GPU Computation with Cg

Jason C. Gaulin
Advisor: K. R. Subramanian

May, 2004

University of North Carolina at Charlotte

GPU Computation with Cg

Jason Gaulin
Advisor: Dr K. R. Subramanian
May, 2004

Department of Computer Science
The University of North Carolina at Charlotte
Charlotte, NC 28223-0001

Table of Contents

Chapter 1 - Introduction..1

Chapter 2 - Background Information......................................2

2.1 - GPU Computation...2

2.2 - The Test GPU..4

2.3 - Cg Language...4

2.4 - Convolution...7

Chapter 3 - Timing Mechanism..9

Chapter 4 - The 2D Application..10

4.1 - Description...10

4.2 - Implementation...11

4.2.1 - User Interface..11

4.3 - Performance...13

Chapter 5 - The 3D Application..15

5.1 - Description...15

5.2 - Application Implementation.......................................15

5.2.1 - User Interface..15

5.2.2 - Core...17

5.3 - CPU Computation...17

5.4 - GPU Computation...18

5.4.1 - Application..18

5.4.2 - Component Convolution......................................19

5.4.3 - Vectorized Convolution.......................................20

5.4.4 - Packed Textures...21

5.4.5 - Convolution in Multiple Passes............................26

5.5 - Performance Comparison...28

Chapter 6 - Future..32

6.1 - GPU Limitations...32

6.2 - Potential Optimizations...33

6.3 - Future GPUs..33

Chapter 7 - Conclusion...*35*

Appendix A - Cg Program Examples.......................................36

Appendix B - References..39

Table of Figures

Figure 2.1 – The Programmable Graphics Pipeline................3

Figure 2.2 - Supported profiles in Cg 1.2.1............................6

Figure 3.1 - Timer and Simplified OpenGL Pipeline...............9

Figure 4.1 – “GPU Filters” Application...................................12

Figure 4.2 – 2D Operation Performance................................14

Figure 5.1 – “Volume Viewer Filter” Application.....................16

Figure 5.2 – Per-component Convolution (3x3 represented). .19

Figure 5.3 – Vectorized Convolution (3x3 represented)..........21

Figure 5.4 – Original Texture Representation in OpenGL.......22

Figure 5.5 – Packed Texture Representation in OpenGL.......23

Figure 5.6 – Packing, Window Size Comparison...................24

Figure 5.7 – Unpacking Operations.......................................25

Figure 5.8 – Packed Texture...26

Figure 5.9 – Multi-pass Convolution (5x5 represented)..........28

Figure 5.10 – Tooth Dataset Used for Testing.......................29

Figure 5.11 – 3x3x3 Gaussian Kernel Performance...............30

Figure 5.12 – Packed Texture Performance..........................31

Figure A.1 – 2D Grayscale Filter..36

Figure A.2 – “get_fragments” Support Function.....................36

Figure A.3 – “ConvolutionFilter3” Per-Component.................37

Chapter 1 - Introduction

Demands in computer graphics have caused great advances in computer graphics hardware over the past fifteen years. With the increasing power of computers, visualization of three-dimensional data by a computer has become important in various fields of science. What was once an expensive and complicated field is now easier and more affordable as commodity hardware is able to process data at acceptable speeds and precision.

This report documents my senior project at the University of North Carolina at Charlotte related to a promising area of computer graphics, GPU computation, and ways in which this can support data processing for visualization. Convolution was chosen as a practical application of GPU computation. A two-dimensional convolution program, and a three-dimensional convolution program were written using Cg, a high-level language for writing programs which execute on graphics cards.

As commodity graphics hardware is rapidly evolving, there are still many limitations on the current generation of this hardware. This led to many unforeseen difficulties in reaching the goal of practical GPU computation. These problems, as well as solutions, are documented.

Chapter 2 - Background Information

2.1 GPU Computation

A GPU, or a Graphics Processing Unit, is a processor on a graphics card [1]. Advanced commercial GPUs, from companies such as SGI, and modern commodity GPUs, from companies such as NVIDIA and ATI, support advanced computation through the use of fragment and vertex programs. A fragment program processes fragments in a frame, returning color information for a given fragment's coordinates [2]. A vertex program performs operations on all vertices in a given context [2]. Fragment and vertex programs are often referred to as “shaders.”

GPUs are stream processors. They are given a list of data (fragments or vertices), and the programs are executed on each element of the list. GPUs have multiple pipelines so data can be processed in parallel [3]. Due to the specialized nature of the data, GPUs are greatly optimized compared to traditional CPUs. “Stanford's Ian Buck calculates that the current GeForce FX 5900's performance peaks at 20 Gigaflops, the equivalent of a 10-GHz Pentium” [3].

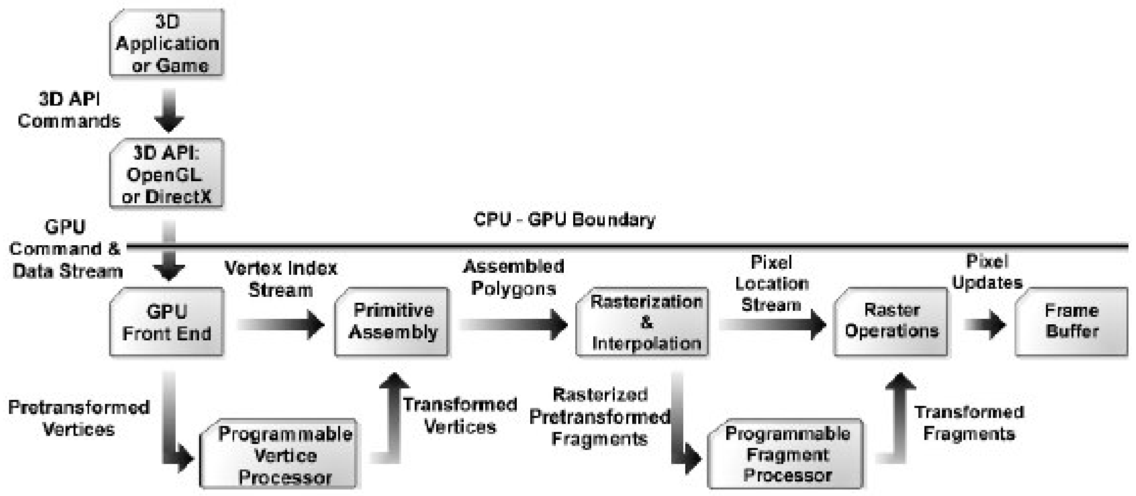


Figure 2.1 – The Programmable Graphics Pipeline [4]

Geometry and pixel data is streamed from the application (CPU-side) to the graphics card. In a fixed-function pipeline, geometry in the form of vertices and texture coordinates are transformed to prepare them for rasterization [10]. Parts of this transformation process can be configured by the application, but these capabilities are limited. The primitive assembly stage involves performing operations to generate fragments from the geometric data in relation to the viewport [10]. These fragments are later applied color values dependent on variables such as light sources and textures [10].

The programmable graphics pipeline (see Fig. 2.1) is similar to the traditional fixed-function pipeline. A vertex processor carries out the operations as specified by a vertex program. This program processes the geometric data and transforms vertices [4]. The primitive assembly stage performs the same way as in a fixed-function pipeline, generating fragments for the fragment processor. A fragment processor carries out the operations specified by a fragment program. The fragment program returns parameters for each fragment [4]. Usually, as in a fixed-function pipeline, this could simply be interpolated color values from vertices or a color value from a texture. However, due to the programmable nature of a fragment processor, more complex operations can be performed on fragments.

There are limitations with this kind of architecture. Some features of general-purpose CPUs are not available on GPUs. See section 6.1 for more information.

2.2 The Test GPU

The programs were developed for use on the NV30 GPU. This processor was developed by NVIDIA and is used in the GeforceFX 5200 and GeforceFX 5600.

2.3 Cg Language

While commodity GPUs began to support execution of external programs, these programs had to be written in assembly for that particular type of GPU. High-level programmable shader languages, such as RenderMan, have been popular in slower, off-line rendering applications [4]. The need for a high-level shading language supporting commodity GPUs from multiple vendors led to the creation of the Cg shading language [4].

Cg is a GPU programming language created by NVIDIA. The language has a C-like syntax and philosophy in that it is hardware oriented, rather than application oriented [12]. Rather than have data types for vertices and colors, Cg supports generic types defined by the programmer with types such as arrays, which can be used to store vertices and colors [12]. This freedom allows for more computation-oriented programs rather than just traditional rendering-specific programs. The syntax is modified to an extent from C in that Cg deals with streams of data.

The creators of Cg had many goals for the language. Having a high-level shading language simplifies the work of the programmer and increases portability [12]. Programmers do not have to learn a different language for each GPU, and programs written in Cg work on multiple GPUs.

To support past, present, and future GPUs and multiple computer graphics APIs, the language makes use of a concept known as “profiles.” In the Cg 1.2.1 release, there are six vertex profiles and eight fragment profiles [5]. Each profile has a set of supported features. This allows Cg to be expanded as new GPUs are released, but fragments the Cg language since a developer can not be sure what features the end-user's GPU will have.

Vertex Profiles	***Platform***
vs_1_1	DirectX 8 and DirectX 9
vs_2_0 and vs_2_x	DirectX 9
arbvp1	OpenGL ARB_vertex_program
vp20, vp30	NV_Vertex_program 1.0 and 2.0

Fragment/Pixel Profiles	***Platform***
ps_1_1, ps_1_2 and ps_1_3	DirectX 8 and DirectX 9
ps_2_0 and ps_2_x	DirectX 9
arbfp1	OpenGL ARB_fragment_program
fp20	NV_register_combiners and NV_Texture_shader
fp30	NV30 OpenGL fragment programs

Figure 2.2 - Supported profiles in Cg 1.2.1 [5]

Cg profiles can be grouped into two basic fields, differentiated by their function and place in the pipeline: vertex profiles and fragment profiles. In the pipeline, the vertex program is executed first, performing all geometric operations. The fragment program is executed last, coloring the fragments in the frame.

Cg programs can be compiled before running the CPU application, or while the CPU application is running. Waiting to compile at CPU application runtime slows the program, but allows for hardware and runtime dependent settings to be set. For example, a Cg program may use features available only on advanced GPUs. At application runtime, the application can determine what support is available in the GPU and modify the Cg program in such a way as to limit it to the user's GPU.

Data is passed to a Cg program in two ways. The CPU application may pass “uniform parameters.” A uniform parameter is a constant passed to all calls in a given fragment or vertex program. This could be the position of a light source, the dimensions of a volume, a texture, or anything that is constant for the stream of data passed to the Cg program. Uniform parameters may not be modified by a Cg program itself.

"Varying parameters" are passed to a Cg program by the GPU itself. A Cg program accesses these by looking at registers in the graphics card. A varying parameter could be texture coordinates, vertex coordinates, a color, or any other data which is accessible from a Cg program through the graphics card's registers. In general, the value of a varying parameter changes as different elements of a stream are processed by a Cg program.

For examples of Cg fragment programs, see Appendix A.

2.4 Convolution

In general, "convolution is a mathematical operator which takes two functions *f* and *g* and produces a third function that in a sense represents the amount of overlap between *f* and a reversed and translated version of *g*" [7]. Image processing involves using a discreet convolution method, based on the following functions [8]:

$$\text{2D:} O_{i,j} = \sum_{q=1}^{m} \sum_{r=1}^{n} I\ (\text{i+q-1, j+r-1}) K(q\ , r\ )$$

$$\text{3D:} O_{i,j,k} = \sum_{q=1}^{m} \sum_{r=1}^{n} \sum_{s=1}^{o} I\ (\text{i+q-1, j+r-1, k+s-1}) K(q\ , r\ , s)$$

In this case, a kernel K is applied to an image I in the neighborhood of $(\text{i},\ \text{j},\ \text{k})$ and returns a single value at that location. The kernel's size, and therefore the window size, is $m\ X n\ X o$ in three dimensions.

The following example applies a 3x3, two-dimensional kernel to a single color channel of a 4x4 image at a single point.

$$Image\begin{bmatrix}20 & 32 & 47 & 2\\ 58 & 120 & 200 & 87\\ 33 & 8 & 74 & 72\\ 4 & 2 & 65 & 3\end{bmatrix} Kernel \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \rightarrow$$

$$Output_{2,2} = \sum_{q=1}^{3}\sum_{r=1}^{3} I(1+q-1, 2+r-1) K(q, r)$$

$$Output_{2,2} = \sum \begin{pmatrix}(20*1) & (32*2) & (47*3)\\ (58*4) & (120*5) & (200*6)\\ (33*7) & (8*8) & (74*9)\end{pmatrix}$$

$$Output_{2,2} = 3218$$

Convolution simplifies applying filtering operations to a set of data in spatial domain. There are many well known filtering kernels. The “VolumeViewerFilter” application generates averaging and Gaussian [9] filters. For reference, this is how i an j are defined for the following examples:

$$\textit{Coordinates for a } 3x3 \textit{Kernel}$$
$$(\mathrm{i, j}) = \begin{bmatrix}(-1,-1) & (0,-1) & (1,-1)\\ (-1,0) & (0,0) & (1,0)\\ (-1,1) & (0,1) & (1,1)\end{bmatrix}$$

$$\textit{Averaging Filter}$$
$$\text{2D}: K_{i,j} = \frac{1}{(m*n)}$$
$$\text{3D}: K_{i,j,k} = \frac{1}{(m*n*o)}$$

$$3x3 \textit{ Averaging Kernel}$$
$$K = \begin{bmatrix}.1111 & .1111 & .1111\\ .1111 & .1111 & .1111\\ .1111 & .1111 & .1111\end{bmatrix}$$

$$\textit{Gaussian Filter}$$
$$\text{2D}: K_{i,j} = \frac{e^{-\frac{i^2+j^2}{2*\sigma^2}}}{2*\pi*\sigma^2}$$
$$\text{3D}: K_{i,j,k} = \frac{e^{-\frac{i^2+j^2+k^2}{2*\sigma^2}}}{2*\pi^{\frac{3}{2}}*\sigma}$$

$$3x3 \textit{ Gaussian Kernel } (\sigma=1)$$
$$K = \begin{bmatrix}.0586 & .0965 & .0586\\ .0965 & .1591 & .0965\\ .0586 & .0965 & .0586\end{bmatrix}$$

Chapter 3 - Timing Mechanism

For accurately measuring performance of the Cg programs, special considerations had to be made. Cg does not have access to a timer on the video card, so all timing is performed on the CPU.

Immediately before rendering the frame, the timer is started. After the rendering functions are called, the “glFinish” function is called to force all operations in the OpenGL pipeline to complete. This guarantees that the Cg program will have finished execution by the time “glFinish” returns control to the calling function. Finally, the timer's value is recorded.

Due to the way the timer is started and stopped, it does not record only the execution time of the Cg program. It records the time it takes to render the entire frame (see Fig. 3.1).

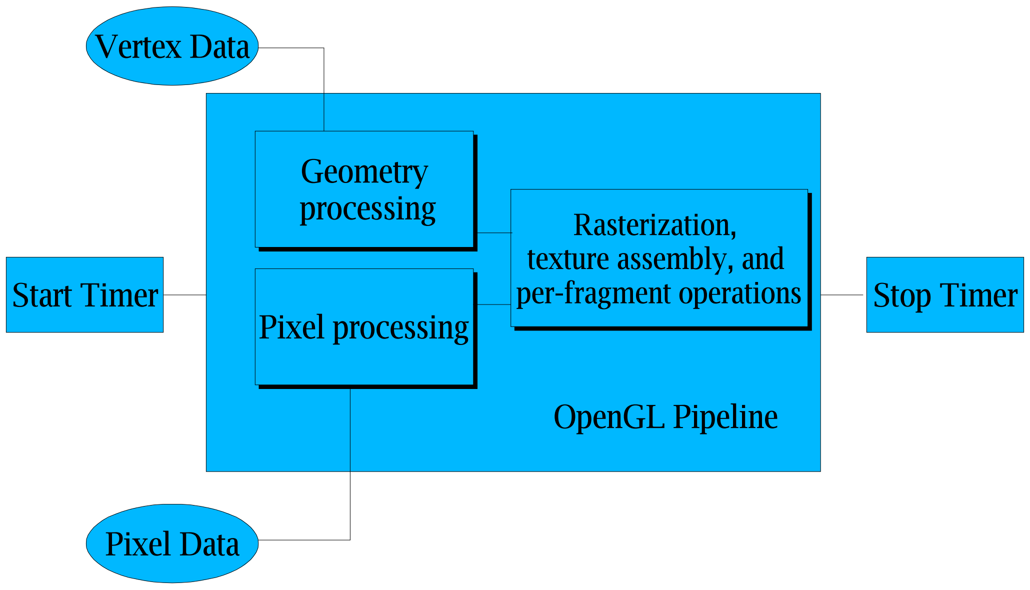


Figure 3.1 - Timer and Simplified OpenGL Pipeline [10]

Chapter 4 - The 2D Application

4.1 Description

The first application, “GPU Filters” performs filtering operations on two-dimensional color images on the GPU. This application's purpose was to show that GPU convolution operations are feasible.

“GPU Filters” has five built-in filtering operations, as well as a convolution mode. The built-in filtering operations are “Average”, “Grayscale”, “Highpass”, “High-boost”, and “Median.” The convolution mode only supports 3x3 kernels.

The “Average” filter applies a 3x3 convolution kernel which averages all the pixels in the window together. The “Grayscale” filter simply averages the red, green, and blue components of each pixel together, and returns the resulting intensity value as the color of the pixel.

The “Highpass” filter applies the 3x3 convolution kernel below. This has the effect of sharpening the image by increasing the weight of the center pixel in relation to its surrounding pixels.

$$HighPass\,K = \frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The “High-boost” filter is similar to the “Highpass” filter. The high pass filter loses some low-frequency components of the image. To overcome this, a weighted version of the original image is added with the filtered image.

$$HighBoost \quad K = w * I_{(j,k)} * \frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The “Median” filter is a complicated test for the GPU due to its non-linear nature. This filter retrieves the 3x3 neighborhood of pixels, sorts them, then returns the central pixel of the sorted list as the color at that location.

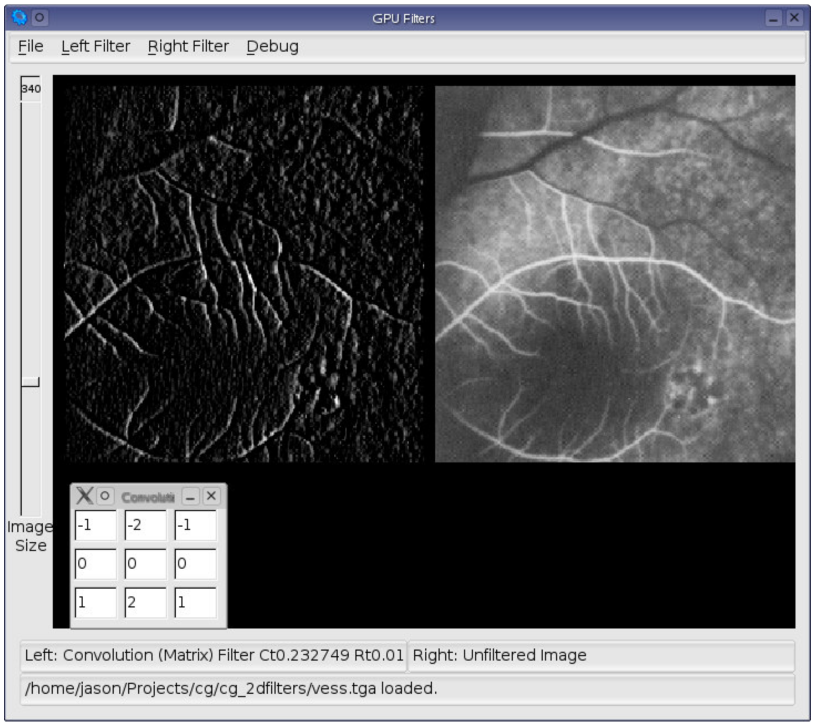
The “Convolution” mode simply creates a small window for the user to enter a 3x3 kernel. The filtered image is calculated when the view window is refreshed.

4.2 Implementation

“GPU Filters” is written in C++ and uses FLTK for window management, OpenGL for rendering the images, Cg for GPU computation, and the timer describe in chapter three.

User Interface

“GPU Filters” uses the Fast Light Toolkit for drawing and managing windows and widgets. The interface is based off of “JDraw” by Joshua Foster.



-1	-2	-1
0	0	0
1	2	1

Left: Convolution (Matrix) Filter Ct0.232749 Rt0.01 Right: Unfiltered Image
/home/jason/Projects/cg/cg_2dfilters/vess.tga loaded.

Figure 4.1 – "GPU Filters" Application

Images may be opened from the "File" menu. The only image format supported are targa (tga) files. The next two options in the menu bar are for applying filters. The program displays the image in two different "slots." Each slot may have a different filtering operation performed, or none at all.

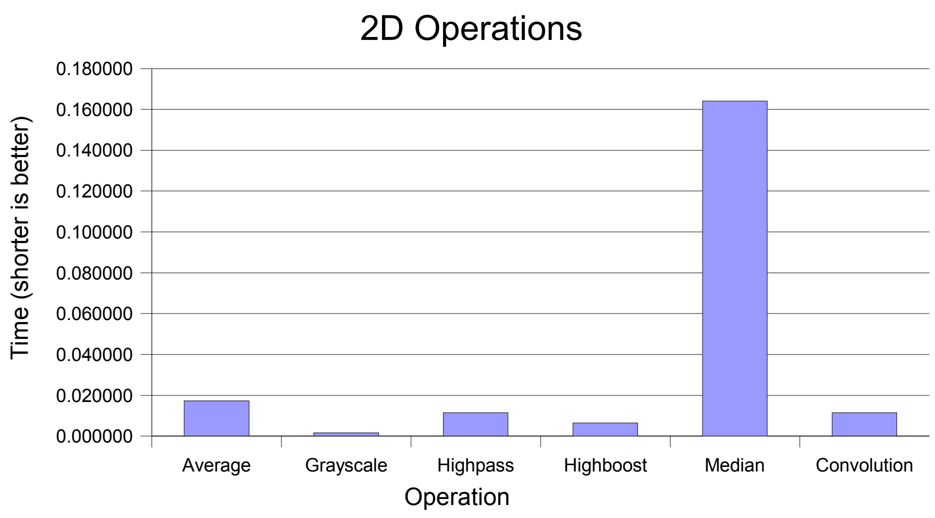
On the left of the window is a sliding bar. This is for scaling the images. When an image is loaded, the bar will be moved so that the images are displayed at their original resolution. Sliding the bar down increases their sizes, while sliding the bar up decreases the image sizes.

The main area of the window contains the images themselves. Below each image is a status display showing which filter is currently applied, the length of time it took to compile the Cg program being used, and the length of time it took to execute the Cg program. Below these status bars is another status bar showing the path of the currently loaded image.

When one of the image slots is in “Convolution” mode, a separate window is shown containing a 3x3 list of text boxes. Each text box represents a value in the convolution kernel to be applied to that image slot. A “-” (minus sign) may be used to indicate negative numbers. Decimal numbers are accepted, but fractions are not allowed. The convolution kernel is applied as soon as a request to redraw the window is issued, such as by resizing the image or dragging the matrix window over the image area of the main window.

4.3 Performance

All tests were performed with an NVIDIA GeForce 5600FX, which has a NV30 GPU. Each filter was applied three times to find an average run-time for each filter. The image operated on is the one shown in Fig. 4.1 . All times are reported in seconds using the timer described in chapter three.



Operation	First Run	Second Run	Third Run	Average
Average	0.0229	0.0119	0.0171	0.0173
Grayscale	0.0015	0.0015	0.0015	0.0015
Highpass	0.0114	0.0112	0.0118	0.0115
Highboost	0.0068	0.0064	0.0063	0.0065
Median	0.1635	0.1646	0.1643	0.1641
Convolution	0.0113	0.0118	0.0113	0.0115

Figure 4.2 – 2D Operation Performance

The "Average", "Highpass", "Highboost", and "Convolution" operations perform roughly the same. The "Grayscale" operation is very quick because all it does is average each color channel at a particular pixel and returns those results. The "Median" operation takes a very long time due to its heavy use of conditional operators for sorting.

Chapter 5 - The 3D Application

5.1 Description

"Volume Viewer Filter" is an application written in C++ which performs three-dimensional convolution operations on a volume using either the GPU or the CPU. The window size can be dynamically set, from 3x3x3 to 7x7x7, however not all kernel sizes are supported on the GPU. The values for the kernel can be entered manually, or one of a set of default kernels can be generated.

CPU computation is facilitated by the use of the Visualization Toolkit (VTK). VTK is an object-oriented toolkit consisting of functions and classes used to create, process, and visualize data [6].

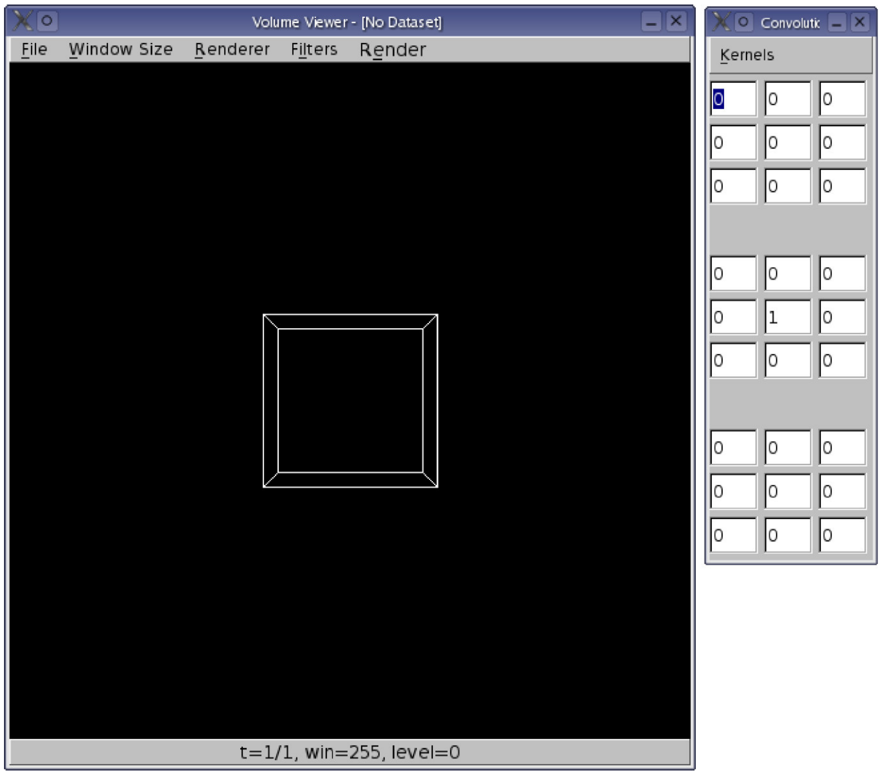
GPU computation makes use of NVIDIA's Cg language in the form of various fragment programs. One program is generic and able to, in theory, handle any window size. Others are specialized to handle only certain window sizes, but are faster compared to the generic version.

5.2 Application Implementation

There are two parts to the application: the CPU executable, and the GPU fragment shaders. The GPU shaders are executed by the CPU program.

User Interface

"Volume Viewer Filter" is based from code by William Carruthers. The "Fast Light Toolkit" is used for drawing windows. There are two windows: the render window and the convolution matrix window.



0	0	0
0	0	0
0	0	0
0	0	0
0	1	0
0	0	0
0	0	0
0	0	0
0	0	0

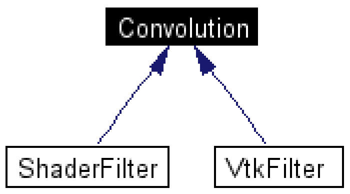
Figure 5.1 – "Volume Viewer Filter" Application

From the render window, datasets may be loaded, the window size can be set, the filter can be turned on and off, and the convolution method can be switched between CPU and GPU computation.

The render window is represented by the "ConvolutionViewer" class. This class is a subclass of the "Viewer" class, and is similar to the "VolumeViewer" class.

The matrix window can generate default kernels, or the user can set values for the kernel. The "Gaussian" option generates a three-dimensional Gaussian kernel. The "Default" option fills the kernel with zeros, except for the center value. This has the effect of displaying the unfiltered volume. The "Average" option generates a simple averaging filter, equally averaging all voxels in the volume.

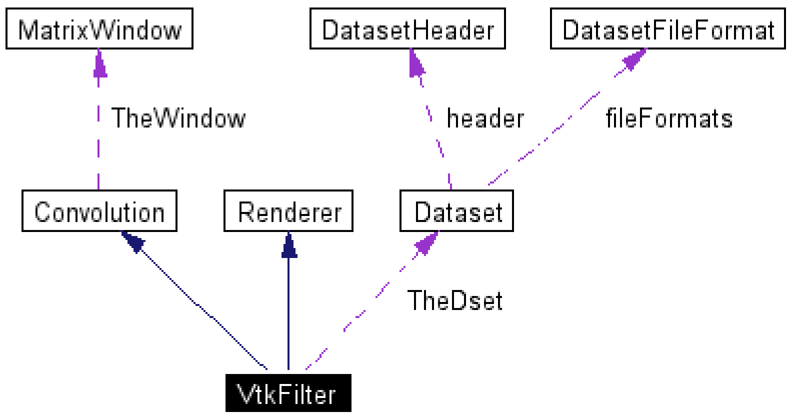
Core



The "Convolution" class is the primary class which groups convolution methods with the user interface. This class stores the window size and the convolution kernel. Since all convolution methods are subclassed from this class, it contains a generic interface for all convolution methods.

One function of the "Convolution" class is to display the window for managing the convolution matrix, through the use of the "MatrixWindow" class.

5.3 CPU Computation



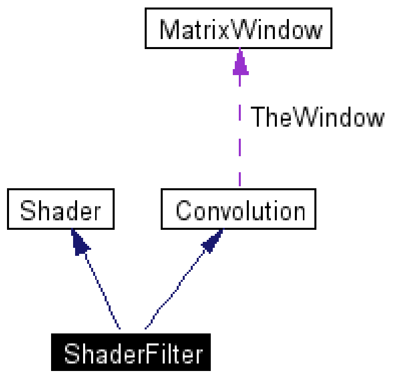
The “VtkFilter” class handles software computation through the use of the Visualization Toolkit. It is subclassed from the “Convolution” class and the “Renderer” class.

The first step in the CPU computation is to convert the dataset into a format VTK can understand. The dataset is converted into a VtkImageData as a series of unsigned characters. At this stage, the convolution operation may be performed using VTK functionality. The VtkImageConvolve class can apply convolution kernels with window sizes 3x3x3, 5x5x5, or 7x7x7. The resulting image is rendered using VTK's rendering capabilities.

5.4 GPU Computation

Application

The first component of GPU computation is the “ShaderFilter” class. This class is subclassed from the “Convolution” class and the “Shader” class.



All of the Cg convolution programs have similar interfaces. Each retrieves the current texture coordinate from the GPU. Also, each is passed a series of uniform parameters: the texture, the estimated size of a fragment, and the convolution matrix. Each program also has a single output: the calculated value of the fragment.

Component Convolution

The first Cg convolution program, “ConvolutionFilter3”, supports variable size kernels. Since current generations of GPUs do not support loops in fragment programs, this is accomplished by setting the window size at shader compile time. Loops may then be unrolled by the Cg Compiler. In theory, any odd-numbered window size is supported, but in practice, due to the limits of the GPU in regards to the number of instructions, the only window size supported with this Cg program is 3x3x3.

“ConvolutionFilter3” works by performing NxNxN texture lookups, where “N” is the size of the window. As each fragment is retrieved, it is multiplied by the applicable value in the convolution matrix. The resulting value is added to a variable which is used to output the final color.

$$Convolve(I_{4,4}) = \begin{bmatrix} I_{3,3} I_{3,4} I_{3,5} \\ I_{4,3} I_{4,4} I_{4,5} \\ I_{5,3} I_{5,4} I_{5,5} \end{bmatrix} \quad K = \begin{bmatrix} I_{1,1} I_{1,2} I_{1,3} \\ I_{2,1} I_{2,2} I_{2,3} \\ I_{3,1} I_{3,2} I_{3,3} \end{bmatrix}$$

$$\begin{array}{lll} I_{3,3} * K_{1,1} = O_1 & I_{3,4} * K_{1,2} = O_2 & I_{3,5} * K_{1,3} = O_3 \\ I_{4,3} * K_{1,1} = O_4 & I_{4,4} * K_{1,2} = O_5 & I_{4,5} * K_{1,3} = O_6 \\ I_{4,3} * K_{1,1} = O_7 & I_{4,4} * K_{1,2} = O_8 & I_{4,5} * K_{1,3} = O_9 \end{array}$$

$$Convolve(I_{4,4}) = O_1 + O_2 + O_3 + O_4 + O_5 + O_6 + O_7 + O_8 + O_9$$

Figure 5.2 – Per-component Convolution (3x3 represented)

This method is extremely inefficient. Not only are there N^3 texture lookups per fragment, there are N^3 multiplications for applying the kernel. For a 3x3x3 kernel, this translates to twenty seven texture lookups and twenty seven multiplications, generating a fragment program that is 183 instructions long. For a 5x5x5 kernel, 125 texture lookups and multiplications are needed, generating a fragment program 1063 instructions long. This is too long as the NV30 only supports fragment programs 1024 instructions long.

Vectorized Convolution

"ConvolutionFilter3v" is a more optimized version of "ConvolutionFilter3." Rather than immediately multiplying values retrieved from the texture with the kernel, it stores each value in a vector. Cg supports a maximum vector size of four values, so the kernel is broken into a series of vectors. Finally, the dot product is calculated between the vectors.

$$Convolve(I_{4,4}) = \begin{bmatrix} I_{3,3} I_{3,4} I_{3,5} \\ I_{4,3} I_{4,4} I_{4,5} \\ I_{5,3} I_{5,4} I_{5,5} \end{bmatrix} \quad K = \begin{bmatrix} K_{1,1} K_{1,2} K_{1,3} \\ K_{2,1} K_{2,2} K_{2,3} \\ K_{3,1} K_{3,2} K_{3,3} \end{bmatrix}$$

$$Vi_1 = \begin{bmatrix} I_{3,3} \\ I_{3,4} \\ I_{3,5} \\ I_{4,3} \end{bmatrix} \quad Vi_2 = \begin{bmatrix} I_{4,4} \\ I_{4,5} \\ I_{5,3} \\ I_{5,4} \end{bmatrix} \quad Vi_3 = \begin{bmatrix} I_{5,5} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$Vk_1 = \begin{bmatrix} K_{1,1} \\ K_{1,2} \\ K_{1,3} \\ K_{2,1} \end{bmatrix} \quad Vk_2 = \begin{bmatrix} K_{2,2} \\ K_{2,3} \\ K_{3,1} \\ K_{3,2} \end{bmatrix} \quad Vk_3 = \begin{bmatrix} K_{3,3} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$Convolve(I_{4,4}) = Vi_1 \cdot Vk_1 + Vi_2 \cdot Vk_2 + Vi_3 \cdot Vk_3$$

Figure 5.3 – Vectorized Convolution (3x3 represented)

Vector manipulation is highly optimized in the GPU. Four-component vector multiplication takes roughly as much time as simply multiplying two floating point numbers. For a 3x3x3 kernel, this program still requires twenty seven texture lookups per fragment, but decreases the number of multiplications to seven. The resulting assembly program is 119 instructions.

Vectorizing the program gives a great performance increase, but due to the number of texture lookups, the speed is still much slower than the CPU method. This also decreases the number of instructions, which is important if larger kernel sizes are required. Another problem is vectorizing does not reduce the total number of instructions enough to support kernels larger than 3x3x3.

Packed Textures

The limiting factor up to this point is texture lookups. They slow the fragment shader down and they increase the program size. Texture lookups return a four component vector. Each component represents a color: red, green, blue, and alpha. The test volume is read assuming the "GL_LUMINANCE" format in OpenGL. This format assumes each value as the same across the red, green, and blue channels. Since the volumes being used store only intensity values, and are therefore gray-scale volumes, each component returned by a texture lookup contains the same value. The green, blue, and alpha components are ignored.

$$\text{Original Image}\ I = [I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8]$$

$$\text{GL\_LUMINASITY Texture}$$

$$\begin{array}{ll} O_1=[I_1 I_1 I_1 1] & O_2=[I_2 I_2 I_2 1] \\ O_3=[I_3 I_3 I_3 1] & O_4=[I_4 I_4 I_4 1] \\ O_5=[I_5 I_5 I_5 1] & O_6=[I_6 I_6 I_6 1] \\ O_7=[I_7 I_7 I_7 1] & O_8=[I_8 I_8 I_8 1] \end{array}$$

Figure 5.4 – Original Texture Representation in OpenGL

To decrease the number of texture lookups, multiple voxels could be packed into a single voxel and be represented using the "GL_RGBA" format. Under such conditions, the best case is texture lookups could be reduced to one texture lookup for every four voxels. For a 3x3x3 volume, the best case is seven texture lookups per fragment rather than twenty-seven, assuming an optimal packing method. Efficiency would be dependent on the packing method, and the dimensions of the volume.

$$\text{Original Image } I = [I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8]$$

GL_RGBA Texture

$$O_1 = [I_1 I_2 I_3 I_4] \quad O_2 = [I_5 I_6 I_7 I_8]$$

Figure 5.5 – Packed Texture Representation in OpenGL

With the packing method shown in Fig. 5.5, the best case for a texture lookup in a three-dimensional texture with a 3x3x3 kernel is nine texture lookups. The worst case with this method is eighteen texture lookups. If the edges are not filtered, 57% of the time it will perform nine texture lookups. The other 42% of the time, it will perform eighteen texture lookups. On average, that is about thirteen texture lookups per voxel. This is less than half of the texture lookups required when not using packing.

A 5x5x5 kernel size will require more texture lookups. If ignoring the edges, it will always require thirty texture lookups to shade a single voxel. This is a great improvement over the required lookups for an unpacked texture, 125 texture lookups.

A 7x7x7 kernel size would be similar to a 3x3x3 kernel in that the number of texture lookups would be variable depending on the location of the fragment in the volume. If ignoring times when the window would overlap the edges, roughly 60% of the time it would require forty-two texture lookups. The other 40% of the time it would perform sixty-three texture lookups. This means that, on average, it would perform fifty-one texture lookup per fragment. This is much better than the 343 lookups required for an unpacked texture. As the size of the window increases, the efficiency of packing becomes more apparent (see Fig. 5.6).

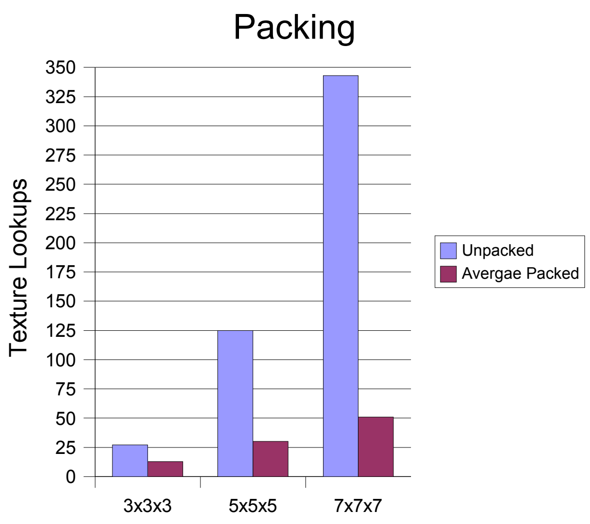


Figure 5.6 – Packing, Window Size Comparison

When such a texture is passed to the fragment program, the fragment program has to unpack the texture, otherwise the texture will be rendered incorrectly. The math involved in unpacking a three-dimensional, packed texture is rather complex. To simplify the process, the coordinates are converted to a one-dimensional representation. Unpacking operations are performed on this value, such as finding neighbors, and the results are converted back to three-dimensions to retrieve the appropriate fragments from the texture. This process could be further simplified by using a one-dimensional texture rather than a three-dimensional texture.

Convert from 3D space to 1D: $\mathrm{i} = x + y*TextureHeight + z*TextureDepth*TextureHeight$

Convert from 1D space to 3D:

$$\mathrm{x} = Index\%TextureWidth$$

$$\mathrm{y} = \mathrm{floor}\left(\left(\frac{Index}{TextureWidth} \right) \%TextureHeight \right)$$

$$\mathrm{z} = \mathrm{floor}\left(\frac{i}{TextureWidth * TextureHeight} \right)$$

Find location of fragment at TextureCoordinate: $TextureLocation = \frac{i}{4.0}$

Extract intensity from four component vector: $TextureComponent = i\%4.0$

Figure 5.7 – Unpacking Operations

Note that unpacking a texture requires the original, unpacked texture dimensions. That would simply be four times the size of each dimension in the packed texture. $TextureLocation$ from Fig. 5.7 returns the location of a voxel containing four packed voxel intensity values. $TextureComponent$ determines which of those four values to use.

Packed 32x32 texture

$$\left[\begin{array}{ccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
7 & T_{56} & T_{57} & T_{58} & T_{59} & T_{60} & T_{61} & T_{62} & T_{63} \\
6 & T_{48} & T_{49} & T_{50} & T_{51} & T_{52} & T_{53} & T_{54} & T_{55} \\
5 & T_{40} & T_{41} & T_{42} & T_{43} & T_{44} & T_{45} & T_{46} & T_{47} \\
4 & T_{32} & T_{33} & T_{34} & T_{35} & T_{36} & T_{37} & T_{38} & T_{39} \\
3 & T_{24} & T_{25} & T_{26} & T_{27} & T_{28} & T_{29} & T_{30} & T_{31} \\
2 & T_{16} & T_{17} & T_{18} & T_{19} & T_{20} & T_{21} & T_{22} & T_{23} \\
1 & T_{08} & T_{09} & T_{10} & T_{11} & T_{12} & T_{13} & T_{14} & T_{15} \\
0 & T_{00} & T_{01} & T_{02} & T_{03} & T_{04} & T_{05} & T_{06} & T_{07}
\end{array}\right]$$

$$T_{26} = \begin{bmatrix} \mathrm{r}=I_{104} \\ \mathrm{g}=I_{105} \\ \mathrm{b}=I_{106} \\ \mathrm{a}=I_{107} \end{bmatrix}$$

Figure 5.8 – Packed Texture

As shown in Fig. 5.8, the packed pixel T_{26} contains four pixels from the original image: I_{104} , I_{105} , I_{106} , and I_{107} . So in a single texture lookup, four intensity values can be retrieved.

Convolution in Multiple Passes

One way to break the instruction limit barrier is to devise a multipass algorithm to filter the volume. Rather than applying the entire convolution kernel to the volume, only parts of it are applied at a time. For each pass, a different part of the kernel is applied to the volume. The results are stored in a separate texture. Finally, after the entire kernel has been applied, the textures are summed together by a separate fragment program for the final image.

$$NumberPasses = \frac{KernelSize}{TextureLookups}$$

$$NumberTemporaryTextures = NumberPasses$$

For a 5x5x5 convolution kernel, which contains 125 elements, 25 elements of the kernel could be passed to the fragment program at at a time. Doing so would require five passes, and five temporary textures. The full neighborhood of voxels would still have to be computed, whether using an unpacked texture model or a packed texture model.

Why only 25 elements at a time? Using only 25 elements would waste three-fourths of the final vector. Why not 32 elements, optimizing for four-component vector operations? If the number of elements of the kernel being passed to the fragment program are not consistent for each pass, a separate fragment program would have to be loaded into the card for the last pass.

For the example in Fig. 5.9, a 5x5 convolution kernel is applied to an image. The kernel is passed five elements at a time, since twenty-five is easily divisible by five. The result is five separate textures. For each pass, the neighborhood of pixels is retrieved, then multiplied by the partial kernel.

$$PartialConvolve(I_{4,4}) = \begin{bmatrix} I_{2,2} I_{2,3} I_{2,4} I_{2,5} I_{2,6} \\ I_{3,2} I_{3,3} I_{3,4} I_{3,5} I_{3,6} \\ I_{4,2} I_{4,3} I_{4,4} I_{4,5} I_{4,6} \\ I_{5,2} I_{5,3} I_{5,4} I_{5,5} I_{5,6} \\ I_{6,2} I_{6,3} I_{6,4} I_{6,5} I_{6,6} \end{bmatrix} \quad K = \begin{bmatrix} K_{1,1} K_{1,2} K_{1,3} K_{1,4} K_{1,5} \\ K_{2,1} K_{2,2} K_{2,3} K_{2,4} K_{2,5} \\ K_{3,1} K_{3,2} K_{3,3} K_{3,4} K_{3,5} \\ K_{4,1} K_{4,2} K_{4,3} K_{4,4} K_{4,5} \\ K_{5,1} K_{5,2} K_{5,3} K_{5,4} K_{5,5} \end{bmatrix}$$

$$Pass\,One \rightarrow K1_1 = [K_{1,1} K_{1,2} K_{1,3} K_{1,4}] \quad K1_2 = [K_{1,5} 0 0 0]$$
$$Pass\,Two \rightarrow K2_1 = [K_{2,1} K_{2,2} K_{2,3} K_{2,4}] \quad K2_2 = [K_{2,5} 0 0 0]$$
$$Pass\,Three \rightarrow K3_1 = [K_{3,1} K_{3,2} K_{3,3} K_{3,4}] \quad K3_2 = [K_{3,5} 0 0 0]$$
$$Pass\,Four \rightarrow K4_1 = [K_{4,1} K_{4,2} K_{4,3} K_{4,4}] \quad K4_2 = [K_{4,5} 0 0 0]$$
$$Pass\,Five \rightarrow K5_1 = [K_{5,1} K_{5,2} K_{5,3} K_{5,4}] \quad K5_2 = [K_{5,5} 0 0 0]$$

$$Vi_1 = \begin{bmatrix} I_{2,2} \\ I_{2,3} \\ I_{2,4} \\ I_{2,5} \end{bmatrix} \quad Vi_2 = \begin{bmatrix} I_{2,6} \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad Vi_3 = \begin{bmatrix} I_{3,2} \\ I_{3,3} \\ I_{3,4} \\ I_{3,5} \end{bmatrix} \quad Vi_4 = \begin{bmatrix} I_{3,6} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$
$$Vi_5 = \begin{bmatrix} I_{4,2} \\ I_{4,3} \\ I_{4,4} \\ I_{4,5} \end{bmatrix} \quad Vi_6 = \begin{bmatrix} I_{4,6} \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad Vi_7 = \begin{bmatrix} I_{5,2} \\ I_{5,3} \\ I_{5,4} \\ I_{5,5} \end{bmatrix} \quad Vi_8 = \begin{bmatrix} I_{5,6} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$
$$Vi_9 = \begin{bmatrix} I_{6,2} \\ I_{6,3} \\ I_{6,4} \\ I_{6,5} \end{bmatrix} \quad Vi_{10} = \begin{bmatrix} I_{6,6} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

First Pass → $PartialConvolve(I_{4,4}) = \sum_{n=1}^{5} Vi_{n*2} * K1_1 + Vi_{n*2+1} * K1_2$

Second Pass → $PartialConvolve(I_{4,4}) = \sum_{n=1}^{5} Vi_{n*2} * K2_1 + Vi_{n*2+1} * K2_2$

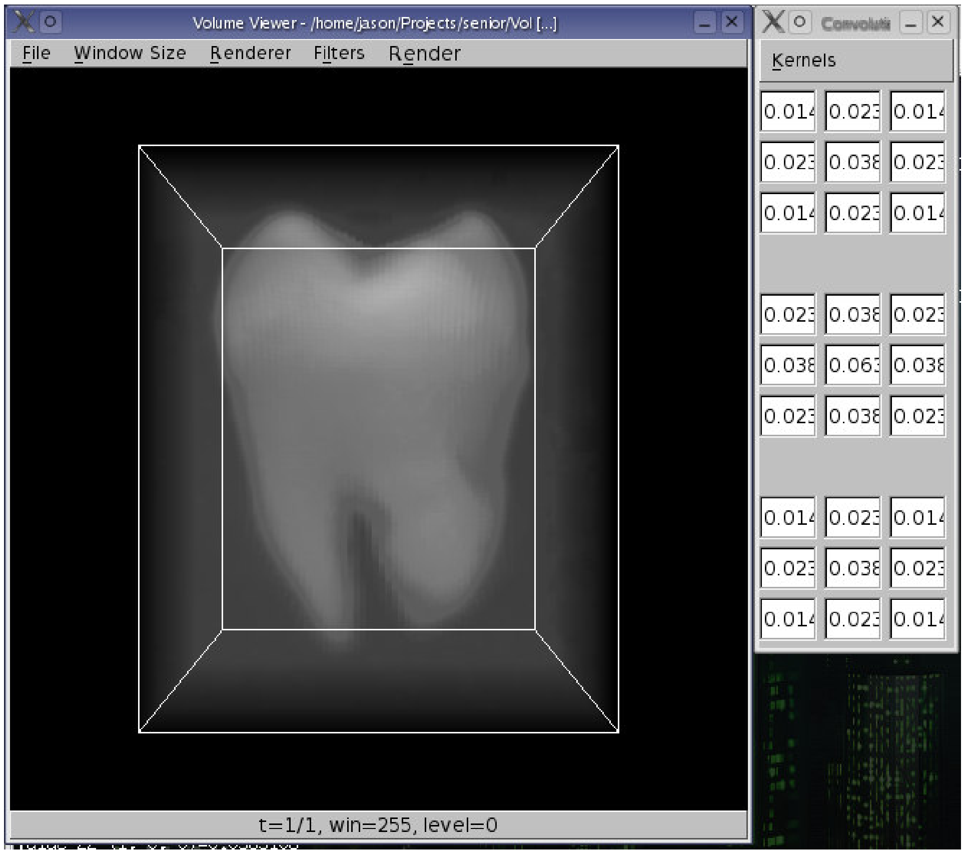
and so on

Figure 5.9 – Multi-pass Convolution (5x5 represented)

5.5 Performance Comparison

The following tests were performed on an Athlon 1800+ XP with 512mb of

RAM and with a NVIDIA GeForce 5200FX, which has a NV30 GPU, using the latest NVIDIA drivers available, 1.0-5336. The operating system was Gentoo Linux with the 2.6.5 kernel.



0.014	0.023	0.014
0.023	0.038	0.023
0.014	0.023	0.014

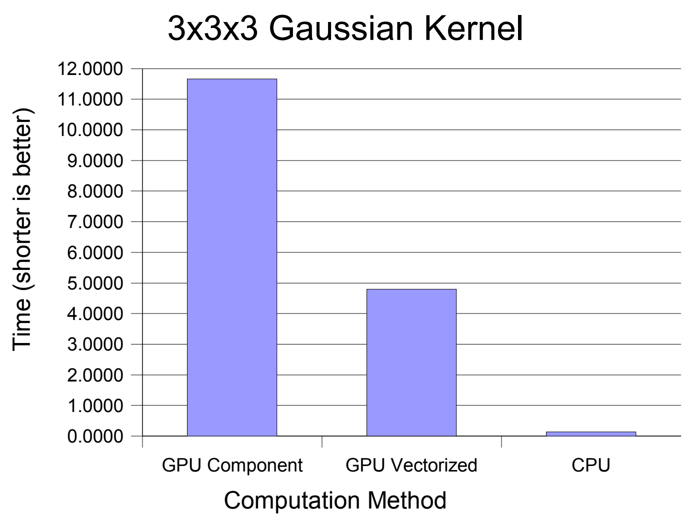
0.023	0.038	0.023
0.038	0.063	0.038
0.023	0.038	0.023

0.014	0.023	0.014
0.023	0.038	0.023
0.014	0.023	0.014

```
t=1/1, win=255, level=0
```

Figure 5.10 – Tooth Dataset Used for Testing

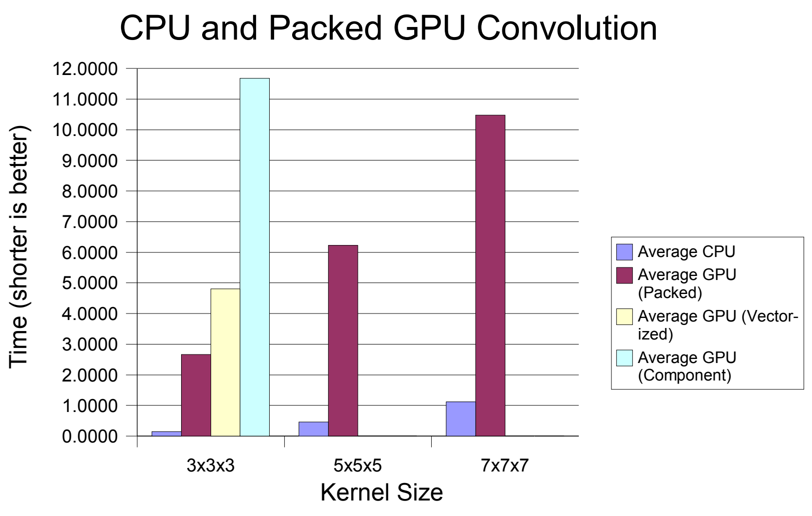
The dataset used for testing was a tooth stored as a list of intensity values. The size of the volume was 55x50x81 voxels. All times are measured in seconds using the timer described in chapter three.



Computation Method	First Run	Second Run	Third Run	Average
GPU Component	11.6633	11.6630	11.7516	11.66
GPU Vectorized	4.7891	4.8056	4.8108	4.8
CPU	0.1399	0.1396	0.1520	0.14

Figure 5.11 – 3x3x3 Gaussian Kernel Performance

Figure 5.11 shows two fragment programs, "ConvolutionFilter3" and "ConvolutionFilter3v", as well as the CPU fragment operation. Vectorizing operations in the GPU greatly increases performance. Performance increased by nearly 60 percent in this example. However, this is not enough to approach the speed of the CPU, which is 98% faster than the vectorized fragment program.



Kernel Size	Average CPU	Average GPU (Packed)
3x3x3	0.1438	2.6597
5x5x5	0.4652	6.2273
7x7x7	1.1162	10.4762

Figure 5.12 – Packed Texture Performance

Fig. 5.12 shows how a packed-texture convolution method would perform. For each value, the program was run three times with the results averaged. These results were obtained by a fragment program which performs the average number of texture lookups required for a packed texture given a specified window size. With a 3x3x3 window size, packing cuts the length of time to process a volume by about half the time compared to the vectorized convolution method. It is, however, still not close to the speed of the CPU method.

Chapter 6 - Future

6.1 GPU Limitations

Currently much of the difficulty is due to the limited feature set of both the GPU and the Cg language. The latest NV30 GPU from NVIDIA supports fragment programs up to 1024 instructions. This limits the ability to execute complex programs on the GPU. The NV30 also does not support loops in fragment programs. Cg works around this by unrolling loops at compile time. This increases the size of programs, and such a loop's length must be able to be determined at compile time.

Conditional operations are allowed in fragment programs, but are very limited. The fragment program is executed for both possible results of an “if” statement, for example. Such limitations greatly decrease the speed of fragment programs; the use of branching operations in fragment operations is undesirable.

Another area lacking in current GPUs is the ability to temporarily store data between iterations of a fragment program. There is no direct way to pass data between fragments. The best way to work around this limitation is the use of multi-pass methods, storing partial results in an external buffer. This requires more complex CPU programs, and more memory bandwidth is used.

Cg itself is a relatively new language, and is currently in development. The development tools and the language itself are still rapidly evolving.

6.2 Potential Optimizations

The main reason the GPU convolution programs are so slow is due to the large number of texture lookups. The GPU is optimized for reading one fragment at a time, in order, but modern GPUs such as the NV30 allow random texture access. This is why it is possible to perform convolution in a single pass at all.

Various multi-pass methods may increase the speed of the program, such as was discussed in chapter 5.3.4. For example, performing each pass with only one value of the convolution kernel, generating a texture for each value in the convolution kernel. If the number of temporary textures is too large to fit all at once in the GPU, a series of calls of a texture summation fragment program could be executed in a tree-like manner. The resulting convolution function would be very simple, comprising of a single texture lookup and a multiplication. Managing the large number of temporary textures in an efficient manner would require the most attention.

6.3 Future GPUs

Consumer graphics cards are in a period of rapid advancement. The concept of the ability to modify the graphics pipeline in the card is a new one to commodity graphics cards. The feature set in such hardware is changing rapidly as developers express demands for new functionality in the hardware.

Cards such as the recently announced NVIDIA GeForce 6800 should solve many of the problems one is faced with performing computation on a GPU.

The GeForce 6800 will have a NV40 GPU, which supports more operations, contains more pipelines, and has a larger instruction cache [11].

The NV40 will support loops in fragment programs thanks to a new loop count register, and will support dynamically indexing any element in an array [11]. This should remove the limitation in a program such as “ConvolutionFilter3,” which requires recompilation for different window sizes. Improvements in dynamic flow control should allow a program such as the median filter in “GPU Filters” to execute much more quickly.

Chapter 7 - Conclusion

“GPU Filters” and “Volume Viewer Filter” demonstrate that it is possible to perform complex operations on a GPU, however special considerations must be made due to the specialized nature of a GPU and limitations in the Cg language.

Even with the limitations of the current generation of Cg and GPUs, this area of computer graphics is showing much potential. New GPUs on the horizon will allow more traditional general-purpose techniques to be applied to GPU applications, such as looping and branching. This will allow complex mathematical operations to be applied to large datasets in real-time.

Appendix A - Cg Program Examples

```
void Grayscale(float2 TexCoord : TEXCOORD0,
               uniform sampler2D decal,
               out float4 oColor : COLOR)
{
   float4 value4=tex2D(decal, TexCoord);    //Retrieve pixel
   float value=(value4.r+value4.g+value4.b)/3.0;
   oColor=float4(value, value, value, value4.a);
}
```

Figure A.1 – 2D Grayscale Filter

Fig. A.1 shows a very simple two-dimensional filter which retrives the color value from the texture at the current fragment, average the red, green, and blue channels together, and places that value in the “COLOR” semantic. This is where the card determines the color for the fragment at the location stored in the TEXCOORD semantic.

```
void get_fragments(uniform sampler2D decal,       //texture
                   float2 TexCoord,      //central pixel location
                   float2 pixelsize,     //pixel skip
                   out float4 colors[9])//output, surrounding pixels
{
   float lowerboundx=TexCoord[0]-pixelsize[0];
   float lowerboundy=TexCoord[1]-pixelsize[1];
   float upperboundx=TexCoord[0]+pixelsize[0];
   float upperboundy=TexCoord[1]+pixelsize[1];
   colors[0]=tex2D(decal, float2(lowerboundx, upperboundy));
   colors[1]=tex2D(decal, float2(TexCoord[0], upperboundy));
   colors[2]=tex2D(decal, float2(upperboundx, upperboundy));
   colors[3]=tex2D(decal, float2(lowerboundx, TexCoord[1]));
   colors[4]=tex2D(decal, TexCoord);
   colors[5]=tex2D(decal, float2(upperboundx, TexCoord[1]));
   colors[6]=tex2D(decal, float2(lowerboundx, lowerboundy));
   colors[7]=tex2D(decal, float2(TexCoord[0], lowerboundy));
   colors[8]=tex2D(decal, float2(upperboundx, lowerboundy));
}
```

Figure A.2 – “get_fragments” Support Function

The function in Fig. A.2 is used by most of the filtering operations in the

“GPU Filters” program. This function takes the location of the current fragment and returns the 3x3 neighborhood.

```
float get_fragments3op(//grabs all of the pixels needed for convolution
                  uniform sampler3D decal,    //texture
                  float3 TexCoord,            //central pixel
                  float3 pixelsize,           //pixel skip size
                  float matrix[WINDOW_ARRAY_SIZE])
{
   int color_count=0;
   float output=0;
   for(int cx=0-WINDOW_OFFSET_SIZE;cx<=0+WINDOW_OFFSET_SIZE;cx++)
   for(int cy=0-WINDOW_OFFSET_SIZE;cy<=0+WINDOW_OFFSET_SIZE;cy++)
   for(int cz=0-WINDOW_OFFSET_SIZE;cz<=0+WINDOW_OFFSET_SIZE;cz++)
   {
      float3 coordinates=float3( TexCoord[0]+cx*pixelsize[0],
                                 TexCoord[1]+cy*pixelsize[1],
                                 TexCoord[2]+cz*pixelsize[2]);

      output+=tex3D(decal, coordinates).r*matrix[color_count];
      color_count++;
   }
    return(output);
}
void ConvolutionFilter3(   float3 TexCoord : TEXCOORD0,
                     float4 color : COLOR,
                     uniform sampler3D decal,   //image to filter
                     uniform float3 PixelSize,   //Pixel skip size
                     uniform float ConvolutionMatrix[WINDOW_ARRAY_SIZE],

                     out float4 oColor :COLOR0
                     )
{
   float intensity=get_fragments3op(decal,
                                    TexCoord,
                                    PixelSize,
                                    ConvolutionMatrix);
   oColor=float4(intensity, intensity, intensity, intensity * 0.1);
}
```

Figure A.3 – “ConvolutionFilter3” Per-Component

Fig. A.3 is the source for the slow “ConvolutionFilter3” program. For this program to compile, three preprocessor values must be set. For a NxNxN kernel, “WINDOW_SIZE” is “N.” “WINDOW_ARRAY_SIZE” which is of size N^3 is the size of the one-dimensional array needed to store the convolution matrix and the number of iterations needed to process all of the pixels in the window. "WINDOW_OFFSET_SIZE" is the length from the center of the window to one of the edges. The basic formula to find this is $floor\left(\frac{\text{WINDOW\_SIZE}}{2.0}\right)$.

For example, to build a 3x3x3 fragment program, the following parameters would be passed to the compiler: WINDOW_SIZE=3 WINDOW_ARRAY_SIZE=27 WINDOW_OFFSET_SIZE=1.

Appendix B - References

[1] "Graphics Processing Unit." Wikipedia. 30 March 2004. 12 April 2004. <http://en.wikipedia.org/wiki/Graphics_processing_unit>

[2] Kilgard, Mark. Cg in Two Pages. 16 January 2003. 12 April 2004. <http://developer.nvidia.com/attach/6043>

[3] Macedonia, Michael. The GPU Enters Computing's Mainstream. October 2003. 12 April 2004. <http://www.computer.org/computer/homepage/1003/entertainment/>

[4] Kilgard, M. & Fernando, R. (2003) *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley.

[5] Cg Downloads. 8 April 2004. 13 April 2004. <http://developer.nvidia.com/object/cg_toolkit.html>

[6] Schroder, W. ; Martin, K. ; & Lorensen, B. (2002) *The Visualization Toolkit: 3rd Edition.* Kitware, Inc.: United States.

[7] "Convolution." Wikipedia. 20 March 2004. 17 April 2004. <http://en.wikipedia.org/wiki/Convolution>

[8] Fisher, Perkins, Walker, & Wolfart. "Convolution." Hypermedia Image Processing Reference. 17 April 2004. <http://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm>

[9] Fisher, Perkins, Walker, & Wolfart. "Gaussian Smoothing." Hypermedia Image Processing Reference. 17 April 2004. <http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm>

[10] Shreiner, Woo, Neider, Davis. (2004) *The OpenGL Programming Guide, Fourth Edition.* Addison-Wesley.

[11] Weinland, L. “NVIDIA GeForce 6800 Ultra Performance Leap.” Tom's Hardware Guide. 14 April 2004. 25 April 2004. <http://www.tomshardware.com/graphic/20040414/geforce_6800-01.html>

[12] Mark, Glanville, Akeley, & Kilgard. “Cg: A system for programming graphics hardware in a C-like language.” SIGGRAPH 2003.