

# Fast Matrix Multiplies using Graphics Hardware

E. Scott Larsen  
Department of Computer Science  
University of North Carolina at Chapel Hill  
Chapel Hill, NC 27599-3175 USA  
larsene@cs.unc.edu

David McAllister  
Department of Computer Science  
University of North Carolina at Chapel Hill  
Chapel Hill, NC 27599-3175 USA  
davemc@cs.unc.edu

We present a technique for large matrix-matrix multiplies using low cost graphics hardware. The result is computed by literally visualizing the computations of a simple parallel processing algorithm. Current graphics hardware technology has limited precision and thus limits immediate applicability of our algorithm. We include results demonstrating proof of concept, correctness, speedup, and a simple application. This is therefore forward looking research: a technique ready for technology on the horizon.

## Keywords

Matrix Multiplication, Graphics Hardware

## 1. INTRODUCTION

We present a technique for multiplying large matrices quickly using the graphics hardware found in a PC. The method is an adaptation of the technique from parallel computing of distributing the computation over a logically cube-shaped lattice of processors and performing a portion of the computation at each processor. Our technique is essentially to visualize this process. We represent matrix elements as colors and render an image of this cube in a rendering mode that properly computes the matrix product, which can then be read from the screen memory. Graphics hardware is specialized in a manner that makes it well suited to this particular problem, giving faster results in some cases than using a general-purpose processor.

When we refer throughout this paper to “graphics hardware,” we refer to hardware that is found in common desktop PC machines. The performance available will vary from vendor to vendor. For our research, we have used cards with nVidia’s GeForce3 GPU (Graphics Processing Unit). Other common PC hardware companies include ATI, Matrox, and 3D Labs. Additionally, UNIX workstations typically have graphics hardware comparable to PCs. In this work, we have emphasized portability and simplicity and have not used any specific vendor’s extensions. We suspect that improved per-

formance may be available using some vendor specific extensions.

We use the graphics hardware for numerical computation, rather than rendering images. This was also done by Hoff[4], who showed that graphics hardware was well suited to the problem of rapidly computing Voronoi regions of geometric primitives. Peercy et al.[6] and Proudfoot et al.[7] have developed shading languages that run on standard graphics hardware. A shading language is a specialized programming language used for describing the appearance of a scene, typically used in off-line rendering for computer animation. These papers illustrate that traditional fixed-function graphics hardware can be used much more generally than intended, in the proper circumstances. Hart[3] and Kautz et al.[5] have also used the multi-texturing capability of current graphics hardware (described in Section 2) in ways that go beyond those envisioned by the hardware architects.

Our work is somewhat forward looking, since current graphics hardware provides limited output precision for the pixel operations that our technique uses. Typically 8-bit fixed point output is provided, rather than the 32-bit floating point needed as a minimum for most numerical applications. However, we believe that floating point pixel operations will be available in future graphics hardware since game developers, the primary demand for PC graphics functionality, have shown great interest in this capability for quite some time.

Although graphics hardware is not the ideal hardware specialization for large matrix operations, we show that it possesses the necessary capabilities and yields competitive performance. It is also inexpensive and pervasive, unlike most specialized computational hardware.

In the remainder of the paper we give an overview of graphics hardware, then present our matrix multiplication technique. We then discuss precision issues, review our results, and finally contrast graphics processors with general-purpose processors to show under what circumstances the presented technique is competitive.

## 2. THE GRAPHICS PIPELINE

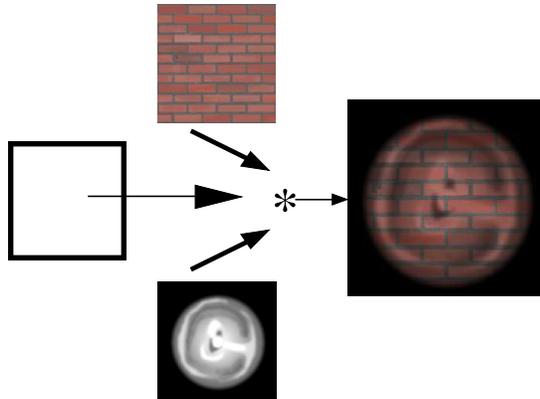
In this section, we overview the graphics pipeline used by graphics hardware. This is intended to give a brief and high level understanding of how the hardware renders scenes. We will not cover everything. For full treatment, we refer the interested reader to Foley[2] and Woo[8]. In order:

**Polygons** Surfaces of objects in the scene are represented as polygons. The hardware draws polygons by converting them to pixel fragments - a process known as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2001 November 2001, Denver

Copyright 2001 ACM 1-58113-293-X/01/0011 ...\$5.00.



**Figure 1: Multi-Texturing** Beginning with a simple white rectangle, we combine the brick texture with the flashlight texture in modulate mode to produce a wall with a flashlight pointed at it. The ability to modulate with two textures provides us with the ability to multiply values from two separate textures, using the graphics hardware.

“rasterization”. The polygon description includes the position of the vertices. The polygon description can also include information on how to texture the polygon.

**Texture Mapping** A “texture” is an image, usually 2D, which alters the appearance of the fragment. For example, a photograph of a brick wall may make a simple rectangle look more like a brick wall. Another application is the use of a texture whose intensities correspond to the amount of light falling on a surface. If this texture modulates the color of the fragment, then the appearance of lighting can be given without complex lighting equations. Figure 1 illustrates the use of two textures to alter the appearance of a single white rectangle. In the illustration, the two textures are multiplied together (pointwise). There are many different ways that the two textures can be combined. Multiplicative combination is what we are interested in here. The term “multi-texturing” refers to the application of more than one texture to the same surface.

**The Framebuffer and Blending** The fragments are then written into the framebuffer, which is a memory buffer in the graphics hardware’s local memory. The incoming fragment may optionally be blended in a variety of ways with the value already stored at that pixel of the frame buffer. The blending mode we use is to add the two fragments together, storing the result in the frame buffer. Pixel colors can be represented in the framebuffer using four channels: red, green, blue, and alpha. The alpha channel has various uses outside the scope of this paper. We treat all four channels identically.

**Read back** The contents of the framebuffer are most commonly scanned out to the screen for display. In our case, we want to copy the contents to main memory

also, in order for our program to have the results of our computations.

### 3. THE TECHNIQUE

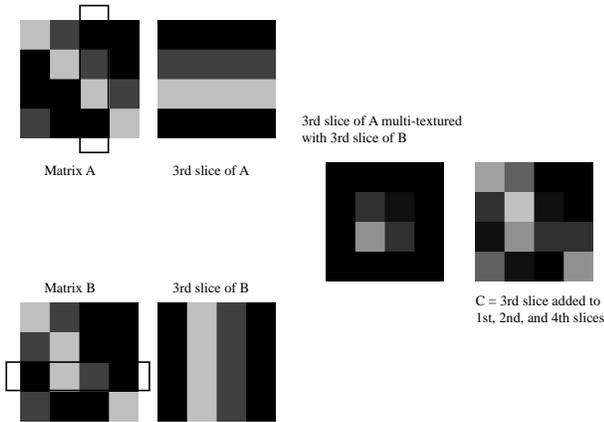
Multiple processors are often used in parallel to compute the multiplication of two matrices. One simple method starts with imagining processors arranged to fill a cube. We can then imagine the first matrix, lying horizontally, distributed in the usual chunk manner across all the processors on the top face of the cube. This distribution is replicated downwards, so that each horizontal “layer” is a distributed copy of the matrix on the top face. Likewise, we can imagine the second matrix, transposed, on a side face, distributed among the processors on that face, and replicated sideways through the cube. Each processor then performs the small matrix-matrix multiply of its sub-matrices. Finally, these results are summed onto the front face of the cube, and this now holds our answer. Our technique is to simply visualize this.

The naive visualization is to draw a little square for each processor in the cube. Let us first assume we have  $x \times y \times z$  processors, so each will only get one element of  $A$  and one element of  $B$  to multiply. This is done by creating two texture maps, one holds the data from  $A$  and the other  $B$ . Then we set the multi-texturing mode to “modulate” and assign the corresponding element of  $A$  and  $B$  to each processor (little square). We axis align the cube with our viewing window so that the front face of the cube is all we see, observing that the processors on the front face now occlude all the other processors. We use an orthographic view (no perspective adjustments) so that the little squares line up where they should. Finally, we set the blend mode to “sum,” and draw all of these little squares so that their individual results are added into the correct place on the screen, at the front face of the cube. The resulting matrix is retrieved as a memory copy from the graphics card to main memory, there is no requirement for a human to try to interpret the values seen on the monitor.

To simplify things, and speed them up, we first combine all the little squares that are in planes parallel to the front face into large squares – there are  $z$  of them. One column of texture  $A$  and one row of texture  $B$  are used for each large square, with the textures spreading perpendicularly to replicate the data. Figure 2 illustrates this. In short, the technique is to render  $z$  parallel rectangles (each  $x \times y$ ) one behind the other – we can only see the first, and it fills our view. Then texture map both matrices  $A$  and  $B$  onto each one, and sum them up onto the screen. The answer is then there on the screen. Hence, we say that our technique is to literally “visualize a simple parallel algorithm.”

### 4. PRECISION

We mentioned in Section 1 that this is somewhat forward looking research. This is because of the precision issues discussed in this section. We begin by tracing all data through the pathway. We specify two texture maps based on the two matrices we want to multiply. Historically, texture mapping in hardware has used fixed point arithmetic. These texture maps are limited to 8-bit precision. First, we multiply texture values. Experiments can show that the GeForce2 uses 14 bits of precision at this step, so underflow errors can occur here (e.g.  $1_8 * 1_8 = 00_8$ ). Experiments can also show



**Figure 2: Graphics Hardware Matrix Multiplication** The two matrices,  $A$  and  $B$ , are shown (white is 1 and black is 0). The 3rd column of  $A$  is replicated across a slice, and the 3rd row of  $B$  is replicated down the slice. The two are then shown multiplied together via multi-texturing. Finally, the result matrix,  $C$ , is the sum of this 3rd slice with the other three.

that the GeForce3 has this internal precision increased to 16 bits, correcting this issue.

Next, we sum up  $z$  of these values. We designate  $k$  to be the number of bits required to represent  $z$ . The correct sum requires  $k + 16$  bits, where only 16 are given. To make matters worse, the graphics hardware uses saturation arithmetic (i.e.  $FF_{16} + 01_{16} = FF_{16}$ ). This makes sense in graphics as adding intensity to an already saturated color should not cause it to wrap and turn black. But this makes it harder to design a higher precision fixed-point implementation. We can prevent the overflow in the adds from occurring by setting the polygon color to  $1/z$ . Since the textures are being multiplied by the polygon color, this effectively shifts off the least significant  $k$  bits before any of the adds take place. This gives us an answer that has not been saturated, but can be erroneous in the least significant  $k$  bits (because the  $k$  truncated bits may have introduced carries that should be visible). This is less useful when  $k$  approaches 8, because then all bits are shifted off and only zeros remain. Finally, the output, stored in the framebuffer, reduces the precision back to 8 bits. Further, what is really needed is floating point support, not just higher precision fixed point.

Assuming 8 bits in and 8 bits out are satisfactory for the application, the major source of error is overflow. As the occurrence of overflow is a result of adding  $z$  values with fixed point arithmetic, using values within the range  $[0, 1/z)$  prevents overflow. On matrices for which overflow does not occur, a reference implementation and our graphics accelerated implementation give identical results.

There is growing demand in the graphics industry for higher precision graphics arithmetic, and even for floating point arithmetic. There are various classic algorithms for improving precision, but we have found the primary obstacle here is the saturation arithmetic. A synthetic approach may be found, but we have strong suspicions that the graph-

ics hardware industry will provide more native support in the near future.

## Precision Effects and Applications

Eight bits is not satisfactory for many scientific applications. We first note that a variety of general-purpose processor vendors have introduced SIMD multimedia extensions, including the Intel MMX, MIPS MDMX, SPARC VIS, and HP PA2 architectures. Many of these provide for, and many applications have found good use for, 8-bit precision with optional saturation arithmetic. Applications using these architectures include audio, communications, graphics, image processing, video, etc. A discussion as to why the graphics hardware can be efficient for matrix multiplication, in the right application, is found in Section 6.

In theory, any large matrix operation would be a candidate for our technique. In practice, and with current technology, the right application needs to be resilient to the limited arithmetic mentioned already. Some applications that currently use MMX could be candidates. Some uses of Markov processes, which require raising a matrix to various powers, depend on which elements of the matrix go to one and zero the fastest rather than the precise value of the less important elements. All this said, application of our technique is certainly limited using current technology. The elegance and results shown in Section 5 suggest that this method of computing large matrix-matrix multiplications is useful, assuming that the future provides the technology. We believe this to be the case.

As a sample application, we have implemented the problem of finding connected components in a graph. This problem includes raising a square matrix to a power. Since the matrix is simply an adjacency matrix, with zero and non-zero values denoting connectivity, it can be implemented using the currently available fixed point precision. We discuss our results and implementation of this application in Section 5.

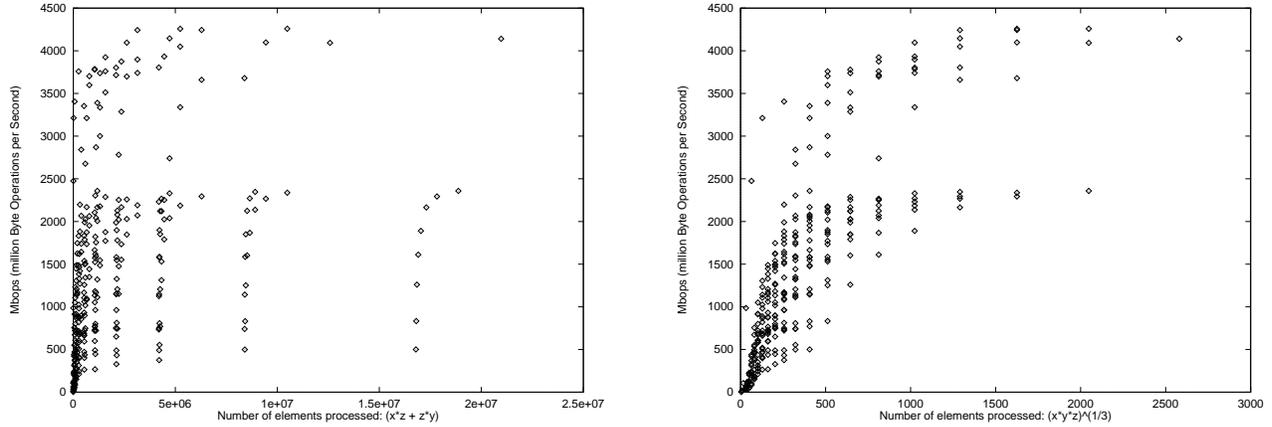
## 5. RESULTS

After hardware initialization, our program can compute many matrix matrix multiplies. As this is a constant time, one time setup cost, we did not consider it in our comparisons. This step takes about  $0.20 \pm 0.05$  seconds.

We focus on operation speed in order to make the results more generally comparable. Dongarra [1] has shown that a P4 with the SSE2 extensions can get 4.0 Gflop/s on ATLAS with 32-bit precision ( $n = 1000$ ). We would like to compete with this. In considering the results here, we again caution the reader regarding the precision of the resulting matrices, as discussed in Section 4.

When the graphics technology on the horizon becomes reality, then more significant tests can be done. At that time, we expect the competing technologies to have improved also. Competition includes general purpose processors, the MMX, SSE, and SSE2 extensions and SIMD instructions and extensions for other platforms, including MIPS MDMX, SPARC VIS, and HP PA2.

The operations performed on graphics hardware are byte sized. This puts serious comparisons at a significant disadvantage. Hence, we have measured Bops (byte operations per second). Our timings include the time to convert from matrix to texture map, time to send the textures to the graphics hardware memory, perform the calculations, copy



**Figure 3: Performance** Performance for matrix-matrix multiplies, in MBops (Byte Operations per Second).  $x$  ranges from 4 to 1024 and  $y$  and  $z$  range from 4 to 4096. In the left graph, the performance is shown as a function of the number of the number of elements passed into the computation. In the right graph, performance is shown as a function of the number of computations. The dips in the left graph and the lower strata of the right graph both correlate to oblong sized matrices. We believe this is caused by cache misses with oblong sized matrices.

the framebuffer back to main memory, and convert back to matrix format. In determining operation speed, we used an operation count of  $2 \times x \times y \times z - x \times y$  ( $x \times y \times z$  multiplies and  $x \times y \times z - x \times y$  adds). Using an nVidia GeForce3 graphics chip, we have achieved 4.4GBops performance. Performance as a function of matrix size is plotted in Figure 3. A texture can be laid out in texture memory in layouts which are optimal for common graphics applications. These layouts are often not the traditional linear layout. This can cause unusual cache misses with oblong textures. In the figure (3), there are two strata of results, with the lower representing oblong textures and the upper representing more square matrix sizes.

## Sample Application

An adjacency matrix raised to a power  $p$  yields a matrix with non-zero entries indicating paths of length  $p$ . Adding matrices of all powers less than and equal to  $p$  tells us which nodes cannot be reached by which other nodes in  $p$  steps. For our implementation of this application we raise an adjacency matrix to various powers. Note that  $M^p = M^a \times M^b$  where  $p = a + b$ . Using this, matrices can be raised to large powers using few matrix multiplies. We show timing results for multiplying a  $1024 \times 1024$  matrix by itself twenty times, using each result in the next multiplication.

To raise a matrix to successive powers requires copying the result matrix from the frame buffer to a texture so that it may be used as input. This memory copy is done locally in video memory and is approximately twice as fast as the time to copy a matrix to or from application memory. This is illustrated by the times shown in Figure 4.

All three kinds of memory copies - loading matrices into textures, copying the result from the frame buffer to a texture for successive powers, and copying the final result from the frame buffer to application memory - are a small frac-

tion of the total time for reasonably sized matrices. This is shown in Figure 5.

## Implementation

We mention here some observations we made during our implementation that may be of interest to those duplicating our results.

**Refresh Rate** We found that setting the refresh rate on the monitor as low as possible made marginal improvements (about 10%).

**RGBA** We found that 4 numbers can be packed into a single pixel, by setting the red, green, blue, and alpha channels to different values.

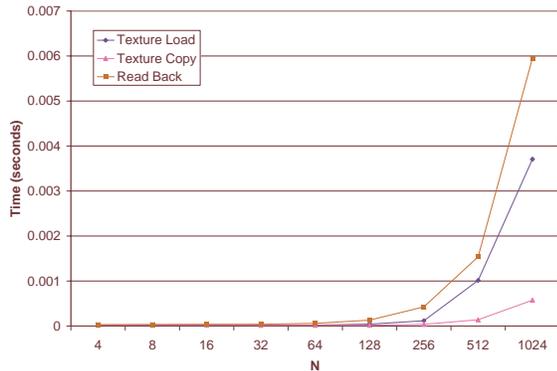
**Texture Format** Changing the format of the texture creation and read-back from RGBA to ABGR\_EXT (in OpenGL) gave about 40% improvement on our hardware. This is because the hardware driver avoids reformatting the data from the application format to the card format. There is a number of options here, with near equal performance for each option except the one used natively on the specific hardware. The native format should give significant improvement.

**Full Screen** Running full screen instead of in a window provides improved performance.

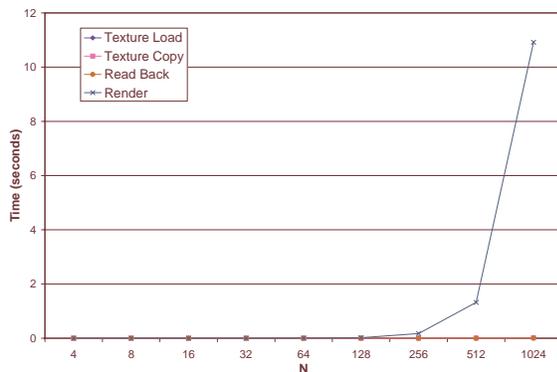
Various other optimizations yielded minor (<1%) improvements.

## 6. CONCLUSION

We expected our graphics hardware matrix multiplication technique to provide speedup over CPU implementations. The primary reasons for this are the memory system and the specialized processing of the graphics hardware. The



**Figure 4: Comparison of memory copy times** Texture Load is the time to copy matrices from application memory to the graphics card. Copy is the time to copy a result matrix from the frame buffer to texture memory, potentially without a round trip to host memory. This is only done when raising a matrix to a successive power. Read Back is the time to read a final result from the frame buffer to application memory.



**Figure 5: Breakdown of times in our sample application** For performing twenty matrix multiplies, the time to perform the multiplies and accumulates vastly dominates the time to set up the problem by copying the matrices to and from the graphics memory.

problem of large matrix-matrix multiplies is traditionally memory limited. Since the matrices are not large enough to store in level 2 cache, the problem becomes one of streaming memory. 3D computer graphics also includes the problem of streaming memory, and the specialized memory subsystem on graphics cards is made for this problem.

The memory subsystem on current graphics hardware is 128 bits wide, running at double data rate, allowing 256 bits to be read or written per clock. The GeForce3 memory clock is 233 MHz, yielding 7.4 GB/sec memory bandwidth. With this particular chip, the 128 bit interface is divided into four independent 32-bit memory busses, so four memory streams may be read or written at burst speeds. Compare this to the memory interface of an AMD Athlon CPU, which is 64 bits wide, at 133 MHz, optionally double data rate, yielding up to 2.1 GB/sec memory bandwidth. Likewise, the Intel Pentium IV with PC800 (RAMBUS) memory benchmarks at 1.5GB/sec.

Note that although CPUs have much higher clock rates than graphics chips (e.g., 1.4 GHz vs. 200 MHz), this has little impact because in a memory streaming problem, memory reads dominate. Even with memory Pre-fetching, the up to 10:1 ratio between CPU clock speed and memory bus clock speed prevents the CPU from going full speed.

The processors in the graphics hardware do not have to be as general purpose as CPUs. They optimize for a different set of operations that are more common for graphics. The highly accelerated operations of multiplication of texture maps and accumulation into the frame buffer is essentially the same as the multiply-accumulate used in matrix matrix multiplication.

Unfortunately, most graphics cards must perform a frame buffer read to implement the accumulate, whereas a CPU can accumulate into a register. An embedded frame buffer graphics architecture such as the Sony PlayStation2 might alleviate this problem. Specifically, the time to compute a single matrix multiply for a  $1024 \times 1024$  matrix is 0.546 seconds on the GeForce3. With the 7.4 GB/sec memory bandwidth, this yields a theoretical bandwidth consumption of 4.1 GB. Since the processor performs  $1024^3$  multiply-accumulate operations, this is an average of 3.79 bytes read or written per multiply-accumulate. This corresponds very closely to reading one element from each texture (operand matrix) and one element from the frame buffer (result matrix) and writing one element to the frame buffer. Texture cache efficiencies seem to be nearly offset by inefficiencies elsewhere in the pipeline. This shows that our hardware is bandwidth limited for this application. On the other hand, an embedded framebuffer would avoid the framebuffer reads and writes and could significantly improve performance.

When comparing the results presented in this paper to those of Dongarra [1], it is important to note that our matrix elements are 8 bits, whereas the cited results of Dongarra are for 32 bit elements. Both implementations achieve the same number of operations per second, although our elements are one fourth as large. Thus, when graphics hardware supports 32-bit arithmetic we believe that despite the much higher memory bandwidth of graphics hardware, our technique will not be competitive against cache aware CPU implementations unless frame buffer reads and writes can be performed at register speeds and graphics chip core clock rates increase significantly.

## Summary

We demonstrate that literal “visualization” of the work done in a simple parallel algorithm for matrix-matrix multiplication can calculate solutions quickly using common 3D graphics accelerators. These “solutions” currently have limited precision (which limits applicability), but the graphics hardware industry is making strides towards changing this. Graphics hardware is low cost and computationally powerful. We use this inexpensive specialized hardware to solve a numerical problem to which it is also well suited.

## 7. ACKNOWLEDGMENTS

We want to acknowledge the aid and support of Amy Larsen, Herman Towles, and Eldon Larsen, Henry Fuchs, and also those who have reviewed and given constructive criticisms on this work.

## 8. REFERENCES

- [1] J. Dongarra. An update of a couple of tools: ATLAS and PAPI. DOE Salishan Meeting (Available from <http://www.netlib.org/utk/people/JackDongarra/SLIDES/salishan.ps>), April 2001.
- [2] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, second edition, 1996.
- [3] J. C. Hart. Perlin noise pixel shaders. In *Graphics Hardware*, Eurographics/SIGGRAPH Workshop Proceedings, August 2001.
- [4] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Computer Graphics*, SIGGRAPH Annual Conference Proceedings. ACM, 1999.
- [5] J. Kautz and M. D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In *Rendering Techniques*, Proceedings of the 10th Eurographics Workshop on Rendering, pages 281–292, June 1999.
- [6] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multipass programmable shading. In *Computer Graphics*, SIGGRAPH Annual Conference Proceedings, pages 425–432. ACM, 2000.
- [7] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time programmable shading system for programmable graphics hardware. In *Computer Graphics*, SIGGRAPH Annual Conference Proceedings. ACM, 2001.
- [8] M. Woo, J. Neider, T. Davis, and O. A. R. Board. *OpenGL Programming Guide*. Addison-Wesley, second edition, 1997.