GPU Processing Methods for Machine Vision

David Riley

1 Abstract

I present a novel model for performing 2D Gabor filtering for images on the GPU. Ideally, the model is built upon the linear-time recursive model from Young, van Vliet and Glinkel in [Young et al. 2002]. I describe how the recursive, feedback-based model upon which the algorithm is built can be adapted to the GPU despite the parallel, independent nature of the shader units. I examine the properties of this implementation vs. a GPU convolution implementation.

2 Introduction

The Gabor filter is a separable filter (extendable to multiple dimensions) whose kernel is composed of a Gaussian modulated by a sinusoid. In machine vision, it is particularly popular due to the resemblance if its output to that of a number of structures in the visual cortex of many animals (including humans) which detect edges at a given orientation for high-level feature recognition. The typical method of processing an image (or any data set) with the Gabor filter is through convolution. In recent times, a linear- time method of closely approximating the Gabor filter through an IIR-like feedback structure [Young et al. 2002] has been developed which produces a significant speedup over the relatively expensive convolution method.

The increasing programmability of the processing units of consumer-level GPUs has shown them to be of tremendous power when applied to parallel processing, especially processing based on floating-point math. The highly distributed nature of GPU processing, however, presents a problem for the recursive Gabor algorithm: most of the processing elements operate fully independently of each other and are generally incapable of seeing each others' outputs. This makes it quite difficult to feed the results from previous stages into the outputs of the recursive Gabor implementation. These obstacles, however, are not insurmountable.

The motivation for implementing this algorithm on the GPU is quite strong. The convolution-based method's run time is directly proportional to the number of pixels in the kernel; for a typical 32x32 kernel, this results in 1024 complex multiplications. Each dimension of the recursive filter requires a forward pass of 3 complex multiplications and a reverse pass of 4; for a 2D image, this results in 14 total complex multiplications per output pixel. In theory, this method should outperform even a 4x4 filter kernel for convolution, which would generally be too small to be useful.

3 Previous Work

The particular method of filtering I am attempting to implement on the GPU is (as mentioned) not new. [Young et al. 2002] describes a linear-time (with respect to the number of input pixels) method of performing Gabor filtering using a recursive feedback method similar to an IIR filter. The filtering process requires two passes in each dimension, and a 2D filter can be accomplished by running the filter on each dimension successively. The method is difficult to implement on the GPU for the reasons mentioned before, but the method for implementing recursive filters given in [Green 2005] turns out to work quite well (though it does introduce a fair amount of overhead).

Others have done Gabor shading on the GPU as well, typically using convolution. [Fung and Mann 2005] is a document describing the OpeNVIDIA project, which aims to do a number of machine vision tasks on the GPU. The document claims that the project does Gabor and Chirplet filtering in the GPU, but I have yet to find the implementation of such in the Sourceforge repository containing their code.

Most computer vision papers which use Gabor filters still use the convolution method for their filtering (for example, [Huang and Xie 2005], which was published significantly after Young et al's work, and [Liu and Wechsler 2002], which was published before or concurrently with it). For real-time tasks, the linear-time method can provide a significant speedup over convolution (even a 8x8 filter kernel, which is usually too small to be useful, requires 64 complex multiplications, while the linear-time method requires about 16 total complex multiplications). The recursive method also tends to be much more bandwidth-friendly as compared to the convolution method, since it primarily uses the results of the previous three calculations as its inputs instead of a great deal of memory locations.

There are a number of papers dealing with general-purpose computing efficiency on the GPU, especially for matters of memory bandwidth and cache management. In [Govindaraju et al. 2006], Govindaraju et al present a framework for effectively utilizing the caches and texture units on modern GPUs for efficient scientific computation. This appears to be a great part of the success of the FFT method presented in [Govindaraju et al. 2008]. The methods found could prove quite useful if cache issues arise or if large areas of memory end up needing to be used.

[Buck 2005] also supplies a number of helpful tips for executing general- purpose code quickly on GPUs. It largely deals with problems which are familiar to authors of conventional software but accentuated on GPUs, such as branching and memory bandwidth issues. Many of these tips were likely gleaned from the development of the framework described in [Buck et al. 2004], which details a stream-computing framework for GPUs.

[Fatahalian et al. 2004] details a number of issues regarding memory bandwidth when doing high-throughput calculations on GPUs. The paper notes a number of issues with GPU bandwidth, among which are the fact that the arithmetic units often cannot acquire data fast enough to feed their processors. It is worth noting that most of the GPUs involved in the study are somewhat older, and some of the issues may be somewhat ameliorated. The methods given in [Govindaraju et al. 2006] may be useful in avoiding some of the problems presented here; since this implementation of the filter is not likely to consume a great deal of texture memory bandwidth (except perhaps on the writing end).

Little information is available in the current academic literature on methods for doing feedback-based filtering in the GPU, presumably due to the difficulty of enforcing order of operation on the pixels. The game development community, however, seems to have come up with a few methods summarized in [Green 2005]. Most of the methods appear to be for real-time blurring operations (e.g. for depth-of-field approximations, among other things), however Green offers examples of recursive, resonance-based filters similar to the linear-time Gabor filter implemented here. The examples in that paper form the basis of this work.

4 Implementation

The important aspects of this algorithm are in two parts: the image processing algorithm itself, and the method by which the recursive linear-time version is implemented in the GPU. We will first present the various methods of obtaining a filtered image (culminating in the recursive method from [Young et al. 2002]), then present the obstacles to implementing them on a massively data-parallel machine such as a GPU. Finally, we will present the implementation of the linear-time algorithm, surmounting the issues present in GPU filtering.

4.1 Algorithm Descriptions

4.1.1 Typical Convolution-Based Method

The method used for most image filtering algorithms employs convolution, either using one 2D pass or two 1D passes (the latter is only possible if the filter kernel is separable). Convolution itself is a simple operation; an area of pixels is multiplied piecewise with another area of pixels and summed together. This is especially simple on the GPU; the filter kernel can be implemented as a small 2D texture (or two 1D textures, in the case of separable filters), and the convolution is a relatively straightforward function to implement in a fragment shader.

The problem with convolution-based methods, however, is their computational complexity (particularly their memory bandwidth requirements). A typical Gabor filter useful for machine vision will be between 16x16 and 64x64 pixels. A 2D convolution for these kernels would require between 256 and 4096 texture fetches for both the kernel and the source image as well as a comparable number of multiplication operations *per pixel*. Even the dual 1D version will require between 64 and 256 fetches and multiplications per pixel. This puts considerable strain on the memory bandwidth of the GPU, especially considering that the units will likely thrash the local texture cache due to the large area over which pixels must be pulled. One could, conceivably, optimize the convolution pixel shader to behave better with the cache, but the main problem remains the fact that such a large amount of data must be gathered from the source image and texture.

2D convolution filters have long been a bottleneck for machine vision research due to their high computational expense (though, depending on the task, the filtering is not always the most significant bottleneck). Thankfully, Young and Van Vliet developed a linear-time (in terms of total source pixels) Gabor filter implementation in [Young et al. 2002].

4.1.2 Linear-Time Recursive Method

Most of the details of the Young and Van Vliet method of filtering are better left to the original paper, but the core of the algorithm are two recursive, IIR-type filters (one each for the forward and backwards directions):

$$w[n] = in[n] - (b_1 * e^{j\Omega} * w[n-1] + b_2 * e^{j2\Omega} * w[n-2] + b_3 * e^{j3\Omega} * w[n-3]) \quad (1)$$

$$out[n] = B * w[n] - (b_1 * e^{-j\Omega} * out[n-1] + b_2 * e^{-j2\Omega} * out[n-2] + b_3 * e^{-j3\Omega} * out[n-3]) \quad (2)$$

The forward filter 1 is applied directly to the source image, while the reverse 2 is applied to the intermediate (w) produced by the forward filter. This bidirectional filter is common to recursive image filters due to the phase shifting which occurs in the forward direction. The filter is applied separately to the horizontal and vertical dimensions; the vertical filter is applied to the horizontal filter's result (though the order is not important).

Note also that the numbers produced are complex quantities, which makes the processing fairly daunting on a CPU (multiplies now take 4 multiplies and 2 adds). The operation isn't considerably better on a GPU; the GPU can perform some operations in parallel, but by and large a complex multiply must still be computed via 4 multiplies and 2 adds.

The recursiveness, however, introduces issues when transitioning to a GPU. The filter requires knowledge of its previous outputs, and this is difficult to obtain in a GPU; most graphics frameworks do not permit rendering to a texture that is being read from, or if they do, the results are frequently undefined because:

- The parallel nature of the processing means that the previous pixel is not even guaranteed to be complete before the next pixel begins
- The GPU's cache subsystem is often developed under the assumption that the textures are not being modified, thus there is no concept of making a "dirty" cache block (this problem is noted in [Green 2005], which also notes that this is implementation-dependent; their summed area table implementation that wrote back to the same texture worked on one generation of hardware and not the next)

Fortunately, methods of bypassing these issues (with some performance overhead) exist, again as detailed in [Green 2005].

4.2 Implementation Details

4.2.1 Framebuffer Magic

The solution to the problem lies in restricting the area to which the GPU renders. If we force the GPU to render one vertical line at a time, we can rotate destination buffers and source textures so that the shader used to compute the Gabor kernel uses the previous three lines as source textures and the next line as a destination texture. This way, we are never rendering into a texture we are reading from (as long as we have enough buffers, and generally the spacing is enough to keep the cache from interfering anyway), and by rendering individual lines, we enforce the order of rendering pixels to make the process go left to right (or top to bottom).

For the reverse process, we can simply move in the opposite direction (reversing the order of buffer rotation) and render the filter again with different coefficients. Looking at the algorithms, we discover several optimizations:

- Since the b_n and $e^{\pm jn\Omega}$ terms are constant per filter, they can be precomputed and pre-multiplied together, essentially forming single -term b_n coefficients for each recursive pixel
- Considering the above modification, formula 1 is basically formula 2 with a B value of 1.0 and different b_n values

Thus we only need one shader, with provisions for moving vertically as well as horizontally and the ability to change out the coefficients (accomplished through uniforms in the shader).

Additionally, the initial pixel values (the first three pixels defined in either direction, important since they use "previous" pixels that don't really exist) are computed by dividing the first input pixel by

a complex value. A complex division is a very expensive optimization, but it can be converted to a multiplication by simply storing the reciprocal of the divisor as a coefficient. For example, the initial pixel in the forward dimension is defined by [Young et al. 2002] as:

$$\frac{in[1]}{(1 + b_1 * e^{j\Omega} + b_2 * e^{j2\Omega} + b_3 * e^{j3\Omega})} \quad (3)$$

If we store $\frac{1}{(1+b_1*e^{j\Omega}+b_2*e^{j2\Omega}+b_3*e^{j3\Omega})}$ as a coefficient called $init\_factor$ supplied to the shader, the equation in the shader simply becomes $in[1] * init\_factor$.

4.2.2 Further Optimization

Singe GPUs typically actually render blocks of pixels in parallel, the lines method is inefficient. Rendering to a 2xn series of rectangles performs nearly twice as fast, but several problems are introduced:

- Half of the previous pixel values (the odd ones) come from different source textures depending on whether the destination is on the left or right half of the quad
- When the destination is on the right half of the quad, the problem is compounded by the fact that the previous pixel sample must come from the same texture as the source input (which is not being written to)

The first problem is fairly easily surmounted by setting a color in the left and right halves of the rectangles to 0 and 1, then using the color as the parameter for the mix() instruction in GLSL; the fetch can be performed from both prospective textures, and the mix parameter determines which one is eventually used.

The second problem is a bit more difficult, since reading the source from the destination buffer is not an option. The most useful option turns out to be "unrolling" the shader somewhat: A fourth previous pixel is retrieved (in addition to the three already being fetched) and the previous pixel is reconstructed from the values loaded. The overhead for this is actually surprisingly low compared to the overhead contributed from other sources (particularly the fill rate and vertex setup overhead).

When rendering a pass, the new values of the left/right indicator are computed and stored in the new alpha so that it will be ready for the next pass. For example, when rendering right to left (the first pass), the indicator should be 0 on the left and 1 on the right. The pass takes in a parameter for the delta direction of the next pass to determine how to generate the selector values; in this case, the left-right filter will generate the 0s on the right and the 1s on the left, since the other filter moves in the opposite direction. The right-left filter generates 0s on the bottom and 1s on the top, as the next pass will be bottom-top.

The filter, as you will recall, must be run in both the horizontal and vertical dimensions. The filter leaves its output behind in "stripes" across each of the textures it renders to, and the Gabor passes expect the source data to be laid out in the stripes. In the preparatory filter for the horizontal pass, which mainly does the job of converting the color values to luminosity and generating the first left-right selector values, the striping is accomplished simply by virtue of the fact that the render target changes every 2 pixels. To convert the left-right stripes to bottom-top stripes, we need to make a separate pass in order to pick up the data from the left-right stripes and, rendering from the bottom to the top in stripes, deposit them back. This requires picking up all four textures in the "tween" shader and selecting which one to pull the data out of by using the texture coordinate to determine which stripe is currently under the pixel. Note that the source textures (for obvious reasons) do not rotate when performing this pass. After this pass, the source textures are ready for the bottom-top pass.

Once the filters have been rendered, the results must be collated into the framebuffer in a final blitting pass. This pass is necessary because the shader cannot render into a single buffer due to the problems retrieving the previous pixels from one. There are advantages to performing this pass, however, as the visual representation of complex numbers is generally not particularly appealing, and the blit shader can be useful for extracting only the real or imaginary parts (or generating the magnitude). In the current implementation, the blit shader places the magnitude of the complex data ($\sqrt{re^2 + im^2}$) into the pixel and multiplies it by a gamma correction value (since the filters tend to produce somewhat dark images).

The final pass order, then, is:

1. YUV conversion (saving only Y) and preliminary striping
2. Gabor horizontal forward pass (left-right)
3. Gabor horizontal reverse pass (right-left)
4. Horizontal-to-vertical conversion
5. Gabor vertical forward pass (bottom-top)
6. Gabor vertical reverse pass (top-bottom)
7. Complex-to-magnitude conversion with gamma correction and blitting

5 Results

5.1 Filter Implementation

5.1.1 Texture Juggling

The filter turned out to be remarkably difficult to implement properly; there are a variety of preturbations that must be made to the texture-juggling routine depending on whether we are going in the reverse direction, the next pass will be going in the reverse direction, whether the pass needs a constant texture for a "backup" source (true for Gabor for the init pixel, false for all others), whether the textures should be rotated at all (no for the "tween" pass) and how many previous textures needed to be used.

The rotation logic actually ended up needing to use a queue to ensure that all textures were rotated in in the proper order; this is not strictly necessary, but provides an overall cleaner implementation than hackery with counters would; the queue is present in the first place because the Gabor filter requires the following textures:

- A "base source" texture to pull the init values from when necessary (tex0)
- A source texture for pulling input pixels from (tex1)
- The first previous texture, for pulling the second previous pixel (and possibly the first and third previous pixels) from (tex2)
- A second previous texture, for pulling the fourth previous pixel (and possibly the third previous pixel) from (tex3)

As it turns out, the "source" texture (tex1) and the first previous texture (tex2) are the same texture most of the time, since the pass is usually writing into the texture one ahead in the rotation of the source; the next stripe will therefore take (tex1 + 1) as the previous value, since it has just been written and take (tex1 + 1) as the new tex1, since the texture indices increment in a linear fashion.

5.1.2 Shader details

The shader implements the linear gabor filter as described in [Young et al. 2002]. The underlying hardware must support floating-point textures in order to give adequate visual quality; implementations which convert the shader's internal floating-point values back to 8-bit integers suffer an unacceptable degradation of quality (on the author's MacBook with an Intel GMAX3100, which does not support floating-point textures, the results fail to even look like Gabor- filtered images).

As mentioned before, in order to support the use of 2-pixel-wide "stripes" in rendering the passes, the filter is "unrolled", reconstructing the first previous pixel half the time in order to meet the requirements of shaders. The 0-or-1 indicator value present in the alpha value of the pixel helps determine whether the pixel is left or right; it is fed into the control parameter of the `mix()` linear interpolation function (`mix(x, y, a)`: $out = x * (1 - a) + y * a$) in order to switch between the two sources without using conditionals such as `if()` or ternary expressions (which incur a large performance penalty on most shader units). The left-right indicator also determines which pixel is used for the source for the third previous pixel, as it can be pulled from either tex2 or tex3 depending on the position; the shader just pulls both and switches afterwards.

The init pixel values, as described before, are computed by multiplying the init source pixel (the first in the line or column) by a pass-specific coefficient in order to get an appropriate starting value. This init value, as described in [Young et al. 2002], is used for the first three pixels as there is not enough previous data to properly construct them. Calculating the source pixel coordinate is not particularly elegant; one minus the absolute value of the delta vector (the direction in which we are proceeding) must be multiplied by the texture coordinate to obtain the correct line (for the horizontal pass) or column (for the vertical pass), and then the absolute value of the delta vector is multiplied by a pass-specific parameter indicating the extent of the source texture (as the shader has no information on this):

$$init\_coord = tex\_coord * (< 1.0, 1.0 > - abs(delta)) + (extent * abs(delta)) \quad (4)$$

The "tween" pass for converting the vertical stripes from the horizontal pass to horizontal stripes for the vertical pass has a difficult job. It must pull from different textures depending only on its location in the horizontal direction. Therefore, we cannot parameterize which texture the stripe pulls data from, as it pulls it from all of them (we can and must, however, send parameters describing the order of textures to pull from). The shader itself, then, decides which texture it must pull pixels from by its texture coordinate. Because most hardware implementations do not support variable indexing of array elements (i.e. if the shader pulls all four textures into an array of four pixels, the shader cannot just assign the output to be `pix[i]`), we use a series of `if()`/`else if()` statements. This theoretically has performance implications, since shader units do not branch well, but the approach seems to work quite well and may prove useful in handling some of the performance issues mentioned later.

5.2 Performance

Method	Recursive	Conv. 4x4	Conv. 8x8	Conv. 16x16
FPS	31	234	88	impossible

5.2.1 The convolution method runs faster?

The results table above bear some explanations and disclaimers. The first is the obvious problem: The convolution method seems faster. However, the convolution could only be implemented at a size of up to 8x8 on the test hardware (owing to limits on the size of a program and the fact that loops actually multiply the instruction count inside them). Anything larger would not execute. In any case, the performance decreases by a factor of 3 between the 4x4 and 8x8 cases, and it is reasonable to expect that a 16x16 kernel might reduce performance by a factor of 2-4 from the 8x8 case.

An 8x8 kernel is too small to be used on any useful machine vision tasks; it could realistically represent a sigma of about 2 or 3 without clipping the Gaussian, and 2 or 3 only covers the most minute of features. In the pictures later on in this section, you will see sigma values up to around 17 being usefully obtained from the data (and on higher resolutions, even larger sigmas would be appropriate). This would require at least a 32x32 kernel and more likely a 64x64 kernel (which may even surpass the limits of most newer hardware, since 4096 pixels must be calculated).

A separable kernel convolution implementation has not been tested; this would likely get better results (and would only require two passes to calculate).

5.2.2 Optimizations awaiting implementation

The recursive method used could be optimized somewhat. Its primary limitation is fill rate; at seven passes, even if drawing all passes as single quads (which does not produce correct results), the maximum frame rate is a bit over 100 fps at 640x480. The computational complexity of the Gabor passes seems to have very little effect; when a simple pass-through shader is used, the frame rate barely changes.

The fill rate issues, unfortunately, cannot be resolved unless a method to use fewer passes can be determined (which is not impossible, but is difficult). Our remaining major bottleneck is setup overhead; the 3-fold difference in performance when drawing as a single quad instead of many 2-pixel-wide strips should indicate this.

One potential optimization is to use wider strips. This reduces the number of calls made to `glDrawArrays()`, and experimentally does improve things somewhat; if the stripe width is set to 4 instead of 2, the frame rate jumps to over 40. Some hardware also renders pixels in 4x4 blocks instead of 2x2; this hardware would benefit much more from a wider stripe. The shader would have to be unrolled further, however, and some of the texture lookups could get somewhat complicated, but it would be surprising if this had a major impact on performance (also, we could get away with using only two alternating textures instead of four rotating ones, which ought to make things a little easier).

A further optimization might be to do the texture lookup determination inside the shader instead of manually reassigning texture units on the CPU before drawing each strip. Theoretically, this would allow us to make a single call to `glDrawArrays()` to draw the entire strip of quads, assuming that the hardware will still draw all the quads in sequence. Truthfully, this is probably where most of the bottleneck is, since the vertex setup and texture unit reassignment overhead from drawing single quads must be enormous.

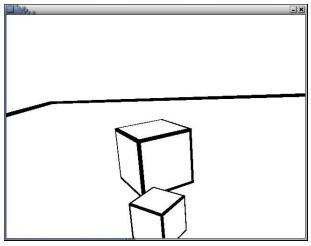

Figure 1: Our base image

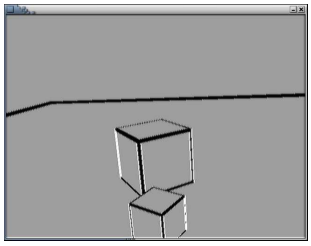

Figure 2: $\sigma = 1.5, \theta = 0\,\text{deg}$

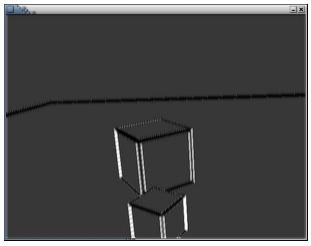

Figure 3: $\sigma = 3.375, \theta = 0\,\text{deg}$

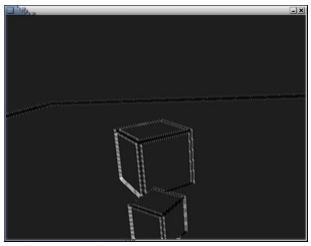

Figure 4: $\sigma = 3.375, \theta = 30\,\text{deg}$

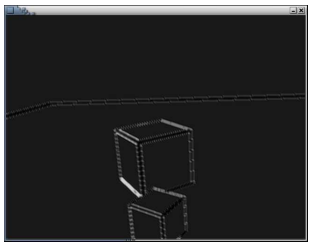

Figure 5: $\sigma = 3.375, \theta = 45\,\text{deg}$

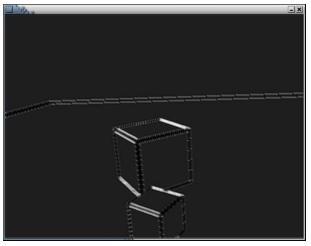

Figure 6: $\sigma = 3.375, \theta = 60\,\text{deg}$

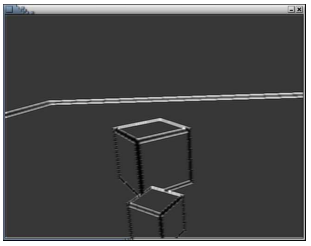

Figure 7: $\sigma = 3.375, \theta = 90\,\text{deg}$

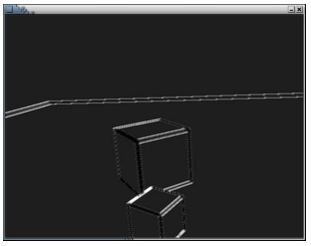

Figure 8: $\sigma = 3.375, \theta = 120\,\text{deg}$

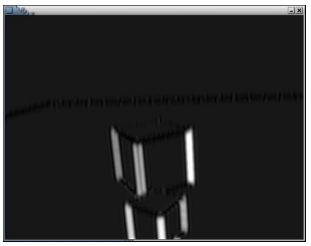

Figure 9: $\sigma = 7.5, \theta = 0\,\text{deg}$

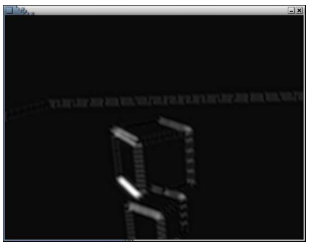

Figure 10: $\sigma = 7.5, \theta = 45\,\text{deg}$

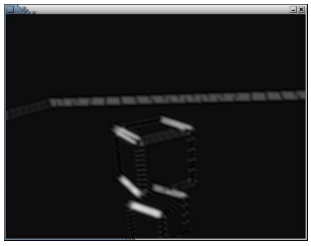

Figure 11: $\sigma = 7.5, \theta = 60\,\mathrm{deg}$

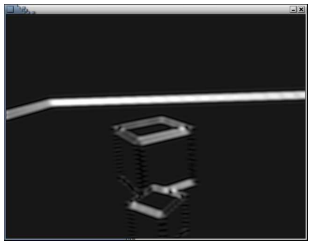

Figure 12: $\sigma = 7.5, \theta = 90\,\mathrm{deg}$

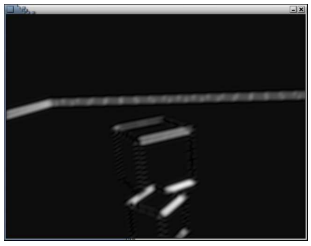

Figure 13: $\sigma = 7.5, \theta = 120\,\mathrm{deg}$

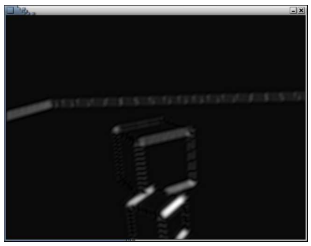

Figure 14: $\sigma = 7.5, \theta = 135\,\mathrm{deg}$

Figure 15: $\sigma = 11, \theta = 0\,\mathrm{deg}$

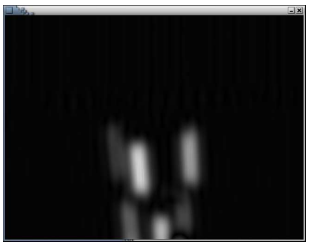

Figure 16: $\sigma = 17, \theta = 0\,\mathrm{deg}$

5.3 Pictures

References

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, ACM SIGGRAPH, 777–786.

BUCK, I. 2005. Gpu computation strategies & tricks. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, ACM SIGGRAPH, 134.

FATAHALIAN, K., SUGERMAN, J., AND HANRAHAN, P. 2004. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, New York, NY, USA, ACM SIGGRAPH, 133–137.

FUNG, J., AND MANN, S. 2005. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, ACM, New York, NY, USA, ACM SIGGRAPH, 849–852.

GOVINDARAJU, N. K., LARSEN, S., GRAY, J., AND MANOCHA, D. 2006. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, ACM, 89.

GOVINDARAJU, N. K., LLOYD, B., DOTSENKO, Y., SMITH, B., AND MANFERDELLI, J. 2008. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, Piscataway, NJ, USA, IEEE, 1–12.

GREEN, S. 2005. Image processing tricks in opengl. In *GDC 2005 Presentations*, NVIDIA, Game Developers Conference.

HUANG, Y., AND XIE, M. 2005. A novel character-recognition method based on gabor transform. *Communications, Circuits and Systems, 2005. Proceedings. 2005 International Conference on 2* (May), 815–819.

LIU, C., AND WECHSLER, H. 2002. Gabor feature based classification using the enhanced fisher linear discriminant model for face recognition. *Image Processing, IEEE Transactions on 11*, 4 (Apr), 467–476.

YOUNG, I., VAN VLIET, L., AND VAN GINKEL, M. 2002. Recursive gabor filtering. *Signal Processing, IEEE Transactions on 50*, 11 (Nov), 2798–2805.