AN ABSTRACT OF THE THESIS OF

Ankit Khare for the degree of Master of Science in Computer Science presented on February 13, 2006.

Title: Volume Analysis and Visualization

Abstract approved: ______________________________

Mike Bailey

3D datasets acquire great importance in the context of medical imaging. In this thesis we survey and enhance solutions to problems inherently associated with 3D datasets-processing time,noise and visualization. Efforts include development of a tool kit to provide a multi-threaded processing platform to cut processing time, produce real time visualization and the use of the Graphics Processing Unit as a general purpose computing device.

Volume Analysis and Visualization

by

Ankit Khare

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented February 13, 2006
Commencement June 2007

Master of Science thesis of Ankit Khare presented on February 13, 2006.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

Ankit Khare, Author

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my academic and research advisor, Dr. Mike Bailey, for his encouragement, motivation, exceptional guidance and support in helping me to complete this work.
I would like to thank the faculty and my peers here at EECS department, Oregon State University for providing a wonderful learning yet competitive environment for graduate studies.
I would also like to thank Krishnan Kolazhi, Rohit Kamath, Roshan Urval, Santosh Tiwari, Anand Venkataraman, and Nilesh Araligidad for being such supportive yet critical peers during my term as a graduate student at Oregon State University.
Finally, I would also like to thank my other committee members Dr Ron Metoyer, Dr Michael J. Quinn and Dr Michael K. Gross for their time and inputs.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Chapter 1 – Introduction

Extracting relevant information from 3D datasets is an essential goal in scientific visualization. Another important aspect is creating tunable parameters and the ability to manipulate them in a user-friendly environment.

This project describes the creation of a novel data-flow volume *filtering workbench* that provides a user-friendly, yet sophisticated, interface and feature set. Filters themselves are pluggable modules and are developed independently of the main application. We also have been experimenting with moving volume filtering operations from the CPU to the faster GPU to enhance user interaction.

1.1 The Problem

Volume datasets, such as MRI and CAT scan datasets, have problems which are inherent to the nature of data acquisition.

1.1.1 Noise

The finite sampling rate of even the most sophisticated devices causes undesired artifacts in the gathered datasets. Thus a verbatim representation of the object in question is impossible, however the noise in the final dataset can be reduced by post-processing to reveal a structure which is closer to the continuous object in question. In the context of digital image processing, the term noise usually refers to high frequency random perturbations of sample values close to one pixel. There are other artifacts of similar appearance which are referred to with different terms to underline their origin. Some of the most commonly occurring types of noise are described below.

White Noise White noise a is completely random signal containing random combination of all possible frequencies.

Gaussian Noise Gaussian noise is essentially white noise with probability distribution equivalent to that of Gaussian distribution.

Impulse or Shot Noise This is essentially random and has potentially large variation between adjacent values.

Periodic Noise Periodic noise is similar to white noise however it has some repetitions in it and is perhaps the easiest to counter. [29].

1.1.2 Visualization

Visualization of 3D datasets poses a challenge because of the amount of information which needs to be conveyed to the viewer. The idea of a volume is somewhat different from the real world scenario where we only notice the silhouettes of a 3D object. To convey information which is embedded within the 3D data, techniques involving transparency and transfer functions are used. These are dealt with in more detail in a later chapter.

1.1.3 Feature Enhancements

An important subset of visualization is to identify relevant features. Features in a volume are domain specific, enhancing them may include making boundary lines more prominent or focussing on a subset of the dataset. By domain specific, we mean the origin of the dataset, e.g medical or scientific domain.

Techniques include manipulating transfer functions and extensions of noise removal techniques.

1.1.4 Performance

3D datasets are inherently large and getting bigger with improvements in scanning devices and compute engines, hence, any kind of processing needs a lot of computing power. This thesis also delves into techniques into transforming the Graphics Processing Unit into a general purpose computing device to speed up the processing.

Chapter 2 – Volume filtering

Filtering essentially encompasses suppressing high frequency or low frequency components in the data set. High frequency components include noise e.g. white noise. Suppressing such components makes the image *smooth*. Suppressing low frequency components enhances edges or boundaries in the image.

2.1 Averaging or Mean Filter

Averaging filter is one the most simplest filters. The basic idea is to reduce the intensity variation across the data set, This is achieved by replacing the voxel with average value of the surrounding voxels.

2.2 Gaussian Filtering

Gaussian filtering is similar to mean filtering - functioning as a low pass filter. However, it uses a different kernel based on the Gaussian distribution.

3D convolution for Gaussian filtering is also separable into x, y and z components. Thus the 3-D convolution can be performed by first convolving with a 2-D Gaussian in the x-z direction and finally convolving in the z direction. 1-D Gaussian can be represented by the following distribution, where σ and μ control the width of the Gaussian distribution. However for purposes of this thesis, a 3D

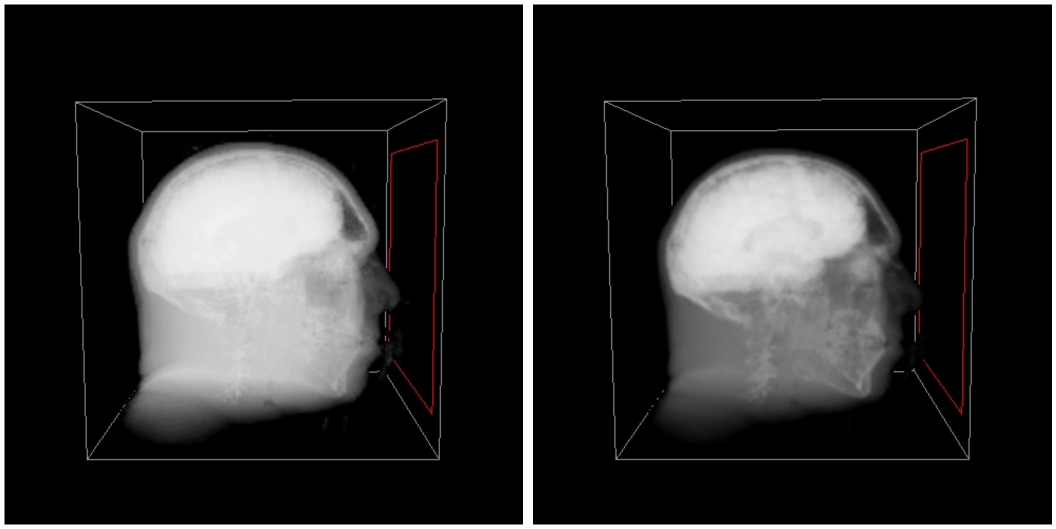


Figure 2.1: Left: Original, Right: After mean filtering

kernel was directly used.

$$G(x) = \frac{1}{\sqrt{2\Pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

2.3 Median filter

The Median filter is a *nonlinear filter* which works similarly to the averaging filter however, instead of replacing the voxel value with the mean of the surrounding values, it replaces the voxel value with the *median* of the surrounding voxel values. There are multiple advantages of the median filter, elucidated in the Hypermedia

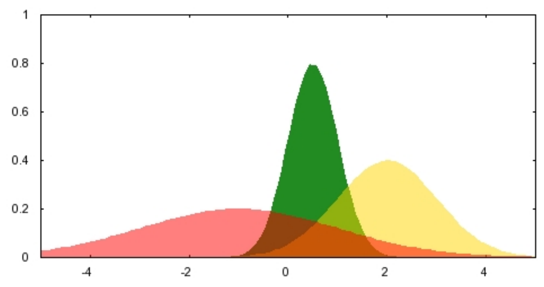


Figure 2.2: This figure shows different gaussian distributions

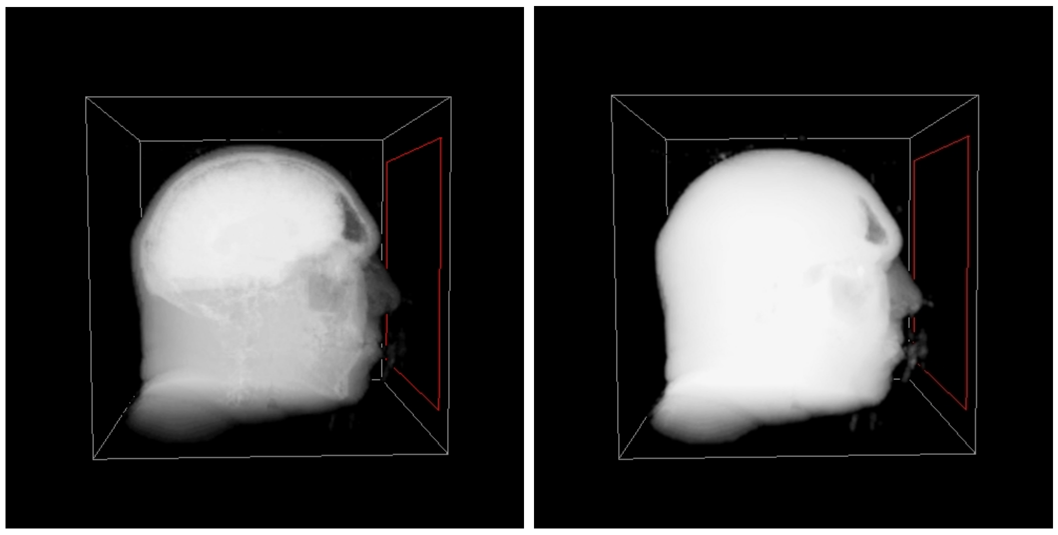


Figure 2.3: Left, Original, Right: After gaussian filtering

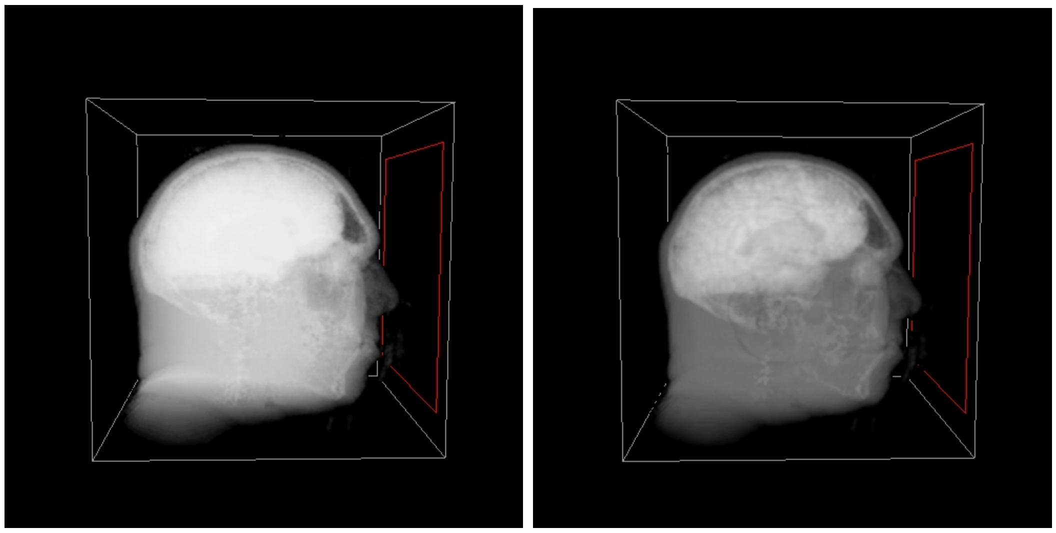


Figure 2.4: Left: original, Right: After median filtering

Image Processing Reference[12] are as follows,

1. Median is a better statistic than mean or average as it's not skewed by extreme values.

2. Median is part of the dataset whereas mean is not necessarily present in the dataset, hence median preserves the sanity of a dataset in some respects. This includes preserving sharp edges or sudden changes in values of a dataset.

However these advantages also mean that this filter is very slow because of the sorting of surrounding values at every step. Fast implementation using the GPU is discussed later.

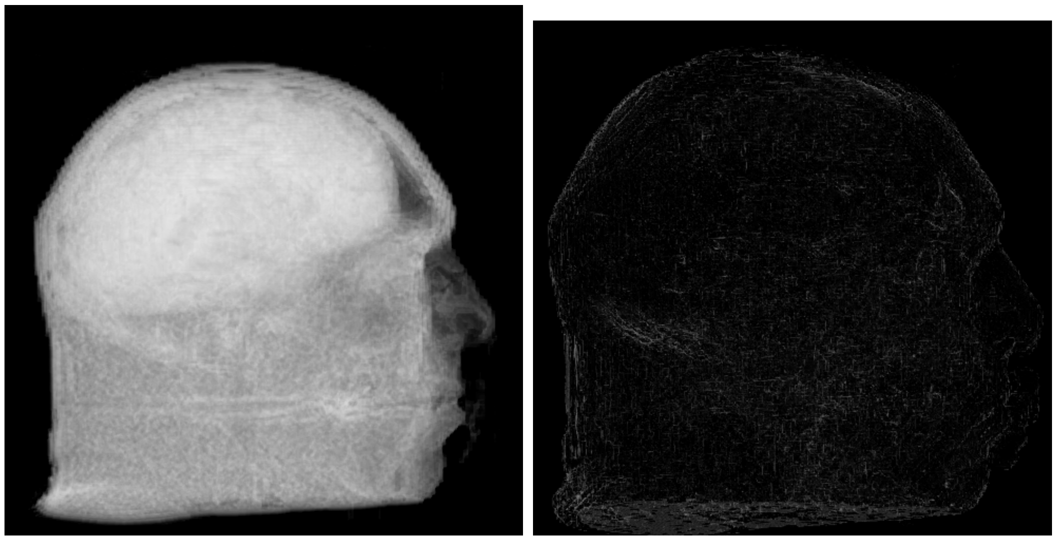


Figure 2.5: Left: Original, Right: After edge detection with using sobel filter, visualization is done using OSU's Volume Explorer to do a more accurate rendering of the fine boundaries created by sobel filtering

2.4 Sobel Filtering

Sobel Filtering allows us to provide an approximation of the gradient of the image. This is especially useful for accentuating different features in the datasets.This gradient, which is a 3D vector, is given by derivatives in all three directions. These derivatives are then approximated by finite differences. Each component represents the rate of change of values in each direction.

This implies that the result of Sobel filtering will accentuate sudden changes in sample values in any direction and suppress regions of constant values. We use a 3x3x3 kernel for our purposes.

2.5 Frequency filtering

2.5.1 Fourier Transform

Fourier transform is used to transform time or spatial domain continuous function to frequency domain. This process breaks down a function into the sum of sinusoidal functions each representing a particular frequency. Discrete Fourier transform(DFT), is user for discrete functions, eg a 3D image or digitized audio.

DFT works on sampled values and hence cannot fully reproduce all the data however it is sufficient to analyse and filter out frequencies present in the dataset.

For a cube image of size NxNxN, the 3-dimensional DFT is given by:

$$F(x,y,z) = \frac{1}{N^3} \sum \sum \sum f(i,j,k) e^{-i2\pi(\frac{xi}{N}+\frac{yj}{N}+\frac{zk}{N})}$$

In a similar way. The inverse Fourier transform is given by [12]:

$$F(x,y,z) = \frac{1}{N^3} \sum \sum \sum f(i,j,k) e^{i2\pi(\frac{xi}{N}+\frac{yj}{N}+\frac{zk}{N})}$$

2.5.2 Filtering

Convolution in the spatial domain is same as multiplication in frequency domain, and hence in theory all frequency filters can be implemented in spatial domain. However, as we only consider a finite convolution kernel, this filtering can only be approximated by spatial domain filter. One thing to note is that, it would be efficient to implement small convolution kernels in the spatial domain, rather than implementing them a frequency filter for bigger convolution kernel it would be more efficient to use frequency space filtering($O(n^2)$ Vs $O(n \lg n)$) Even in frequency space filtering there is approximation as we use finite sampling.

Frequency filters can essentially be divided into three category, low pass, high pass and band pass - each satisfying some criteria. As we have already defined noise as high frequency components, a low pass filter will attenuate noise and a high pass filter will accentuate sharp edges or features in a dataset. A band pass filter will more selectively highlight certain frequencies (which may represent certain sections in a dataset).

Another type, band reject, is useful for eliminating artificial frequencies which might contaminate a dataset, e.g. 60Hz from line voltage and/or interference other electronic sources. Filters may also be of higher orders - which can help to manipulate the slope of curve near the cut-off frequencies and can prevent effects like ringing in the final result[12].

2.6 Combining filters to achieve better results

All the above filters have one particular advantage and can be used in conjunction to achieve a particular visual output. This is specially useful when outputs of various filters can be linearly combined. Results found by mixing the filters have been arguably better and are discussed in the Results chapter.

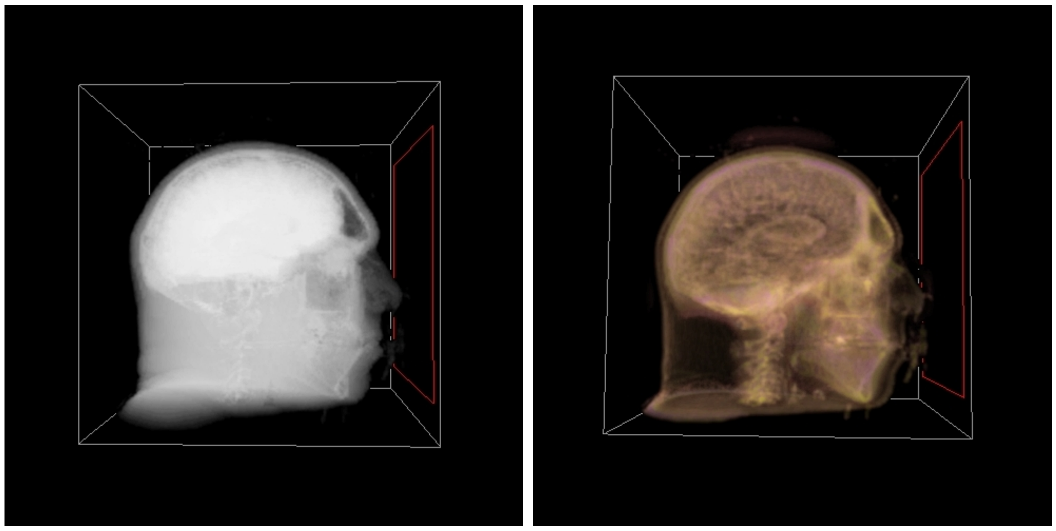


Figure 2.6: Left: Original, Right: After a High pass filtering

Chapter 3 – Volume Visualization

Volume visualization is the process of creating images from multidimensional scalar or vector data grids. This process generally involves projection of 3D datasets onto a 2D image plane to gain understanding of the structure contained within the data. Most techniques are applicable to a uniformly sampled 3D such as obtained MRI and ultrasound.

3.1 Techinques

Volume visualization techiniques can be divided in two types.

- Surface Fitting Algorithms
- Direct Volume Rendering.

Surface Fitting Algorithms include isosurface generation using marching cubes [7] and contour tracking, Direct Volume Rendering includes methods like splatting [17] and Ray Tracing. As part of this thesis, two methods were explored for visualization: Terarecon's realtime ray tracing system and a 3D texture based, view aligned plane method[5]. Both these systems allow realtime visualization. OSU's Volume Explorer based on Terarecon's system [31] extensively used as an application to analyze volumes.

3.2 3D Texture Based Volume Visualization

Many 3D graphics systems use texture mapping to apply images, or textures, to geometric objects. Commodity PC graphics cards are fast at texturing and can efficiently render slices of a 3D volume, with realtime interaction capabilities. There are two types of 3D texture based volume visualizations, view aligned and object aligned. In view aligned each slice is drawn perpendicular to the view vector and in object aligned the slices are constant with respect to objects orientation.

One major advantage of using 3D texture support of current graphics cards is the hardware interpolation of data points in the 3D dataset (*trilinear interpolation*), without any extra code. A disadvantage of such a system is that shading cannot be done on the fly, the input dataset need to be preprocessed, This removes the possibility of realtime changes in lighting conditions.

Hardware support for 3D textures allows the use of view-aligned slices. The slices are always drawn parallel to the viewing plane, eliminating the popping when moving from different axes as seen in the object/volume aligned texturing. This done by drawing quadrilaterals (known as proxy geometry) aligned with the user's view and using texture matrix operations to rotate the volume texture. A view aligned approach was used in this implementation.

3.3 Implementation

In this project, the implementation was done using OpenGL [3] and C++. OpenGL capabilities for texture coordinate generation and clipping planes were used extensively to provide realtime cropping of volume datasets. The Terarecon VOX data format was used as input data format. To provide cross-platform and easy access to OpenGL extensions, GLEW [22] an abstraction layer over OpenGL Extensions was also used.

The rendering of the texture-plane based process is.

1. Setup the clip planes transform matrices.
2. Setup the texgen planes transform matrices.
3. Setup the texture coordinate generation using *glTexGeni.* - in eye Linear mode.
4. Configure the clip planes using *glClipPlane.*
5. Enable them using *glEnable.*
6. Enable the alpha blending/testing using *glEnable*(GL_ALPHA_TEST) and *glAlphaFunc.*
7. Setup the blending function using *glBlendFunc.*
8. Setup the texture matrix.
9. Set the ModelView matrix to identity.
10. Set the texture coordinate generation using *glTexGenfv.*
11. Draw the slices.

12. Disable the clip planes.
13. Draw the slice plane across the volume to get a better image.
14. Draw the box framing everything.

3.4 Image processing

3.4.1 Transfer functions

Transfer functions form an integral part of any volume rendering. These functions transform scalar data values into (RGBA) optical properties. Volume Explorer allowed us to examine the volume dataset with different standard and handcrafted transfer functions.

3.4.2 Image processing Filters

One the framework for rendering volumes is ready, various image processing filters can be applied. These include:

- Colour Adjustment.
- Brightness control.
- Saturation control.
- Hue control.

- Cropping.
- Equalization.

3.4.3 Experiments with Non-Photorealistic rendering

NPR is sometimes of great help in visualizing the overall structure of 3D datasets. Experiments were done by clamping scalar voxel values to specific values and results obtained were arguably pleasing. Figure 7.4 shows one of the images obtained.

Chapter 4 – CPU Based Approach

This chapter presents one of the most important aspects of this project, a plugin-based architecture for analyzing and processing 3D datasets. During the initial phases, requirements for this toolkit were laid out - a user friendly interface, cross platform and efficient processsing. Taking cues from previous efforts like OpenDX and Vtk [11][21], a data flow paradigm was decided upon.

To provide a consistent user interface and cross platform functionality, the Qt library was chosen. Various other cross platform GUI toolkits like FLTK and TK were also investigated, but were not taken into consideration because of various reasons - complexity and lack of documentation being the most important. Qt provided a host of other APIs for multithreading and dynamic library loading which were integral to this application. The availability of Qt under Gnu Public License was also an important factor in using the library.

This application provides a framework for processing element (plugins). These plugins are developed independently of the application and are loaded into memory on startup. The design is such that everything from loading to display is handled by plugins.

4.1 User Interface

The user is presented with an empty screen with floating widgets, each representing plugins or processing elements. These widgets can then be arranged in the desired order with drag and drop operations. The data flow is then indicated by drawing flow-lines. Individual parameters for each plugin (represented by widgets) can be then set using right click operations. Default parameters can also be set separately as XML files.

Once the data dependency and hence the execution order is provided to the application using the flowlines, presence for loops is tested to avoid any circular dependenies using depth first search.

Once the appropriate flow-graph has been constructed, it can be saved as an XML file. Such an XML file can also be loaded into the application.

4.2 Plugins

Leveraging Object Oriented concepts, each plugin derives the same *Base* class which is visible to the application. The child class implements functions which provide inputs to the plugin and actual execution and output after processing. The application then handles the flow of data from one plug-in to another. These separate pieces of code are compiled as libraries and are linked dynamically with the main application during startup. Each plug-in maintains its own copy of data or state,which can grabbed at any stage during execution. The following plug-ins were implemented and loaded by default.

Load loads a .vox format volume dataset

Save Saves the input to this plugins on the disk as .vox formate volume dataset

Sobel Performs Sobel Filtering on the data set.

Median Performs Median filtering on the data set

Mean Perform mean filtering on the data set

Gaussian Performs gaussing filtering on the data set.

FFT Performs a band pass filtering on the data set.

Mixer Outputs a linear combination of the 2 inputs provided.

4.3 Execution Management

4.3.1 Background

Static scheduling of a program represented by a directed task graph on a multiprocessor system to minimize the program completion time is a well-known problem in parallel processing[34]. Finding an optimal schedule is an NP complete problem [33].Many Heuristic algorithms have been provided that provided satisfactory results. What is presented below is a greedy approach, which tries to minimize the

the completion time and solves the problem sufficiently at hand. The complexity of the algorithm is

$$O(\alpha^2 n)$$

where n is the number of nodes and α is the branching factor.

4.3.2 Description

To provide efficient plugin execution a multithreaded approach is used. Whenever it's possible to execute two or more plugins in parallel i.e., they are not dependent on each other, appropriate actions are taken such that they are executed independently. This is achieved by maintaining two separate Directed Acyclic Graphs, dependency and child graphs. These two could have been merged together in a single graph, however, to maintain algorithmic and implementation clarity two separate instances were used. Moreover using two separate graphs did not affect the runtime complexity.

At first, plugins which have no dependencies are executed as threads. On completion of execution, each thread generates an event. This event is then caught and each of its children (using the child graph) are checked for possible execution. These children may have other parents which might not yet have been executed or are still running. For a particular node, if all of its dependencies are satisfied, the corresponding thread is allowed to run. This process continues until all nodes have completed their execution. As a visual cue, the corresponding widget changes its color signifying its current state - execution complete, currently executing or yet to be executed.

The above procedure guarantees that execution order of each plugin is optimal in terms of parallelism. After execution, changes in the flowgraph can be accommodated if needed. As with any multithreaded system, problems of synchronization come up. Each plugin maintains its own copy of data which is then fed as input to its children. This effectively eliminates any contentions between multiple writers as children can only read the data and not write into it.

4.4 Implementation

Each of the nodes inherit from the QtThread class and has a data member which identifies the dynamically loaded library associated with the given node. Once the node has been identified as runnable, an object of the processing filter is created by a factory method in the DLL and passed on to the main application. The application then passes in a pointer, which is pointing to the output data from previous computation, to this object. Internally the library object uses dynamic_cast to check whether the data pointer which is being passed into has the right type. The data is then processed and a new output copy is created. Once this processing finishes, an event is generated which is caught by the main application. The output pointer is then grabbed from the object for further processing.

4.5 Typical Usage

At startup, the application loads all the plugins available in the plugin directory.

The user is presented with an empty drawing area. The user then right click and selects the appropriate plugin from the list. Generally the first node is the Load plugin. To configure this Load plugin, the user right clicks on the Load node (which is now available on the drawing area. An editable dialog box pops up with configuration file of the Load plugin. This is then edited to provide the name of the file to be loaded.

Next, a desired plugin eg sobel plugin is selected like above. The user can now connect the Load plugin and sobel plugin, via flow lines. For connecting the flow lines the user middle clicks the source node and then the target node. A dialog asks to which input the flow line needs to be connected. The user inputs the input number and accordingly updates the configuration file. This is necessary as these plugins can take any-number of inputs and have only one output. Finally the last node, which is generally the Save node is added in a similar fashion. Different Options are shown in figures 4.1 to 4.4.

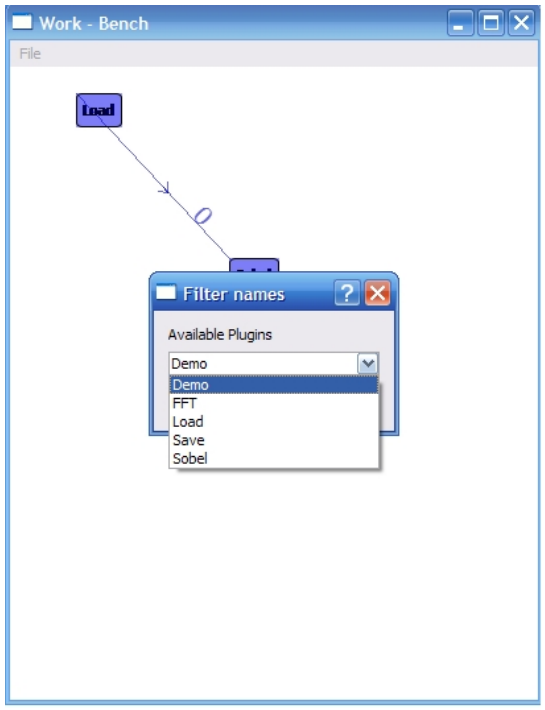
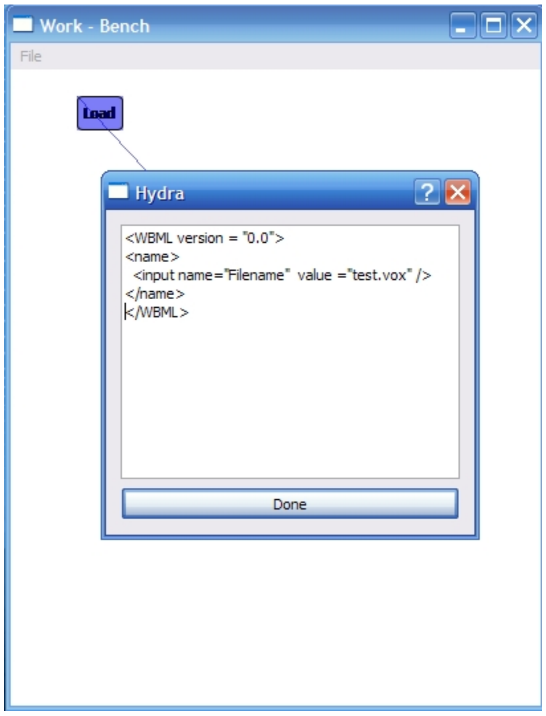


Figure 4.1: This drop down menu appears when the user right clicks in the empty drawing area



```
<WBML version = "0.0">
<name>
 <input name="Filename"  value ="test.vox" />
</name>
</WBML>
```

Figure 4.2: The configuration dialog appears when the user right clicks on an existing node

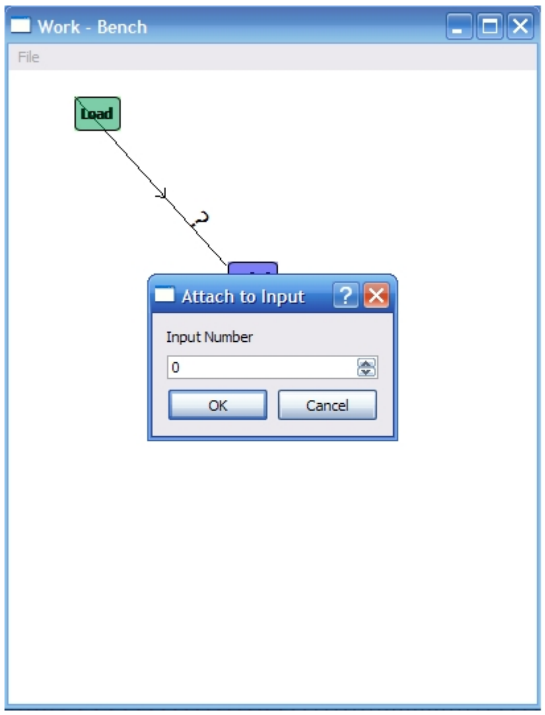


Figure 4.3: This dialog appears when the user connects two nodes

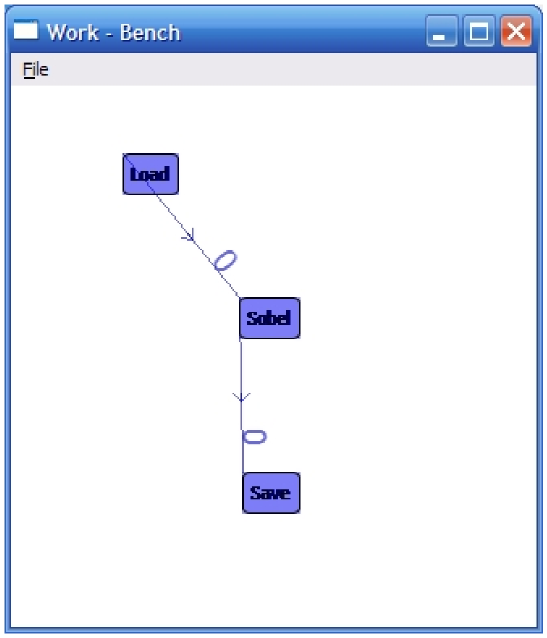


Figure 4.4: The final connected flow diagram

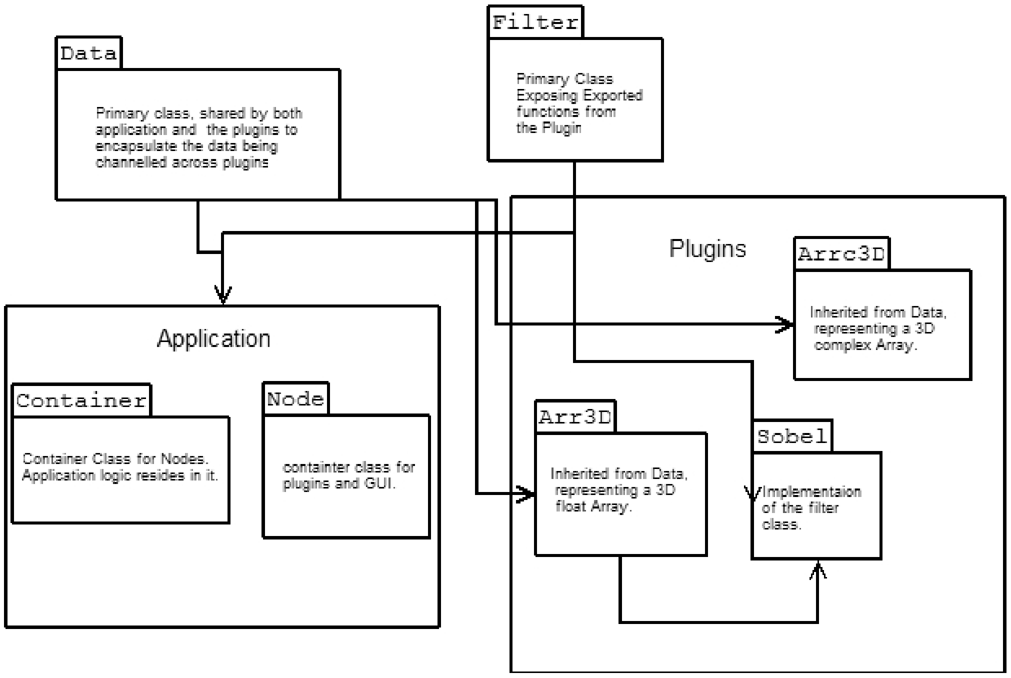

Data
Primary class, shared by both application and the plugins to encapsulate the data being channelled across plugins
Filter
Primary Class Exposing Exported functions from the Plugin
Plugins
Arrc3D
Inherited from Data, representing a 3D complex Array.
Application
Container
Container Class for Nodes. Application logic resides in it.
Node
containter class for plugins and GUI.
Arr3D
Inherited from Data, representing a 3D float Array.
Sobel
Implementaion of the filter class.

Figure 4.5: various software components in the application

4.6 Software Engineering Aspects

The choice of GUI toolkit Qt and Programming language C++ dictated the choice of object oriented paradigm while constructing this application. The design decision of the ability to do development of plug-ins independent of the application was achieved using a set of classes related to each other in the inheritance tree while the application just had the knowledge of the root class.

Application The main application is composed of two parts, The main container and the nodes.

- **container** The class contains the application logic and acts as a container for all the nodes in the application. The graph data structure resides in this class. Input from the GUI is handled in this class. The model-view-controller methodology separates the logic and GUI handling. This class understands the data type and provides nodes with access to it.
- **Nodes** This class acts as a building block. In this, objects of the plug-in class are instantiated using a class factory provided in the Filter class. This class also invokes configuration mechanism of the filter class when directed by the user through the UI.

Plugins This component inherits from the filter class and implements the actual image processing algorithms. It understands custom data types inherited from Data and operated on them. e.g Custom data types including arrays of 3D floats or array of 3D complex numbers. This component is separately build as dynamically loaded library and the required functions are exported so that they are visible from the Application.

3rd Party Components Various freely available components were used.

fftw,fftw+.h[18] fftw or *fastest fourier transform in the west* is an highly optimized library for doing FFT transforms. During early stages of this thesis, a FFT implementation was handcoded but fftw was significantly faster. fftw++.h provides an easy C++ abstraction over this C++ library.

Qt[19] is a cross platform GUI library which also provides features like multi- threading in a platform independent manner. All functions including dynamic library loading has been done using Qt.

Chapter 5 – General Purpose Computation on The GPU

Graphics Processing Units have now evolved into extremely fast processors with a slightly different architecture paradigm than that of general CPUs. They are now programmable and have parallel processing units. This particular advantage of parallelism is extremely useful for programs that can be cast as stream processing problems[9]

Figure 5.1 shows the growth of sheer speed of GPU's as compared to CPUs. As we can see the GPU growth follows growth pattern higher than the Moore's law prediction for CPUs.

5.1 Graphics Pipepline

Owen et. al in their Survey of General Purpose Computation on Graphics Hardware describe graphics pipeline and its hardware implementation:

> All of today's commodity GPUs structure their graphics computation in a similar organization called the graphics pipeline. This pipeline is designed to allow hardware implementations to maintain high computation rates through parallel execution. The pipeline is divided into several stages. All geometric primitives pass through every stage. In hardware, each stage is implemented as a separate piece of hardware on the GPU in what is termed a task-parallel machine organization[1].

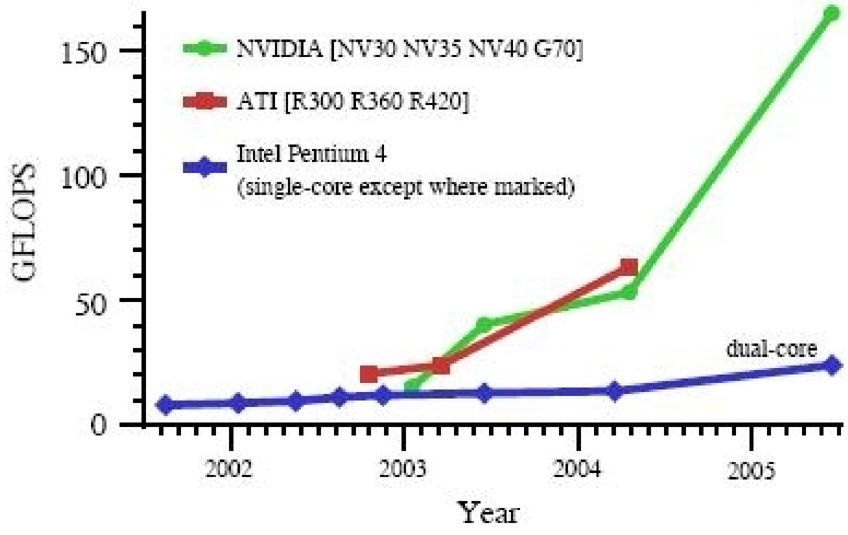


Figure 5.1: GPU speed Vs CPU speed [1]

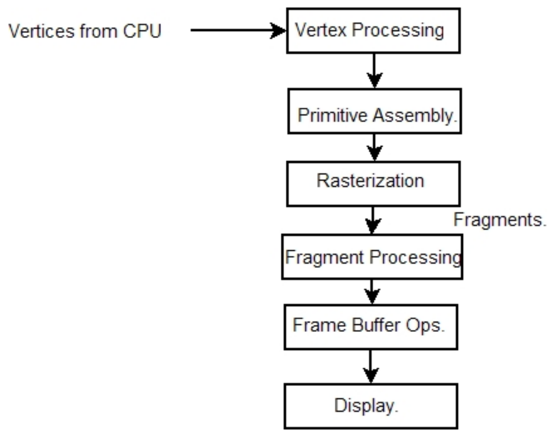


Figure 5.2: Graphics Pipeline

Vertex Buffer and Vertex Processor The programmable vertex processor replaces the following functionalities of the fixed OpenGL Pipeline.

- Vertex transformation
- Normal transformation and Normalization
- Texture coordinate generation and transformation.
- Per-vertex lighting.

Rasterization This stage determines the screen position covered by each triangle and interpolated per-vertex parameters. Each of these color values (fragments) are then passed to the fragment processor for per-fragment operation.

Fragment processing In this stage, color for each fragment is computed. This computation can also use values from global texture memory to compute color values.

The programmable fragment processor replaces the following functionality of the fixed OpengGL pipleline.

- Color computation
- Fog
- Texture Application
- Normal/Lighting computation for per-pixel lighting.

5.2 Mapping of programmable graphics pipline to General purpose computation.

5.2.1 Programmable hardware

The programmable part of the GPU are the two processors, fragment and vertex, each capable of replacing some functionality of the fixed function pipeline. Over time, with more sophisticated technology, the functionality and generality of these processors has increased. Currently, these GPU's support SIMD (Single Instruction Multiple Data) instructions, ever increasing amounts of inputs, outputs and the number of processors working in parallel. This parallelism is, in fact, more important with respect to GPGPU than raw computational speed. The availability of high level languages for programming these GPU's is the single most important facilitator for general purpose computing on GPU.

However, there are other issues which limit the usage of GPU as general purpose computation device. Most importantly is that they require an entirely different programming model. Much research is going into mapping problems onto the GPU while optimally using the speed. Some promising areas which have come up include numerical computation (PDE solving, matrix operations), physical simulation, ray tracing (traditionally a CPU intensive task), image processing etc. As part of this thesis we have tried to assess the feasibility of using the GPU for processing 3D data sets in a much faster way than is possible with CPU.

5.2.2 General GPU Programming Model

A typical GPGPU program uses the fragment processor as its main computational engine. This is because a typical graphics scene has more fragments than vertices hence they need to be more and faster fragment processors than vertex processors. However, highly efficient implementations try to balance the load on both vertex processor and fragment processor. The programming model derives from the streaming computation model, where programs are considered as *kernels* and data on which they operate are termed as *streams* The structure of a GPGPU program is [1][10],

- The Application is segmented in independent parallel tasks called *kernels.* input data is transformed into textures that can be considered as streams. On these streams, kernels are invoked to perform necessary computation.
- These kernels are then programmed as shader programs and invoked by passing the vertices to the vertex processor. A typical invocation might include drawing a textured quadrilateral, equal to the size of the stream data (in the textures) that needs to be processed.
- The rasterizer then generates millions of fragments and color values (output values) which are then written to framebuffer memory.
- The framebuffer memory can be retrieved as a texture used as output, or again as input to another pass.

5.3 Shader Programs

In context of GPGPU, vertex programs often emulate the fixed function pipeline in all, but the most optimally designed programs. Fragment programs are generally the main processing units because the graphics hardware is such that there are more fragment processors and they also work at higher speeds.

Multiple options are available for coding these programs or shaders which include GLSL or GL Shader Language with OpenGL origins with Open Standards, Cg which is a Nvidia proprietary langauge with bindings for both DirectX and OpenGL and HSLS - specifically for DirectX.

As the name suggests vertex shaders are used for programming the vertex processor and fragment shaders for the fragment processor. Both these processors emulate the fixed graphics pipeline and accordingly behave in terms of input and output. As mentioned earlier, most of the processing is done using fragment shaders and it forms the core of any GPGPU application. These fragment shaders expect input in the form of interpolated values from the vertex shader, infrequently changing values from the host program and possibly per vertex information which does not need to be interpolated. These fragment shaders also have access to textures or areas of memory through input data encoded as an image can be passed in.

5.3.1 Textures

Textures form an integral part of the GPGPU pipeline. Input data is encoded as an image, which is then operated upon by the shader programs. Rather than rendering on the screen - the output is rendered onto a off-screen buffer which is then read back as textures and output encoded as color values is read back. OpenGL supports multiple types of textures which provide different precisions eg floating point and unsigned byte. For this project, we used floating point textures exclusively because of the higher precision of floating point values.

5.4 pBuffers and Frame Buffer Objects

Output from a GPGPU application is generally an image in which the result of a computation is encoded. This image is generally rendered to an off-screen buffer because a windowing context is not necessary. This non-visible rendering context provided by the OpenGL renderer is known as Pixel Buffer or pBuffer. Allocating or deallocating such buffers or even switching between multiple buffers is an expensive operation as the OpenGL context in question could be potentially a heavy weight entity. The original goal was to have a static area for rendering.

In GPGPU applications, there's generally a feedback loop in which the output is again used as input for further iterations (Ping Pong)[10]. This makes maintaining multiple buffers a necessity. To overcome the efficiency issues related to pBuffers, they have now been superseded by FrameBuffer Objects. FBO is currently implemented as an OpenGL extension and should be standardized soon.

The framebuffer object (FBO) extension presents a much better and more simplified method of doing render-to-texture. Its advantages include:

- FBO only requires a single OpenGL context and hence avoids the need for expensive context switches in case of pBuffers.
- FBO model is similar to DirectX's render to target model making the possibility of porting code or abstracting both OpenGL and DirectX easier.
- FBO is more memory efficient as supporting buffers like depth and color can be shared across multiple rendering targets[13]

For reasons mentioned, we have exclusively used Framebuffer Objects in this implementation.

Chapter 6 – Using The GPU for Volume filtering

In this chapter we will discuss how a 3D texture is transformed by using GPU code.

6.1 3D Textures

Texture Mapping is a method of adding detail or surface texture onto a surfaces. A texture is essentially an ordered data set which is indexed by Texture coordinates. These texture coordinates are then used to map this data on to a surface for adding detail. These datasets can contain color data ,luminance , transparencies, etc and in our case scalar values of a 3D dataset like MRI.

Other ways of storing a 3D dataset includes a tiled approach where the 2D slices of the 3D dataset are tiled to form a big 2D dataset. This approach was preferred when hardware support for 3D textures was not available. The biggest advantage of using 3D texture as such is the trilinear filtering on hardware which is not possible with the series of 2D datasets.

6.2 Computation loop

Each slice of the 3D texture is rendered on a Quadrilateral in such a way that there's one to one correspondence between pixels and Texels. This is necessary because the fragment processor needs to be invoked for every texel or datum in the dataset. The result is then copied back to a 3D texture slice by slice and displayed on the screen using the visualization routines described in chapter 3.

For every slice in the input 3D texture

1. Set up a 2D FrameBuffer Object using glFramebufferTexture2D.

2. Set up a RenderBuffer object using glRenderbufferStorage

3. Attach RenderBuffer to framebuffer using glFramebufferRenderbuffer

4. Make the framebuffer object, the current rendering context.

5. Set up Projection matrices. As mentioned earlier we want a 1:1 mapping between pixels (to which we want to render) and texels (from which we access data. The key here is to choose an orthogonal projection and a proper viewport that will enable a one to one mapping. A typical way of achieving that would be

```
glViewport(0, 0, texSize, texSize);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

6. Set the input texture as the active texture.
7. Enable the fragment shader code.
8. Draw a quadrilateral.
9. Disable fragment shader code.
10. Set the output texture as active texture.
11. Copy the content of framebuffer on the the output texture using *glCopyTexSubImage3D*

OpenGL 2.0 also supports directly rendering to each 2D slice of the output 3D texture, creating a 3D framebuffer. However at this point, in most cards it is emulated by drivers and is extremely slow or, at best, equivalent to the copying mechanism negating any possible theoretical performance gain we might have.

The filtering computations in the fragment program is which invoked for each and every datum in the data set. Filters like Mean, Median and Gaussian were implemented. To accentuate the boundaries, gradients were also calculated and then used as input to the transfer function.

Chapter 7 – Results

Two datasets, benoit.vox and smallhead.vox were analyzed and representative Images are shown.

7.1 Dataset - benoit

benoit.vox is a 3D ultrasound dataset. Because of its ultrasound origins, this dataset is more or less binary, making analyze difficult. However various attempts were made to analyst the structure and are shown in Figures 7.1 to 7.8.

7.2 Dataset - smallhead.vox

Smallhead.vox is an MRI based dataset. Because of its highly organic nature analyzing internal boundaries is difficult. Problems associated with it are noise and very fine detail, specially around the folds in the brain. Figures 7.9 to 7.15 show the various results obtained.

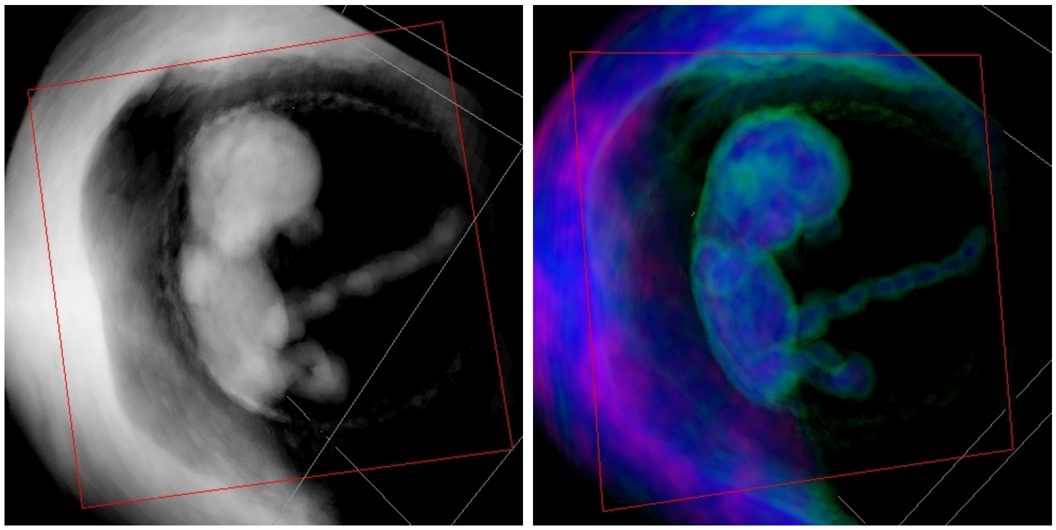


Figure 7.1: Base gray scale image and color visualization after applying mean filter

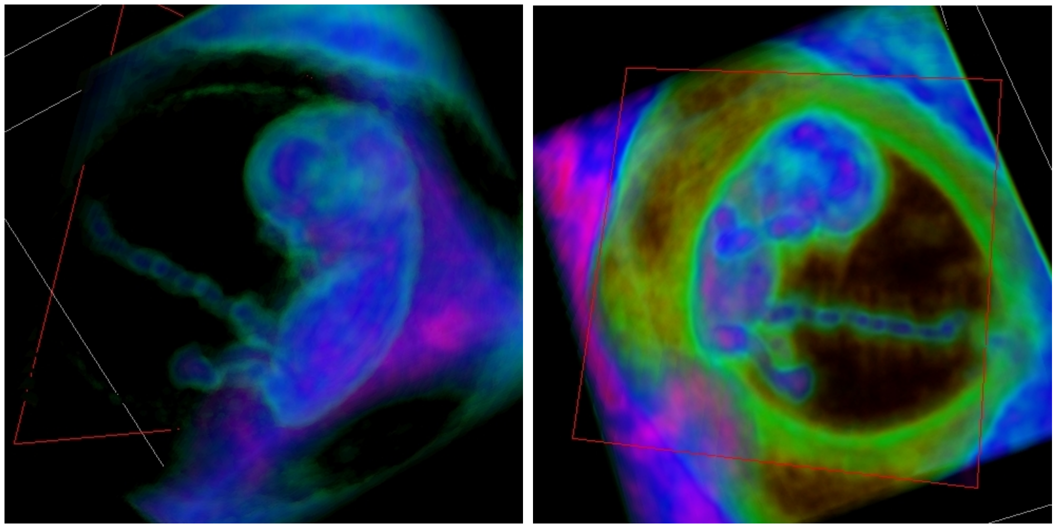


Figure 7.2: Same dataset from a different view and accentuated by gradients

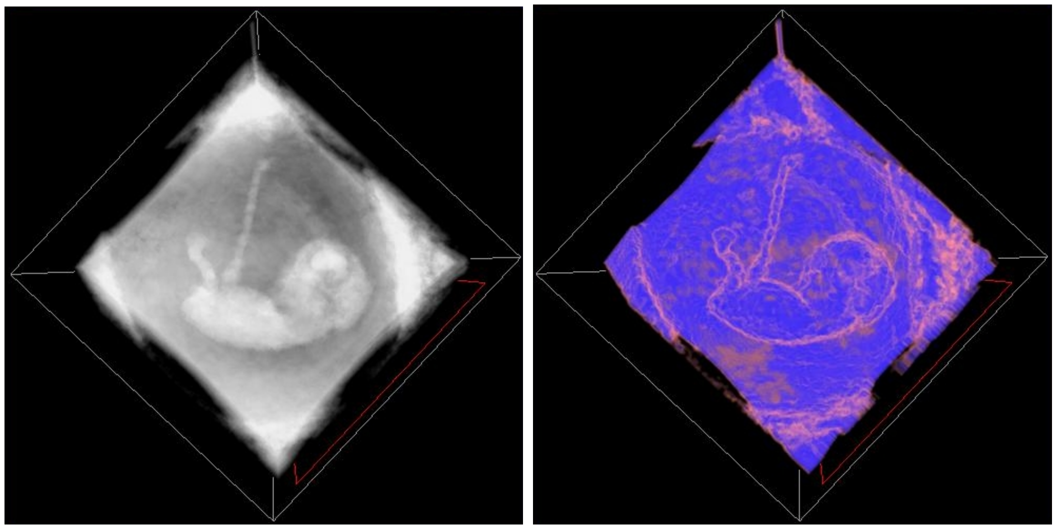


Figure 7.3: Mean filtering with a larger window - white noise has been considerably reduced however image detail has also reduced (Umbical cord thickness has been reduced

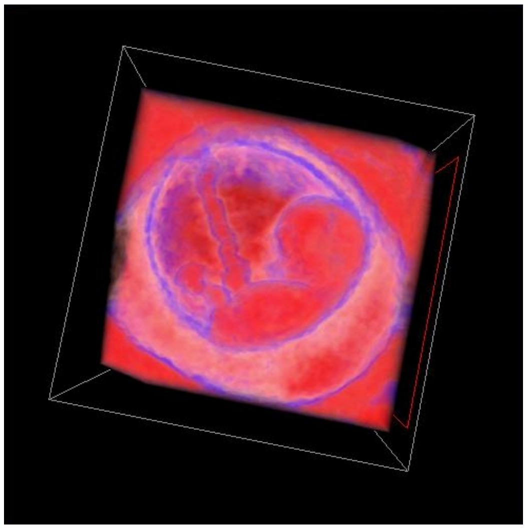


Figure 7.4: NonPhoto realistic style visualization obtained by clamping to color values obtained from transfer functions

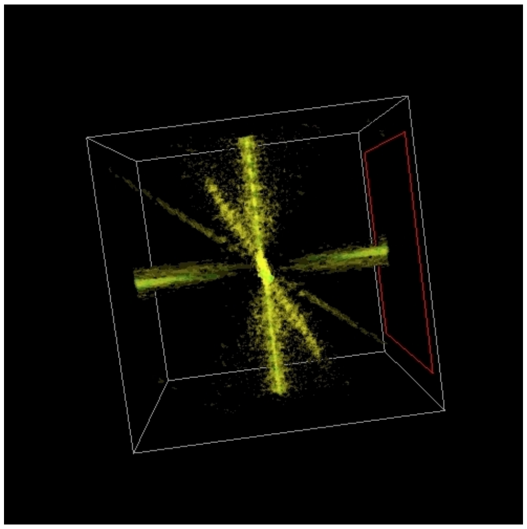


Figure 7.5: Frequency Plot obtained from FFT of the benoit dataset, the spikes across the x,y and z axis represent the discretization. The diagonal spike represent the direction in which the ultra-sound was sampled. This tells us that resampling in this direction would not lead to any more detail

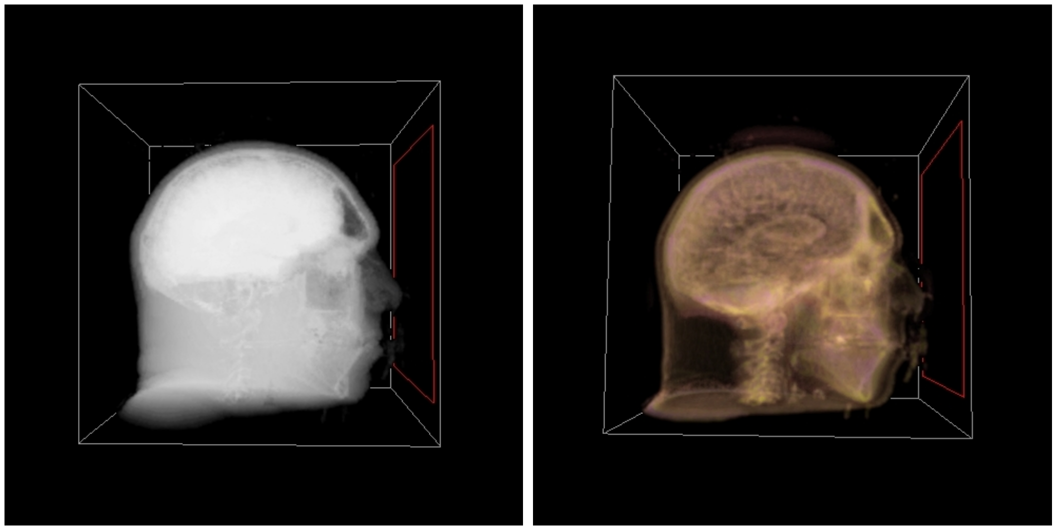


Figure 7.6: Left:Unmodified dataset,Right: After a high pass FFT filter accentuating internal structure

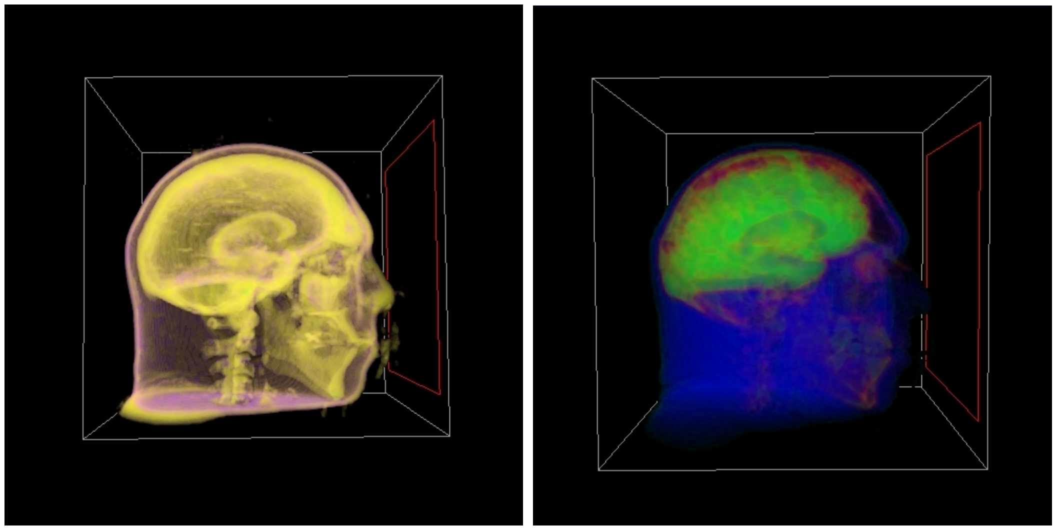


Figure 7.7: Left: Applying Median filter to the high pass filtered dataset, boundaries are more pronounced. Right: Result after applying the mean filter

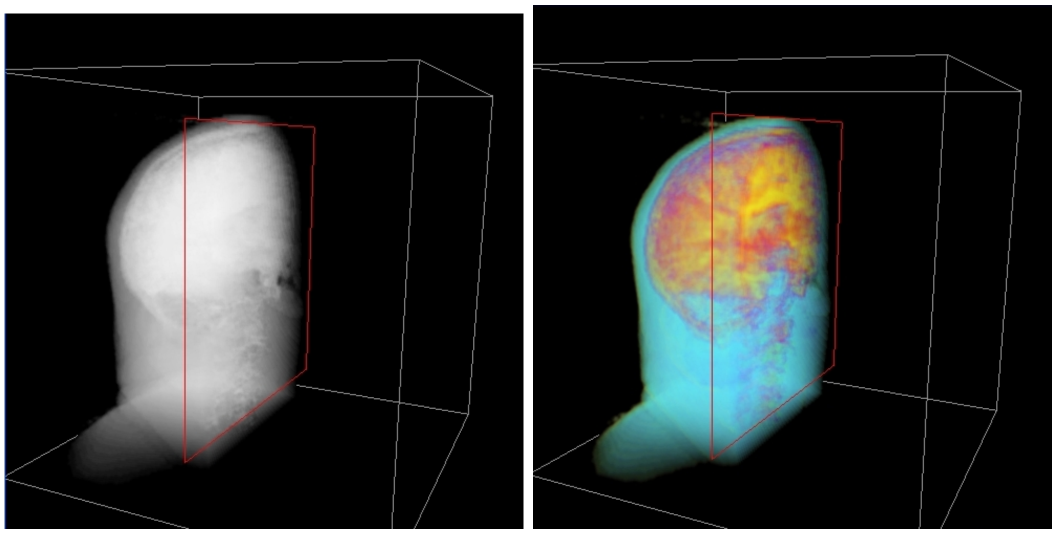


Figure 7.8: Clipping plane in action

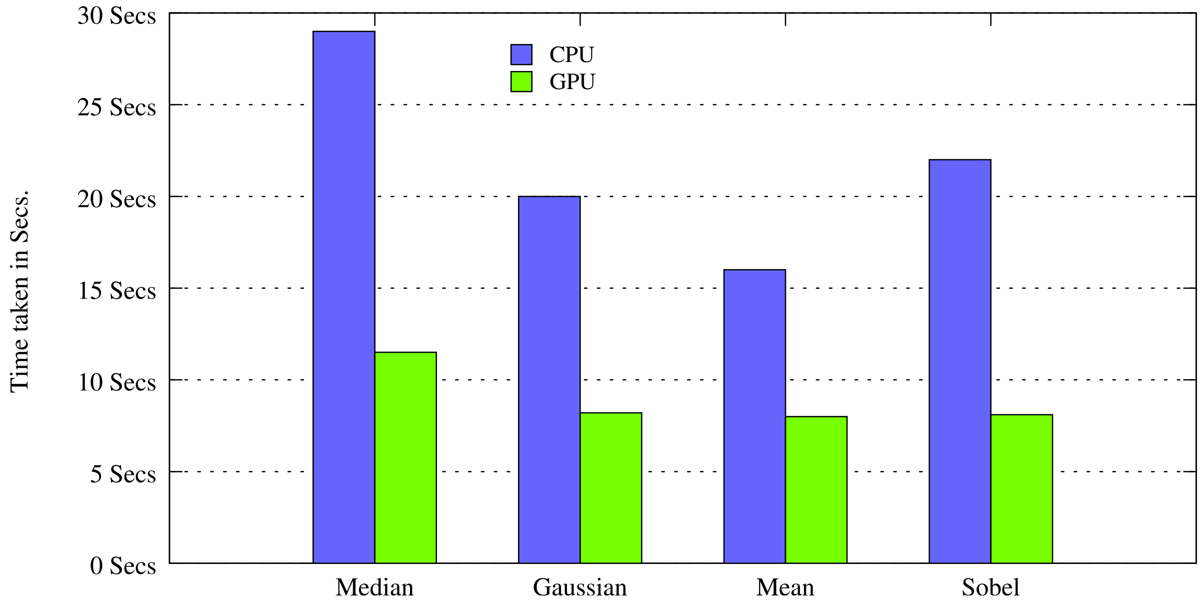


Figure 7.9: This graph shows the time taken by some filters on both 3.2 Ghz dual core Intel CPU and Nvidia 3400 QuadroFx GPU on a 128x128x128 size dataset (Benoit)

Chapter 8 – Conclusion and Future work

This thesis has presented techniques to optimize processing and visualization of 3D datasets. It has also been shown the Graphics Processing Unit is a viable co-processor to the CPU in context of processing of 3D datasets. Experiments have suggested that the GPU provides an order more floating point operations than the CPU. Current trends in GPU architecture suggests that this gap will continue to widen.

However it remains imperative that we strive towards parallel processing of datasets. This is because of the shift in CPU architecture towards multi-core architectures and innovative methods need to be found for using the CPU's. Our CPU based application is a first step towards that.

Future work includes combining GPUs and CPUs in such a way the that stream processing model is abstracted from the user. Current limitations of bandwidth between CPU and GPU and an entirely different computing model are the biggest challenges. Recent developments, specifically Nvidia's CUDA technology [26] seems an interesting step forward. The availability of C compiler and possibility of threads running on GPU to cooperate when solving a problem could make GPU an extremely fast and able co-processor.

Bibliography

[1] John D Owens,David Lubekke,Naga Govindraju, Mark Harris, Jens Kruger, Aaron E Lefon and Timoth J Prucell. A Survey of General Purpose Computation on Graphics Hardware, Eurographics 2005, State of Art Reports, pp. 2151, August 2005.

[2] Randy Roast, OpenGL Shading Language, Addison-Wesley, 2006.

[3] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, OpenGL Programming Guide Fifth Edition, Addison Wesley, 2005.

[4] Peter Shirley, Fundamentals of Computer Graphics, Second Edition, AK peters Ltd, 2005.

[5] Orion Wilson and Allen VanGelder and Jane Wilhelms, Direct volume rendering via 3D textures, University of California at Santa Cruz, 1994.

[6] Gonzalez, R.C. Digital Image Processing Prentice Hall, 2nd Edition, 2002.

[7] W.Lorensen, H.Cline Marching Cubes: A High Resolution 3D Surface Construction Algorithm Computer Graphics, 21 (4): 163-169, July 1987

[8] 6800Shiaofen Fang and Tom Biddlecome and Mihran Tuceryan, Image-Based Transfer Function Design for Data Exploration in Volume Visualization, IEEE Visualization 1998.

[9] Timothy J. Purcell and Craig Donner and Mike Cammarano and Henrik Wann Jensen and Pat Hanrahan,Photon mapping on programmable graphics hardware, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2003

[10] Matt Phar, GPU Gems 2 ,Addison-Wesely 2005

[11] OpenDX, http://www.opendx.org

[12] Digital Filters. http://homepages.inf.ed.ac.uk/rbf/HIPR2/filtops.htm

[13] Simon Green, Nvidia, The OpenGL Framebuffer Object Extension http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL Day/OpenGL FrameBuffer Object.pdf

[14] David Laur and Pat Hanrahan, Hierarchical splatting: a progressive refinement algorithm for volume rendering, SIGGRAPH 1991

[15] Arie E. Kaufman,Volume Visualization ACM Comput. Surv , 1996

[16] Klaus Engel and Martin Kraus and Thomas Ertl, High-quality pre-integrated volume rendering using hardware-accelerated pixel shading, HWWS 01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware.

[17] Westover, L., Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm. PhD Dissertation, July 1991.

[18] “Fastest Fourier Transform in the West” http://www.fftw.org

[19] Qt Library, TrollTech Inc, http://www.trolltech.com

[20] Hanspeter Pfister Et. al., Visualization Viewpoints, IEEE 2001

[21] KitWare Inc, The Visualization Toolkit,

[22] GLEW, http://glew.sourceforge.net

[23] Nvidia 6800 Specifications, http://www.nvidia.com/object/geforce6techspecs.html

[24] Mercury Computer Systems Inc, Amira - Advanced 3D Visualization and Volume Rendering

[25] Apple Inc. Shake - Advanced Digital composition

[26] Compute Unified Device Architecture, http://developer.nvidia.com/object/cuda.html

[27] Volume Rendering, ”http://en.wikipedia.org/wiki/Volume_rendering”.

[28] Sobel Operator, ”http://en.wikipedia.org/wiki/Sobel”

[29] Noise, ”http://en.wikipedia.org/wiki/Noise”

[30] Simon Green, FrameBuffer Objects , http://www.gamedev.net/columns/events/coverage/feature.asp?feature_id=75

[31] Terarecon Inc, http://www.terarecon.com/

[32] Min Yoh Wu, On parallelization of static schedulling algorithms, tech report , Department of Computer science,SUNY - Buffalo.

[33] M R Gary and D S Johnson, Computers and Intractability A Guide to the Theory of NP Completeness WH Freeman and Company, 1979.

[34] Yu-Kwong Kwok, Ishfaq Ahmad Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors, (1999) ACM Computing Surveys, 1999.

APPENDICES

.1 Median Filtering Shader, also representative of techniques used in other shaders

```
// median filtering shader.
uniform sampler3D tex;

const int dim = 3;
const int size = dim*dim*dim;
const int sizeminusone = dim*dim*dim-1;
const float halfsize = (dim*dim*dim)/2;
float data[size];

vec3 HsvRgb(vec3 hsv) ;

void bubblesort()
{
  // bubble sort values.
   float minor, major;
   for( int i=0; i<size; ++i) {
      for( int j=0; j<sizeminusone; ++j) {
         minor = min( data[j], data[j+1] );
         major = max( data[j], data[j+1]);
         data[j]   = minor;
         data[j+1] = major;
      }
   }
}


void main(void)
{
 const float offset = 1.0/256.0 ;
 vec3 grad ;
 vec4 C = texture3D ( tex, vec3( gl_TexCoord[0].stp )) ;


// compute average.
int a = 0 ;
for ( float i = -1 ; i < 2 ; i++ ){
    for ( float j = -1 ; j < 2 ; j++ ){
        for ( float k = -1 ; k < 2 ; k++ ){
            vec4 gxc1 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
                        +vec3 ( offset*i  , offset*j , offset*k ));
            data[a] = gxc1.a ;
            a++;
        }
   }
}

 bubblesort();

 // pick the median.
 vec3 hsv = vec3( 200*data[d3],1.0,1.0 ) ;

// compute gradients.
vec4 gxc1 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
            + vec3 ( offset , 0 , 0 )) ;
vec4 gxc2 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
            +vec3 ( -offset , 0 , 0 )) ;

grad.x =  ( gxc1.a + gxc2.a )/2  ;

vec4 c2 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
            +vec3 ( -offset , 0 , 0 )) ;

vec4 gyc1 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
                + vec3 ( 0 , offset , 0  )) ;
    vec4 gyc2 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
                + vec3 ( 0 , -offset , 0 )) ;

    grad.y =  ( gyc1.a + gyc2.a )/2.0  ;

    vec4 gzc1 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
                +vec3 ( 0 , 0,  offset   )) ;
    vec4 gzc2 = texture3D ( tex, vec3 ( gl_TexCoord[0].stp )
    ‘           +vec3 ( 0 , 0, -offset   )) ;

    grad.z =  ( gzc1.a + gzc2.a )/2.0  ;

    float intensity = sqrt ( grad.x*grad.x + grad.y*grad.y + grad.z*grad.z );

    // clamp values.
    vec4 color ;

    if (intensity < 0.26)
          color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity < 0.36)
          color = vec4(0,0,01,1.0);
    else if (intensity < 0.9)
          color = vec4(0.4,0.2,0.2,1.0);
    else
          color = vec4(0.2,0.1,0.1,1.0);



     // final setting of gl_FragColor.
     gl_FragColor = color + vec4 ( HsvRgb(hsv) , intensity )  ;
     gl_FragColor.a =C.a + 2.0*intensity ;
}

vec3 HsvRgb( vec3 hsv )
{
```

```
// HSV to RGB routine.
   vec3 rgb ;
   float h, s, v;           // hue, sat, value
   float r, g, b;           // red, green, blue
   float i, f, p, q, t;     // interim values

// guarantee valid input:

   h = hsv.x / 60.;
   while( h >= 6. ) h -= 6.;
   while( h <  0. )     h += 6.;

   s = hsv.y;
   if( s < 0. )
      s = 0.;
   if( s > 1. )
      s = 1.;

   v = hsv.z;
   if( v < 0. )
      v = 0.;
   if( v > 1. )
      v = 1.;


// if sat==0, then is a gray:

   if( s == 0.0 ){
      rgb.x = rgb.y = rgb.z = v;
   }
   else {
    // get an rgb from the hue itself:
      i = floor( h );
      f = h - i;
      p = v * ( 1. - s );
      q = v * ( 1. - s*f );
      t = v * ( 1. - ( s * (1.-f) ) );

    if (i == 0.0 ){
      r = v;    g = t;  b = p;
    }
    if (i == 1.0 ){
      r = q;    g = v;  b = p;
     }
    if (i == 2.0 ){
      r = p;    g = v;  b = t;
    }
    if (i == 3.0 ){
      r = p;    g = q;  b = v;
    }
    if(i == 4.0 ){
      r = t;    g = p;  b = v;
    }
    if(i == 5.0 ){
      r = v;    g = p;  b = q;
    }

    rgb.x = r;
    rgb.y = g;
    rgb.z = b;
  }
  return rgb ;
}
```