ELECTRONIC PUBLISHING, VOL. 7(4), 217–229 (DECEMBER 1994)

Automatic generation of script font ligatures based on curve smoothness optimization

MICHAEL KOKULA

Fraunhofer Institute for Computer Graphics
64283 Darmstadt
Germany

SUMMARY

The idea of type as a fixed geometrical object is shown to be inadequate for script types. The method presented creates ligatures between script font glyphs on-the-fly, i.e. as a part of the glyph rasterization process. This is done by manipulation of an existing font. So the process described here can be used to give existing fonts the intelligence to join characters correctly when being interpreted by a standard font rasterizer or print server.

Of vital importance to the method is the natural appearance of the curve serving as the 'ligature backbone'. In this article, a new smoothness criterion for curves is developed. Then, a method is presented that creates a curve connecting two given curves in a natural-looking way — this is done by optimizing a parametric curve by means of the new criterion. With this algorithm being integrated into on-the-fly generation of script font ligatures, these ligatures get the required level of quality.

KEY WORDS Automatic ligatures Script fonts Connecting curves Curve smoothness Curve optimization

1 INTRODUCTION

In printing, text is traditionally built by juxtaposition of letters with constant shape. However, this is inadequate for script types, where the joining stroke between two neighbouring characters is influenced by the shape of both characters. Section 2.2 describes this problem. In Section 2.3, a method is presented that can overcome this anachronism by automatic generation of adaptive character joins between character pairs. With this method built into an (algorithmic) font description, a script font can be given the intelligence to join adjacent characters in a very natural looking way by on-the-fly generation of adaptive character joins. It is shown how this method can be integrated into a user-defined (Type3-) PostScript font that can then be used on any PostScript interpreter.

The appearance of the curve that is used as a centre line for the stroke between two adjacent characters is vital to the visual quality of the ligature as a whole. However, a curve that is generated in a straightforward manner, e.g. by simply preserving some order of continuity, seldom gives a natural appearance. In Section 3.3 we try to formulate *natural appearance* of a curve in a mathematical way; i.e. we define an aesthetically motivated smoothness criterion for curves. This enables a smooth, natural-looking curve connecting two other curves to be generated by optimizing a parametric curve, thanks to the use of the

CCC 0894–3982/94/040217–13

Received 15 August 1993
Revised 1 December 1993

new criterion (Section 3.4). With this algorithm being integrated into on-the-fly generation of script font ligatures, these ligatures get the required level of quality.

2 ON-THE-FLY GENERATION OF SCRIPT FONT LIGATURES

2.1 Script fonts in typography

At the dawn of movable type, letter shapes were designed to give an impression of the handwriting of their time. It is well known that Gutenberg made several slightly different versions of the same letter to blur the characteristics of the new technology. Good scribes, however, try to make a letter look the same every time; their idea of the letter is a fixed geometrical form. So it was soon accepted quite naturally that printing produces letters with constant shape.

Beneath the formal scripts there have always been cursive script styles. Originally they emerged from the need to write faster, and one of their characteristics is that the characters are connected. This is the origin of the script styles we call (cursive) scripts, which of course by now have their own raison d'être as a style in its own right.

Making types for printing a cursive script is far more natural than doing so for formal scripts. With these scripts, a character is not simply a fixed form. Not only does the joining stroke between two characters depend on these characters, but this stroke in turn influences the appearance of the characters. Therefore, a technology that builds words by juxtaposition of fixed forms is not suitable for the artistic process of joining two (artistic) shapes.

Nevertheless, it has of course been done! And in fact some script fonts are beautiful enough to use. But they all suffer from the problem mentioned above.

2.2 Who needs script fonts?

Script fonts are not appropriate for writing a book. In a book, even a headline set up in script font types would look strange. In the office as well, script types are seldom used. But a set of fonts used in an office always includes at least one decorative font, e. g. for invitations to festivities. This will often be a script font [5]. More often you will find script fonts in newspapers or magazines, though almost always in headlines. Their typical domain is of course advertisement. Especially when food is concerned, on almost half of what you can buy you will find at least one word printed in a script font.

Advertisement design is expected to be done by professional designers. And these should definitely avoid script fonts problems by joining the characters by hand. You may sometimes find the name of the company or product — set up in a script font — without any corrections, with ugly character joins, and a stroke dangling away from the last character in search of a successor it will never find.

2.3 Drawbacks of the traditional approach

Of course you can set script font characters, as is done with any other font, by juxtaposition of fixed shapes. You must simply ensure that the end of every character meets exactly with the start of every other; the joining stroke becomes part of the character shapes. So normally when designing a script font, first you have to define a rule for how characters fit together, a *meeting-place rule* such as 'characters meet at half x-height ascending at an angle of 55 degrees with a stroke width of five percent of x-height'. All characters that are meant to

join properly have to follow that fixed rule: capitals at the end of their stroke, lowercase letters at both ends. In fact, they do follow that rule even if they are not connected; think of an ‘s’ at the end of a word having an absolutely superfluous stroke directly going nowhere!

Of course there have been attempts to avoid these problems in the times of metal type by using additional fixed shapes, e.g. some ligatures (like ‘be’) and some variations of characters (e.g. an alternative ‘s’ for word ends). But in practice, these techniques are seldom used. And apart from that, this approach can only avoid very few extremely bad cases, but does not overcome the basic problem.

This conventional method leads to the following problems:

- *Curve quality*: The joins are often bad, with curves and inflection points that would never have been drawn by a scribe or a designer. And, what is worse, they irritate the reader who recognizes geometrical structures at unexpected places. This fact is quite remarkable for anyone who has learned how fussy type designers are about the curves within their characters. I really wonder where this difference in aesthetic requirements comes from. It must have something to do with the fact that typography is extremely conservative (just compare our capitals with the script of the Romans two thousand years ago!). Typesetting of script fonts has been done this way for a very long time now.
- *Word endings*: At the beginning or end of a word a characer needs a different shape than the one that follows the font’s meeting-place rule. You would, for example, start your stroke at the bottom line; you would choose to have no stroke at the end of ‘s’ and ‘p’, etc.
- *Freedom of design*: This is a question of aesthetic quality. It might be the most important point, although it cannot be recognized directly. The need to follow what we have called the meeting-place rule has quite an impact on script type design. For example, most script fonts have a meeting-place rule quite similar to the example mentioned above. Evidently these fonts have problems with characters that have their natural exit stroke in a relatively high position, for instance ‘b’, ‘o’, ‘v’, ‘w’, and a common form of ‘r’, but also the capital ‘A’. You will often find strange shapes at the end of these characters which the type designer would never have chosen if he had had the choice. The meeting-place rule has a more subtle impact on the type design as a whole. The type designer must have the rule in mind throughout the design process, and will develop only shapes that fit with the rule.
- *No kerning*: The character width, and thus the distance between characters, are completely determined by the character shape. So with conventional script fonts, kerning is simply not possible! The same holds for narrow, respectively wide, spacing. Nevertheless, in practice, wide spacing of script fonts occurs frequently, which then lets the individual characters of a word fall apart.
- *Raster fit*: This is an effect that has come up only with digital technology being used in typography. Normally a text rasterizer accepts floating-point coordinates for the character origin and rounds these coordinates before rasterization. This ensures that the raster representation of a character (with given size and orientation) is exactly the same every time. The consequence is that the distance between two characters randomly varies by one pixel, an effect that can still be noticed at 300 dpi. If you look carefully at the joins within a conventional script font, you will detect that only fifty percent of the characters really meet exactly.

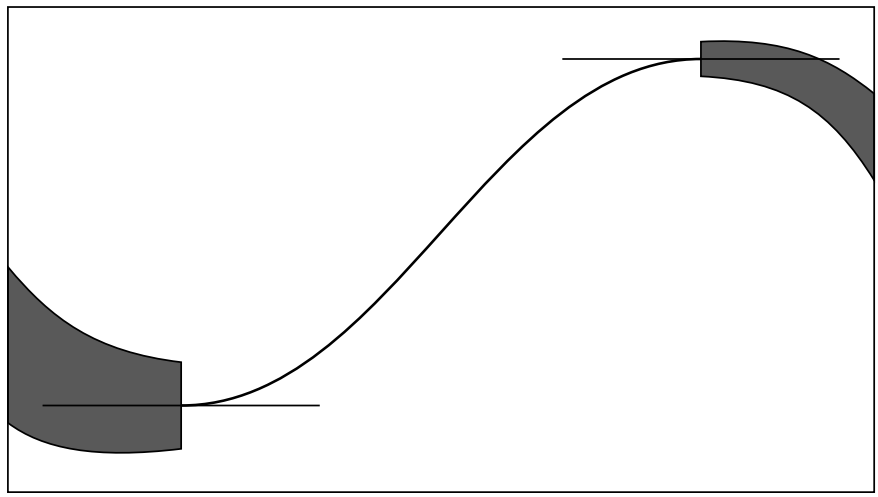

Figure 1. Ligature centre line

2.4 Joining script font characters on the fly

The concept of a character as a fixed shape is a tribute to metal type technology. It is an anachronism nowadays when a character shape is a mathematically described computer object. We can reduce the character to the part that will not change (we will call this part the *character kernel*). If we now succeed in describing the rules for a nice character ligature as an algorithm, the computer can do the rest for us.

Of course it is not possible (I hope) to completely reduce an artistic process to an algorithm. But if you see what script fonts look like today, you will notice that even a rough approximation of artistic behaviour will do much better. This paper presents a method of generating joining strokes between characters on the fly, i.e. as part of the character rasterization process. With fonts considered as a special kind of program, the process described here is used to give *existing* fonts the intelligence to join characters correctly when being rasterized. With this method it becomes possible to do kerning and even wide spacing within script fonts; the algorithm automatically adjusts its behaviour to the character distance.

2.4.1 Generation of a stroke joining a pair of script font characters

The characters of the given script font need to be reduced to their character kernels. Additionally, we need a table that tells us which characters are to be connected to their neighbours: normally this is the case for small letters at both ends and for capitals at the character exit, whereas other characters are not connected at all. This table must also contain the joining characteristics of each character, i.e. the geometry of the junctions at the character entrance and exit. We will call this table the *character junction geometry table.*

When creating a stroke connecting two characters, first the algorithm creates the 'backbone' of the stroke, a kind of centre line, which is computed from the data of a phantom centre line of each of the two characters (Figure 1). This data is stored in the character junction geometry table. This is the most critical step: this curve must be smooth, and it

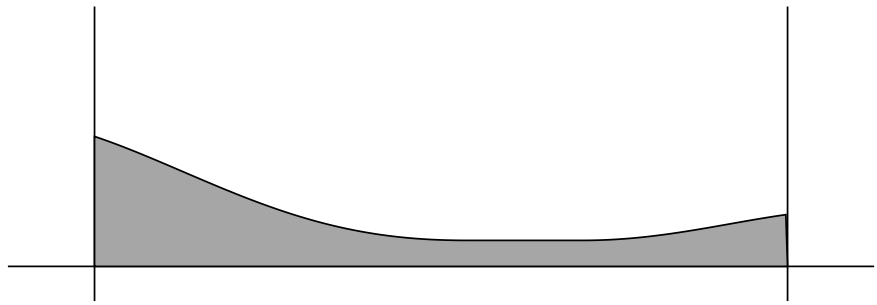

Figure 2. Stroke width function

must be well-behaved — e.g. there must be no loops or cusps. Furthermore, it must have an overall natural appearance. The algorithm for the generation of the centre line is described in Section 3.

In the second step, the stroke width is computed as a function of the arc length of the centre line. This is done as follows: for both character ends, the character junction geometry table contains information about the stroke width as a function of the arc length of the character middle line. This data is used to compute a stroke width function that guarantees a smooth connection to the character, and approximates (on the other side) a *minimal stroke width*, which is a parameter of the font. So the two adjacent characters both define a stroke width function; one fits at the end of the first, the other fits at the beginning of the second character. Now these two functions are merged into a new stroke width function which now guarantees smooth connection to both characters, and approaches the minimal stroke width of the font in the middle of the stroke (Figure 2).

Next, the width function is applied to the centre line to get an outline description of the resulting stroke that now correctly joins the two characters (Figure 3). Of course, the last steps must preserve at the ligature outline the kind of geometric continuity that is guaranteed for the centre line by the first step.

2.4.2 Integrating ligature generation into a script font

Consider the font to be a program that is called using a parameter which is itself a word to be rasterized. The positions of the individual characters may be given implicitly (by character width or kerning information), explicitly by individual coordinates, or by some sort of additional kerning information. The normal behaviour of such a font would then be to rasterize the individual characters and return the resulting raster.

Now let us take such a font and reduce all characters to their character kernels. Additionally, the font would contain the *character junction geometry table*, describing the geometry of the junctions at the character entrance and exit. To rasterize a word, first the individual character kernels have to be rasterized. Then for each pair of adjacent characters the exit geometry of the first, and the entry geometry of the second character is extracted, and a stroke smoothly connecting the two characters is generated as described above.

However, this does not work for the beginning and end of the word. Of course it is not sufficient to draw only the kernel of the first, or last, character. There are several ways of handling the first and last character. There could for example, be for every character

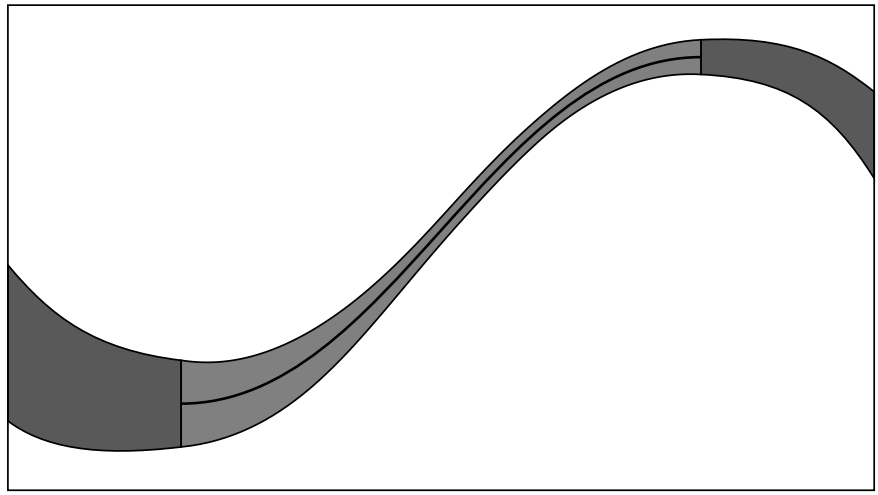

Figure 3. Character ligature

a special version for the beginning, or end, of the word (and one version for the isolated character) which would then be used automatically. Or the font could have for every character a special shape that is to be appended automatically to the character kernel at the beginning of the word (and at the end of the same word of course).

In our prototype implementation, we found it sufficient to define for every character the geometry of the junction of an imaginary predecessor, respectively successor. So if the kernel of the first character of a word has been rasterized, a circle with diameter equal to the minimal stroke width of the font is painted on a position that depends on the character (and is stored in the character junction geometry table). Then a stroke is generated joining this circle with the character kernel. This is done with the same algorithm as is used for ligatures within the word.

2.5 Implementation

Our example font is a Type3 PostScript font, a so-called *user-defined font*. As such a font simply defines a subroutine that is called when a character is to be processed. We are given the full freedom of the PostScript programming language to implement our method. So it is possible to get correctly joined script font text on a standard PostScript printer by simply downloading the enhanced script font.

PostScript does not support the notion of a ‘word’. So the font is called with single characters instead of complete words. As it is the goal of this font to get the new properties within standard text processes (i. e. without changes to the document generation or the document printing process), the font has to *guess* when two characters are to be connected! This can be done, for example, by controlling the distance between the two characters; the distance must not be greater than the default distance (character width and/or kerning value) plus, for example, the width of a space character. However, a real solution to this problem is the introduction of a consistent notion of a ‘word’ throughout the document generation and printing process.

These facts also imply that the font program must be context-sensitive. It is a so- called dynamic font [1]; rasterization of a glyph does not produce the same geometry every time it is performed. Of course, font caching must be disabled! The font contains the definition of a *context dictionary*. This is the place where the character generation routine stores information about the current character. This information can then be used when the next character is processed, i.e. on the next call to the same routine.

The character generation routine is called *BuildChar* in PostScript. Normally this routine simply looks up the character encoding table and then calls the routine that processes the selected character. This routine has to be changed. After processing the character kernel, BuildChar checks whether it needs to connect the current character with its predecessor by an appropriate stroke. The following conditions are tested:

- the actual character belongs to the group of characters that can be connected to predecessors (e.g. a lower case letter)
- its predecessor (that is now looked up in the context dictionary) can be connected to successors
- both characters belong to the same word

For testing the last condition, as well as for the creation of the stroke, position orientation and size of both characters are needed. As far as the predecessor is concerned, this data is stored in the context dictionary. If all three conditions hold, BuildChar extracts the relevant data from the character junction geometry table and calls the ligature creation routine. If not all conditions hold, but at least the first or the second does, the special handling of characters at word boundaries is triggered by BuildChar.

The computations are all carried out in device coordinates, so the process takes into account changes to the *current transformation matrix* that PostScript uses to map user space to device space. This is necessary to support effects commonly used in advertisement, e.g. for setting text along an arbitrary curve.

The following changes have been made to the font:

- characters are reduced to their *character kernel*
- the *character junction geometry table* is built
- the *context dictionary* is inserted
- the *routines for ligature creation*, respectively handling of word boundaries, are added
- the *BuildChar routine* is changed as described above

2.6 Example

Figure 4 shows a printout of both the original font (upper row) and the manipulated version (lower row). Note that we have done a bit of kerning to a few character combinations (e.g. ‘jj’ is set wider). The right column shows the effect of wide spacing, which of course does not work with a standard script font.

3 CONNECTING TWO CURVES SMOOTHLY

When creating a stroke connecting two characters, first the ‘backbone’ of the stroke is determined, i.e. the effective centre line, which is computed from the data of a phantom

Figure 4. Sample font; top: original font; left: wide spacing

centre line of each of the two adjacent characters (see Section 2.4). This is the most critical step of the method described in Chapter 2: this curve must be smooth, and it must be well-behaved — there must, for example, be no loops or cusps. On top of that, this curve must have an overall natural appearance. This chapter deals with the problem of connecting two given curves by means of another parametric curve in an aesthetically acceptable manner — a problem that is by no means limited to script font ligatures. In particular, it should be noted that the method developed here is not limited to two dimensions.

3.1 GC^1 continuity: a first approach

In a first approach, a cubic spline was chosen, that connects to the given curves preserving GC^1 continuity (first-order geometrical continuity, also known as first-order arc length continuity[3, pp. 210–213]). This leaves two degrees of freedom, which were satisfied through a small set of simple rules that guaranteed at least predictable behaviour without claiming to produce a curve giving a natural appearance.

In fact, the curves derived this way did not have the required visual quality. The most serious drawback was the lack of curvature continuity, i.e. second-order geometrical continuity (GC^2) at the connection points with the given curves — it is well known, that human perception perceives such lack of continuity as an angle. So at least GC^2 continuity is required.

3.2 GC^2 continuity: optimization of a parametric curve

To get GC^2 continuity, at least a cubic spline is needed. But for many combinations of given curves, such a spline does not exist. (And if it does, it often takes unexpected shapes like enormous loops etc.) To be able to create a connecting curve for any given pair of curves, you need at least a quintic spline. This can easily be shown: in Figure 5, the two curves that are to be connected point directly towards each other. If the connecting spline were given in Bézier representation with degree n, the following holds: the second vertex as well as

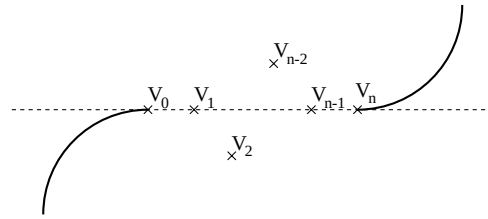

Figure 5. Required spline order

the one before the last (V_1, V_{n-1}) will be on the common tangent, V_2 has to be below and V_{n-2} has to be above the tangent. So $n-2$ must be greater than 2, i.e. $n \geq 5$. On the other hand, the existence of a quintic spline with GC^2 continuity can be proven by construction of the (determined) quintic spline with C^2 continuity, as C^2 implies GC^2 (at least, as long as the second derivative does not disappear [3]). Of course, curves with lower order are easier to handle, and the results show that an order of 5 is in fact sufficient.

Using a quintic spline with GC^2 continuity leaves four degrees of freedom. We decided to choose these by optimizing a function of a set of geometric criteria:

1. arc length minimization
2. integral (with respect to arc length) over curvature, respectively square, of curvature minimization
3. maximum curvature minimization

However, the results were not satisfying. We found two main reasons for that. The method has an exaggerated tendency to curvature uniformity throughout the curve because of criteria 2 and 3. This can clearly be seen when looking at curves that need a much higher curvature on one of the connection vertices than on the other. Although criterion 1 counteracts to a certain point the influence of criteria 2 and 3, there is still a distorting influence that cannot be eliminated by changing the individual weights of the criteria. A simple interpretation of what happens is that a certain curvature is weighted without respect to its neighbourhood. A local curvature maximum deserves much more attention than the same curvature in the neighbourhood of stronger curvature.

The other reason is, that GC^2 continuity refers to properties of the given curves at the connection points, but completely ignores the overall shape of these given curves. The consequence is that at the extremities, a straight line has exactly the same influence as a small, sharp curve that by chance ends in an inflection point (for examples, refer to Figure 7, second and third row). This is an unexpected and unwanted behaviour — at least from an aesthetic point of view.

3.3 A criterion for curve smoothness

The criteria used so far did not match our subjective judgement of aesthetic quality very well. We achieved much better results using a curve quality criterion based on the following simple model.

3.3.1 *Definition*

As a point is travelling along the curve, it is given two constraints:

1. limited centrifugal acceleration, and
2. limited acceleration/retardation in direction of the velocity vector (i.e. limited derivative of the absolute value of velocity)

The curve quality rating is the minimum time the point needs to move along the curve. This time can easily be determined: for every location on the curve a maximum velocity $v_{max}(s)$ is defined as a function of the current position on the arc $s(t)$. It is determined by the curvature in that point. This may lead to absolute accelerations that violate the constraints. The respective velocity restrictions lead to a new function $v(s)$. The equation $v = \frac{ds}{dt}$ now enables us to formulate an arc length function $s(t)$ describing the movement of the imaginary point on the curve as a function of the time parameter t. The time t_{max} needed to travel the whole curve with arc length s_{max} is obtained by $s_{max} = s(t_{max})$.

The time parameter t of the point travelling along the curve should not be mixed up with the parameter of the spline function. Instead, the function *x(t)* providing the location of the point as a function of time is a reparamterization of the original spline function.

This model has a strong similarity with the question of how fast a car can travel through a given curve. However, there are substantial differences. The most important are these. With cars, maximum acceleration is not constant (as velocity is limited). And the absolute value of the acceleration vector as a whole is limited; acceleration as a vector is the vector sum of the centripetal acceleration and the acceleration in the direction of travel. (You should avoid accelerating when driving round a curve as fast as possible.) In addition, maximum acceleration of a car is different from its maximum retardation. Nevertheless, thinking of driving a car can be helpful when thinking about the model presented here.

The process developed here as a criterion for curve smoothness has as its design parameter the maximum acceleration. It can be interpreted as a measure of hastiness or fidgetiness (as it is, again, with cars). Increasing this value leads to the promotion of straighter, more vivid, less smooth lines. The maximal acceleration is to be adapted to the respective application.

Note that the curve smoothness criterion, respectively the process, described here is by no means limited to two dimensions.

3.3.2 *Rationale*

This optimization criterion tends towards something like an "efficient" behaviour. But does efficient mean good-looking? This model does not attempt to mathematically describe an artistic process. In fact, what we need here are *inconspicuous* curves, so we rather try to approximate some kind of natural harmony. But again: does efficient mean harmonic?

We think that the human idea of a natural curve is closely coupled to movements in nature, in particular to one's own movements of, for instance, the hands. Nature, in turn, has a strong tendency towards efficiency. So we think it is no surprise if this model provides a good approximation of humans' subjective judgement of curve quality. However, it should be remembered that the goal of this criterion is not to model the characteristics of human handwriting, but to give a general measure for the aesthetic quality of curves.

3.4 Connecting curves using the new criterion

A method has been developed to generate smooth, good-looking curves connecting two given curves. This is done by optimizing splines by means of the criterion described above.

We take quintic splines and connect them to the given curves preserving GC^2 continuity. This leaves four degrees of freedom (details can be found in [3], pp. 215–218). This quadruple is now derived by optimizing the criterion developed in Section 3.3. It becomes apparent, that this eliminates the drawbacks of the optimizations tried before. Curvature at a certain point has an impact only if it reduces the maximum velocity at this point; which is not the case in the neighbourhood of significantly greater curvature.

It is most interesting that this method takes into account not only the characteristics of the given curves at their end points, but also the overall course of these curves. This is done by considering the computation of the time needed by the moving point. In fact, the computation can be extended onto a complete curve path (think of the centre path of a pen stroke representing a complete word). Note that a maximal speed notion is not necessary, because velocity is always limited within a path with finite velocity at both ends. In fact, for most applications the velocity at the ends of a path will be zero.

3.5 Looking inside the optimization process

It is quite interesting to look at the interaction between the curve representation and the optimization criterion. The spline is responsible for ensuring GC^2 continuity as well as for local smoothness properties, and the optimization criterion is responsible for the overall character of the spline.

There is a certain correlation between the method described here and a type of curve called *clothoid curve* or *Curnu's spiral*. These curves are used in surveying, e.g. for motorway approaches or exits (yes, for cars!), but also within font design [4]. A clothoid is the curve you get by moving a point exactly along the limits of the constraints of the model described above, i.e. with *constant* acceleration along a curve in such a way, that the absolute value of centripetal acceleration remains *constant*.

However, the curve derived from the method described here is not a clothoid curve; instead, it is a quintic spline. The optimization only leads to the *promotion* of curves similar to clothoids. Furthermore, curvature or curvature alteration within the spline are not limited by clothoids. The spline is not directly influenced, but only *rated* by the process. Curvatures that do not fit within the constraints of the model simply slow down the moving point of the underlying model, and thus have an impact on the rating.

In fact, the constraints of the model alone would be unsuitable for the generation of curves; note that the model allows angles within the curve, as it only *limits* the relation between velocity and curvature. Clothoids themselves are unsuitable as well. These curves cannot connect to a straight line preserving GC^2 continuity, and inflection points are not realizable. The properties of the curve come from the interaction between the quintic splines guaranteeing GC^2 continuity and local smoothness properties, and the optimization criterion that is responsible for the overall character of the spline.

Whereas within the curve the new criterion promotes smooth, natural-looking courses, at the connection vertices it sometimes favours rapid changes of curvature. In fact, when you enter a straight street, the fastest route may have a certain curvature until you really enter the street, where you have to turn the steering wheel sharply (which stands for a discontinuity

in the second derivative). In several cases the optimization process of a quintic Bézier spline succeeded in producing such an angle by shifting one or two neighbouring vertices towards the connection vertex; in this way, GC^2 continuity is lost! This has been avoided by restricting the set of quintic splines to a subset of 'well-behaved' splines, for example, by defining a minimum distance between a connection vertex and its direct neighbour.

3.6 Results

The process works well for a wide range of connection conditions. In most cases, problems with local minima of the rating function do not occur. As a start value for the optimization, a nearly straight curve (satisfying GC^2) works well. The connection conditions that occurred when applying the method to the script font problem, presented no problems.

Figure 6 gives insight into the optimization process. The upper part shows the progress of the optimization, whereas the resulting curve can be seen below. The marks on the curve are evenly spaced with respect to the time parameter of the model.

Figure 7 shows some sample curves. Within one row, the same given curves are to be connected by the respective method. In the left column, the given curves are connected through a simple cubic spline that meets GC^1 continuity. The middle column is done by optimizing a quintic spline as described in Section 3.2, and the right column uses quintic splines, optimized with respect to the new smoothness criterion developed in this article.

Please note the difference between the examples in the second and third row, respectively. Although the curves that are to be connected are quite different, their first and second derivate at the connection vertices are the same. These examples are treated differently only by the approach developed in this paper (right column).

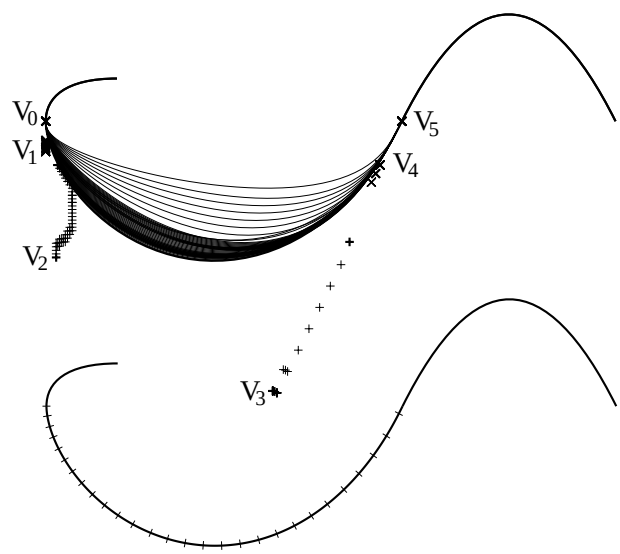

Figure 6. Optimization process

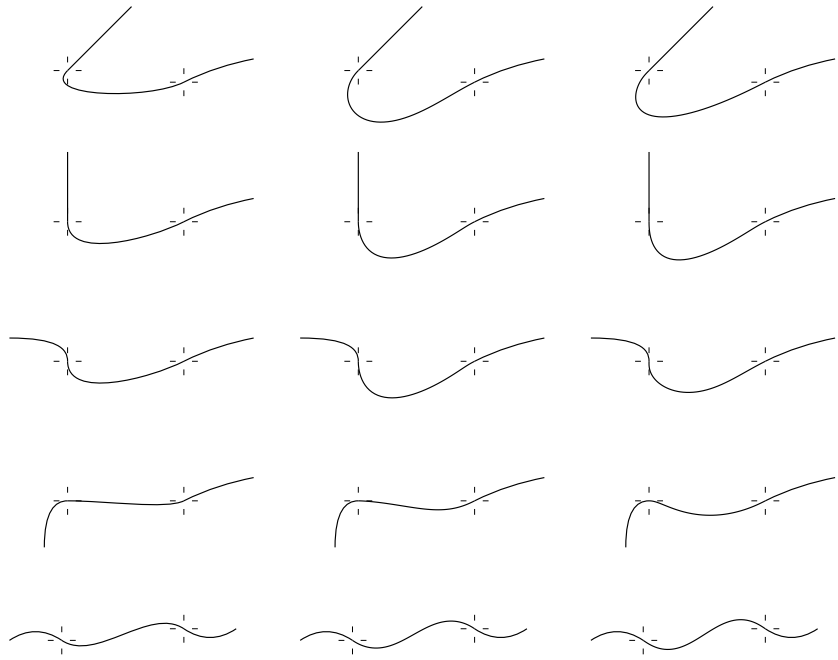

Figure 7. Sample curves (left: cubic spline; middle: quintic spline, optimized by length and curvature; right: quintic spline, optimized by new criterion)

REFERENCES

1. Jacques André and Bruno Borghi, 'Dynamic fonts', *PostScript Language Journal — International Edition*, **2**(3) (1990).
2. Richard H. Bartels, John C. Beatty and Brian A. Barsky, *An Introduction to Splines for use in Computer Graphics and Geometric Modelling*, Morgan Kaufmann, Los Altos, CA, 1987.
3. Josef Hoschek and Dieter Lasser, *Grundlagen der geometrischen Datenverarbeitung*, 2nd edn, Teubner, Stuttgart, 1992.
4. Peter Karow, *Digitale Schriften: Darstellung und Formate*, Springer-Verlag, Berlin, 1992.
5. Richard Rubinstein, *Digital Typography*, Adison-Wesley, Reading, MA, 1988.