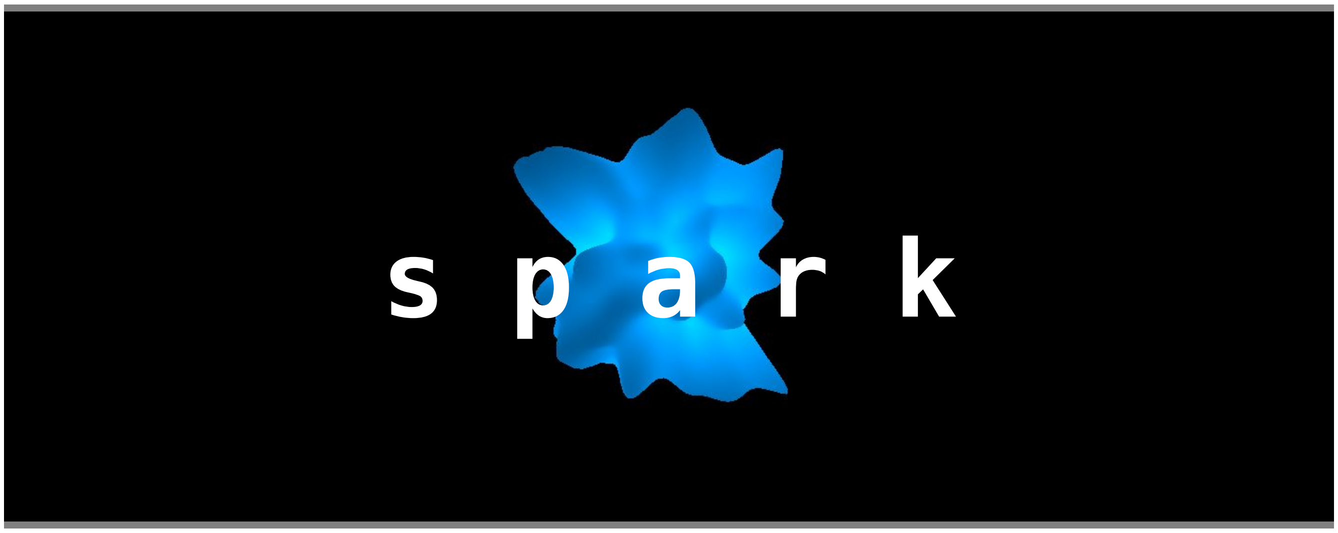
Aug 01, 2004

A Stream Programming Abstraction and Rendering Toolkit



for Programmable Graphics Hardware

Masters Project
MSc Computer Animation
NCCA Bournemouth University

Derek Gerstmann - C1405511

Table of Contents

Illustration Index

1 INTRODUCTION

This thesis describes the theory, mechanics, and design of an abstract toolkit for stream processing and rendering using programmable graphics hardware, with an emphasis on procedural texture synthesis.

The goal of this project was to develop a toolkit which utilizes programmable graphics hardware to do high quality texture synthesis.

Rather than writing simplified programs in a high level shading language which executes in a single pass and conforms to the hardware limitations that currently exist, this project aims to provide a framework which uses a stream processing model for general purpose computation allowing high quality textures to be created without sacrificing detail.

The implementation presented in this thesis, *“A Stream Processing Abstraction and Rendering Toolkit (SPARK) for Programmable Graphics Hardware”*, is a minimal framework which provides an API and abstraction layer for stream oriented programming.

Also included is a demonstration application which uses this framework to illustrate the functionality and feasibility of this solution. This solution also demonstrates an incredible performance advantage over traditional CPU based solutions for certain types of spectral synthesis.

The main topics covered in this thesis include streaming architectures and stream oriented programming, general purpose computation using programmable graphics hardware, and the design and implementation of SPARK.

Additional information has been provided on a DVDROM included with this thesis. This DVDROM contains the full source code of SPARK and the demonstration program, video files showcasing the functionality of the demonstrations, and the full API documentation for SPARK.

1.1 OVERVIEW

Graphics processing units (GPUs) are specialized processors designed specifically for real-time graphics applications. However, with the recent addition of programmability in certain stages of the graphics pipeline, there exists the possibility to perform general purpose computation on a GPU (GPGPU).

Many researchers have considered using graphics processing units (GPUs) as an additional source of processing power to offload work traditional sent to a computer's central processing unit (CPU). With GPU performance now exceeding that of CPUs in floating point arithmetic, GPUs offer an economically attractive source of additional processing power.

1.2 PERFORMANCE TRENDS

For several years the performance trend of GPUs has exceeded the characteristic “Moore's Law” for CPUs which describes the yearly exponential growth in the number of transistors per integrated circuit for commercially available CPUs.

As evidence of this trend, a comparison of raw floating point performance trends between CPUs (Intel's Pentium 4) and GPUs (NVidia NV30, NV35, NV40, and ATI's R300, R360, R420) for vector multiplication can be seen in the following chart [Hanrahan 2004].

FLOATING POINT MULTIPLIES PER SECOND

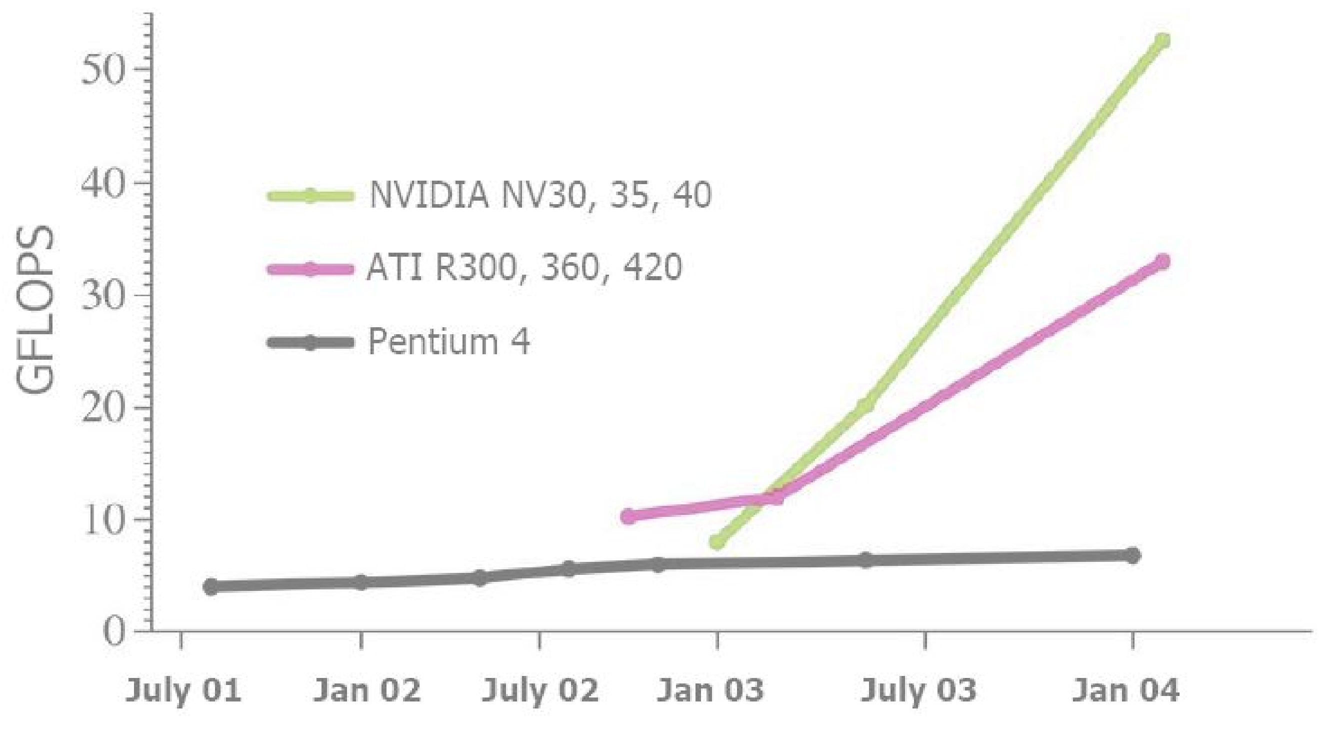


FIGURE 1 Performance trends of floating point multiplication of CPU's vs GPUs. [Hanrahan, 2004].

Based on current manufacturers specifications, an Intel Pentium 4 SSE processor can theoretically perform a floating point vector multiply at a rate of 6 GFLOPS (giga, or 10^9, floating point operations per second). In contrast, a previous generation GPU, the NVidia GeForce FX 5900 (NV35), demonstrated a performance rate of 20 GFLOPS in a single fragment program [Buck, 2003].

Furthermore, in a cost efficiency analysis of these processors based on current consumer prices and the measurements listed above, the Pentium 4 SSE rates at 24 MFLOPS/DOLLAR (mega, or 10^6, floating point operations per dollar) compared to 80 MFLOPS/DOLLAR for the NVidia GPU.

CPU VS GPU PERFORMANCE AND COST EFFICIENCY

Intel Pentium 4 SSE 3 GHZ ($250)	NVidia GeForce 5900 ($250)
– ~ 6 GLOPS	– ~20 GFLOPS
– ~24 MFLOPS/DOLLAR (peak)	– ~80 MFLOPS/DOLLAR (peak)

FIGURE 2 CPU vs GPU performance and cost efficiency for certain applications [Buck 2003][DG].

1.3 COMPLICATIONS

Unfortunately, the raw processing potential of a GPU is not easy to exploit. Fundamentally, this is because GPUs were developed around a fixed function pipeline for real-time graphics, not for general purpose computation. Only recently has programmability been added, but most GPUs continue to retain a large portion of fixed functionality for graphics processing.

As a result, only applications which fit within the imposed limits of a GPU and are designed to exploit the GPU's underlying architecture will be computationally efficient. This requires algorithms to be optimized and performed in parallel, and data to be managed effectively.

Many researchers have identified that these problems can be solved in an elegant manner if a “stream processing” model is applied to the underlying system architecture. A brief analysis of general streaming architectures will be useful in order to gain an insight into how a GPU should be managed and to aid in the development of an abstract framework for processing data in a stream oriented manner.

2 STREAMING ARCHITECTURES

Nearly all modern computers use an architectural model developed by John von Neumann. This type of “von Neumann” architecture places both the instructions and the data required (and/or generated) by an operation in the same storage space [Burks et al Page 35].

With few exceptions, this architectural design fundamentally requires that operations be performed on a single unit of data. As a result, instructions tend to “flow” through the system, while data tends to stay relatively still. By providing a substantial set of fixed function operations, a high performance processor can be built relatively inexpensively by performing computations in serial, or one after another.

In contrast to the von Neumann architecture, a streaming architecture is designed to perform the same operation on a large set of data in a parallel manner (at the same time). This model allows data to flow through the system, while operations tend to stay relatively still. This can be extremely effective for processing large amounts of data, if the operations are powerful, and the flow of data is minimized [Brooks 2004].

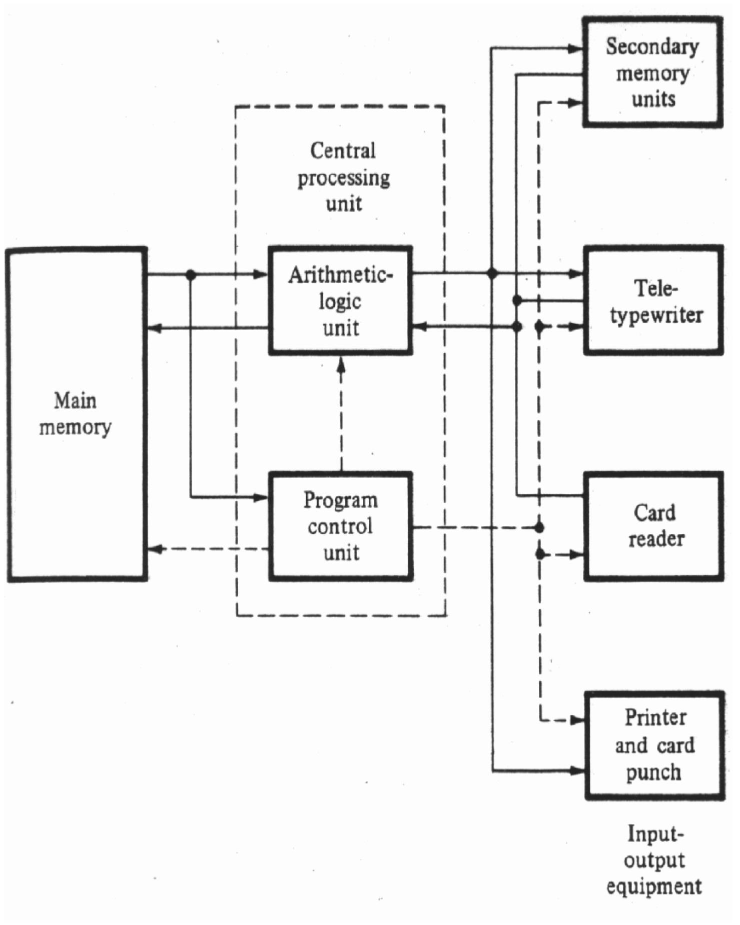
2.1 EARLY STREAMING SYSTEMS

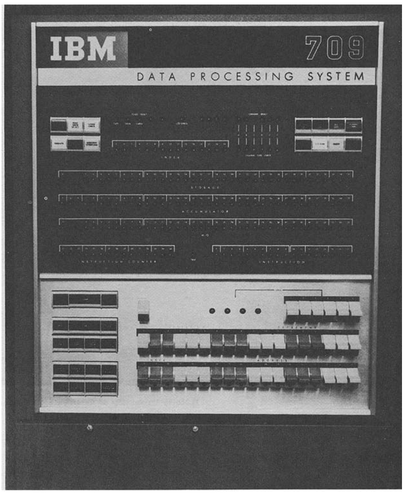
One of the first examples of a general purpose computer with a streaming architecture can be traced back to the IBM 709 mainframe data processing system introduced in 1957.

2.1.1 IBM 709

The IBM 709 included three CONVERT operations which could be applied to streams of data by performing a table lookup operation on each data value. In this instance, a single 6 bit byte value from a data stream could be indexed into a table for specific applications, such as character translation, binary-to-decimal conversion, and decimal addition. This streaming table-lookup offered a powerful and flexible way to apply very basic commands to large amounts of data [Brooks 2004].

IBM 709 STREAMING ARCHITECTURE





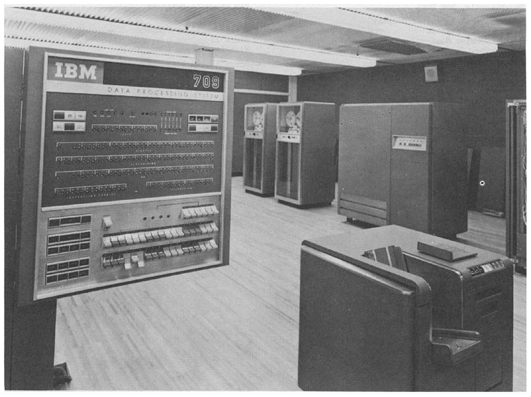


FIGURE 3 Block diagram of the IBM 709 system architecture, and photos [Salomon][Harper].

Another notable implementation of an early streaming architecture was the IBM 7950 (HARVEST) plug-in card for the IBM 7030 (STRETCH) computer. It was designed and built exclusively for the United States National Security Agency (NSA) and was put into operation in 1961. Although only a single unit was ever built, it was operated for over 14 years doing byte-by-byte cryptographic analysis before being decommissioned in 1976 [Brooks 2004].

IBM 7950 STREAMING ARCHITECTURE

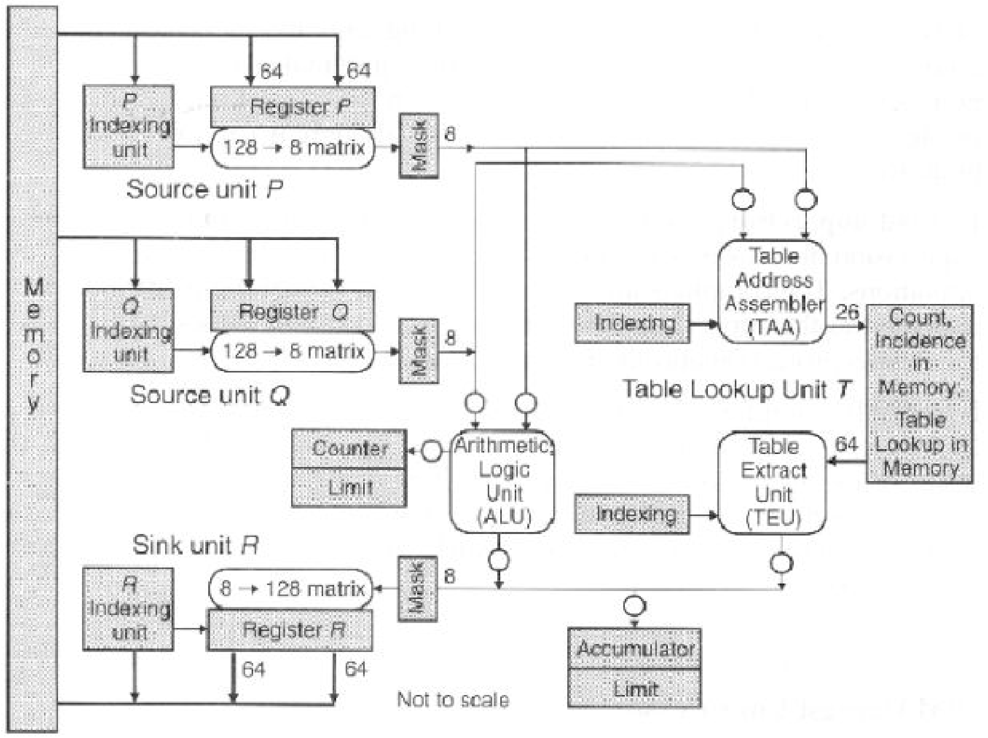


FIGURE 4 Block diagram of the IBM 7950 HARVEST streaming data path architecture [IBM 1957].

2.1.2 IBM 7950

The IBM 7950 HARVEST was unique in that it allowed the processor to be pro grammatically configured to operate on a set of streaming data read from memory. After performing the computation, the data could either be combined with the initial input, used to lookup values in a table (streaming look-up), or possibly counted, before being written back to a separate stream of memory. This allowed a well informed programmer to develop finite state machines for various operations using this architecture.

A list of some of the applications developed using these powerful streaming operations are included in the list below.

IBM 7950 STREAMING APPLICATIONS

- Sort, Merge, and Collate Large Amounts of Data
- Count and Accumulate, Incidence in Memory
- Convert Roman Numerals to Arabic
- Flow Data through Multiple Permutation Tables
- Detect Low-Probability Sequences [Bayes]
- Determine the Language of a Text [Bayes]
- Randomly create valid hymn tunes [Markov]

FIGURE 5 Applications using the IBM 7950's streaming architecture [Brooks, '04].

2.2 MODERN STREAMING SYSTEMS

More recently, in 2002, Stanford University and Texas Instruments (TI) of the United States used some of the early architectural models to jointly design a modern streaming architecture as a test bed for novel microprocessor design, and, in the process, address some of the restrictions of conventional semiconductor technologies.

VLSI BANDWIDTH LIMITATIONS

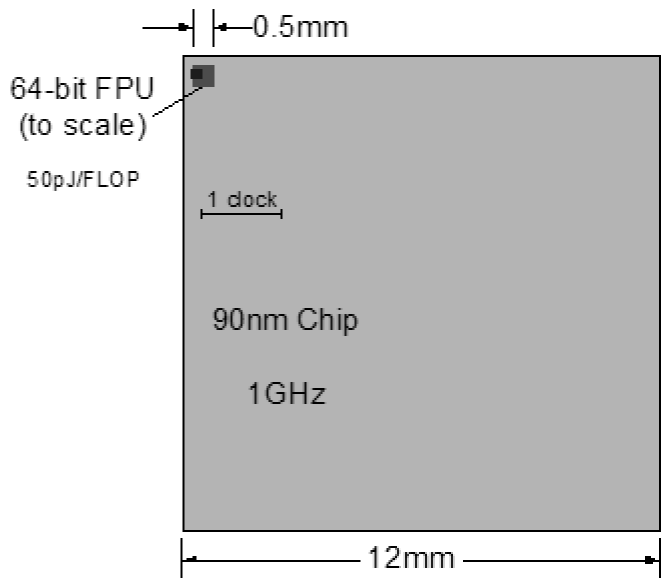


FIGURE 6 Relative distance data can travel in one clock for a 90nm chip at 1GHz [Brooks 2004].

2.3 VLSI AND STREAMING ARCHITECTURES

One of the primary issues with traditional microprocessors is the expense of traditional VLSI (Very Large Scale Integration) technology in regards to bandwidth and data locality. Modern microprocessors use VLSI technology to organize a massive number of semiconductor integrated circuits into individual logic units. Using recent semiconductor manufacturing processes, incredible amounts of arithmetic processing power can be packed into a single chip, with relatively little expensive.

Unfortunately, a consequence of this technology is that, although arithmetic operations are relatively cheap to perform due to the abundance of arithmetic logic units (ALUs), bandwidth becomes an extreme limiting factor for performance [Dally et al, 2003]. Parallelism can reduce this affect, as shown by vector processors which operate on multiple data for a single instruction [Sima et al 1997], but the locality of data still remains to be an issue.

The solution developed by the Stanford group utilizes a streaming architecture to exploit data parallelism and locality through a set of on-chip local registers. This reduces the distance that data must travel between operations, and minimizes the performance degrading effects of limited global bandwidth.

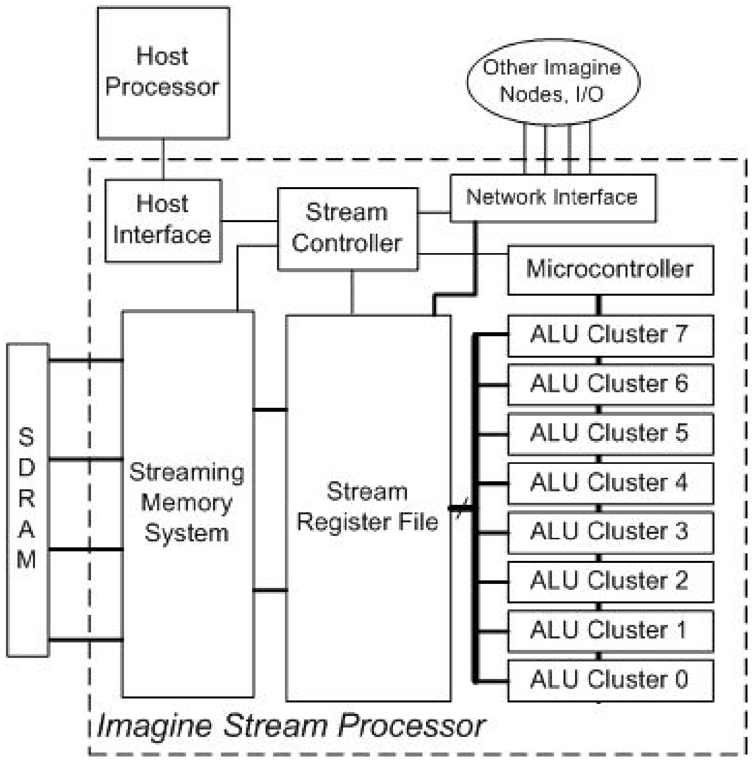
2.3.1 THE IMAGINE STREAM PROCESSOR

The initial prototype developed by Stanford and TI, the IMAGINE stream processor, was built to test the architectural design and its applicability towards general purpose processing. It's main applications include digital image and signal processing.

The IMAGINE prototype packs 21 million transistors onto a silicon die size of 16mm x 16mm in a 0.15 micron standard cell technology, and operates at 400MHz with a power dissipation of 10 Watts. It has demonstrated a sustained performance of 18.3 GFLOPS on certain applications, which is comparable to conventional CPUs operating at more than 2x the clock speed, with nearly 10x the number of transistors [KAPASI et al 2002].

The IMAGINE architecture pioneered the use of a storage bandwidth hierarchy, which enabled the processor to efficiently operate 48 floating point ALUs in parallel. The hierarchy consisted of three tiers of storage: a streaming memory system with a maximum throughput of 2.1GB/sec, a 128KB stream register file (SRF) operating at 25.6GB/sec, and local register files (LRFs) for directing results between ALUs, with a maximum throughput of 435GB/sec [Kapasi et al 2002].

STANFORD/TI IMAGINE STREAM PROCESSOR



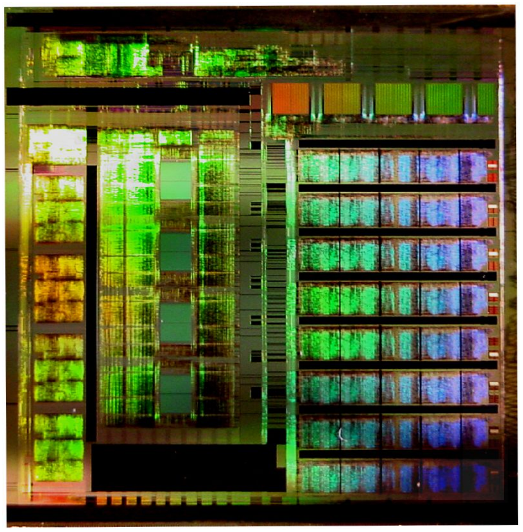


FIGURE 7 Block diagram for the Stanford/TI IMAGINE stream processor, and photo. [Brooks, 2004].

2.3.2 THE MERRIMAC STREAM COMPUTER

Stanford later demonstrated the processing potential of their stream architecture by building an advanced, scalable, streaming scientific computer, called MERRIMAC.

This system utilizes advanced interconnection networks to harness the streaming architecture of multiple stream processors, enabling a platform which can be scaled from a 2 TFLOPS (tera, or 10^{12}, floating point operations per second) workstation, to a 2 PFLOPS (peta, or 10^{15}, floating point operations per second) supercomputer [Dally et al 2003].

STANFORDS MERRIMAC STREAM COMPUTER

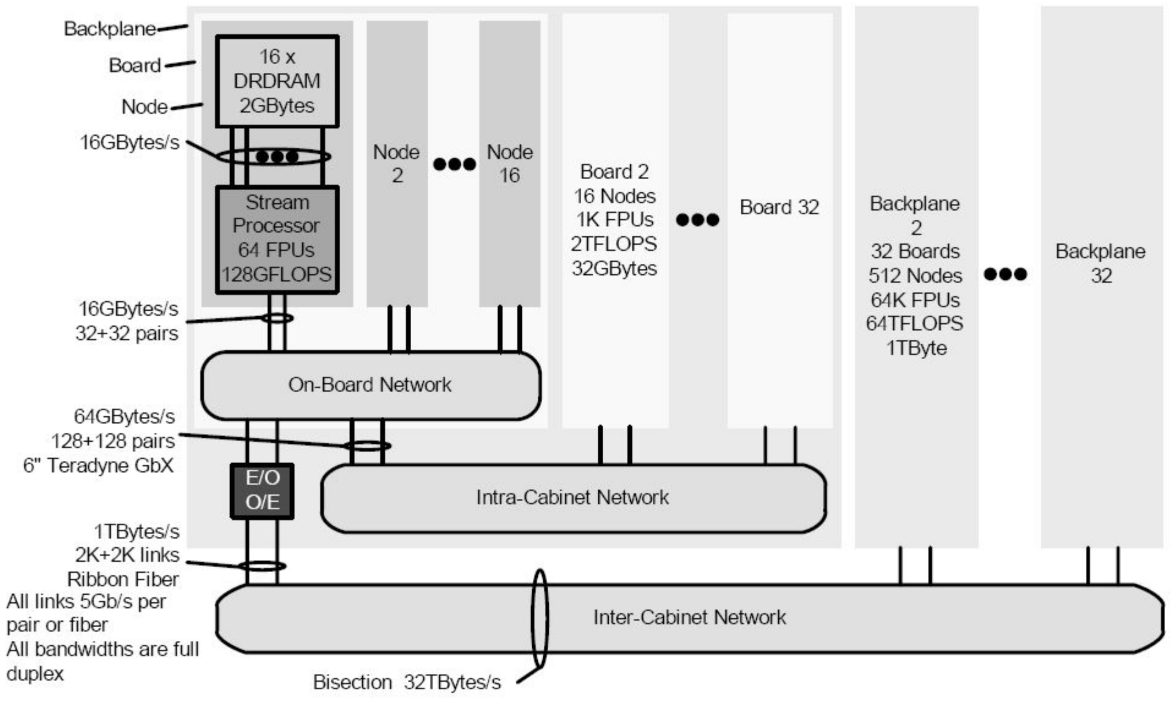

Backplane
Board
Node
16GBytes/s
16 x DRDRAM 2GBytes
Stream Processor 64 FPUs 128GFLOPS
Node 2
Node 16
Board 2 16 Nodes 1K FPUs 2TFLOPS 32GBytes
Board 32
Backplane 2 32 Boards 512 Nodes 64K FPUs 64TFLOPS 1TByte
Backplane 32
16GBytes/s 32+32 pairs
On-Board Network
64GBytes/s 128+128 pairs 6" Teradyne GbX
E/O O/E
Intra-Cabinet Network
1TBytes/s 2K+2K links Ribbon Fiber
All links 5Gb/s per pair or fiber
All bandwidths are full duplex
Inter-Cabinet Network
Bisection 32TBytes/s

FIGURE 8 System overview block diagram of Stanford's MERRIMAC [Stanford].

Each node in a MERRIMAC system consists of an advanced stream processor similar to the IMAGINE prototype, but uses a 90nm CMOS fabrication process and a modified system architecture with 64 FPUs (floating point arithmetic units) operating at 1GHz. This results in a peak performance of 128 GFLOPS, a peak efficiency of 128 MFLOPS/DOLLAR and a sustained efficiency of 23-64 MFLOPS/DOLLAR for specific applications [Dally et al 2003].

STANFORDS MERRIMAC STREAM PROCESSOR

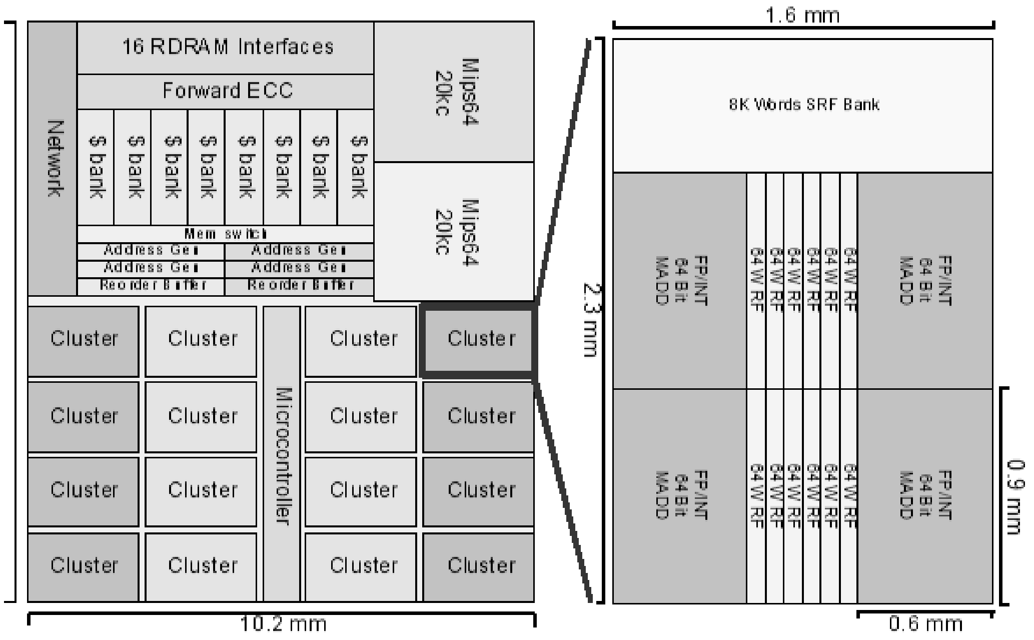


FIGURE 9 Individual processor block for a MERRIMAC stream processor [Dally, 2004].

3 PROGRAMMABLE GRAPHICS HARDWARE

Programmable GPUs are not stream processors, however, they do have similar capabilities [Hanrahan 2004][McCool 2004], and will likely approach a stream processor's abilities in the near future as they approach a more generalized vector processing architecture [Rost 2004].

Nevertheless, with the current GPUs operating in excess of 400-500MHz with a fabrication process of 0.13 microns, GPU vendors have produced some incredibly high performance specialized processors, with high economic values.

There are several underlying architectural design decisions which allow a GPU to exhibit such incredible performance benefits for graphics, but at the same time limit their applications for general purpose computing.

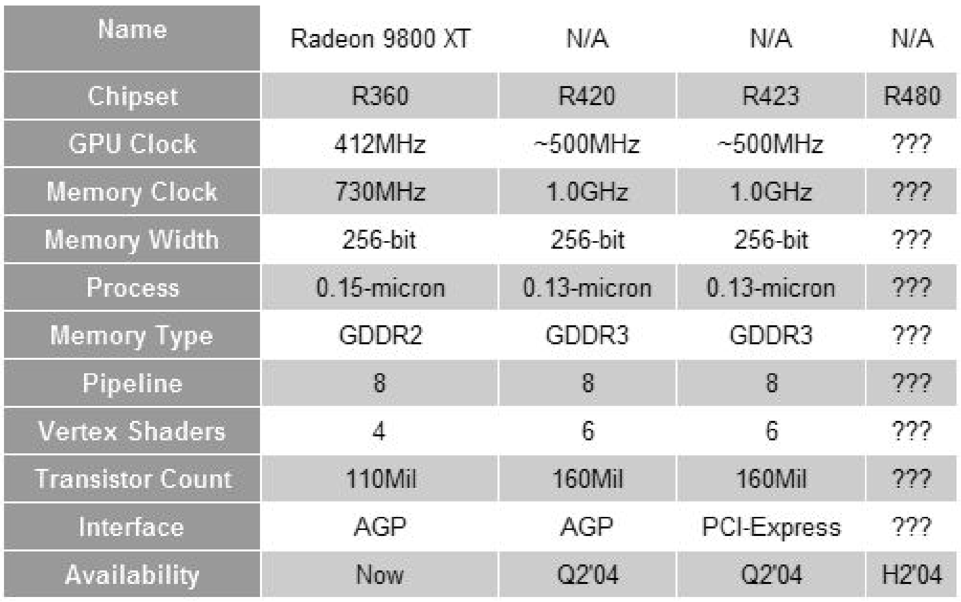
In order to provide a general understanding of how this enhanced performance is obtainable, a brief explanation of the graphics pipeline follows.

3.1 GRAPHICS PIPELINE

The ultimate responsibility of a GPU is to determine the color values for every picture element (pixel) for a visual display or output device. Fortunately, the vast majority of these calculations can be done independently, allowing for a great deal of parallelism to be incorporated into the hardware architecture.

Since many operations, such as rasterization, consist of a linear series of operations, an assembly line can be constructed of multiple “pipelines” which act on different parts of the same data flow. This divide and conquer approach has led to modern GPU architectures which provide a large number of logical units for performance critical stages in the graphics pipeline.

ATI ARCHITECTURAL ROADMAP Q2 2004



Name	Radeon 9800 XT	N/A	N/A	N/A
Chipset	R360	R420	R423	R480
GPU Clock	412MHz	~500MHz	~500MHz	???
Memory Clock	730MHz	1.0GHz	1.0GHz	???
Memory Width	256-bit	256-bit	256-bit	???
Process	0.15-micron	0.13-micron	0.13-micron	???
Memory Type	GDDR2	GDDR3	GDDR3	???
Pipeline	8	8	8	???
Vertex Shaders	4	6	6	???
Transistor Count	110Mil	160Mil	160Mil	???
Interface	AGP	AGP	PCI-Express	???
Availability	Now	Q2'04	Q2'04	H2'04

FIGURE 10 Roadmap for ATI hardware, Q2 2004 [Anand 2004].

For example, the NVidia NV30 line of graphics cards is capable of calculating up to 4 pixels per draw cycle with 2 textures per pixel (also referred to a 4x2 pipeline). The ATI R300/R420/R480 use a similar architecture, composed of an 8x1 pipeline.
The latest NV40 architecture from NVidia places a lot of emphasis on programmable shader pipelines, which quadruples the number of pixel pipelines that were available on the NV30 and doubles the capacity of the GPU to handle textures (making it a 16x1 architecture). In addition, the NV40 is flexible, and can operate as an 8x2 pipeline, dedicating more resources to texturing when required.

NVidia NV40 SYSTEM ARCHITECTURE

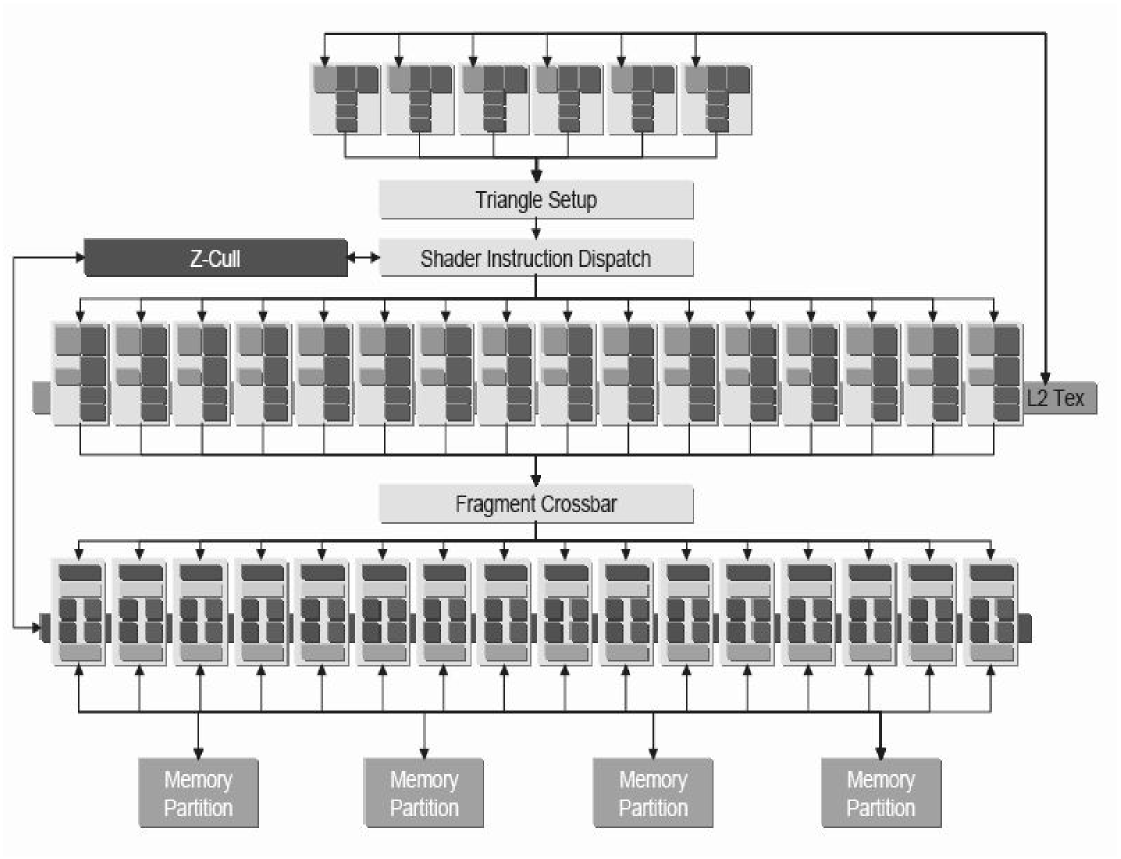


FIGURE 11 Block diagram for the NVidia GeForce 6800 (NV40) architecture [Hanrahan, 2004].

3.2 PROGRAMMABILITY

Even with all of these pipes, it remains essential to minimize the amount of data flow. All VLSI semiconductors suffer from bandwidth bottlenecks, and GPUs are no exception. In order to reduce the effects of limited bandwidth, GPUs tend to operate like a vector processor: they do as many calculations as possible in parallel. With the traditional fixed function pipelines, GPU vendors could fine tune the pipeline in hardware to optimize performance.

GPU PERFORMANCE TACTICS

Increase Data Transfer Rates

- Advanced Protocols
- Wide Data Bus (256 bit)
- Point to Point Wiring

Decrease Amount of Data Being Transferred

- Multiple Pipelines
- Structured Cache
- Algorithmic Optimizations

FIGURE 12 Optimizations used to obtain high performance on GPUs [Harris 2004].

Two tactics which GPU vendors readily exploit to obtain high performance are data locality, and data transfer rates. By decreasing the amount of data being transferred, and increasing data transfer rates, GPU vendors can minimize some of the effects of VLSI bandwidth. Some notable examples of theses types of optimizations are listed in table 4.

Unfortunately, as clock speeds increase, this fixed functionality can lead to underutilization and scheduling problems. If a fixed function pipeline executes, and the results are not needed, the hardware goes unused and clock cycles are wasted. It also means that the hardware is restricted to the tasks that it was designed for, and can never be used for anything else.

These limitations have led to the inclusion of limited programmability within certain stages of the graphics pipeline. Most GPUs now have programmable vertex units, and programmable fragment units which can execute a limited number of user provided instructions.

3.3 HARDWARE ABSTRACTION

The power of a GPU is hidden behind an industry standard application programming interface (API), such as OpenGL or DirectX, and a vendor supplied graphics driver which interfaces the two. This allows a GPU manufacturer to drastically change and optimize their hardware without requiring an independent software vendor (ISV) to modify their applications.

With the release of DirectX 9.0, Microsoft has required GPU manufacturers (also referred to as independent hardware vendors, or IHVs) to include certain features in hardware both to insure that particular API capabilities will be available, and to provide a certain level of abstraction to a developer.

HLSL VERTEX SHADER CAPABILITIES

	VS 1.1	VS 2.0	VS 3.0
Instruction Slots	128	256	>= 512
Program Length	128	256	65535
Temporary Registers	12	12	32
Constant Registers	96	>= 256	>= 256

HLSL VERTEX SHADER CAPABILITIES

Static Flow Control	YES	YES	YES
Dynamic Flow Control	NO	NO	YES (24 Levels)

FIGURE 13 Capability matrix for Vertex Shaders under HLSL [DG].

These capabilities, which Microsoft has named Shader Models, can also serve as a means of classifying GPU hardware. For example, if a GPU vendor specifies that their hardware supports Shader Model 2.0, then a developer can be confident that all the requirements of Vertex Shader (VS) v2.0 and Pixel Shader (PS) v2.0 have been met.

HLSL PIXEL SHADER CAPABILITIES

	PS 1.1	**PS 2.0**	**PS 3.0**
Instruction Slots	8	64	>= 512
Program Length	8	96	65535
Temporary Registers	12	12	32
Constant Registers	12	32	>= 256
Static Flow Control	YES	YES	YES
Dynamic Flow Control	NO	NO	YES (24 Levels)

FIGURE 14 Capability matrix for Pixel Shaders under HLSL [DG].

Capability lists such as these are necessary, since GPU manufacturers consider their core architectural designs to be trade secrets and do not provide detailed hardware specifications for their processors.

This restricts any third party from attempting to write a fully optimized driver, compiler, or debugger for a GPU since the documentation needed to access the device at the hardware level, is not available.

As a result, the only way to currently program a GPU is through the available graphics APIs, their corresponding assembly level shading languages or high level shading languages, or through an abstraction layer built upon either of the latter.

3.4 SHADING LANGUAGES

Programmable shading languages are not new. Since the late 1980's, programmability has been at the core of most modern rendering systems. The following table lists some milestones that have been made over the past 20 years in shading language development.

SHADING LANGUAGE DEVELOPMENT TIMELINE

1984	Shade Trees (Rob Cook)
1985	Image Synthesizer - Noise (Ken Perlin)
1988	Renderman - SL (Pixar)
1998	PixelFlow - First real-time shading language (UNC)
1998-2000	OpenGL Shader (SGI)
1999-2001	Stanford Real-Time Shading Language
2000	Non-standard Vertex Programs in Assembly
July 2001	GLSL Development Starts (ARB)
Oct. 2001	GLSL White Papers released (3Dlabs)
June 2002	Cg Specifications Released (NVidia)
Nov. 2002	HLSL Specifications Released (Microsoft)
Feb. 2003	OpenGL v2.0 Specification Finalized (ARB)
Aug. 2004	OpenGL v2.0 Formally Introduced (ARB)

FIGURE 15 Development timeline of shading languages [Rost, 2004].

Of these examples, Stanford's Real-time Shading Language (RTSL), Microsoft's High Level Shading Language (HLSL), NVidia's C for Graphics (CG) and OpenGL Shading Language (GLSL) were designed specifically for programmable GPUs [McCool 2004].

However, each of these languages has a different set of platforms, APIs and GPU vendors which they support. Furthermore, RTSL is no longer under active development and CG is nearly identical semantically to HLSL. The table below identifies the particular Platform/API/Vendor combinations that each language supports.

HIGH LEVEL SHADING LANGUAGE CAPABILITIES

	RTSL	HLSL	CG	GLSL
Active	-	X	X	X
DirectX-Supported	-	X	X	-
OpenGL-Supported	X	-	X	X
Multi-Platform	X	-	X	X
Vendor-Neutral	X	X	-	X

FIGURE 16 Capability matrix for several high level shading languages [DG].

Although not designed for GPUs, Pixar's Renderman Shading Language (SL) has been widely adopted in the film industry [Apodoca 2000]. Many real-time shading languages attempt to provide similar functionality since there is an abundance of high quality production level Renderman shaders already written. Being able to emulate or even process a Renderman shader continues to be an interesting challenge for many GPU vendors [ATI 2004].

The real-time oriented shading languages are very similar in their structure and feature set. They each bind to an underlying graphics API and expose certain sections such that a user may have control over the state of the API with a certain degree of programmability.

Being specifically designed for graphics applications, they all offer vector based arithmetic operations, n-tuple data types for variables, and basic matrix calculations that may be performed on whatever data is exposed. They have all chosen to base their language syntax on the C programming language, with some additional modifications and enhancements to help reduce tedious tasks and optimize code written for GPUs [NVidia 2004].

The main differences between the languages is the level at which they are integrated into their corresponding graphics API. For example, Microsoft's HLSL is translated into machine code outside of the DirectX API through a proprietary assembler written and maintained by Microsoft. In contrast, GLSL gets translated inside of the OpenGL API, thus requiring the GPU vendor to supply a GLSL assembler as part of their graphics driver.

Both approaches have their advantages, but one of the more attractive features of the GLSL approach is that it allows the GPU vendor to optimize GLSL code for their hardware, whereas the HLSL approach relies on the DirectX assembler written by Microsoft to optimize for all vendors.

HLSL/CG VS GLSL API INTEGRATION

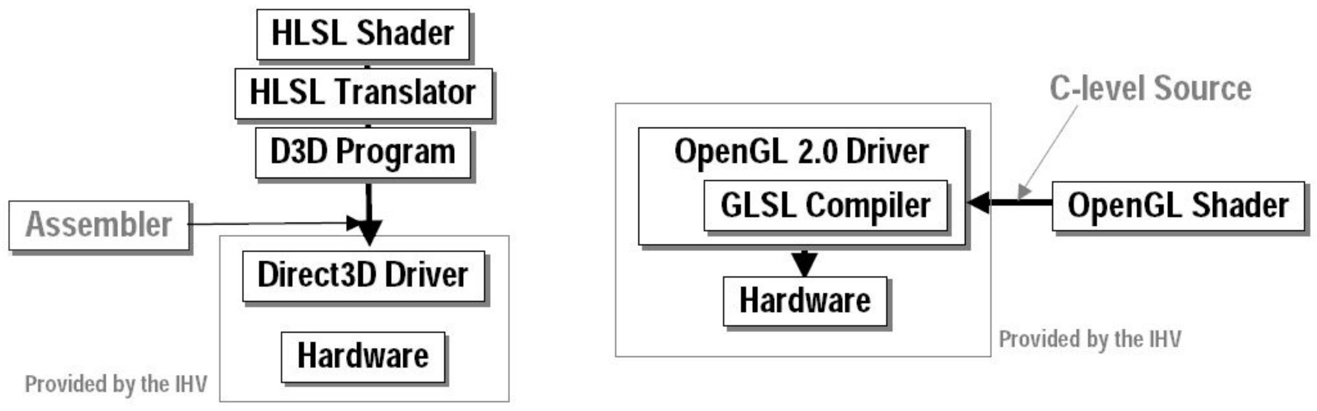


FIGURE 17 Block diagram showing integration levels for HLSL/CG and GLSL [Rost 2004].

3.4.1 HLSL/CG

NVidia was the first major GPU vendor to release a high level shading language for real-time graphics. It was submitted to the OpenGL architectural review board (ARB) in 2002 as a candidate for an integrated OpenGL shading language [Rost 2004], but was not accepted due to problems regarding the ARB's goals for vendor neutrality.

Since then, NVidia has partnered with Microsoft and jointly developed the High Level Shading Language (HLSL) for Microsoft's DirectX 9.0 graphics framework. As a result, the two languages are nearly identical and share 99% of the same semantics [McCool 2004].

CG LOGICAL PROGRAMMING MODEL

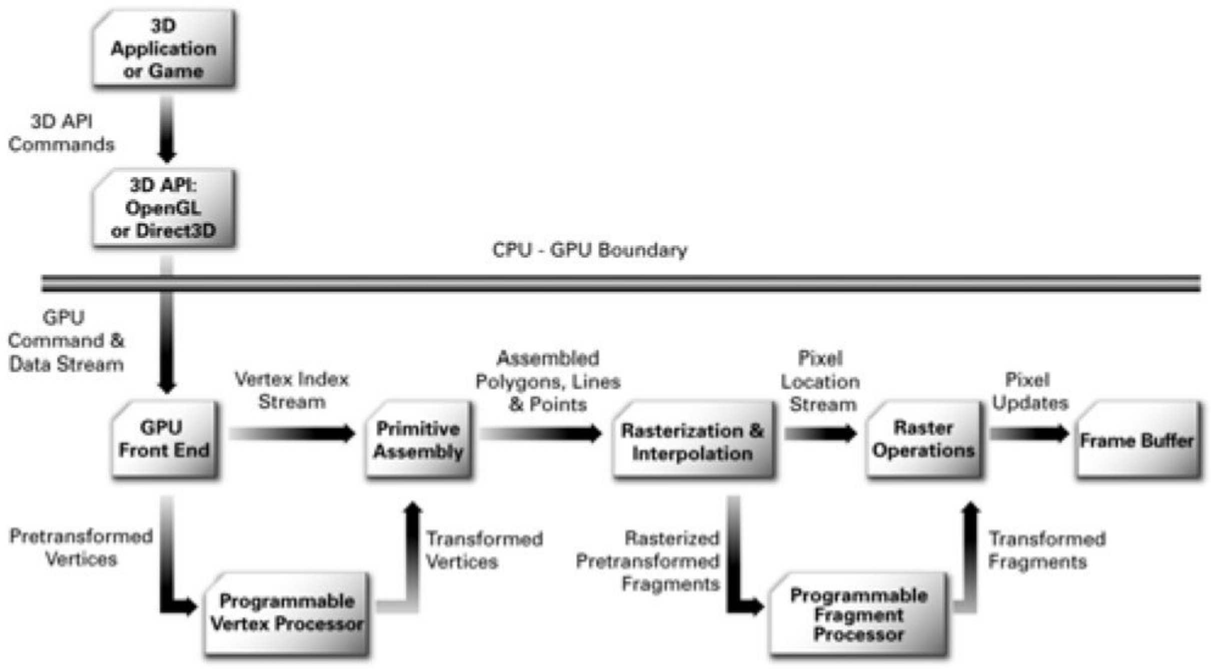


FIGURE 18 Block diagram for the logical programming model for Cg/HLSL [NVidia, 2004].

3.4.2 OPENGL SHADING LANGUAGE

Since GLSL is the only active, vendor neutral, cross platform shading language currently available, it has some competitive advantages over HLSL/CG. It is also now included as a part of OpenGL v1.5, and was the shading language of choice for this project.

OPENGL 2.0 STATE MACHINE

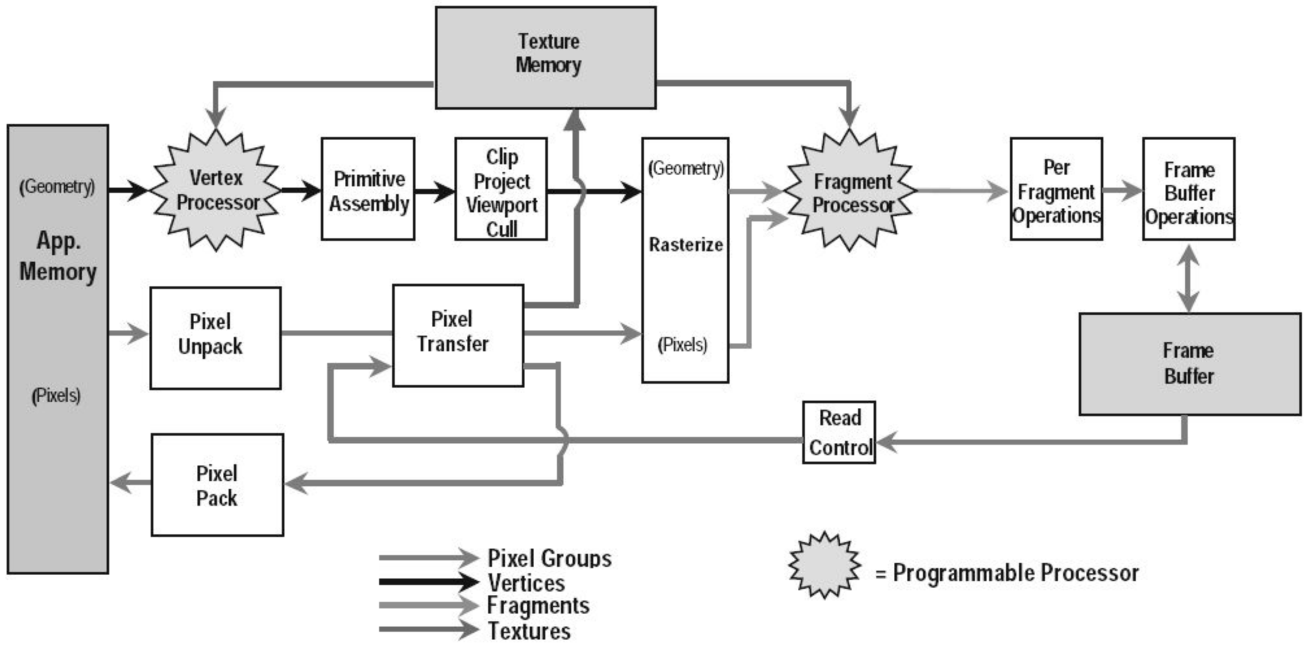


FIGURE 19 Logical diagram of OpenGL v2.0 state machine [Rost, 2004].

3.5 LIMITATIONS AND CONSTRAINTS

With the introduction of programmability for the pixel and vertex stages of the graphics pipeline, the responsibility for exploiting locality and minimizing data flow rests with both the user and the GPU vendor.

Programmers should be aware of the limitations of the hardware and be informed of the best practices for performance optimizations. Likewise, GPU vendors should take an interest in the applications that are being developed on their hardware so that they can incorporate new features and design decisions into next generation products.

Using programmable graphics hardware for general purpose computation and/or stream processing can easily exceed the capabilities and features of GPUs. As a result, there are a fair number of limitations to be aware of and some constraints to be familiar with when designing a stream processing framework.

3.5.1 DEBUGGING TOOLS

Programming in a high level programming language is not easy. The tools are not mature and there are numerous abstraction layers which must be navigated when attempting to deduce the source of programming errors.

Currently, there are only two vendor supplied debugging tools: NVidia's NVPerfHud, and Microsoft's HLSL Shader Debugger. Neither offer functionality for interrupted execution, stack tracing, or watch variables, features which developers have come to rely upon in traditional programming environments.

However, there are a few initiatives in the research community which look promising: Stanford's Shadesmith Fragment Program Debugger [Purcell et al 2004], and the publicly available Image Debugger utility from University of North Carolina at Chapel Hill [Baxter 2004]. Both offer features which will likely be incorporated into vendor supplied tools sometime in the near future.

3.5.2 READBACK PERFORMANCE

One of the biggest bottlenecks in the graphics pipeline is reading back data that has already been processed. Although the transfer rate from system memory on the host computer to the GPU is relatively fast (2GB/sec), reading data back from the GPU to the host can be orders of magnitude slower. There are several methods for doing so such as **glReadPixels()**, and **glCopySubTex2DImage()**, but each has it's own technical restrictions.

OPENGL READBACK PERFORMANCE

```
WinXP (WGL)
- NVidia
  - 1215 MB/sec Texture Copy
  - 221 MB/sec Read Pixels
- ATI
  - 926 MB/sec Texture Copy
  - 180 MB/sec Read Pixels
Linux (GLX)
- ATI
  - 680 MB/sec Read Pixels
```

FIGURE 20 Read back performance measurements for GPUs under WinXP and Linux [Buck, 2004].

One possible resolution to this limitation may be the recently introduced PCI-Express (PCI-E) expansion slot, which offers a symmetric connection with 4x the bandwidth (4GB/sec in both directions, or 8GB/sec total) compared to the widely adopted AGP-8x expansion slot (2GB/sec, one direction at a time).

Manufacturers of GPUs which have released PCI-E graphics cards so far do not take advantage of the full bandwidth potential. Instead, they simply route their existing AGP-8X card through an adapter (or bridging chip) which reduces the effective bandwidth of the PCI-E to the AGP-8x interface. The resulting PCI-E bridging chip combination has very little performance advantage over the existing AGP-8X cards other than to provide a card that works with PCI-E. This has lead to much confusion in the consumer market [Rost 2004].

AGP 8X VS PCI EXPRESS WITH BRIDGING CHIP

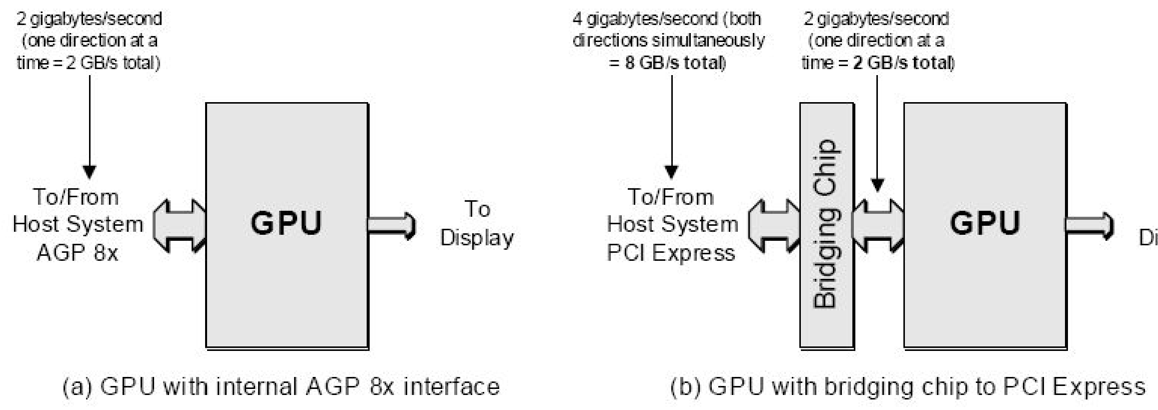

2 gigabytes/second (one direction at a time = 2 GB/s total)
To/From Host System AGP 8x
GPU
To Display
(a) GPU with internal AGP 8x interface
4 gigabytes/second (both directions simultaneously = 8 GB/s total)
2 gigabytes/second (one direction at a time = 2 GB/s total)
To/From Host System PCI Express
Bridging Chip
GPU
To Display
(b) GPU with bridging chip to PCI Express

FIGURE 21 Comparison of AGP vs PCI-Express w/bridging chip results in same performance [Rost, 2004].

3.5.3 OFFSCREEN FRAME BUFFERS

An alternate solution is to do as much processing as possible on the GPU, minimizing the number of times data gets transferred from the graphics card back to host memory. In a stream processing system, this is known as a feedback mechanism, since it allows results to be saved and used again as an input stream.

In OpenGL and DirectX, there are two methods for saving data that has been processed by the graphics pipeline: Copy To Texture (CTT), and Render to Texture (RTT).

CTT is supported on multiple platforms and is fairly flexible in terms of its output format. Unfortunately, it's relatively slow, and consumes internal GPU memory bandwidth [Lefohn 2004].

COPY TO TEXTURE FEEDBACK MECHANISM

- Save Current Viewport Size
- Initialize Viewport to Stream Buffer Size
- Render Stream Geometry
- Copy Current Frame buffer to Texture
- Restore Viewport Size

FIGURE 22 Basic Copy-To-Texture (CTT) feedback mechanism under OpenGL [DG].

From a different perspective, RTT is attractive because it uses off-screen frame buffers, or pixel buffers. These pixel buffers reside in GPU resident memory and offer a fast non-visable rendering target to save results that are processed through the graphics pipeline. At this point, there exists no common cross-platform interface for pixel buffers or RTT. Each platform (WGL, AGL, GLX) has a different implementation, and none of them match any formal specification.

Furthermore, pixel buffers were never intended for general purpose computation. Most implementations require a OpenGL rendering context to be associated with the buffer, and switching between contexts is slow. In order to maintain the speed benefits that pixel buffers offer, usually only a single context can be used (with a single buffer). In addition, executing a frame buffer read operation may flush the entire graphics pipeline.

Some tricks have been developed to avoid these limitations that a context switch may impose. These include packing multiple arrays of scalar data of the same size into multiple color channels (4 floats -> RGBA), and using non-standard multi-surface pixel buffers which are supported on some graphics cards (ATI, NVidia) [Lefohn 2004].

There is currently a proposal pending before the OpenGL Architectural Review Board (ARB) regarding an open specification for a new memory model, called Superbuffers, which could be used for CTT/RTT. Unfortunately, it was not accepted for OpenGL v2 and there is no estimated time for when and/or if it will be adopted [Mace 2004].

However, with the latest release of DirectX (v9.0c) and Shader Model v3.0, there is added support for multiple render targets (MRT). Most vendors have exposed this functionality through vendor extensions under OpenGL (GL_ATI_draw_buffers). This allows multiple RGBA results to be written through a single fragment program and reduces the number of rendering passes for applications which produce more than four outputs.

3.5.4 CONTROL STRUCTURES

The latest generation of GPUs allow conditional expressions and iterative loops within fragment and vertex programs, but at a severe cost, especially when executed within a fragment program (between 2-6 clock cycles per branch) [NVNEWS 2004].
Furthermore, all cards have a limited number of assembly level instructions which can be executed within a shader program, which range from 8 to 65k instructions depending on the hardware (see the section on Abstraction).

Although these limitations will continue to be reduced (NV40 already supports full-speed branching in vertex programs) there will likely be applications which exceed these capabilities and instruction limits in the foreseeable future.

3.5.5 FLOATING POINT PRECISION

A fundamental requirement for applications which rely on numerical accuracy is floating point precision. Although many GPU vendors claim to support 32-bit IEEE 754 “single” floating point precision, and even the 16-bit “half” precision type, there still remains some inconsistencies between vendors [Hillesland et al 2004].

Furthermore, both the scientific and the high performance computing communities rely on 64-bit “double” and sometimes 128-bit “long double” for many of their calculations [Heroux 2004][Seagar 2004]. Neither of these formats are supported on any commercially available GPU to date, and there are no formal announcements regarding whether or not they will be added.

3.6 GPGPU APPLICATIONS

Even with all of these constraints and limitations, a number of researchers have successfully used a GPU for general purpose computation. Some examples include real-time fluid dynamics, cloth simulations, particle simulations, radiosity and ray tracing [Harris 2004].

4 SPARK

In order to take advantage of a GPU's full capabilities, and to be fully prepared for the next generation of more general purpose graphics processing, a stream programming model can be used to provide an abstraction layer to program a GPU in an efficient manner.

This project, *“A Stream Processing Abstraction and Rendering Toolkit for Programmable Graphics Hardware, (SPARK)”*, uses such an approach, and was designed as a bare bones API for doing stream processing on a GPU, with an emphasis on rendering and procedural texture synthesis.

This architecture was designed as a minimalistic framework, with the hope that it would be extended in the future, depending on the particular needs of the developer.

As a demonstration of SPARK's capabilities, an application was written which can generate complex procedural textures interactively.

In the following sections, a brief review of some existing tools and libraries related to stream processing on the GPU will be examined, followed by an explanation on how SPARK maps a stream processing model onto programmable graphics hardware. This is examined in more depth in the next section, which discusses the structure of the SPARK API. Finally, an explanation of the demonstration application and the process of using SPARK for procedural texture generation is provided.

4.1 PREVIOUS WORK

Two notable projects, BROOK for GPUs (BROOK) and libsh (SH), attempt to provide an abstraction layer oriented towards stream programming. They do this by hiding the underlying graphics API that is actually used to program the GPU. However, both are relatively young and still early in their development cycle.

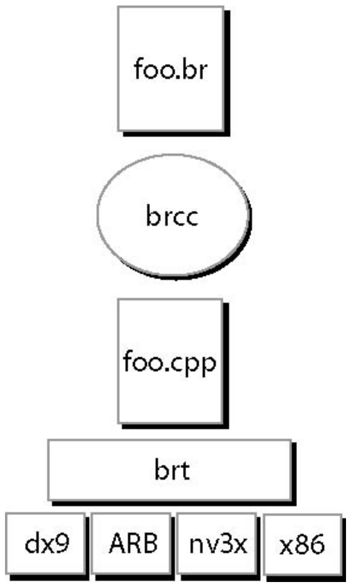
4.1.1 BROOK

Designed as an extension to ANSI C, BROOK was written by Ian Buck as part of his PhD thesis at Stanford and has been used in conjunction with Stanford's Streaming Supercomputer project for performing high performance scientific calculations [Hanrahan 2004].

BROOK provides a non-graphical programming environment which incorporates data parallelism and arithmetic intensity (compute-to-bandwidth ratio) into a general computational model [Buck 2003].

It was designed to hide GPU resources and API state management so that developers familiar with C could easily use a GPU for additional processing power. A consequence of this design is that it does not expose any graphical subroutines routines and requires code to be translated from a proprietary format into standard C++.

BROOK TRANSLATION PROCESS



- BROOK Source Files (br)
  - C Based Source Files
- BRCC
  - Source to Source Compilation
- BRT
  - Brook Runtime Library
  - Runtime Targets (Back ends)
    - DirectX 9.0
    - OpenGL ARB
    - Cg NV3X
    - CPU

FIGURE 23 BROOK to GPU translation process [Buck 2004].

The resulting C++ source can then be compiled and linked against the BROOK Runtime Library (BRT) so that the binary program can be executed on the configurable runtime target. In addition to the GPU runtime targets, BROOK also includes support for a CPU back end, allowing comparisons to be made relatively easily between CPU and GPU performance.

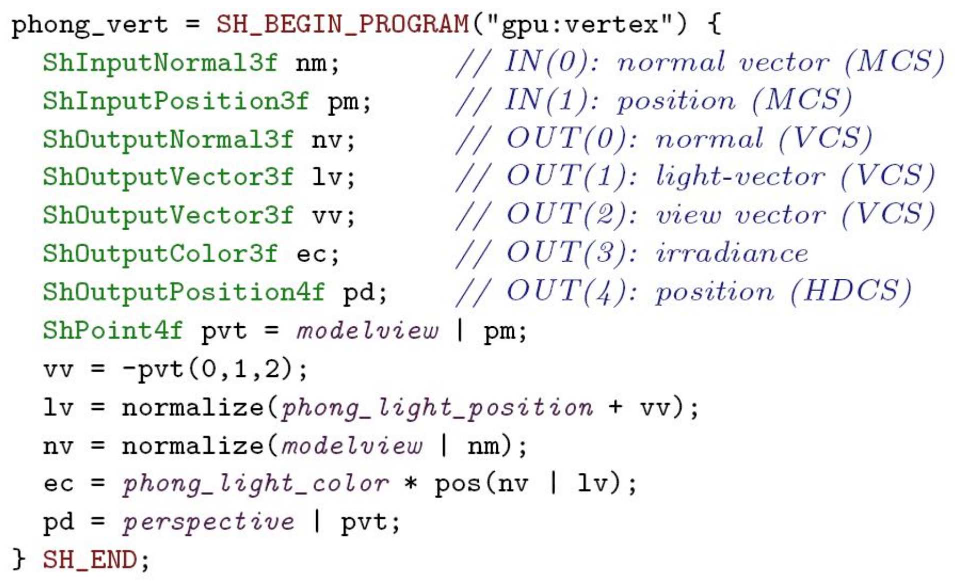
4.1.2 SH

Another approach to provide a high level abstraction for stream processing has been realized by Michael McCool et al with the recent release of the Sh Metaprogramming Toolkit, or libsh (SH).

The SH framework uses an embedded metaprogramming approach that exploits the functionality of C++ templates to provide an object oriented library for advanced real-time graphics applications with a unified programming model for both GPUs and CPUs [McCool 2004]. Stream programs built with SH are declared inside a set of C++ macros. These programs can then be stored as program objects and are translated into a format suitable for the designated target platform at runtime (either GPU or CPU based).

SH does not allow programs to be instantiated via a source code based interpreter, since all functionality is defined as an integral part of C++. Therefore, the traditional source based approach where programs are stored as text files and interpreted at runtime is not supported. Classes must be derived or built out of various components that are provided in the SH library.

SH EXAMPLE PHONG PROGRAM



```
phong_vert = SH_BEGIN_PROGRAM("gpu:vertex") {
  ShInputNormal3f nm;        // IN(0): normal vector (MCS)
  ShInputPosition3f pm;      // IN(1): position (MCS)
  ShOutputNormal3f nv;       // OUT(0): normal (VCS)
  ShOutputVector3f lv;       // OUT(1): light-vector (VCS)
  ShOutputVector3f vv;       // OUT(2): view vector (VCS)
  ShOutputColor3f ec;        // OUT(3): irradiance
  ShOutputPosition4f pd;     // OUT(4): position (HDCS)
  ShPoint4f pvt = modelview | pm;
  vv = -pvt(0,1,2);
  lv = normalize(phong_light_position + vv);
  nv = normalize(modelview | nm);
  ec = phong_light_color * pos(nv | lv);
  pd = perspective | pvt;
} SH_END;
```

FIGURE 24 An example program written using the Sh library [McCool 2004].

By providing overloaded operators (+, -, *, /, <<, >>) for all program objects, SH can be used to perform “shader algebra” and complex functional composition through object based program manipulation [McCool 2002]. The end result allows complex objects to be created easily and used with legible syntax.

Unfortunately, the developers of SH chose to publish a book [McCool et al 2004] which details features which are not yet supported in the library. Since the software is currently in the alpha stage of development and constantly changing, many of the examples described in their book do not work as expected.

In short, a lot of work remains before they meet their full project goals. The name, however, which is an abbreviation of Serious Hack, seems quite appropriate.

4.2 DESIGN AND IMPLEMENTATION

The development approach of SPARK attempts to provide abstract interfaces oriented towards stream processing on the GPU using existing GPU (shader) programs, which are loaded from source files and interpreted at run time.

The goals for this project were to minimize the complexity of doing GPU based stream processing, while still provide all of the necessary components to design and build full stream oriented applications.

It does not attempt to compete with existing stream processing solutions for GPUs, such as BROOK or SH which have been in development for over two years, but does provide a core framework that could be extended to accommodate additional features.

Furthermore, there is a specific focus on procedural textures, and as such, some features are specifically included for doing spectral synthesis.

4.2.1 STREAM PROCESSING

Streaming architectures require a radically different programming model than traditional programming languages such as C/C++. Some problems quickly arise when attempting to apply these concepts to a GPU with limited programmability, most notably conditional operations which modify the state of the data flow, and iterative loops which operate on segments of a stream calculation.

GPU HARDWARE RESOURCES FOR STREAM PROCESSING

- Processors
  - Programmable Vertex Unit
    - Processes 4 Component Vectors
    - Capable of Changing Vertex Data
    - No Communication Between Neighbors
  - Programmable Fragment Unit
    - Processes 4 Component Vectors
    - Random Access Memory Read via Textures
    - Direct Output Fixed to Pixel Location
- Rasterizer
  - Interpolation
- Texture Units
  - Read Only Memory
  - Limited GPU Memory
- Frame Buffers
  - Write Only Memory
  - Limited GPU Memory

FIGURE 25 Hardware resources on GPUs which can be used for stream processing [Harris 2004].

The design of SPARK, and most other GPU based stream abstraction layers, attempts to utilize all of the GPU hardware resources that are available, and provide a high level interface for managing those resources in a stream processing context.

For performing the actual stream computation, a grid based computational model is used. This type of computation maps well the hardware resource available in a GPU, and is flexible enough to allow general purpose computations to be performed for a wide range of applications [Harris 2004].

With grid based stream processing, there are two additional concepts related to grid communication that are important and worth mentioning. These include stream based gather and scatter operations [Harris 2004].

Gather operations are allowed random access for reading stream data, but do not have any way of modifying their output destination. In contrast, scatter operations can modify the data contained in a stream, but do not have access to any data values other than the ones they receive.

In this context, programmable vertex units are limited to scatter operations, and programmable fragment units are likewise limited to gather operations.

GRID BASED GATHER AND SCATTER STREAM OPERATORS

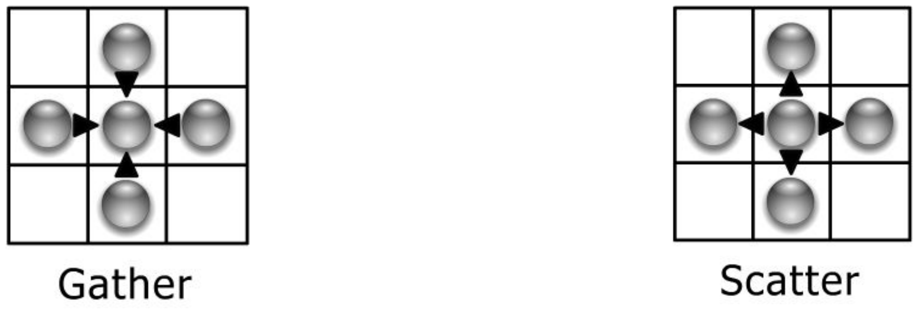


FIGURE 26 Communication diagrams for grid based gather and scatter stream operators. [Harris 2004].

Some analogies can be made by comparing the available GPU hardware resources to the components used in grid based computation, thus identifying how they can be used in a stream processing environment.

Some examples of these conceptual mappings are listed in the following table.

GPU ANALOGIES FOR STREAM PROCESSING

- Texture -> Acts as an Array of Data
- Program -> Acts as a Filter or Kernel
- Geometry -> Forces Data Through Stream
- Frame Buffer -> Provides Feedback Mechanism

FIGURE 27 Analogies for mapping GPU resources to stream processing concepts [DG].

This mapping is critical to the implementation of SPARK, and was used as the context for which the class library was designed.

4.2.2 CONVENTIONS

The entire SPARK library is encapsulated in its own C++ namespace, called *Spark.* Every class name in SPARK starts with the two letter prefix ***Sp***, and certain classes contain suffixes which distinguish their base types. For example, the ***SpVertexNoiseSb*** class is derived from the ***SpStreamBasis*** class interface, and includes the suffix ***Sb***. Similar conventions exist for classes derived from ***SpStreamOperator (Op), SpStreamGeometry (Sg)*** and ***SpStreamFeedback (Sf)***.

4.2.3 STREAM MANAGEMENT CLASSES

The core infrastructure of SPARK consists of several classes for managing streams. These include stream inputs and outputs (defined as ***SpStreamInput*** *and* ***SpStreamOutput***) which represent abstract conduits through which data can flow. This is an important concept, since these classes do not actually store any data themselves and only provide a mechanism for connecting classes which perform the actual data management.

Data management is handled via a hybrid render to texture (RTT) solution, and is encapsulated in the ***SpStreamBuffer*** class. Although ***SpStreamBuffer*** has some built in mechanisms for feedback, it is not derived from ***SpStreamFeedback.*** Instead, ***SpStreamBuffer*** is meant to provide a fast rendering context for storing intermediate results.

These results can be obtained via an ***SpStreamFeedback*** derived class when processing is complete. This approach allows all rendered output to be sent directly to GPU resident memory and remain there while data gets processed (on architectures where RTT is supported).

SPARK STREAM MANAGEMENT CLASSES

SpStreamBasis	*Abstract base class for a stream basis for spectral synthesis*
SpStreamBuffer	*A multi-format OpenGL GPU resident data buffer and render-to-texture (RTT) interface*
SpStreamGeometry	*Abstract base class for generating a data stream by rendering geometry*
SpStreamInput	*Abstract base class for an input stream for loading/generating a data stream*
SpStreamOperator	*Abstract base class for a stream operator which modifies an input stream*
SpStreamOutput	*Abstract base class for an output stream used for saving/viewing stream data*
SpStreamViewer	*Stream viewer output class for visualizing a data stream*

FIGURE 28 Listing of classes defining the core infrastructure of SPARK [DG].

Since it is inefficient to switch contexts, a single instance of ***SpStreamBuffer*** is typically used and passed as a parameter to an ***SpStreamInput*** derived class during an ***updateOutputBuffer()*** function call.

Another major component of the core is the stream operator *(**SpStreamOperator**)*, which is derived from both the stream input and stream output base classes *(**SpStreamInput** and **SpStreamOutput**)*. This interface acts as a filter or kernel, manipulating the data as it flows from the input stream to the output stream. Thus, multiple stream operators to be connected in series to form complex data flow networks (or pipelines).

In order to minimize the amount of re-evaluation, each stream input *(**SpStreamInput**)* class provides a query method to determine whether or not the state of the stream has been modified. In a stream operator, this method checks both the current object as well as the stream input connected to the operator, allowing requests to be propagated throughout an entire pipeline.

A specialized interface for a stream basis (***SpStreamBasis***) is also included in the core. This component can be used to create complexity at various scales through a process called spectral synthesis [Ebert 2002].

Finally, a specialized stream viewer (***SpStreamViewer***) is provided which is derived from the stream output base class (***SpStreamOutput***), allowing objects to be built for visualizing stream based data that has been evaluated through a data flow network.

4.2.4 STREAM FEEDBACK CLASSES

A fundamental concept for stream processing is the notion of a feedback mechanism. This mechanism allows data to be returned after being processed by a stream operator (***SpStreamOperator***). In addition to an abstract feedback base class (***SpFeedback***), SPARK provides two mechanisms for retrieving stream data that has been modified.

SPARK STREAM FEEDBACK CLASSES

SpStreamFeedback	Abstract base class for a stream feedback mechanism for returning results
SpCopyToTextureFb	Copy To Texture (CTT) feedback mechanism via OpenGL
SpCopyToTexture3dFb	Sliced 3D Copy To Texture (CTT) feedback mechanism via OpenGL

FIGURE 29 Listing of classes defining feedback mechanisms in SPARK [DG].

The first mechanism (***SpCopyToTextureFb***) allows data to be retrieved from a ***SpStreamBuffer*** by copying data from memory resident on the GPU, to host or system memory. These data are copied to a pre-initialized texture in OpenGL and are useful for obtaining results which can be represented as 2d arrays of data.

For 3d data, ***SpCopyToTexture3dFb*** can be used to copy a slice of a 3d volume or texture that is resident in a ***SpStreamBuffer.*** The slicing can be performed at multiple locations to retrieve a partial reconstruction of the volume.

4.2.5 STREAM GEOMETRY CLASSES

All of the stream management classes in the core allow connections to be made in preparation for stream processing. In order to actually move data and make it flow, some geometry must be rendered to push data through the pipeline.

SPARK STREAM GEOMETRY CLASSES

SpStreamGeometry	*Abstract base class for generating a data stream by rendering geometry*
SpGridSg	*Multi-resolution grid-based stream geometry*
SpQuadSg	*Quad-based stream geometry*
SpTexturedGridSg	*Multi-resolution grid-based stream geometry w/ texturing support*
SpTexturedQuadSg	*Quad-based stream geometry w/ texturing support*
SpMultiTexturedGridSg	*Multi-resolution grid-based stream geometry w/ multi-texturing support*
SpMultiTexturedQuadSg	*Quad-based stream geometry w/ multi-texturing support*

FIGURE 30 Listing of classes defining stream geometry classes in SPARK [DG].

There are several classes available in SPARK for rendering geometry that is particularly suited for stream processing. These are all derived from the ***SpStreamGeometry*** base class, and include quads (***SpQuadSg***) and multi-resolution grids (***SpGridSg***).

Both representations are implemented in three different forms. These forms differ in the type of data which they submit: vertex coordinates only (***SpQuadSg***, ***SpGridSg***), vertices and texture coordinates (***SpTexturedQuadSg***, ***SpTexturedGridSg***), or vertices + multi-texture coordinates (***SpMultiTexturedQuadSg***, ***SpMultiTexturedGridSg***).

4.2.6 GPU PROGRAM MANAGEMENT CLASSES

SPARK relies on OpenGL for interfacing with the underlying graphics hardware and provides classes for managing GLSL programs.

SPARK GPU PROGRAM MANAGEMENT CLASSES

SpGlManager	*Static manager class for interfacing with OpenGL*
SpGlslManager	*Static manager class for managing GLSL programs*
SpGpuProgram	*Abstract base class of a GPU program*
SpGlslVertexProgram	*Interface for a GLSL vertex program*
SpGlslFragmentProgram	*Interface for a GLSL fragment program*
SpParameter	*Storage class for multiple data types for use with GPU programs*
SpParameterSet	*Container class for parameter data used in a GPU program*

FIGURE 31 Listing of classes defining GPU program management classes in SPARK [DG].

The ***SpGlManager*** static class provides methods for interacting with OpenGL, and the ***SpGlslManager*** provides methods for managing GPU programs with GLSL. These GPU programs can be created via the specific vertex and fragment shaders for GLSL, defined in the ***SpGlslVertexProgram*** and ***SpGlslFragmentProgram*** classes.

It should be noted, however, that the library was designed with abstraction in mind, such that an alternative shading language could easily be added by simply deriving the ***SpGpuProgram*** class and implementing the necessary methods.

There are also two classes for managing parameters for GPU programs. These include the ***SpParameter*** storage class for multiple data types, and the ***SpParameterSet*** for maintaining a list of parameters for a program.

These parameter classes are not used internally for the current implementations of the ***SpGpuProgram*** classes in order to avoid parameter proliferation and ensure a lightweight interface. However, they can be extremely helpful for applications and utility classes which need to maintain program state information.

4.2.7 MATH CLASSES

In addition to the stream oriented classes, SPARK also includes a template based math library for n-tuples (***SpTuple, SpTuple2, SpTuple3, SpTuple4***), vectors (***SpVector, SpVector2, SpVector3, SpVector4***), and matrices (***SpMatrix, SpMatrix2, SpMatrix3, SpMatrix4***). There is also a static class containing relevant mathematical constants (***SpMaths***).

SPARK MATH CLASSES	
SpMaths < Real>	*Static utility class for mathematical constants*
SpMatrix < N, Real >	*NxN Matrix base class for vector algebra*
SpMatrix2 < Real >	*2x2 Matrix class for vector algebra*
SpMatrix3 < Real >	*3x3 Matrix class for vector algebra*
SpMatrix4 < Real >	*4x4 Matrix class for vector algebra*
SpTuple < N, Real >	*Nd Tuple class with support for vector mathematics*

SPARK MATH CLASSES

SpTuple2 < Real >	*2d Tuple class with support for vector mathematics*
SpTuple3 < Real >	*3d Tuple class with support for vector mathematics*
SpTuple4 < Real >	*4d Tuple class with support for vector mathematics*
SpVector2 < Real >	*2d Vector class for vector algebra*
SpVector3 < Real >	*3d Vector class for vector algebra*
SpVector4 < Real >	*4d Vector class for vector algebra*

FIGURE 32 Listing of classes defining mathematical classes in SPARK [DG].

4.2.8 PROCEDURAL METHODS

To aid in the development of procedural textures, an extensive list of procedural methods has been included. Continued development of SPARK will in part focus on emulating the functionality of many of these methods by restructuring them into a stream processing environment. Consequently, these methods serve as a good reference and offer the ability to perform partial calculations, or pre-compute certain data on the CPU which can be fed to GPU programs. There is also a high quality pseudo random number generator included in ***SpRandom.***

4.2.9 UTILITY CLASSES

Several utility classes are included with SPARK for system level interaction and redundant tasks.

To support shared objects, SPARK provides a template class (***SpReference)*** with built in garbage collection for managing shared object references.

The ***SpTimer*** class offers a cross platform interface for obtaining performance measurements, ***SpResourcePath*** allows datafile resources to be accessed easily, and the ***SpString*** class provides additional methods for character based data that is not available in the C++ standard library.

Finally, there is a minimal cross platform windowing interface provided in ***SpWindow*** and the derived ***SpGlutWindow*** classes for easily creating interactive applications.

SPARK UTILITY CLASSES

SpReference < Type >	*Reference class with built in garbage collection for supporting shared objects*
SpString	*String utility class for character data*
SpTimer	*Timer for profiling and retrieving performance and system time information*
SpResourcePath	*Resource utility class for querying a file system and loading datafiles*
SpWindow	*Abstract base class for a graphical display window*

FIGURE 33 Listing of classes defining utility classes in SPARK [DG].

4.3 DEMONSTRATION

As a demonstration of the functionality of SPARK, several classes for performing spectral synthesis were written, and a data flow network was constructed for creating procedural textures.

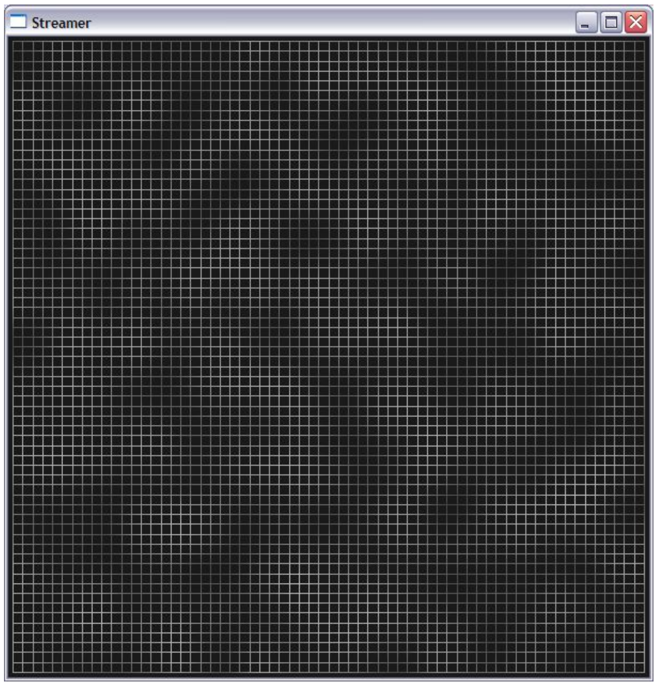
The goal of this demonstration was to perform as many calculations as possible on the GPU and test the feasibility of the general purpose stream processing capabilities of SPARK.

This demonstration program uses a GLSL vertex program encapsulated inside of ***SpVertexNoiseSb*** to calculate color and displacement values by evaluating a gradient noise function (also known as Perlin Noise) at each vertex location. In contrast to most real-time “noise” shaders which rely on a precomputed texture, this implementation performs the entire noise evaluation, including the spline based interpolation, in the vertex program.

However, the GLSL vertex program does use a precomputed table of packed pseudorandom permutations and gradient directions which are provided via a uniform parameter array. This precomputation is necessary, since it is not currently feasible to generate pseudorandom numbers in a vertex program due to the lack of bit shift operations and the high cost of control structures.

Since ***SpVertexNoiseSb*** uses a vertex program to perform it's calculation, it must submit vertex coordinates to the GLSL vertex program. As a result, it uses an ***SbTexturedGridSg*** to render a multi-resolution grid which pushes data through the stream. This approach is advantageous because the desired visual level of detail can be configured by modifying the resolution of the grid accordingly.

SPARK DEMONSTRATION GRID BASED VERTEX NOISE



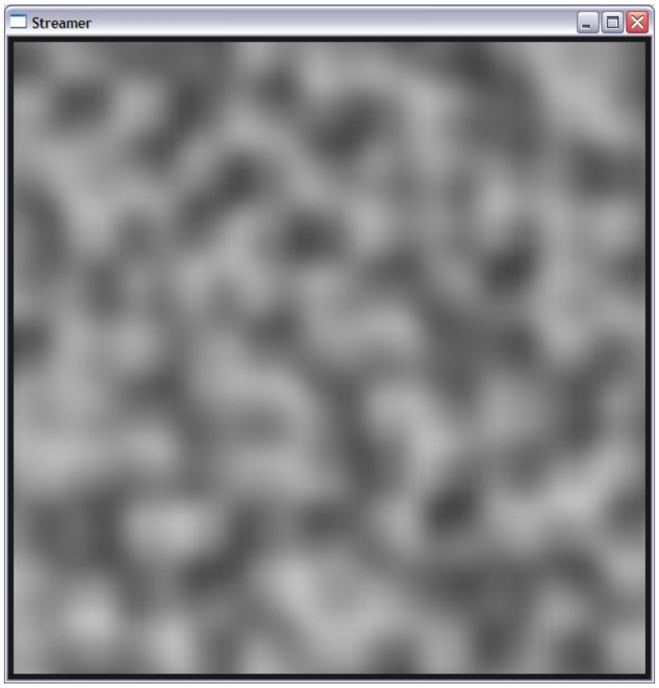


FIGURE 34 Grid based vertex noise used in SPARK demonstration shown in wireframe and shaded[DG].

To construct the data flow network, connections must be made. This is achieved via ***SpVertexNoiseSb.*** which is derived from ***SbStreamBasis.***

This allows ***SpVertexNoiseSb*** to act as an input stream for ***SbTurbulenceOp.*** This abstraction of an ***SbStreamBasis*** interface allows the input to ***SbTurbulenceOp*** to use any arbitrary basis operator.

The final connection in the data flow network is to an ***SbStreamViewer*** which is connected to the output of ***SbTurbulenceOp*** so that the data stream can be visualized.

This resulting data flow network is illustrated in the following figure.

SPARK DEMONSTRATION DATA FLOW NETWORK

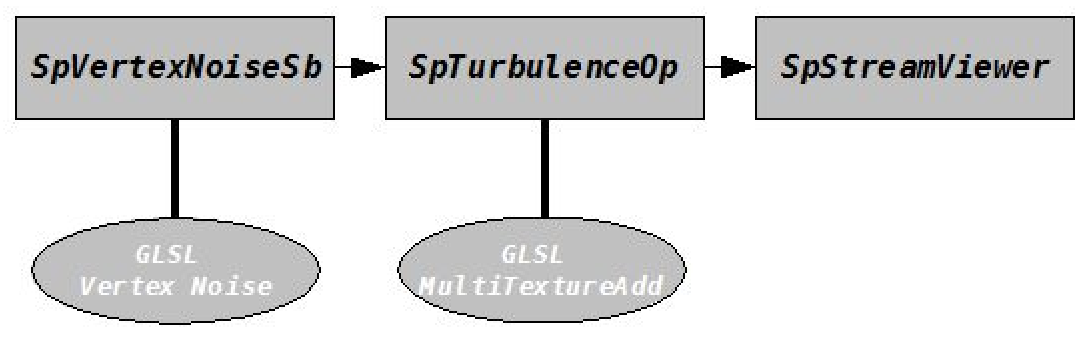


FIGURE 35 The streaming data flow pipeline used for procedural texture generation [DG].

SbTurbulenceOp is a specialized class which evaluates ***SpVertexNoiseSb*** at multiple scales (or octaves) and combines them together through a process known as spectral synthesis.

The end result offers a visually complex solid texture with multiple self-similar levels of detail [Ebert *et al* p584].

SbTurbulenceOp stores each evaluation of ***SpVertexNoiseSb*** as a texture and uses the ***SpCopyToTextureFb*** feedback mechanism to copy the data from the global ***SpStreamBuffer.*** After all of the specified octaves have been calculated, ***SpTurbulenceOp*** combines the results by using a GLSL fragment program to sum the values stored in each texture.

As a result, the maximum number of octaves that ***SpTurbulenceOp*** can calculate is limited by the number of texture units available on the GPU.

The final, low-level pipeline starts with vertex coordinate generation via ***SbTexturedGribSg*** which gets triggered for each octave of noise via ***SbVertexNoiseSg*** when ***SbTurbulenceOp*** requests an update.

For each vertex a noise value gets calculated via the vertex processor. The vertex program then assigns a color value to the surface position of the grid. This color value then gets interpolated via the rasterizer, and passes through the fragment processor unmodified.

The resulting single octave of noise, is stored in the global ***SbStreamBuffer*** object, whose data gets copied into a texture via the managing ***SbTurbulenceOp***.

SPARK DEMONSTRATION PIPELINE

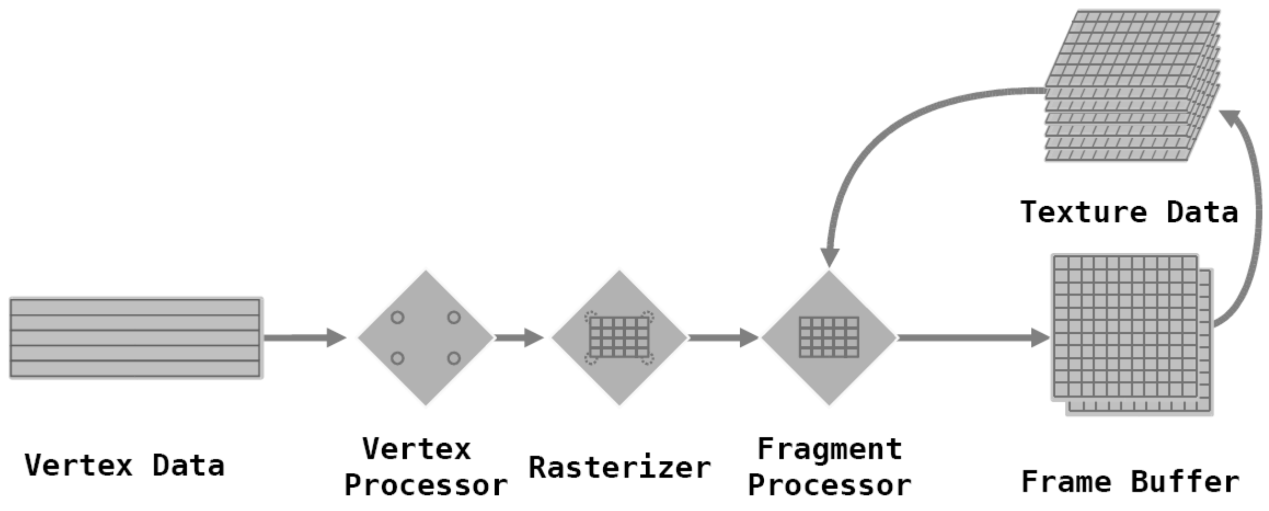


FIGURE 36 The low level pipeline used for procedural texture generation [DG].

Once all octaves have been calculated, ***SbTurbulenceOp*** then passes all of the texture data to a specialized fragment program where the data values are added together on a per-pixel basis and then sent to ***SbStreamViewer*** for display.

SPARK DEMONSTRATION PROCEDURAL TEXTURE

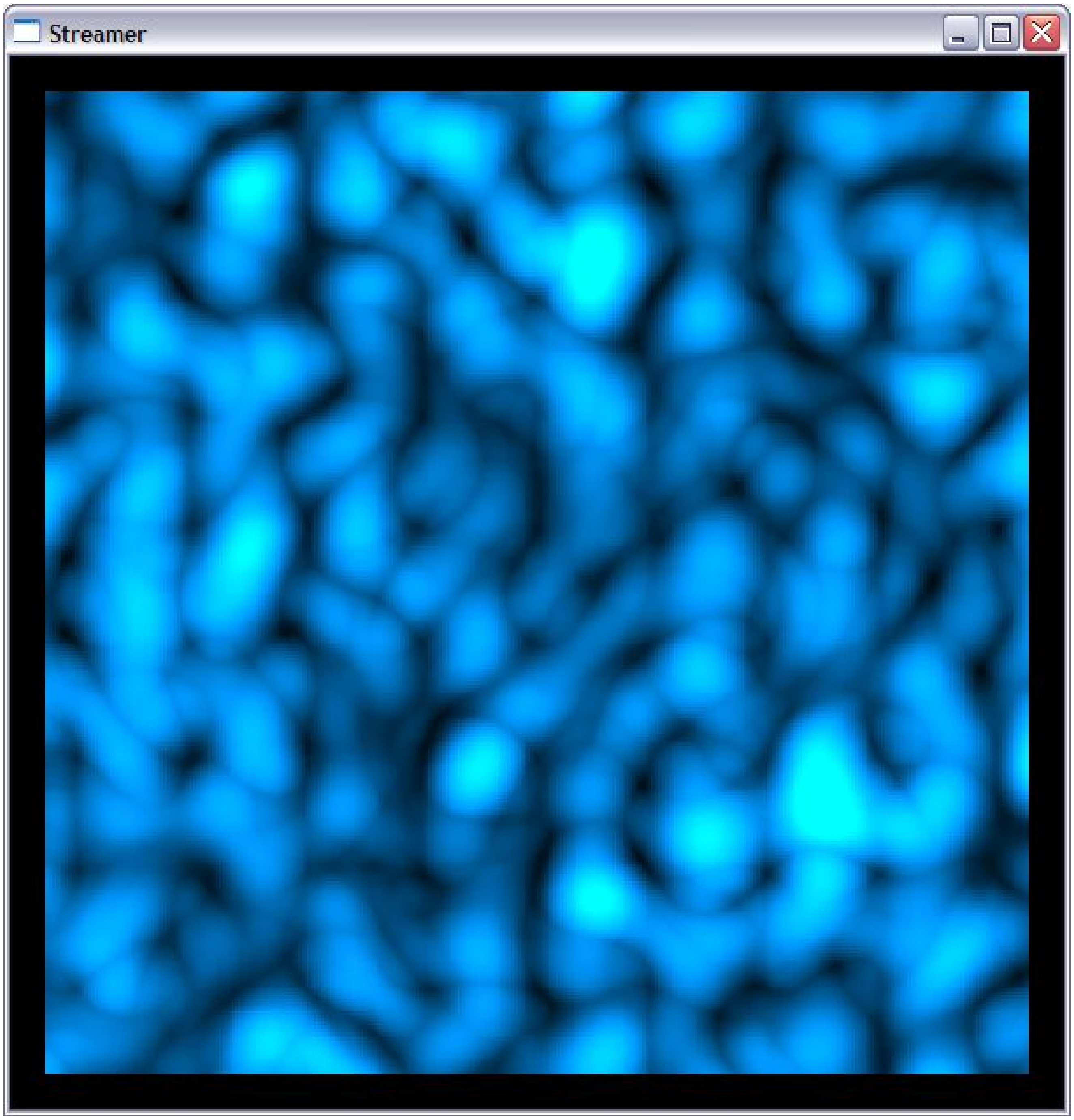


FIGURE 37 Final turbulence texture with 4 octaves of gradient noise for SPARK demonstration [DG].

The end result is a procedural texture constructed of multiple levels of gradient noise, that can be modified and adjusted in real-time. Depending on the bit depth of the pixel buffer, frame rates of more than 30fps for 4 octaves at RGBA 8bpp have been measured on an NVidia 5950.

5 SUMMARY AND CONCLUSION

Stream processing is a powerful model for performing complex computations in an efficient manner. As GPUs continue to outperform CPUs and become more generalized, a lot of responsibility will be placed on the tools necessary to program GPUs. A stream processing abstraction layer such as SPARK, offers a basic framework for developers who wish to explore general purpose computation using programmable graphics hardware.

5.1 FUTURE WORK

SPARK was designed as a bare bones framework for stream processing. Nevertheless, there are a number of features which could be added to make it more robust and usable for other developers. These include additional types of stream management, stream geometry, and support for other shading languages.

Furthermore, SPARK has not been rigorously tested. It would be helpful to get feedback from other developers who are not familiar with the internal workings of the library, and hear what they might have to say about the design.

Theoretically, SPARK could be used for processing any form of streaming media, including both audio and video. It would be interesting to see what kind of performance a stream processing abstraction layer could provide for digital media content.

In it's simplest form, SPARK is similar to a node or tile based system, such as Apple's Shake compositing package, and it would be fairly straight forward to provide stream operators appropriate for doing high quality image compositing.

There are many applications for the interactive high quality procedural texture synthesis. These include such areas as terrain modeling, volumetric hypertexture, natural phenomena. It would also be feasible to use SPARK as a procedural engine for off-line rendering.

Another application for procedural texture synthesis is evolutionary computer graphics, commonly referred to as genetic art. By combining evolutionary algorithms with a rendering engine for procedural textures developed with SPARK, a vast number of generations could be explored within an aesthetic landscape of possible visual art forms.

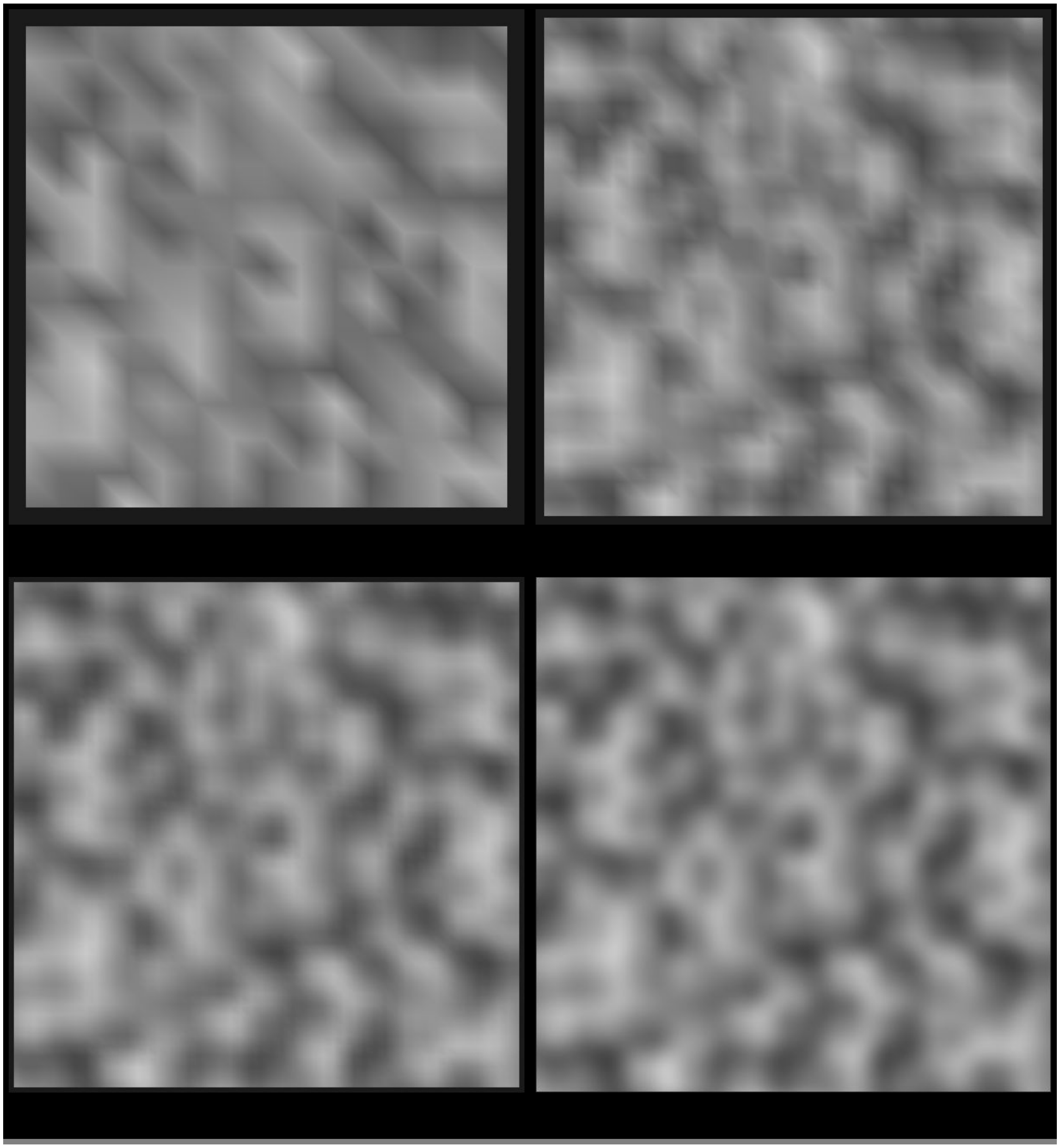


FIGURE 38 Four different levels of detail for one octave of vertex noise [DG].

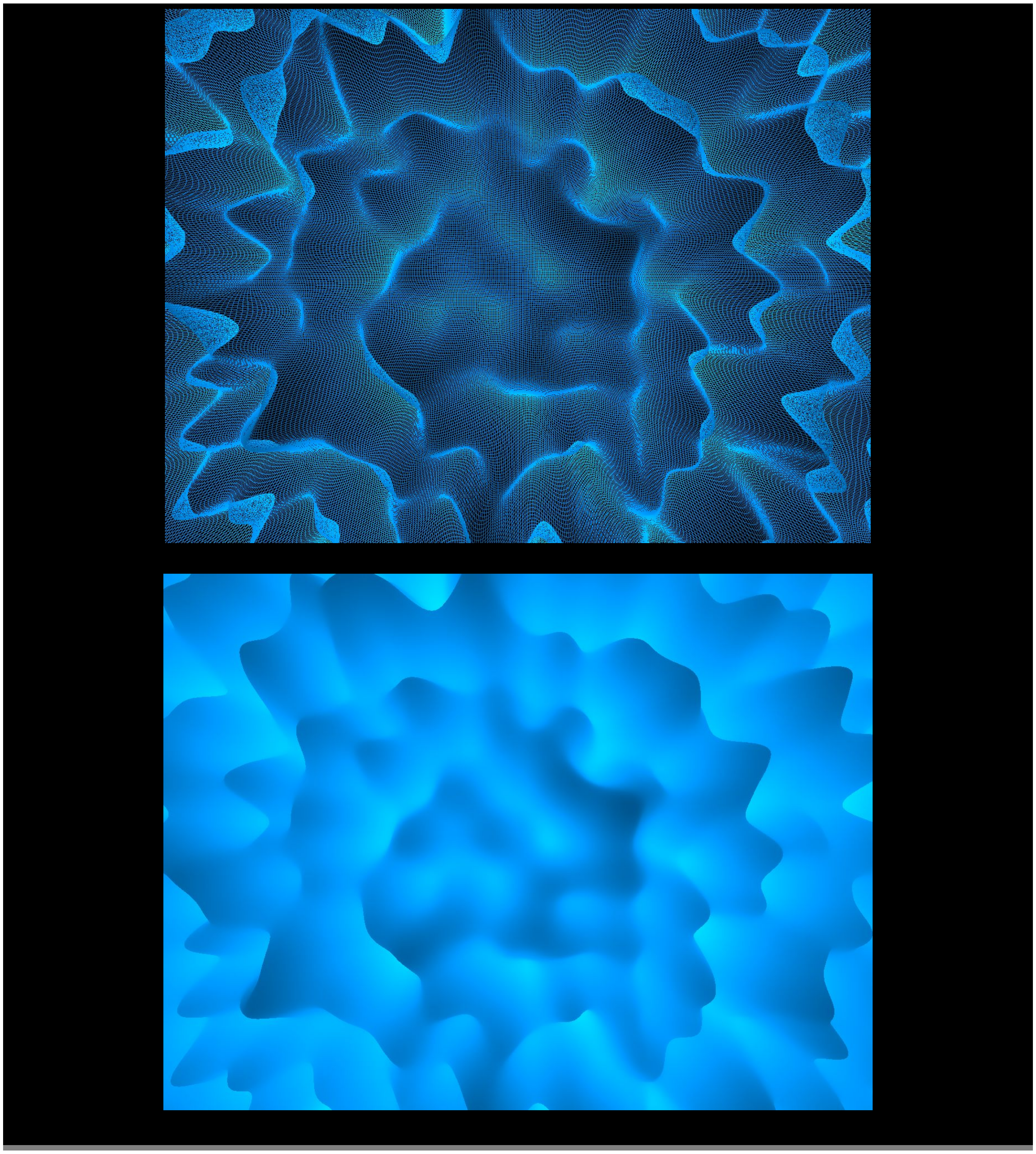


FIGURE 39 Mesh deformation from vertex noise, shown shaded and in wireframe [DG].

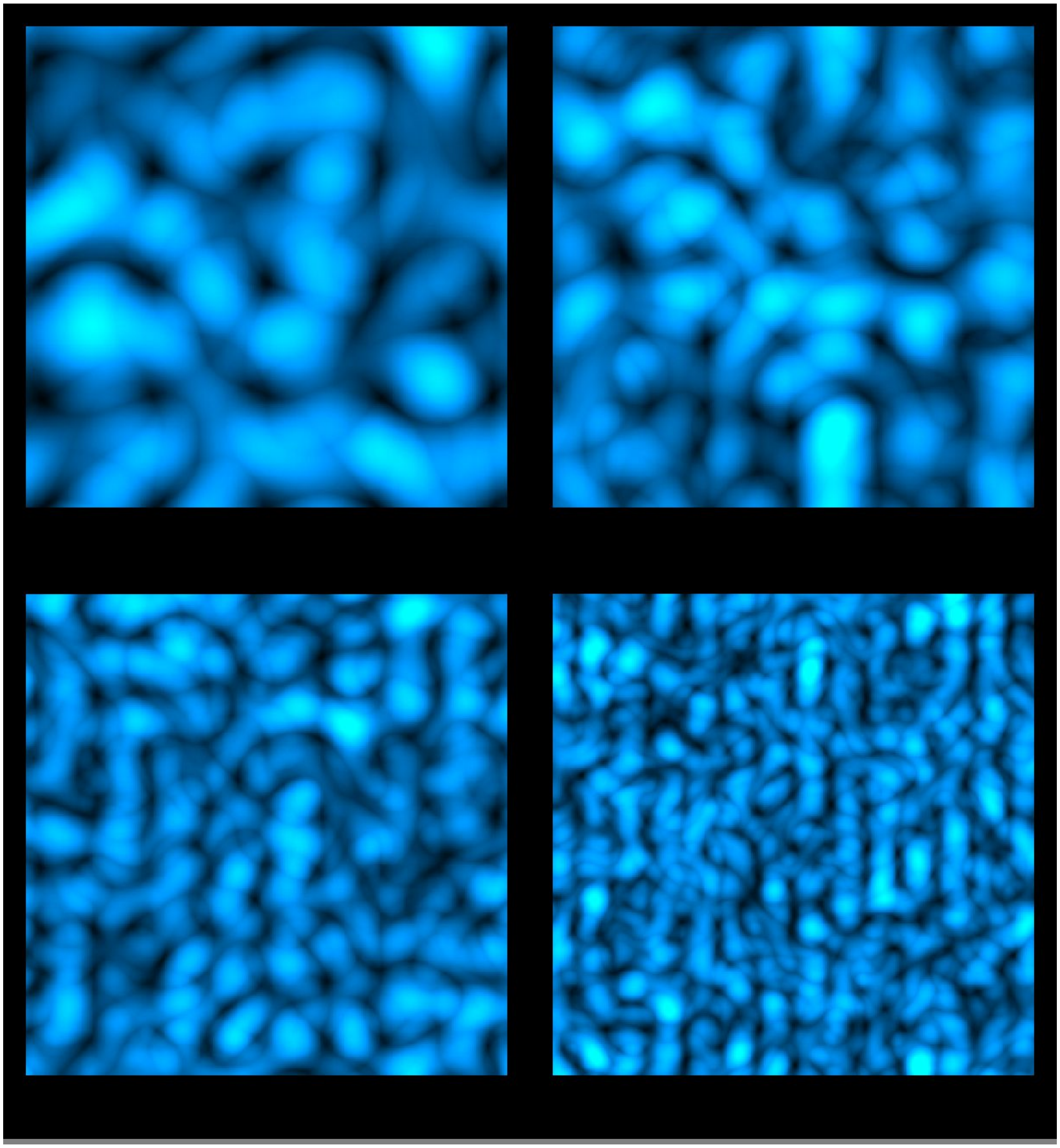


FIGURE 40 Four different views of 4x octaves of turbulence with vertex based noise [DG].

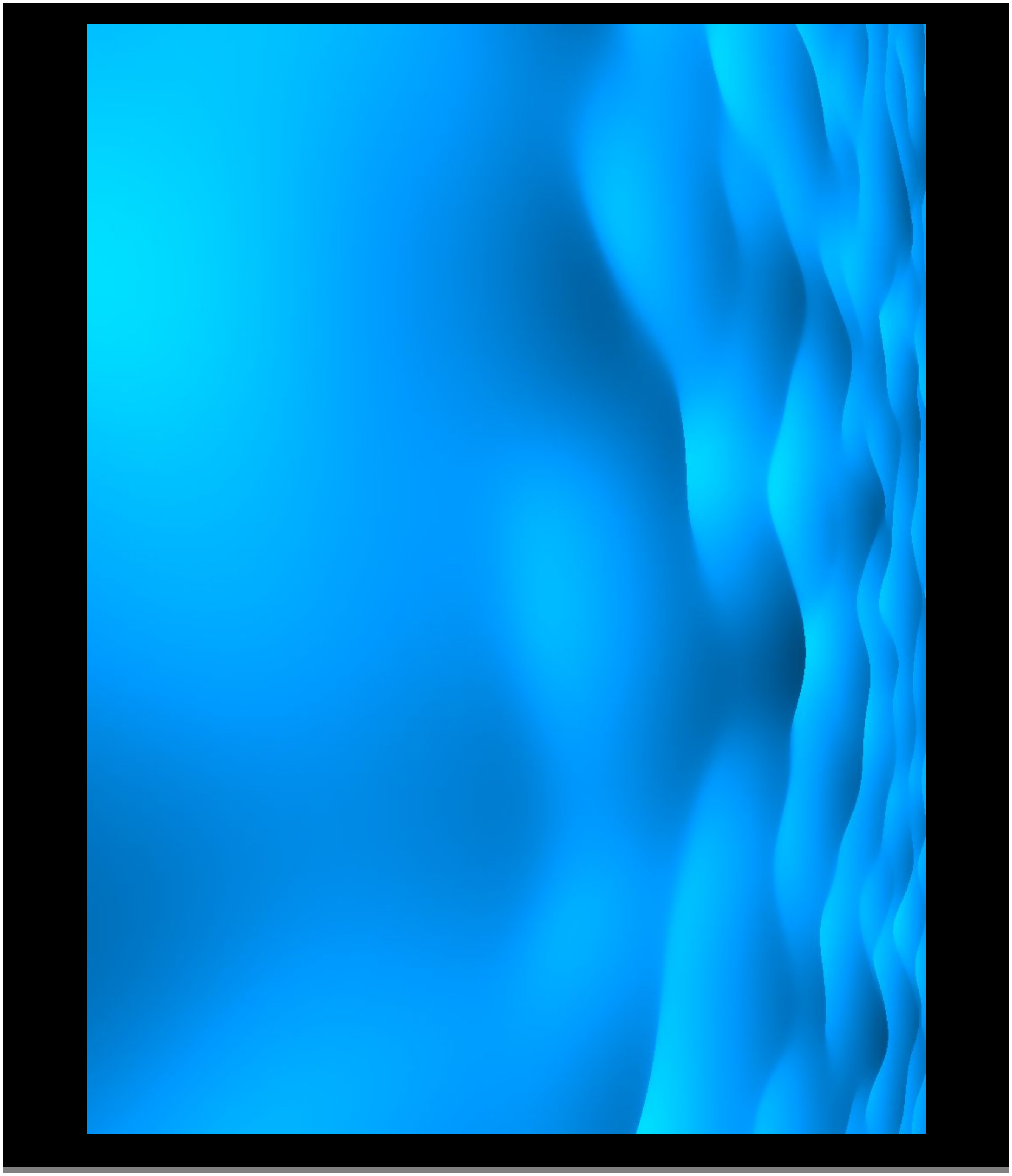


FIGURE 14 A procedural landscape built from vertex noise [DG].

5.2 ACKNOWLEDGEMENTS

Special thanks to Matthew Lewis, Ken Musgrave, and Peter Gerstmann for providing feedback and advice on this project. Special recognition goes to Mark Harris and the GPGPU community, who convinced me that stream processing was the way to go.

6 REFERENCES

APODACA, A. A., 2000. *Advanced Renderman: Creating CGI for Motion Pictures.* Academic Press.

ATI. 2004. *“ASHLI: Advanced Shading Language Interface.”* Available from: http://www.ati.com/developer/ashli.html [Accessed 10 August 2004].

BAXTER, W., 2004. *“The Image Debugger”.* Available from: http://www.cs.unc.edu/~baxter/projects/imdebug/ [Accessed 01 August 2004].

BROOKS, F. 2004. *“On the Power of Streaming Table-Lookup”*. Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

BUCK, I., FATAHALIAN, K., HANRAHAN, P. 2004. *“GPUBench: Evaluating GPU Performance for Numerical and Scientifc Applications”.* Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

BURKS, A. W., GOLDSTINE, H. H., and von NEUMANN, J. 1963. *“Preliminary discussion of the logical design of an electronic computing instrument”*. In Taub, A. H., editor, John von Neumann Collected Works, The Macmillan Co., New York, Volume V, 34-79.

DALLY, W. J., HANRAHAN, P., EREZ. M., KNIGHT, T. J., LABONTE F., AHN J., JAYASENA, N., KAPASI, U. J., DAS, A., GUMMARAJU, J., BUCK, I., 2003. "*Merrimac: Supercomputing with Streams*", SC2003, November 2003, Phoenix, Arizona.

DALLY, W. J., 2004. *“Stream Processors vs Graphics Processing Units”*. Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

EBERT, D., MUSGRAVE, F. K., PEACHEY, D., PERLIN K., WORLEY, S., MARK, W. R., HART, J. C.. 2002. *Texturing & Modeling: A Procedural Approach, Third Edition*. Morgan Kaufmann.

FATAHALIAN, K., SUGERMAN, J., HANRAHAN, P.. 2004 *“Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication”.* Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

FERNANDO, R. 2004. *“GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics”.* Addison-Wesley. Boston, MA.

HANRAHAN, P. 2004. *“Streaming Programming Environments”*. Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

HARPER, J., 2004. *“The IBM 7090/94 System Architecture”.* Available from: http://www.frobenius.com/7090.htm [Accessed 01 Aug 2004].

HARRIS, M. 2004. “*SIGGRAPH 2004 GPGPU Course Notes: Mapping Computational Models to GPUs”.* Proceedings of ACM SIGGRAPH 2004, Aug 2004, Los Angeles, CA.

HEROUX, M. 2004. “*Using GPUs as CPUs for Engineering Applications: Challenges and Issues”.* Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

HILLESLAND, K., LASTRA, A. 2004 *“Floating Point Paranoia”*. Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

IBM. 1957. “*Revised Manual: Harvest System”.* Available from: http://bitsavers.org/pdf/ibm/7950/Harvest_Nov1957.pdf [Accessed 01 Aug 2004].

KAPASI, U. J., DALLY, W. J., RIXNER, S., OWENS, J. D., KHAILANY, B. 2002. “*The IMAGINE Stream Processor*”, Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02); Page 282.

LEFOHN, A. 2004 *“SIGGRAPH 2004 Course Notes: GPGPU”.* Proceedings of ACM SIGGRAPH 2004, Aug 2004, Los Angeles, CA.

MACE, R., 2004. *“OpenGL ARB Superbuffers”.* Available from: http://www.ati.com/developer/gdc/SuperBuffers.pdf [Accessed 01 Aug 2004].

MCCOOL, M. D., Qin Z., POPA, T. S., 2002. *“Shader Metaprogramming”*. SIGGRAPH/Eurographics Graphics Hardware Workshop, September 2-3, 2002, Saarbruecken, Germany, pp. 57-68.

MCCOOL, M. D., DU TOIT, S. 2004, *“Metaprogramming GPUs with Sh”*. AK Peters, Ltd. Wellesley, MA.

MCCOOL, M. D., TOIT, S. S., POPA, T. S., CHAN, B., MOULE, K. 2004. *“Shader Algebra”*. Proceedings of ACM SIGGRAPH 2004, Aug 2004, Los Angeles, CA.

MICROSOFT. 2004. “*Shader Model 3.0”.* Available from: http://www.microsoft.com/whdc/winhec/partners/shadermodel30_NVidia.mspx/

MOORE, G. E. 1965. "*Cramming More Components Onto Integrated Circuits*". Electronics, April 19.

NVNEWS. 2004. “GeForce 6 Series: Questions and Answers with Nvidia”. Available from: http://www.nvnews.net/articles/geforce_6_interview/index.shtml [Accessed 01 August 2004].

NVIDIA. 2004. *“Cg Toolkit: User's Manual v1.2”.* Available from: http://developer.NVidia.com/cg [Accessed 01 August 2004].

PURCELL, T., SEN, P. 2004. *“Shadesmith Fragment Program Debugger”.*

Available from: http://graphics.stanford.edu/projects/shadesmith/ [Accessed 01 Aug 2004].

ROST, R. 2004. *“SIGGRAPH 2004 Course Notes: Real Time Shading”.* Proceedings of ACM SIGGRAPH 2004, Aug 2004, Los Angeles, CA.

SEAGER, M. 2004. *“GPU Requirements for Large Scale Scientific Applications”.* Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, GP2 2004, Aug 2004, Los Angeles, CA.

SALOMON, D., 2004. *“A Short History of Computers”.* Available from: http://www.ecs.csun.edu/~dsalomon/history/histcomp.html [Accessed 01 Aug 2004].