An Artistic Pencil-Sketch Filter in Hardware

Benjamin Stewart Adrien Treuille

December 17, 2003

Abstract

We present a hardware implementation of an artistic filter that converts a video stream to a "sketched" representation of the stream. The design consists of two sliding windows and a lookup table. Interactive speeds are achieved by using precomputed line rasterizations and a precomputed approximation to the arctan function. Each module is extensively parameterized to allow for future extensions to the filter.

1 Introduction

We present a hardware implementation of a filter that converts a video stream into an artistic "sketched" representation of the same video in real time. Our goal is purely aesthetic: we hope for an interesting effect in which real life suddenly appears sketched and cartoon-like.

Artistic filters have a long precedent in computer graphics where they have been studied both for aesthetic purposes [4], and as a way of analyzing the painting process [2]. Typically these algorithms render an image so that it appears to have been created by some traditional technique, such as pencil sketching or oil paint. More recently, researchers have turned their attention to applying these filters to video. Most video algorithms have been off-line (e.g. [1]), but Hertzmann demonstrated a realtime filter [3]. The main difficulty with artistically filtered video is achieving *temporal coherence*: adjacent frames should avoid discontinuity artifacts.

To our knowledge, we are the first to implement an artistic filter in hardware. While we did not address temporal coherence, we were nonetheless pleased with the results. Video streams from the lab exhibited a cartoon-like, sketched appearance when placed through the filter. Feeding a laptop video output through the filter gives the impression of looking at an artist's sketched prototype of the Windows interface, while allowing the user to interact in realtime.

2 Solution Description

The idea behind our filter is that an artist will tend to sketcg lines along the countours of objects. That is, the lines will occur in areas of high gradient and will be oriented perpendiclar to the gradient. Therefore, our filter makes the entire image white (as in paper), and then add short black line segments (as in pencil marks) oriented along the edges. Pseudo code for the filter is given in Figure 1.

```
sketch(inImage, outImage) {
  Make outImage completely white.
  for every point p in inImage {
    g = inImage.gradient(p);
    if (length(g) > threshold)
      Draw a black line in outImage at p perpendicular to g.
  }
}
```

Figure 1: Pseudo-code for the filter.

The implemention uses two sliding windows. One 2×2 window sweeps over the input stream computing the gradient. Simultaneously an 8×8 window slides over the output stream. When the gradient magnitude falls above a threshold,

we place a line segment in the output window oriented perpendicular to the gradient. Instead of using an iterative line-drawing algorithm, we use a lookup table. This table contains precomputed pixel maps of antialiased lines of length 8 that can be rendered on an 8×8 grid. A block diagram of the computation is shown in figure 2.

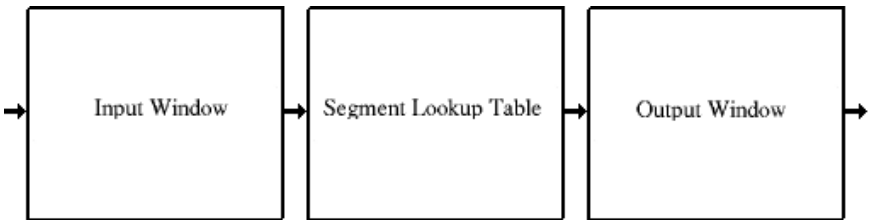

Figure 2: Block diagram of the computation.

We now explain the two main components, the sliding window and the lookup table, in greater detail.

2.1 Sliding Window Implementation

We implemented a general sliding window module parameterized by the size of the window n, the bits per grid cell w, and the scanline width of the image. The input/output pins of our window are shown in Figure 3. The two operations permited are shift and write. To advance the window by one pixel, one must enable the `shift` signal and wait for the `valid_out` signal. Then, one can write to any combination of registers using the `write_enable` mask along witht he $n \times n \times w$-bit input buffer: for each bit position of `write_enable` that is high, the corresponding value from the $n \times n \times w$-bit input buffer is selected and written to the appropriate register within the window.

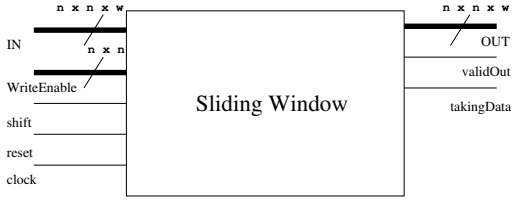

Figure 3: Sliding window input/output pins.

The sliding window hardware consists of $n \times n$ w-bit registers and one $(512 \times w)$-bit single ported RAMs for each window row. Shifting of the sliding window involves three steps. The first step saves the left edge of the window to the RAMs, the second step shifts the data, and the final step loads into the right edge of the window from the RAMs.

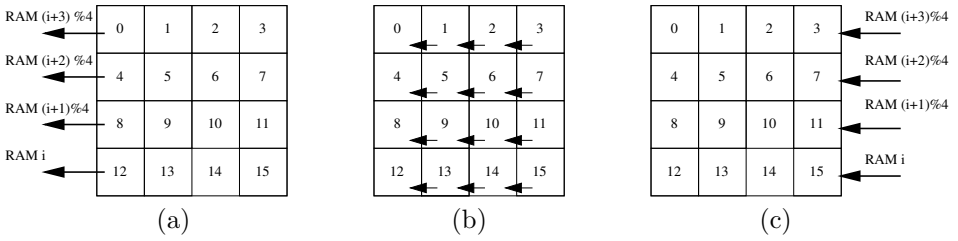

Figure 4: The save step.

Since there are $(n-1)$ values to be saved (the upper left hand corner can be thrown away) and there are n RAMs, all of the values can be saved in parallel. Thus, the first step of the shift takes constant time. Initially, the value in row i is written to the ith RAM. Once the sliding window is advanced to the next row of the image, row i begins writing into the $i+1$th RAM modulo the number of RAMs. Thus each row rotates through the entire set of RAMs as the sliding window is advanced from row to row (Figure 4a).

The second step, shifting the data internally, can be done in a single cycle. This is because the output of a register can be sampled the same cycle that the input changes (Figure 4b).

Finally, reading from RAM, like writing, can also be done in constant time. As with writing, each time the sliding window advances to the next row of the image, the RAMs are rotated (Figure 4c).

2.2 Lookup Table Implementation

The lookup table takes a gradient, and outputs a properly oriented antialiased line if the gradient lies above a threshold. As with the sliding windows, we parameterized the lookup table along a number of dimensions, namely:

- The gradient threshold.
- The length of the line segments.
- The number of line segments in the table.
- The bits per pixel.

The last three parameters are nontrivial, and cannot be implemented with a simple `parameter` statement in verilog. Instead, the lookup table is actually a python script which takes these parameters and outputs a compilable verilog file implementing the lookup table.

The lookup table has input/output pins as shown in Figure 5. When `valid_in` is asserted, `GRAD_X` and `GRAD_Y` represent the gradient as signed 9-bit numbers. Three clock cycles later, `valid_out` will be asserted and `PIXMAP` will contain a set of $n \times n$ 8-bit values to be subtracted from (drawn onto) the output window. If the gradient falls below the threshold, then `PIXMAP` is all zeros, meaning that no line is to be drawn.

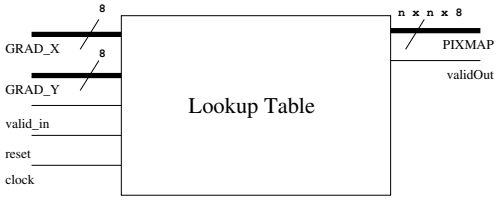

Figure 5: Lookup table input/output pins.

The two interesting parts of the lookup table are the line representation and the gradient to line conversion. We address these in turn.

2.2.1 Line Rendering and Storage

The user specifies the number of lines to store in the lookup table. These lines are distributed evenly over the interval $[-\pi/2, \pi/2]$ by angle. For antialiasing, the lines are discretely rendered onto an $8n \times 8n$ grid (Figure 6a), and then subsampled down to an $n \times n$ grid to get gray values (Figure 6b). This inefficient antialiasing algorithm is not used by most renderers, such as OpenGL, however it is very simple to code. Moreover, we do not care how long it takes as this process is a precomputation.

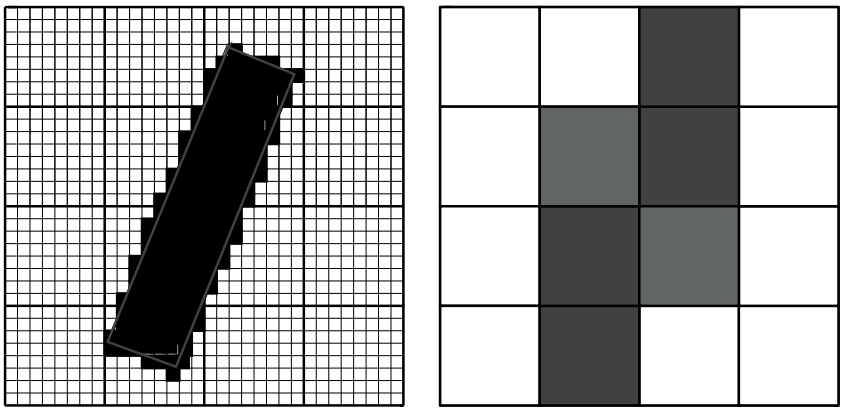

Figure 6: Antialiasing procedure.

Each line segment is stored using a user-specified number of bits per grid cell. After finding the appropriate line segment, the lookup table must decode the image to 8-bits per channel. This may seem like a trivial feature, however it turned out to be very useful. As we implemented the circuit, we found at one point that the timing constraints were not met. Changing parameter allowed us

to move from 4 bits per channel to 3 (substantially changing the lookup table code) and meet the time constraints without any work at all!

2.2.2 Gradient to Line Segment Conversion

After storing the line segments, we must provide a mechanism to translate gradients to the approprate line segment. The gradient is stored as a pair of signed 9-bit integers while the line segments are stored by angle. The appropriate conversion from one to the other is the arctan function, but this is not easily computed in hardware.

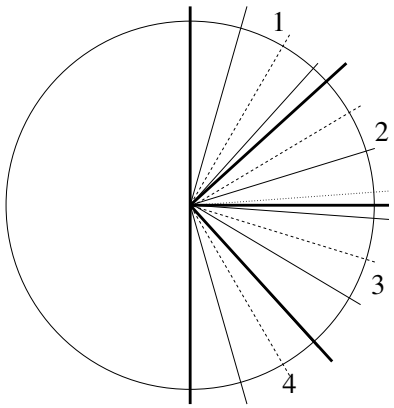

Figure 7: Bins and slices for line segment lookup.

To solve this we take each line segment (shown as thin solid lines in Figure 7), and compute a "bin" for that line segment (bounded by the dotted lines). If θ_1 and θ_2 are the angles of a bin's edges, then the gradient $[g_x, g_y]^T$ falls in this bin if

$$\theta_1 < \arctan(g_y/g_x) \leq \theta_2,$$

but this means:

$$\tan(\theta_1) < g_y/g_x \leq \tan(\theta_2). \tag{1}$$

The solution therefore is for the python script to precompute the tan function for the bin edges. Then we apply Equation 1 to find the right bin. Unfortunately, division is slow in hardware. So rather than divide by the denominator, we find the largest power of two smaller than the denominator (with a find-first-one implementation) and divide by that instead. This is speedy, but becomes more and more innacurate as the denomiator grows larger. Therefore we use Equation 1 only if $|g_y| > |g_x|$. Otherwise we perform the reciprocal division and check:

$$1/\tan(\theta_1) > g_x/g_y \geq 1/\tan(\theta_2). \tag{2}$$

The final wrinkle is that our division implementation works only for positive numbers. So we divide the *absolute value* of both numbers and then compute the sign of the output seperately.

In summary, we take the gradient and check both $|g_x| > |g_y|$ and $g_y/g_x > 0$. This indicates whether the gradient is in slice 1, 2, 3, or 4 of Figure 7. Then, we perform division and reciprocal division on the absolute values $|g_x|$ and $|g_y|$. Using all this information, we can chose the right bin.

3 Analysis and Retrospective

It is always difficult to judge the "correctness" of an artistic filter, in the sense that the output is not well defined. This is particularly true for filters such as ours whose output, a "sketched" rendition of video, does not really have a traditional analogue. That said, we were pleased with the output as a visually stimulating sketched rendition of the input video input stream. An input/output pair is given in Figure 8.

Figure 8: Sample input output pair.

A separate issue is whether correctly implemented the algorithm as stated. This we have checked by inspection of the video, and by running our implementation on various test patterns. The one problem is the approximation of division in the lookup table: our division procedure always *underestimates* the denominator. So our binning is artificially biased towards vertical and horizontal lines, and away from diagonal lines. This is somewhat noticeable in the output, but is not overly distracting.

As for speed, we were able to achieve interactive frame rates of 15 frames per second, but we believe that further pipelining the circuit could yield 30 frames a second. Looking more closely at the time, we see that updating the sliding window takes 7 cycles, while the lookup table computation takes 3 cycles. Our current implementation serially performs the three stages of our algorithm;

input window update, line look-up, and output window update. Therefore, the total time required to process one pixel is 17 cycles. As noted, this yields a frame rate of 15 frames a second. Each stage of the algorithm could be pipelined so that it would take 7 cycles from start to finish. We believe this would allow us to achieve 30 frames per second.

In terms of hardware, the number of registers used to implement the sliding windows is quadratic in the size of the window. The amount of RAM needed grows linearly. Within the lookup table, it seemed the most expensive components were the comparators. The number of comparisons is linear in the number of stored line-segments. This is user-specified, but should grow linearly with the size of the output window. Two multipliers were used to perform the thresholding of the gradient, but this would not increase if the size of the input window changed.

The entire circuit is straightforward to extend in a variety of ways. Each sliding window is parameterized up the declaration of the RAM modules. The Verilog `generate` statement appeared to be well suited to fully parameterize our sliding window circuit, but we found we could not use this statement with Active HDL and Synplicity. Nevertheless, it is trivial to change the sliding window implementations to windows of any size. Since the update of the sliding window takes constant time, total computation time should stay the same. The limiting factor in extending the size of the sliding window is the memory on the FPGA. Changing the output or input window size might allow us to draw longer lines, or perform fancier input calculations, such as the Laplacian. Memory limitations will also be stressed by the lookup table as the size of the output sliding window grows.

Memory issues aside, an interesting and easy extension would be to draw lines in color on the output device. The color with which to draw the line would be sampled from the pixel at the center of the gradient sliding window. The grey levels of the anti-aliased lines in our lookup tables could be translated into color levels of the sampled pixel.

Since lines are stored in look up tables, we could easily replace the lines with other "brush strokes," such as a paintbrush or charcoal. This might require a more complex algorithms to generate the lookup tables, but could easily fit into our framework without any modifications.

One important lesson we learned from the design process is to take advantage of the parallel nature of hardware. For instance, early designs of the sliding window used one large RAM instead of multiple RAMs. This forced the delay of the sliding window to grow linearly with the size of the sliding window. Switching to multiple RAMs allowed us to reduce the delay to constant time because we could perform multiple reads and writes in parallel.

4 Appendices

Attached to this paper is the python script used to generate the lookup table Verilog followed by an example output. We then present the rest of the Verilog,

and finally include the complete synthesis log.

References

[1] Aseem Agarwala. Snaketoonz : A semi-automatic approach to creating cel animation from video. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering*, pages 139–146, June 2002.

[2] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 421–430, August 1997.

[3] Aaron Hertzmann. Fast paint texture. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering*, pages 91–96, June 2002.

[4] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 517–526, July 2000.