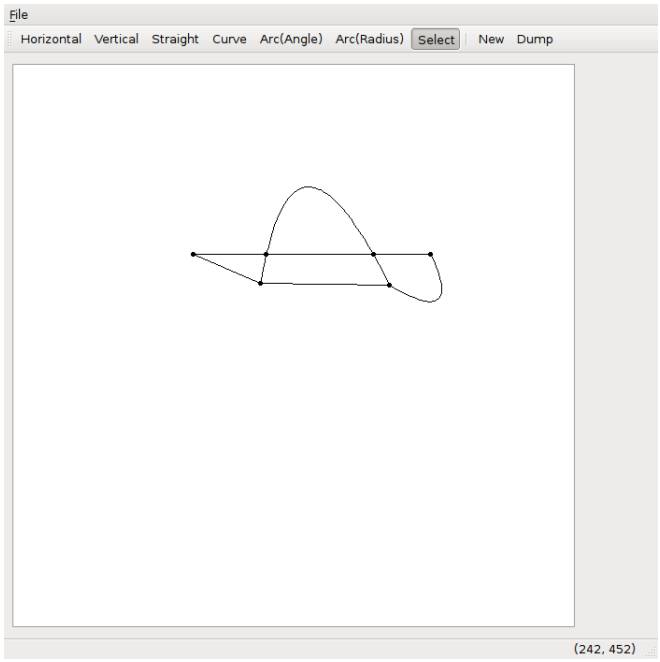

Figure 5.3: Close the figure

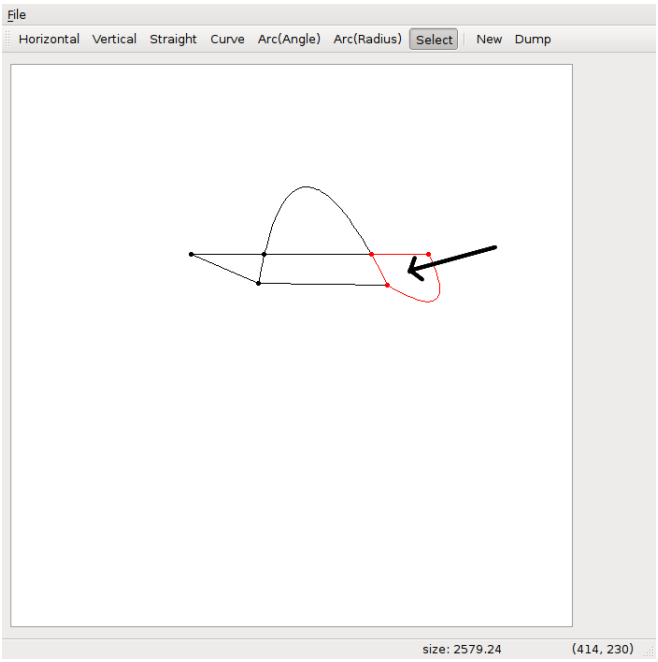

Figure 5.4: Areas are then recognized

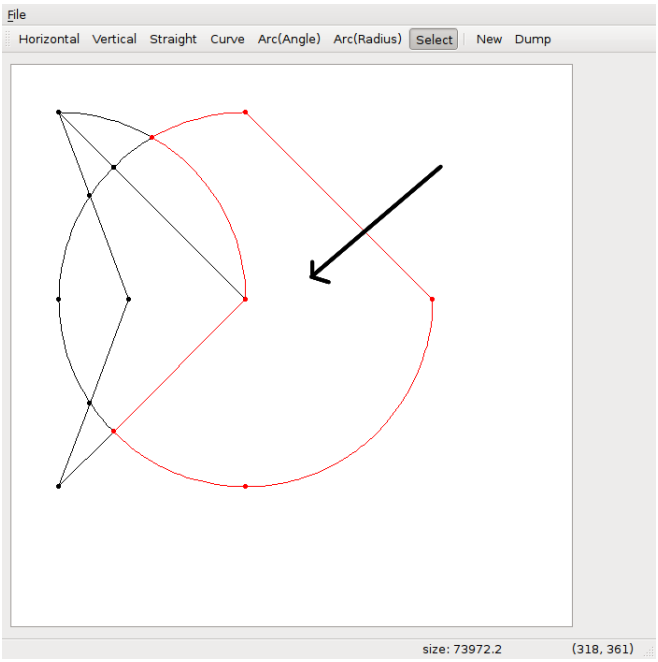

Figure 5.5: Also with complex figures areas are recognized

Chapter 6

Conclusion

In this project we have described how to use Bézier curves in a Computer Aided Design program to be able to create complex figures. We started with creating the mathematical framework for Bézier curves, then we described some algorithms for working with Bézier curves. After that we showed how to work with areas that are enclosed by Bézier curves and explained our new algorithm to determine all the areas. For the last task we developed new algorithms.

We implemented those algorithms in a small program. The program can do basic tasks such as drawing curves, closing them to create areas, determining which areas there are and calculating the size. This shows that our algorithms work in principle and also shows that they are fast enough.

This program is very basic however. It is far from usable by an end user. For example everything drawn are just simple lines with a width of 1 pixel. For a CAD program you will also need to be able to specify different kind of materials, specify whether an area is a window or a wall, etc. The user interface is something that might be a whole different project. Creating a user interface that makes it possible that the user can easily draw a lot different kind curves is an interesting challenge if you consider the numbers of parameters you have.

Another issues that needs attention is precision. We currently just use the float data type that Qt also uses, and approximate Bézier curves by straight lines with a precision of 1 pixel so it will draw nicely on the screen. But if you want to really build the drawing you might need a better precision. All sizes are also in pixels at the moment, but metrics units, like mm, cm, etc. are needed. More polishing is also needed, such as detecting whether intersection points overlap. At the moment it is not detected when you draw three or more lines in such a way that they share the same intersection point. It will just add an intersection for every line that it intersects every time you draw a line, without looking whether it overlaps or is very close to another point.

So there is enough work to be done before there is a program that can be used by an end user. But this project shows that Bézier curves are a good basis for such a program and creates a solid mathematical and algorithmical foundation where all future work can build on.

Bibliography

[1] Duncan Marsh, *Applied Geometry for Computer Graphics and CAD, second edition*, Springer, 2005

[2] Wikipedia, *Surveyor's formula*. URL `http://en.wikipedia.org/wiki/Surveyor's_formula`, accessed 3 september 2010

[3] Wikipedia, *atan2*. URL `http://en.wikipedia.org/wiki/Atan2`, accessed 3 september 2010