Rendering PostScript™ Fonts on FPGAs

Donald MacVicar[1], John W Patterson[1], Satnam Singh[2]

[1]Dept. Computing Science, University of Glasgow, U.K.
{donald, jwp}@dcs.gla.ac.uk
[2]Xilinx Inc., San Jose, California, U.S.A.
Satnam.Singh@xilinx.com

Abstract. This paper describes how custom computing machines can be used to implement a simple outline font processor. An FPGA based co-processor is used to accelerate the compute intensive portions of font rendering. The font processor builds on several PostScript components previously presented by the authors to produce a system that can rapidly render fonts. A prototype implementation is described followed by an explanation of how this could be extended to build a complete system.

Introduction

The way in which electronic documents are utilised has changed as computers have become more powerful. Standard page description languages, such as PostScript or PDF, along with powerful desktop publishing applications have increased the popularity of electronic documents. These documents often contain many different type faces at different sizes.

Fig. 1. Example Character.

Page description languages and computer graphics systems allow characters to be processed in the same manner as any other graphical object. Characters with similar stylistic properties are grouped together to form fonts. Each character within a font is defined in terms of its outline by a number of arbitrary shaped polygons. These outlines can then be scaled to the required size and rasterised when the document is rendered to either a computer screen or a printer. We concentrate on the processing of fonts for printed output by rendering fonts used within PostScript documents.

Some printers do have a small set of fonts built into ROM as pre-rendered bitmaps but other fonts require rendering from outline descriptions. The rendering of fonts adds more computation to the print time for a document. Desktop printers often contain a single processor which is used both to render fonts and interpret PostScript documents. This processor is often an off the shelf RISC processor which may not be able to supply data to the print engine at as high a rate to keep it fully utilised. Commercial high resolution printers often require a separate system to perform the PostScript interpretation. These external systems tend to be very powerful multi-processor workstations with large amounts of both memory and disk space. Even these powerful systems can require hours to render a document. Font processing becomes especially important when handling Eastern or Arabic languages which contain complex characters such as the one

shown in Fig. 1.

Modern printers often have resolutions of around 5000dpi (dots per inch) and impose a severe computational load on the PCs and workstations that prepare images for them. Bitmap fonts at this resolution are large and can be much more efficiently rendered on the fly if the implementation is quick enough. We use FPGA technology to reduce the time spent rendering fonts. Our system does not have enough on-board memory to represent an entire A4 or US letter page for high quality printing. However font rendering makes localised and predictable memory accesses. We exploit this behaviour to transfer and format image memory in a way which reduces over-bus memory transfers.

Careful use of the memory hierarchy continues to the on-chip level where some of the 4K BlockRAMs on the Xilinx Virtex FPGAs are used to cache image data. Stack space needed for a recursive curve drawing algorithm is also implemented on chip.

2 Font Rendering

There are many different formats which can be used to define fonts we concentrate on the simplest which is PostScript Type3. Type3 PostScript fonts are described using the standard set of PostScript path operators with some additional font operators for precise layout control. Other formats are based on similar operators but employ more elaborate encoding schemes and modified operators which help to minimize the data. The FPGA is utilised for the most compute intensive part of the font rendering process which is the actual drawing of the characters.

There is a small basic set of operators required to allow the definition of any character. Only operators which allow the definition of moving to a point, straight lines, curved segments, ending a sub-path and filling the outline are required.

- `x1 y1 moveto` : Sets the current position to `(x1,y1)` used at the start of each subpath.
- `x1 y1 lineto` : Adds a straight line from the current point to `(x1,y1)` to the current subpath.
- `x1 y1 x2 y2 x3 y3 curveto` : Adds a curved segment to the current subpath from the current point to `(x3,y3)` using the points `(x1,y1)` and `(x2,y2)` as control points for a cubic Bézier curve.
- `closepath` : Ends a subpath and joins the current point to the first point of the sub-path if necessary.
- `fill` : Ends a characters description and fills the outline with the current colour.

2.1 A Simple Example

The character shown in Fig. 2 is defined using two subpaths: one defines the exterior outline and the other defines the interior outline. The interior outline uses the `closepath` operator to draw the straight line from the end of the last curve to the start of the first which reduces the data. A `lineto` operator could have been included to perform this. The internal and external outlines are also deliberately defined so that one winds clockwise and the other anti-clockwise. This allows the winding number rules to calculate which areas are interior and which are exterior achieving the same result no matter which rule is used.

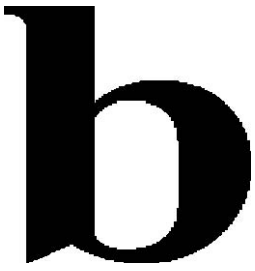
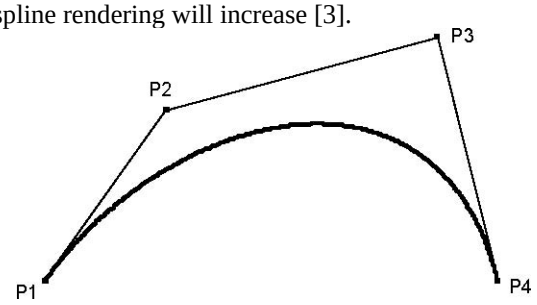


Fig. 2. Simple character extracted from a standard font

Processing Elements

The five operators only require implementations for line rendering, curve rendering and filling. Straight lines within a character are defined by the start and end points of the line. During rendering every point that lies on the straight line between the end points has to be calculated and coloured.

Cubic Bézier curves are described by a parametric (cubic) equation. These curves are used extensively in Postscript and other graphics systems, but are computationally expensive to calculate making them suitable for hardware rendering. As 3D imaging moves to curve rendering in preference to polygon rendering, the need for fast spline rendering will increase [3].



$$Q(t) = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2 (1-t) P_3 + t^3 P_4$$

Fig. 3. Bézier Curve.

Whether rasterising Bézier curves for screen or printed output the general technique is to use a piecewise linear approximation of the true curve. There are then two main algorithms for obtaining this approximation, iterative calculation of points on the curve and recursive sub-division.

3.1 Bézier Recursive Sub-Division

Rather than iteratively evaluating the Bézier equation directly to obtain the points on the curve it is possible to divide the curve into smaller curves until these curves become approximately flat enough to be replaced by a straight line. This is the principle behind the recursive sub-division algorithm. The Bézier curve is divided in half, division by two is used since it involves the minimal amount of arithmetic.

This is the preferred technique for use in software systems as it reduces the amount of computation required while still producing accurate results. The results from this method can be an improvement over the iterative calculation since the frequency of points around a sharp bend in the curve is increased.

3.2 Outline Fill

When performing a fill on arbitrary shaped polygons one of two rules is used to determine which areas are interior and which are exterior. The two rules are even-odd and non-zero winding number, which can produce different results given the same input data. The guide lines for creating fonts say that they should be defined in such a way that the correct result is achieved using either rule. When implementing the non-zero winding number rule the direction of each edge is important and has to be kept and is thus not suited to FPGA implementation because of the large data this would require.

We implement the even-odd rule which allows the outline to be rendered first. The fill is performed by working across each scanline from left to right. Whenever an edge is detected the fill value is inverted. In the bitmap black is represented by 1 and white by 0. As the current position on the scanline moves across from left to right (0 to F) the fill value changes at position 1 from white to black and then back again at position E.

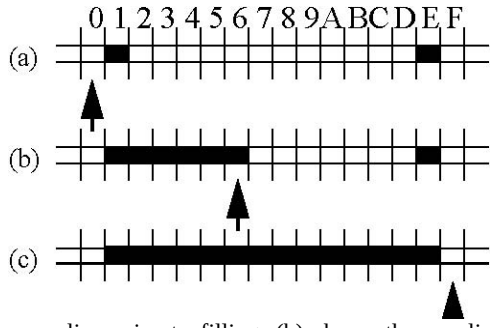


Fig. 4. Scanline Fill. (a) shows a scanline prior to filling, (b) shows the scanline partly filled and (c) shows the completed scanline

4 Target Architecture

The implementations are targeted to the Xilinx Virtex™ [11] series of FPGAs. We assume that the FPGA is situated on a PCI card which provides close coupling with the main system processor. We also assume that an adequate amount of fast SRAM is located on the same card as the FPGA. The host CPU passes the outline definition to the FPGA and then gets the result back when rendering is complete. There are several cards available which meet our requirements including the PCI Pamette with Virtex daughter card from Compaq Research Labs and the Wildstar™ range from Annapolis Micro Systems Inc. The following circuits are all implemented in VHDL and synthesised using either Synopsis Design Compiler or Synplicity for the Virtex XCV300 FPGA at speed grade 4.

5 Implementation

The input data describing the outline of a character is written into a number of blocks of the 4K BlockRAMs. Although the data is actually a number of 8-bit words and are read from the memory as 8-bit words they can be written as 16-bit words due to the dual ports available on the BlockRAMs. The wider data path reduces the time required to write the outline data to the FPGA. Each of the five operations is assigned an op-code which is 8bits allowing extensions at a later date such as the more compact definitions used in Type 1 fonts. Due to the small number of operations the values 1, 2, 4, 8, and 16 are used as the op-codes since this makes the decoding simpler and thus faster.

The character in Fig. 2 requires 80 bytes of memory space. The more complex character in Fig. 1 contains 54 straight line segments, 15 curve segments and nine subpaths requiring 276 bytes. These memory requirements are using 8-bit coordinate values however a complete system would require larger coordinates of perhaps as much as 16-bits.

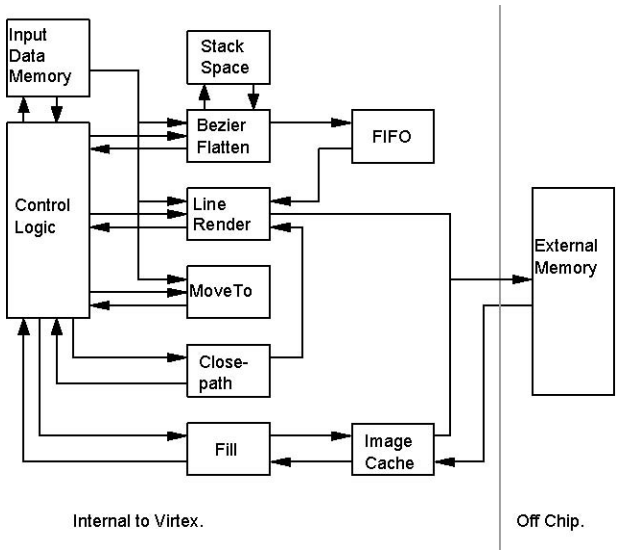


Fig. 5. Font Processor Block Diagram.

The processor is implemented using a fetch, decode and execute style architecture. A finite state machine is used to control the fetch, decode and execute stages. An op-code is read followed by the appropriate number of operands. The number of operands depends on the particular instruction that was previously read. Once an operation and its associated data have been retrieved from the BlockRAM it is passed onto the appropriate processing element where it is executed. Each of the five processing elements, which are Bézier flatten, line render, MoveTo, Closepath and Fill in Fig. 5 has a simple state machine to control the operation of that element. The fetch and execute stages can be pipelined as in modern processor architectures. While one operation is being executed the next can be loaded. This would improve performance but only marginally. The most common operations are the `lineto` and `curveto` which also have long execution times. These operations take many clock cycles each and thus the time taken to load data from memory is significantly less than that of the operation execution. The extra resources required to implement parallel fetch and execution may be better utilised elsewhere.

5.1 Straight Lines

A line rendering circuit is required for several of the operations, lineto, curveto and closepath. A single line rendering circuit is implemented on the FPGA. Depending on the current operation the input to the circuit is changed. When a lineto operation is being executed the current point is the start point and the point defined by the associated data is the end point of the line. When executing a curveto operation the input is taken from a FIFO buffer into which the Bézier curve circuit writes its results. When executing the

closepath operation the start point is the current point and the end point is the first point in the current path. To produce the straight lines we use a simple implementation of Bresenhams Straight line algorithm. An optimised version of which runs at approximately 120Mhz. The output of the line circuit is written to external memory.

5.2 Recursive Bézier Implementation

We now return to implementing the Bézier calculation by using a recursive implementation. Dividing a Bézier curve in half requires very little computation. The only arithmetic involved is addition and division by two. The calculation is shown in Fig. 6

The main drawback with this calculation is the combinational delay through the calculation. The delay could be minimised by using a pipelined circuit, unfortunately this is used inside a loop where the next operation is dependant on the result of the previous operation thus making pipelining difficult.

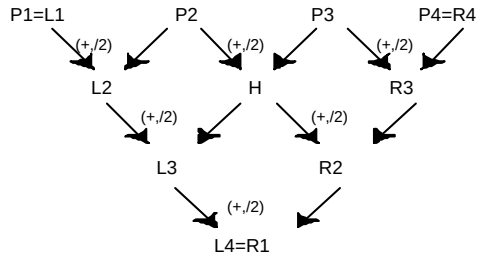


Fig. 6. Bézier division.

The other computational section of the recursive sub-divide method is the flatness test. This is achieve by comparing the gradients of the line connecting the control points. The gradient between P1, P2 and P1,P3 is compared to that of P1,P4 if these are suitably close then we have a straight line.When a curve is flat *delta1* and *delta2* are zero, since we are only calculating an approximation then we can assume the curve is flat when *delta1* and *delta2* are less than some tolerance value.

$$\begin{aligned} delta1 &= [(P2x - P1x) \times (P4y - P1y)] - [(P2y - P1y) \times (P4x - P1x)] \\ delta2 &= [(P3x - P1x) \times (P4y - P1y)] - [(P2y - P1y) \times (P4x - P1x)] \end{aligned} \quad (1)$$

The results from the flatness test are used to decide whether we have a point on the curve or if we need to do another division. When another division is required one curves is pushed onto the stack and the other is used for the next iteration. When a point on the curve is found the last control point is passed to the line rendering circuit and the next

5.3 Outline Fill

The process of filling the outline is relatively simple and thus fast but the data management techniques employed are vital to the performance. The fill process works on a scan-line basis processing one scanline at a time. As an edge is found the position is written onto a small stack and once the end of the line is reached the space between pairs of edges is filled in the image. The outline fill process implemented in behavioural VHDL runs at approximately 90Mhz and uses 262 slices of an XCV300 when synthesised using Synplicity.

5.4 Image Cache

The only other circuit to directly alter the output image is the line rendering circuit. The line rendering circuit as previously described draws directly into the external memory. The arrangement of the image data in the external memory affects both the line rendering process and the filling process. We are only dealing with monochrome images and thus only require one bit per pixel. We assume that the external memory can be accessed using an 8-bit wide data bus and a 16-bit address. Thus each pixel is stored using 8-bits which requires 64Kbytes of memory.

The memory accesses performed by the fill process are highly predictable and thus the data can be cached easily. Two BlockRAMs are used as an image cache for the fill process. This allows for 32 scanlines to be held in the cache at one-time. This is adequate so long as the cache system can keep the fill process supplied with unprocessed scanlines. Assuming a read or a write to either BlockRAM or external memory takes one clock cycle. The load of a scanline takes 256 clock cycles and writeback takes 32 resulting in 73728 clock cycles to completely cache the image. When using 1-bit memory access for the fill process each scanline requires 256 reads and a number of write depending on the data resulting in a minimum of 65536 memory accesses. This requires 8192 writes to balance the access but is also equivalent to the number of cycles taken to load the cache.

6 Summary

The font processor circuit was implemented using the latest version of the Xilinx place and route software. Using the slowest XCV300 speed grade we achieved a primary clock input speed of 55MHz. The layout produced is shown in Fig. 7 Note that related chunks of logic get grouped together. The complete system uses only 1335 of the available 3072 slices on the XCV300 leaving room for a more pipelined approach and other additions.

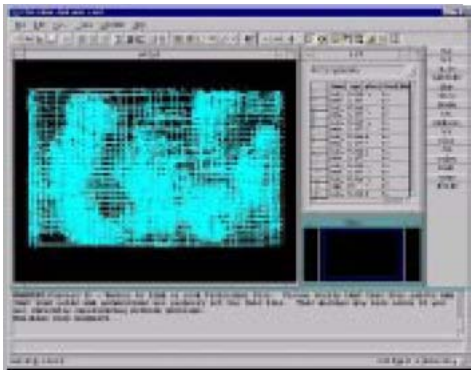


Fig. 7. The layout of the font processor

We have shown how a custom co-processor can be built using FPGA technology to solve the problem of font rendering. The increased size of current FPGAs allows us to implement in hardware algorithms which

would previously only been possible in software. Specifically we show that a recursive algorithm can be implemented efficiently in hardware when fast access memory is available on chip.

There are several optimisations and extensions that can be integrated into the processor described above. The use of external memory which is cached using the BlockRAM could be improved. Since the majority of memory access comes from the fill process optimizing the caching behaviour for this would be optimal. When the memory access of the line rendering are compared which those of the fill process it can easily be seen which is more memory intensive. The line rendering process involves relatively few memory access and it may be worthwhile to pay the cost incurred when the image data is packed even when in external memory.

Many characters use circles or arcs as part of the outline for example the dot on the letter *i*. It is possible to implement circle rendering very efficiently using an FPGA [12]. An extra operator would be added which would allow circles and arcs to be drawn. Bézier curves cannot represent circles perfectly. Four Bézier curves are required to approximate a circle. If a circle operator were added this would not only produce improved visual results for circular objects but would also improve performance due to reduced data and more efficient implementation.

The definition language used is based on the set of PostScript path operators. This could be expanded to allow the definition of PostScript documents and not just Post-Script fonts.

References

[1] Adobe Systems. Adobe PostScript Extreme White Paper. Adobe System Inc. 1997

[2] Adobe Systems. Adobe PrintGear Technology Backgrounder. Adobe Systems Inc. 1997

[3] Smooth Operator. The Economist. Edition 6 March 1999.

[4] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Kirshna Palem, Viktor K. Prasanna, Henk A. E. Spaanenburg. Seeking Solutions in Configurable Computing. IEEE Computer, December, Vol. 30, No. 12. December 1997.

[5] Intel. Accelerated Graphics Port Interface Specification Revision 2.0. December 11, 1997.

[6] M. Sheeran, G. Jones. Circuit Design in Ruby. Formal Methods for VLSI Design, J. Stanstrup, North Holland, 1992.

[7] Satnam Singh and Pierre Bellec. Virtual Hardware for Graphics Applications using FPGAs. FCCM’94. IEEE Computer Society, 1994.

[8] Satnam Singh. Architectural Descriptions for FPGA Circuits. FCCM’95. IEEE Computer Society. 1995.

[9] J.D. Foley, A. Van Dam. Computer Graphics: Principles and Practice. Addison Wesley. 1997.

[10] Xilinx. XC6200 FPGA Family Data Sheet. Xilinx Inc. 1995.

[11] Xilinx Virtex™ 2.5V FPGA Product Specification. Xilinx Inc. 1998.

[12] D. MacVicar, S. Singh. Accelerating DTP with Reconfigurable Computing Engines. FPL’98. Springer-Verlag 1998.

[13] S. Singh, J. Patterson, J.Burns, M. Dales. PostScript™ Rendering with Virtual Hardware. FPL’97. Springer-Verlag 1997.

[14] http://www.xilinx.com/products/virtex.htm