Parallelization of Filling Algorithms on Distributed Memory Machines using the Point Containment Paradigm

ANTONIO ELIAS FABRIS[1] MARCOS MACHADO OTTANI ASSIS[1] A. ROBIN FORREST[2]

[1] Instituto de Matemática e Estatística, Universidade de São Paulo,
Caixa Postal 66281, 05315-970, São Paulo-SP, Brazil
aef,otani@ime.usp.br

[2]School of Information Systems, University of East Anglia
Norwich, NR4 7TJ, U.K.
forrest@sys.uea.ac.uk

Abstract. The Point Containment predicate which specifies if a point is part of a mathematically well-defined object is a crucial problem in computer graphics. Point-driven algorithms can be executed for several points in parallel as there is no interdependence among the computations for different points. This paper presents a variety of parallel configurations to counteract the main disadvantage of the point-driven algorithm: its slowness on a standard uniprocessor software implementation for interactive editing and manipulation.

1 Introduction

Region filling is an important raster graphics transformation of a closed curve into a region, which has many fundamental applications in Computer Graphics. The point-driven approach, that specifies if a point is part of a mathematically well-defined object, is a natural way to implement it.

However, the number of computations involved in the point-driven approach is quadratic with the resolution of the output screen. In the object-driven approach, the scan conversion is usually linear with the output resolution. Therefore a software implementation of the point-driven rendering on a standard uniprocessor is too slow for interactive editing and manipulating. Silicon integration and/or the use of coherence tests seem to be necessary to compete "in the large" with the object-driven systems.

In [2, 3, 4] Corthout et al. present a robust point containment algorithm, which was implemented in dedicated silicon, the Pharos chip, fabricated by Philips, on support of the PostScript language. Corthout and Pol [4] describe a way to reduce the number of tests of the point containment approach for region filling to quasi-linear using quadtrees coherence. Fabris et al. [9] present the Maximal Coherence algorithm, a method whose number of tests complexity depends not on the resolution but only on the perimeter of the polygon boundary.

On the other hand, the point-driven algorithm can be executed for several points in parallel as there is no interdependence among the computations for different points. Ideally, one could reserve a point processor per output pixel or dot. The various UNC PixelFlow machines [5, 15] have a processor per pixel and conceivably could, with a more powerful processor, use the point containment algorithm. Brooks [1] quoting Poulton points out that if current hardware trend continues, the number of pixels per primitive rendered by hardware will approach unity and in such circumstances pixels should be computed directly from the underlying geometry rather than first approximating the geometry by polygons or line segments. The point containment approach is an example of this strategy, enabling a fast parallel implementation and generating pixels directly from curves, which avoids the difficulties in curve rendering tackled by Klassen [13] and Lien et al. [14].

This paper presents different strategies of parallelization for the region filling task according to the following point containment sequential versions: the basic quadratic one and those using quadtree coherence [4] and maximal coherence [9]. These strategies were chosen in order to maximize the gain of performance in each case without introducing much complexity into the algorithms.

2 Discrete Curves

A list of length n is a finite sequence of points given by the function $L : [0..m] \rightarrow \mathbb{Z}^2$. Discrete curves can be described as lists whose distance between consecutive points is less or equal to 1. A discrete curve is closed if $L(0) = L(m)$.

The algorithms described in this paper are based on non-simple 8-connected discrete closed curves, that is, discrete closed curves including self-intersecting ones and whose adopted distance is the well-know 8-distance in $\mathbb{Z}^2$ [11], as exemplified in Figure 1.

The point containment algorithm used in this paper is based on the Discrete Jordan Theorem for Non-simple Closed curves, stated and proved by Corthout and Pol [4].

0-7695-0878-2/00 $10.00 © 2000 IEEE

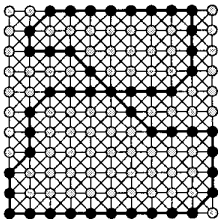

Figure 1: An 8-connected plane discrete closed curve

This mathematical structure and the corresponding algorithm are also briefly described by Fabris et al. [9]. Corthout and Pol's thesis contains details of the overall integer precision required for rendering on a chosen device together with proofs of robustness and accuracy.

In this paper, the point containment test will be used as a primitive operation. From [4] and [9] it is easy to see that if the discrete closed curve has p points, the time complexity of the containment test is of order O(p). This information will be used later in the analysis of the algorithms.

3 Sequential Algorithms

In this section we describe three sequential algorithms for region filling on which the corresponding parallel versions are based.

3.1 Quadratic Filling

This first algorithm is the most natural one, but also the slowest of the three presented in this section. It works testing all points of the image against the curve, marking them either with the object or the background color.

Algorithm 3.1 Quadratic Filling

Input: image I, closed curve C
Output: image I

```
// paint each point of the image
for each point P of the image do
  if P is inside the curve
    then paint it with the object color
    else paint it with the background color
// return result
return I
```

If the image has r points on each axis, then the algorithm will make O(r^2) point containment tests. As each point containment test spends O(p) time (where p is the perimeter of the discrete curve), the total cost time of the algorithm is O($r^2 \cdot p$).

3.2 Quadtrees Coherence

One way to reduce the time complexity is to introduce a coherence test for regions. A region R is coherent with respect to a closed curve C if and only if it is completely inside or completely outside the region delimited by C, Figure 2. This can be easily determined checking if any point of C is inside R. If R is a rectangle, testing if a point is inside R spends O(1) time. Thus a whole coherence test for a curve with p points spends O(p) time.

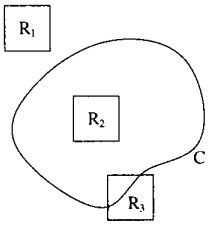

Figure 2: R_1 and R_2 are coherent regions with respect to C, while R_3 is not

The notion of coherence can be used in a recursive algorithm that tests if a rectangle is coherent with respect to a closed curve. If it is, the whole rectangle can be painted with the object or background color, depending if its inside or outside the curve. If the rectangle is not coherent, it can be divided into four quadrants and the algorithm can be called recursively, Figure 3. The initial rectangle is the whole image and the subdivision may stop in a rectangle constituted by only one point.

Algorithm 3.2 Quadtrees Filling

Input: image I, closed curve C
Output: image I

```
if the image is coherent with respect to C
  then do
    // paint all points of the image
    choose a point P of the image
    if P is inside the curve
      then paint all points of the image with
           object color
      else paint all points of the image with
           background color
  end do
  // image subdivision
  else divide the image in four quadrants and
       call the algorithm recursively
// return result
return I
```

Note that in Algorithm 3.2 the initial parameter must be the entire image. Then the following calls will be done only for smaller parts of the image.

In [12] Hunter and Steiglitz proved that the quadtree that represent the recursive calls to the algorithm has O(r +

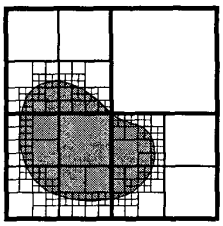

Figure 3: Recursive division of an image

p) nodes, where r is the image resolution and p is the perimeter of the curve. So, the algorithm will be called $O(r + p)$ times, making a coherence test each time. Besides, only $O(r+p)$ point containment tests will be done (on the leafs of the quadtree), each one spending $O(p)$ time. There are also $O(r^2)$ color assignments. Therefore the algorithm spends $O((r+p)\cdot p+(r+p)\cdot p+r^2)$ time, that is, $O(p^2+r\cdot p+r^2)$ time.

3.3 Maximal Coherence

As stated by Fabris et al. [9], the quadtrees subdivision approach used to detect coherent regions is not optimal, because it can find coherent regions that are embedded in bigger coherent regions.

The maximal coherence algorithm finds the bigger coherent regions existing on the image. It uses a propagation approach starting on any point inside the region to be filled. The function is simple: first the curve is plotted on the image. Then for each point of the curve, it propagates the object color from every 8-connected neighbour inside the region. This propagation can be done using a queue.

Algorithm 3.3 Maximal Coherence Filling

Input: image I, closed curve C
Output: image I

```
initialize all image points with the back-
          ground color
// plot the curve on the image
for each point P of the curve do
  plot P on the image with the object color
// start propagation from each interior
// 8-neighbour point of each point of C
for each point P of the curve do
  for each 8-neighbour Q of P do
    if Q is inside the curve
      then start the propagation from Q
// return result
return I
```

The propagation process is similar to a breadth-first search used on graphs. It can be done only for the 4-connected neighbours of each point, to guarantee that the propagation will not cross the 8-connected curve. For more details see [9].

For each of the p points of the curve, up to 8 point containment tests are performed, resulting $O(p^2)$ complexity time. The propagation is performed for $O(r^2)$ points, where r is the image resolution. The time for each propagated point is constant as it requires one removal and up to 4 insertions in the queue. Then the total complexity time of the algorithm is $O(p^2 + r^2)$.

3.4 Theoretical Comparison of Sequential Algorithms

Table 1 shows that for an upper bound limitation, the maximal coherence algorithm is theoretically faster than the quadtrees coherence and that this one is theoretically faster than the quadratic algorithm, although the first difference is smaller than the second.

Algorithm	# of Tests	Total Time
Quadratic	$O(r^2)$	$O(r^2 \cdot p)$
Quadtrees Coherence	$O(r + p)$	$O(p^2 + r \cdot p + r^2)$
Maximal Coherence	$O(p)$	$O(p^2 + r^2)$

Table 1: Comparison of theoretical results

4 Parallel Algorithms

A well-known fact in parallel computing (e.g. [6]), is that the fastest sequential algorithm is not always the fastest parallel algorithm. And the fastest parallel algorithm for one number of processors may not be the fastest for a different number of processors. In the following we present parallelizations for distributed memory machines with arbitrary number of processors for the three sequential algorithms described above.

The correct load balance among the processors is a crucial problem in constructing parallel algorithms. Also, excess of communication, specially during the computation phase, may cause overhead, limitating the gain of performance proportionally to the number of processors. Thus communications among processors must be minimized.

4.1 Quadratic Filling Parallelization

The parallelization of the quadratic filling algorithm is trivial: first the image can be equally partitioned among the processors, then each processor constructs its own part of the image and finally all parts are joined.

One processor is chosen to be the master and is in charge of: taking the curve from the input, dividing the work among the processors, sending the data to them, collecting the results, joining them and sending it to the output. As usual, the master processor in our implementation also takes a part of the work for itself to help the others.

Algorithm 4.1 Parallel Quadratic Filling - Master

Input: image I, closed curve C, int numProcs
Output: image I

```
// divide image among all processors
for each processor p do
  send the curve C to processor p
  send the limits of the image to be filled
       by p to processor p
end do
// execute algorithm
execute quadratic filling on master's image
        piece
// get results from all processors
for each processor p do
  get image piece constructed by p
join all image pieces
// return result
return I
```

Algorithm 4.2 Parallel Quadratic Filling - Others

Input: (none)
Output: (none)

```
// get data from master processor
get the discrete curve C from master
get limits of the image to construct
// execute algorithm
execute quadratic filling on processor's
        image piece
// send results to master processor
send image piece to master
```

In order to simplify the division and joining for any number of processors, we divide the image by vertical strips, Figure 4. Horizontal strips could equivalently be used.

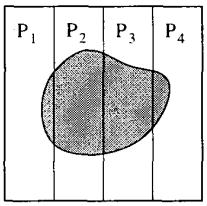

Figure 4: Image division among four processors

4.2 Quadtrees Coherence Filling Parallelization

The main idea of this parallelization is similar to the quadratic one. The image is partitioned equally among the processors either by horizontal or vertical strips and the execution starts. However, due to differences in the execution times for different pieces of the curve, a dynamic load balance is required: when a processor finishes its work, another one has to delegate a part of the work to it, as shown in Figure 5.

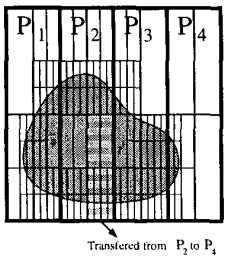

Figure 5: Image division among four processors with work transfer from P_2 to P_4

Algorithm 4.3 Parallel Quadtrees Coherence Filling - Master

Input: image I, closed curve C, int numProcs
Output: image I

```
// divide image among all processors
for each processor p do
  send the curve C to processor p
  send the initial limits of the image to be
       filled by p to processor p
end do
// execute algorithm
do
  execute quadtrees coherence filling on
          image piece
  ask other processors for more work
  choose from which processor to get more work
  tell other processors which one's work was
       taken
while got work to do
// get results from all processors
for each processor p do
  get image pieces constructed by p
join all image pieces, combining repetitions
     using ''or'' combination
// return result
return I
```

Algorithm 4.4 Parallel Quadtrees Coherence Filling - Others

Input: (none)
Output: (none)

```
// get data from master processor
get the discrete curve C from master
get limits of the image to construct
// execute algorithm
do
  execute quadtrees coherence filling on image
          piece
  ask other processors for more work
  choose from which processor to get more work
  tell other processors which one's work was
       taken
while got work to do
// send results to master processor
send all image pieces constructed to master
```

The dynamic load balance implies that the recursivity of the algorithm must be eliminated. Thus, a stack can be used to keep the data of the recursive calls. When a processor needs to delegate some work to another one, it can just take some data from the base of the stack and transfer it to the top of the other processor's stack.

When a processor finishes its job it must contact all the others to know the amount of work that each one may transfer in order to decide from which one to take it. If the amount of work is not advantageous the processor may decide to do not take any work from the others.

After a processor delegates a task to another, it must clean the image piece related to the area transfered with the background color, because when both of them send their results the master one decides from which one to take the results, or just combine them using an "or" operator.

Since a working processor should never be waiting for communication, asynchronous communication must be used. So a processor contacting the others may wait for their answer. But the others, that are still working, may check for messages only when they are able to transfer data. If there is no request, they can continue working.

4.3 Maximal Coherence Filling Parallelization

A parallel algorithm using the maximal coherence approach could simply divide the image among the processors and let each one calculate its own part, limitating the propagation of the object color to each processors' area. It is necessary to avoid that an area without pieces of the curve could be attributed to a processor, since this algorithm only works inside the curves propagating the object color.

Once the propagation starts it is difficult to change its limits, what may increase the time complexity of the algorithm. Thus a static load balance should be used.

The idea of the maximal coherence filling algorithm (see Algorithms 4.5 and 4.6) is to calculate the limits of the curve in one axis, partition the range occupied by the curve among the processors in that axis and let them perform the filling only in that region, limitating the propagation. The area not assigned to any processor must be pre-filled with the background color by the master processor.

If the width of the curve is greater or equal to the number of the processors, vertical strips can be used, as shown in Figure 6. Otherwise horizontal strips must be used.

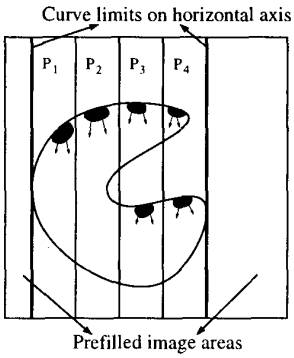

Figure 6: Maximal coherence image division among four processors

Algorithm 4.5 Parallel Maximal Coherence Filling - Master

Input: image I, closed curve C, int numProcs
Output: image I

```
// divide image among all processors
calculate minx and maxx of the curve C
partition minx->maxx range equally among
          the processors
fill areas not assigned with background color
for each processor p do
  send the curve C to processor p
  send the limits of the image to be filled by
       p to processor p
// execute algorithm
execute maximal coherence filling on master's
        image piece, limitating the propaga-
        tion to image piece
// get results from all processors
for each processor p do
  get image piece constructed by p
join all image pieces
// return result
return I
```

Algorithm 4.6 Parallel Maximal Coherence Filling - Others

Input: (none)
Output: (none)

```
// get data from master processor
get the discrete curve C from master
get limits of the image to construct
// execute algorithm
execute maximal coherence filling on pro-
        cessor's image piece, limitating
        propagation to image piece
// send results to master processor
send image piece to master
```

4.4 Theoretical Comparison on the Parallel Algorithms

For the three parallel algorithms described, the curve must be sent by the master processor to all the others, causing an overhead linearly dependent on the perimeter of the curve and on the number of processors, $O(n \cdot p)$. This overhead can be lower if more processors help on this task. We did not use this improvement because in our implementation a low number of processors is used.

The parallel quadtrees coherence algorithm also requires communication among all processors on the job transfer. On each of the $O(r+p)$ subdivisions up to $n-1$ processors may contact the other $n-1$ processors asking for more job, resulting on a $O(n^2 \cdot (r+p))$ overhead.

In the parallel maximal coherence algorithm, the limitation imposed to the propagation of the object color causes a low overhead for each propagated pixel, but high when considering the whole image. It means that the overhead of any parallelization of the maximal coherence algorithm based on area limitation is relatively high and quadratically dependent on the image resolution, $O(r^2)$.

Finally, for the three parallel algorithms, there is a $O(r^2)$ overhead caused by sending the image pieces to the master processor.

Table 2 shows the theoretical upper bound limitations of the overhead of the parallel algorithms with respect to the perimeter of the curve (p), the image resolution (r) and the number of processors (n).

Algorithm	Overhead
Quadratic	$O(n \cdot p + r^2)$
Quadtrees Coherence	$O(n^2 \cdot p + n^2 \cdot r + r^2)$
Maximal Coherence	$O(n \cdot p + r^2)$

Table 2: Comparison on the parallel algorithms overhead

Given a curve and image resolution, the overhead of the parallel quadratic and of the parallel maximal coherence algorithms depend linearly on the number of processors. On the other hand, the parallel quadtrees coherence algorithm depends quadratically on the number of processors. It means that, for a low number of processors, the parallel quadtrees coherence algorithm can be the fastest one. But for a higher number of processors the parallel maximal coherence is better. Near a limit situation, for a number of processors close to the number of pixels of the image the parallel quadratic algorithm is better, due to its simplicity.

5 Practical Results

The algorithms were tested in a Parsytec PowerXplorer with 8 processors and distributed memory. The sequential versions were also implemented and tested on this machine using only one processor. Two images with resolution of 450x300 pixels were used to compare the times spent on each algorithm using different number of processors. These images have complex shapes in order to illustrate the robustness of the algorithms.

Plate 1 shows a curve composed by 47 Bézier curves, with degrees varying from 2 to 10, adding up 211 control points. It has some sharp ends and some curve superposition, but has no self-intersection. The resulting discrete curve has 9,425 points. Figure 7 shows the execution times for each algorithm.

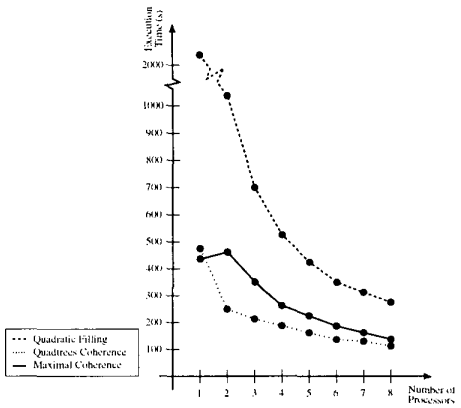

Figure 7: Times for first image

Plate 2 explores an example of a curve composed by 41 Bézier Curves, with degrees varying from 3 to 6, adding up 159 control points. It has many self-intersections and image superpositions. The resulting discrete curve has 5,495 points. Figure 8 shows the execution times for each algorithm.

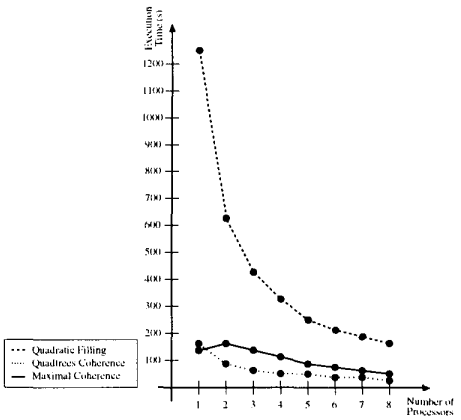

Figure 8: Times for second image test

5.1 Data Analysis

A well known tool that may be used to illustrate the gain acquired with the parallel execution is the speed-up. If t_1 is the time spent for the best sequential algorithm known and t_p is the time spent in a parallel execution using p processors, than the speed-up S_p for p processors can be defined as:

$$S_p = \frac{t_1}{t_p}$$

Figures 9 and 10 show the charts constructed for the speed-ups for both images.

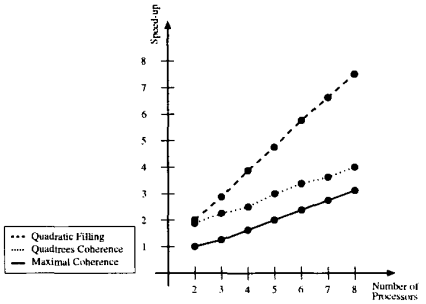

Figure 9: Speed-ups for Image 1

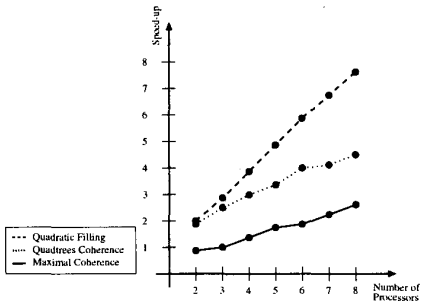

Figure 10: Speed-ups for Image 2

Using the time and speed-up graphics, we can analyze the behavior of the algorithms with relation to the number of processors. They show that, although the maximal coherence is the fastest algorithm for sequential execution, for parallel execution using a low number of processors the quadtrees coherence is faster.

The graphical and theoretical analysis indicate that, for higher numbers of processors, the maximal coherence is faster than the quadtrees coherence algorithm because, as stated before, it has a high overhead but linearly dependent on the number of processors, while the quadtrees coherence algorithm has an overhead quadratically dependent on the number of processors. The quadratic filling algorithm is slower for all tested number of processors.

6 Conclusions and Further Work

Algorithms based on the point containment paradigm are usually simple but slow. In this paper we have presented parallelizations for three filling algorithms based on the point containment paradigm. The gain of performance acquired makes all them profitable despite the fact that each one is the fastest for a certain range of numbers of processors: the parallel quadtrees coherence algorithm is faster for a low number of processors, the parallel maximal coherence algorithm is the fastest for a higher number of processors and the parallel quadratic algorithm is theoretically the fastest for a number of processors approaching the number of pixels of the image.

Current work is going to investigate the parallelization of the stroking algorithms based on the point containment paradigm presented in [10]. By doing this we would be providing a complete parallel system for image renderizations (e.g. a PostScript language interpreter), using simple but efficient and robust algorithms.

Other further work include applications to Point Containment based antialiasing techniques [7, 8] and the parallelization of multidimensional Point Containment based algorithms.

7 Acknowledgment

The work of A.E. Fabris and M.M.O. Assis was supported respectively by grants 97/03055-3 from FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) and 137529/1999-6 from CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

References

[1] F.P. Brooks. Springing into Fifth Decade of Computer Graphics - Where we've been and Where we're going! In *Computer Graphics*, vol. 29 page 513, August 1996.

[2] M.E.A. Corthout and H.B.M. Jonkers. A new Point Containment algorithm for B-Regions in the discrete plane. Em *Theoretical Foundations of Computer Graphics and CAD*, 279-306. Springer-Verlag, 1988.

[3] M.E.A. Corthout and E.J.D. Pol. Supporting outline font rendering in dedicated silicon: the Pharos chip. In *Raster Imaging and Digital Typography II*. pages 177-189. Cambridge University Press, 1991.

[4] M.E.A. Corthout and E.J.D. Pol. *Point Containment and the PHAROS chip*. Joint PhD thesis, University of Leiden. Leiden, March 1992.

[5] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra and N. England. PixelFlow: The Realization. *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*. Los Angeles, CA. August 3-4, 1997, 57-68.

[6] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[7] A.E. Fabris and A.R. Forrest. Antialiasing of Curves by Discrete Pre-Filtering. *Computer Graphics*. 31(3):317–326, August 1997.

[8] A.E. Fabris and A.R. Forrest. High Quality Rendering of Two-Dimensional Continuous Curves. In *SIBGRAPI 97 Conference Proceedings*, pages 10–17. IEEE Computer Society Press, 1997.

[9] A.E.Fabris, L. Silva and A.R. Forrest. An efficient filling algorithm for non-simple closed curves using the Point Containment paradigm. In *SIBGRAPI 97 Conference Proceedings*, pages 2-9. IEEE Computer Society Press, 1997.

[10] A.E. Fabris, L. Silva and A.R. Forrest. Stroking discrete polynomial Bzier curves via Point Containment paradigm. In *SIBGRAPI 98 Conference Proceedings*, pages 118-126. IEEE Computer Society Press, 1998.

[11] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley. Second edition. 1990.

[12] G.M. Hunter and K. Steiglitz. Operations on images using quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 145-153, 1979.

[13] R.V. Klassen. Drawing Antialiased Cubic Spline Curves. *ACM Transactions on Graphics*. 10(1):92–108, January 1991.

[14] S.-L. Lien, M. Shantz and V.R. Pratt. Adaptive Forward Differencing for Rendering Curves and Surfaces. *Computer Graphics*. 21(3):111–118, July 1987.

[15] S. Molnar, J. Eyles and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics* 26(2):231–240, July 1992.

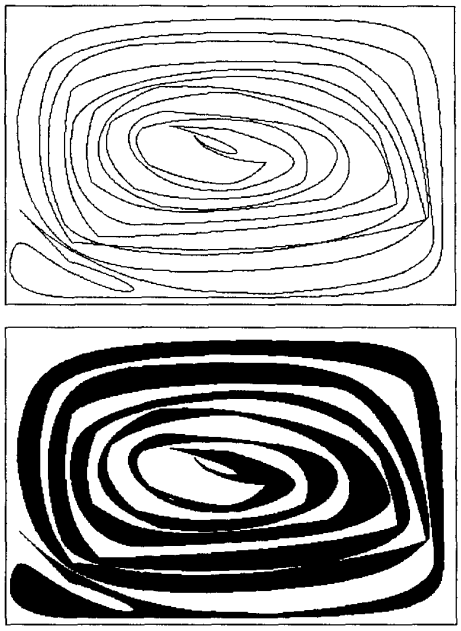

Plate 1: Composed Bézier curves segments

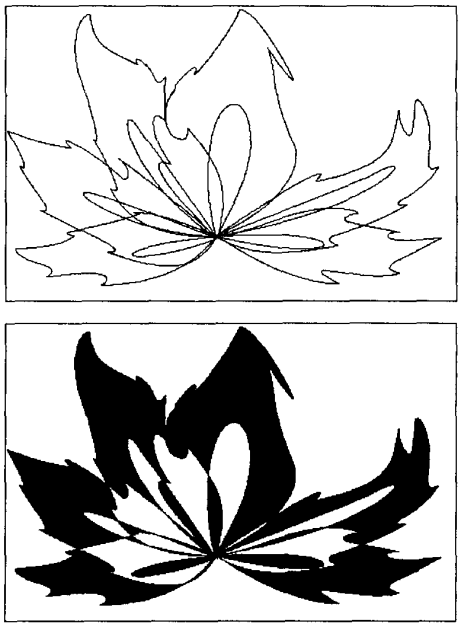

Plate 2: Bézier curves with self-intersecting segments