GPU-ASSISTED RENDERING OF LARGE TREE-SHAPED DATA SETS

A thesis submitted in partial fulfillment

Of the requirements for the degree of

Master of Science

By

PALLAVI R MANGALVEDKAR

B.E., Visvesvaraya Technological University, India, 2004

2007

Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

November 14, 2007

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Pallavi R Mangalvedkar ENTITLED GPU-assisted rendering of large tree-shaped data sets BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

_____

Thomas Wischgoll, Ph.D.
Thesis Director

_____

Thomas Sudkamp, Ph.D.
Department Chair

Committee on
Final Examination

_____

Thomas Wischgoll, Ph.D.

_____

Arthur Goshtasby., Ph.D.

_____

T.K.Prasad., Ph.D.

_____

Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Mangalvedkar, Pallavi. M.S., Department of Computer Science, Wright State University, 2007. GPU-Assisted Rendering of Large Tree-Shaped Data.

Due to their complexity, medical data sets can easily comprise of several gigabytes in size. For example, geometric representations of a coronary arterial tree spanning vessels from the large proximal coronary arteries down to the capillary level consist of 6 gigabytes or more of geometry data, depending on the accuracy of the geometric representation. Visualization of such large data sets using the CPU alone can be inefficient and time consuming since they require special out-of-core techniques due to the size of the data set. The usage of the GPU can help render such data sets faster compared to CPU-based implementations by computing geometric information on-the-fly, thus eliminating the necessity of out-of-core methods. Here, we present a GPU-based algorithm for rendering large-scale tree-shaped data sets which takes advantage of the programmability of modern graphics hardware. The proposed approach makes use of a fragment shader to compute geometric information on-the-fly, thereby allowing faster visualization. Also to make use of the CPU's programmability along with the GPU, we further extend the algorithm such that the computation of the geometry is executed in parallel by the CPU and the GPU, which makes it more efficient. We illustrate this with a vascular tree data set which resembles the entire arterial coronary vasculature.

Keywords: GPU, Visualization, Fragment Shader

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Thomas Wischgoll, for his guidance and input on the thesis development process. I also wish to thank Dr. Arthur Goshtasby and Dr. T.K.Prasad for serving as members of my MS thesis defense committee, giving the constructive criticism necessary to produce a quality technical research document. I would also like to thank the Department of Computer Science and Engineering and Dr. Thomas Sudkamp, the Department Chair, for giving me the opportunity to obtain my degree at Wright State University.

1. INTRODUCTION

1.1 Problem Description

The increased life span in the world leads to a great deal of health services and medical care. Due to significant improvement in the medical imaging modalities, measurements, and resolution of the instruments etc., the volumetric data size keeps on increasing. So one of the major problems the medical field is facing is the data explosion. Previously the major concern was to get as much data as possible, but today the main aim is to traverse through the huge data sets and retrieve the important features in it. But the technological development is not fast enough to keep pace with the rapidly increasing data size. Therefore there is a need for many constraints and problems to be handled.

The visualization of such large volume data poses a challenge, as the accuracy in the visualization is of great important to analyze the data correctly. Though there has been a lot of improvement in the hardware also, the requirement for faster and more efficient algorithms still exists. Since the development of techniques which uses the programming capabilities of graphics hardware (GPU) have been developed, it has been possible to visualize data on a regular PC.

The emerging of the GPU as a powerful subsystem has made the computing industry very flexible. The advanced graphics cards nowadays are relatively cheap and almost all the PCs today have some or the other graphics card in them. Therefore in many computations they can serve as a co-processor to the CPU, sometimes performing even better than the CPU. To make use of the computational power of the GPU, high level

languages like Cg, HLSL, GLSL, Brook and others are available, which allow the users to program the GPU quite easily. Here we take advantage of the graphics processing unit available currently on PCs to minimize the CPU overhead.

Our algorithm uses OpenGL and Cg to make use of the graphics hardware's programmability. Our algorithm uses a texture consisting of triangle strips to represent blood vessels in the data set. As the data set is huge, the calculations will be proportionally huge and the calculation of this geometry has to be fast for efficient visualization. Using conventional methods to calculate such geometries will cause a lot of strain to the system. For example, with a system having 2 gigabytes of memory, a maximum data transfer of 2 gigabytes of geometry at a time is possible. The texture co-ordinates and the coloring effect required to get the actual graphics effect requires more bandwidth. In practice, the bandwidth will be a lot more than this.

Here the focus is on visualizing large scale medical data. Specifically the data set being visualized consists of a pig's heart being detailed to every blood vessel in it. The data set consists of nearly 11 million blood vessels and the size of its geometric representation spans up to 6 gigabytes. We use the pig's heart data set as its structure is quite similar to that of a human heart's structure. GPU-based rendering is done using fragment shaders. A similar rendering is done using the CPU. Both the results are synchronized to compute the desired vascular tree structure as fast as possible.

1.2 Motivation

Though there are many instruments available in the market to capture different kinds of data and though the data captured by each of the instruments may be of different format, type or character, they are all facing the same problem nowadays, and that is managing the huge amounts of data captured by advanced instruments. Another challenge that follows is the visualization of the data. Therefore in order to handle all these problems a system must be able to deal with the following things:

- **It must be able to handle huge amounts of data:** Just the storage of this huge data does not account to managing the data well. Researchers must be able to manage them in a standard and efficient way. They should be able to share the data with one another. The retrieval should also be fast and efficient.

- **It must be efficient and accurate:** Extracting the key features in the data set is a very important issue because the decision-making is dependent on it. This is one of the problems faced by the researchers in the industry. For example, in the medical field, the doctor's diagnosis of the problem might be based on the visualized image of the data set. If there is any fault in it, it might lead to fatal complications. Therefore the visualization of the data set must be accurate and efficient in conveying the information, in order for the right decision to be taken.

- **It must be fast:** The system should be able to deal with displaying the data set as fast as possible. If the process of visualization takes a very long time, it becomes less interactive. If the images are displayed at a very slow frame rate, the

data cannot be analyzed correctly. Hence the visualization must take place at a reasonable speed.

2. RELATED WORK

A lot of work has been done previously in this field. Here we will focus on the efforts done by many researchers on hardware-accelerated visualization of large scale data. Some of the early researchers on rendering 3D texture graphics using the graphics hardware were Cullip et al. [1], Cabral et al. [2] and Akeley [3]. Cullip et al. described a new method of rendering volumes that leveraged the 3D texturing hardware in Silicon Graphics Reality Engine workstations. They defined the volume data as 3D texture and utilized the parallel texturing hardware to perform reconstruction and re-sampling on polygons embedded in the texture. The re-sampled data on each polygon is transformed into color and opacity values and composited into the frame buffer.

A lot of work has been done in this field and the researchers are still working on it. Kiefer et al. [4], propose ways to exploit the standard PC's cache memories and the graphics hardware (graphics processing unit, GPU). Their technique optimizes the utilization of on-board caches, thereby being able to handle large data set, with good speed.

Heng et al. [5] propose a GPU-based rendering algorithm to facilitate the usage of the hardware resource. For rendering, they use the ray-casting algorithm. Their proposed algorithm is implemented completely on the GPU. Their algorithm can deal with many medical data sets on standard, off-the-shelf PCs.

Wang et al. [6] propose texture-based techniques to visualize microscopy data with support of GPU. Their technique uses three sets of 2D textures in the three principal directions, and creates the rendered image via a weighted sum of the images generated by blending the individual texture sets.

Engek et al. [7] have covered algorithms for volume rendering that employ new features and extensions of consumer graphics hardware to achieve high quality and interactive frame rates, in their report.

Loop et al. [8] have made use of the processing power of the graphics hardware to render the curved primitives, thereby avoiding tessellation artifacts and LOD management, unlike the triangle mesh. Their method also required less memory and bus bandwidth than a triangle mesh. They only render the portion of surface inside a bounding tetrahedron, which is analogous to curve segments and surface patches that are the image of a bounded domain.

Loop et al. [9], by making use of the parallelism of the programmable graphics hardware to obtain high performance, have presented a method for resolution independent rendering of paths and bounded regions, defined by quadratic and cubic spline curves. They compute a set of texture coordinates that are attached to the vertices to the quadratic and cubic Bezier curve control points. These coordinates encode a corresponding function for these curves. This is evaluated by a pixel shader, which determines whether the pixel is inside or outside of the curve.

Zhang et al. [10] have presented an algorithm to generate high quality images with a small number of slices by utilizing displaced pixel shading technique. They calculate the actual surface location by linear interpolation between the outer and the inner points, instead of sampling points along a ray with regular intervals. This location is used as the displaced pixel for the iso-surface illumination. They have increased the rendering speed by using multi-pass and early Z-culling.

Kruger et al. [11] have addressed the early ray termination and empty-space skipping into texture based volume rendering on the graphics processing unit. They describe this method as an alternative to object-order approaches. They use the early z-test to terminate fragment processing once sufficient opacity has been accumulated and also to skip empty space along the rays of sight.

Burger et al. [12] present an efficient data structure and algorithms for GPU ray tracing of secondary effects like reflections, refractions and shadows. They have used geometry shaders available in Direct3D 10, to efficiently build LDCs on the GPU. They also show that the traversal of secondary rays is greatly accelerated by exploiting a two level hierarchy and the adaptive nature of the LDC. Their method has obtained high-quality rendering of static and dynamic scenes at interactive rates due to the computational capabilities and also the bandwidth of recent GPUs.

Durikovic et al. [13] introduce a framework for spectrum based rendering on the GPU to compute local illumination. This uses the Phong reflectance model, employing the spectral power distribution of light source and the spectral reflectance of surface simulating the color rendition of light sources and metamerism of surfaces. They have

also proposed the light area defined by the spectral cube environment map and show the method for conversion of RGB environment map into a spectral one.

Hable et al. [14] have proposed a method that renders the primitive, one at a time, to classify the candidate surfels against the primitive and to evaluate the Boolean expression directly on the GPU. The linear formulation of a Boolean expression is combined with depth-peeling called *Blist*, the Blister algorithm renders an arbitrary CSG model of n primitives in at most k steps, where k is the number of depth layers in the arrangement of the primitives. Each of the steps starts by rendering the primitives to produce candidate surfels on the next depth layer.

Viola et at. [15] have introduced frequency domain volume rendering, which is a volume rendering technique with lower computational complexity. This algorithm is accelerated by a factor of 17 by mapping the rendering stage to the GPU. The rendering step is computed completely on the GPU. The rendering stage is implemented through shader programs running on programmable graphics hardware, to achieve highly interactive frame rates.

Wischgoll et al. [25] have proposed a novel method for rendering the entire porcine coronary tree down to the first segments of capillaries interactively. It employs the geometry reduction and occlusion culling techniques. These techniques exploit the geometrical topology of the object to achieve faster rendering speed while still handling the full complexity of the data.

3. THE GRAPHICS HARDWARE

3.1 Overview of Programmable Graphics Hardware

In the early days, due to the limited graphics power, the uses of graphics were very limited. The monolithic graphic chips of the early 1980s and 1990s are the ancestors of today's modern graphics card. The BitBLT support in these cards was very limited. The ANTIC co-processor used in Atari 800 and Atari 5200 is an early example. In the late 1980s and early 1990s, general-purpose microprocessors were used to implement the GPUs. As the technology advanced, the BitBLT and the drawing functions were moved on to the same board.

All along the 1990s the technology of the 2D GUI kept on advancing. The the mid-1990s, graphics assisted by the CPU were very common. This lead to a need for the hardware accelerated 3D graphics. As DirectX advanced, the 3D also evolved along with the time. The support for hardware-accelerated transform and lightning was introduced in Direct3D 7.0.

The advent of the DirectX 8.0 Application Programming Interface (API) and also the OpenGL functionalities, the programmable shading were added to the GPUs. The fragment shader is a short program that processes every pixel and the vertex shader is a short program that processes every vertex in an image before they are displayed on the screen. These shaders can accept additional data like the image textures as input. NVIDIA was the first to market a chip capable of these programmable shading. These GPUs are now becoming almost as flexible as the CPUs.

3.2 GPU Today

Today the computational capabilities of the GPU have lead to another field of research termed as the GPGPU, which stands for General Purpose Computing on the GPU. Due to the expanding gaming industry, the pressure on the GPU manufactures to manufacture better hardware with better designs has increased. This has lead to providing more and more flexibility to the programming capabilities of the GPU. The current chip line in the market has been dominated by ATI (ATI Radeon graphics chip line) and NVIDIA (NVIDIA Geforce graphics chip line.) There are some GPUs built inside the mother board by Intel, but these are not sufficient for playing very demanding 3D games.

The modern GPUs are used to mainly do 3D computer graphics related calculations. Units to accelerate the geometric calculations have been added to memory-intensive work of polygon rendering and mapping textures. Nowadays the GPUs support programmable shaders which can manipulate pixels and vertices along with many operations supported by the CPU also. In addition to all these things, the GPUs today also support the render to frame buffer capabilities. Even high-definition video can be decoded in the graphics card nowadays.

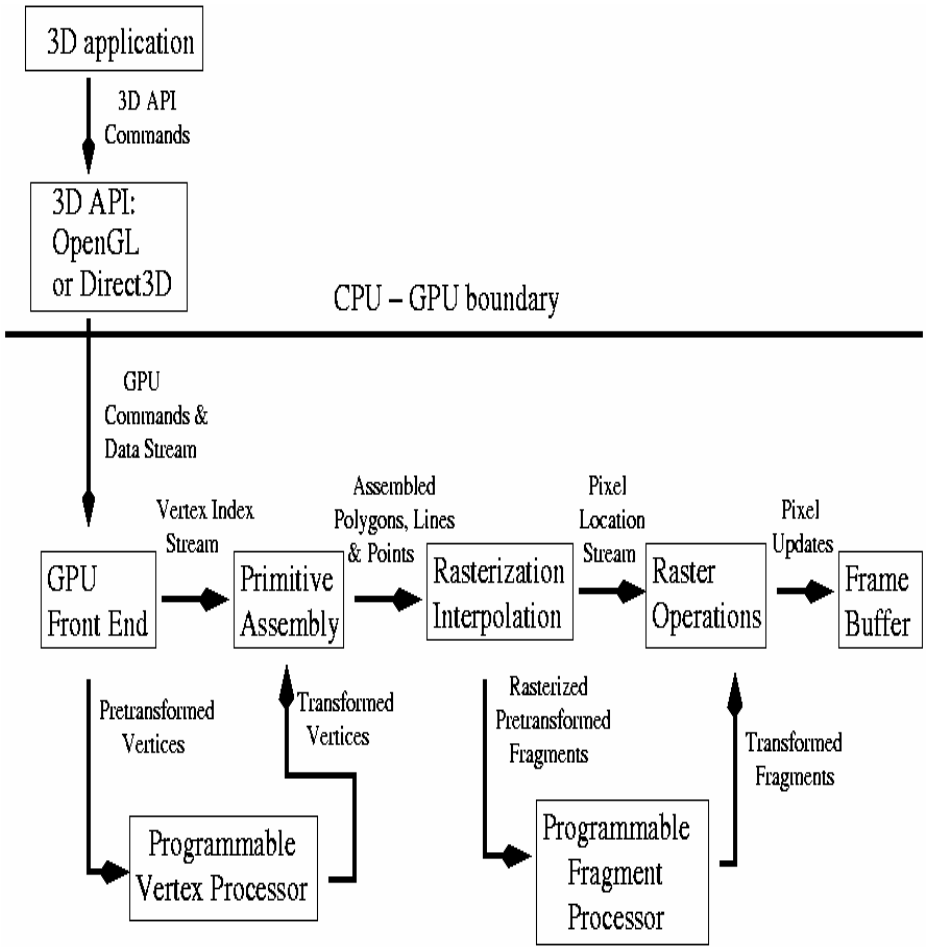
The graphics cards which have RAM that is dedicated to the card's use are the most powerful ones. They do not use the mother board's standard memory. Hence they are termed as 'dedicated graphics card'. There are also shared graphics card. They use the computer system's RAM for its use and do not have any RAM of its own and they are generally not fit to run graphics programs and modern 3D games.

3.3 Graphics Pipeline

All the computation on the graphics hardware is done in a sequence of steps called the graphics pipeline. They are designed so as to exploit the hardware implementation to keep high computation rates through parallel execution. There are several stages in the pipeline. All the primitives in the computation go through these stages. Each stage is implemented as a separate piece of hardware in the GPU.

The inputs to the pipeline are some representation of geometric primitives and the output is a 2D raster image. Generally these primitives are defined by vertices. These vertices then are transformed and per-vertex calculations are performed on it. At this point, a vertex shader can be used to manipulate the 3D vertices. Now the vertices are clipped and rasterized to get fragments. In this stage, another program called the fragment shader can be employed to perform calculations on every fragment before the final result is sent to the frame buffer for display.

The pipeline has very powerful programmable capabilities because the vertices and fragments are considered as a single unit. This allows all the calculations to be performed on each one of them. The hardware used in this project is a NVIDIA Geforce 6600 graphics card with 256 MB of graphics memory.



[16] Figure 3.3 shows the different stages in the graphics pipeline

4. SOFTWARE SPECIFICATION

4.1 Introduction

The system is based on C++ and the graphics library used is OpenGL (Open Graphics Library). OpenGL is a cross-platform API that is used to produce 3D computer graphics. To take advantage of the programmable graphics hardware, Cg (C for Graphics) has been used.

4.2 OpenGL

The Open Graphics Library is a library used to develop 3D computer graphics application. At the most basic level, it is a specification; which contains a set of functions. It also describes the exact behavior of their performance. It was developed by the Silicon Graphics Incorporation in 1992. The interface is made up of more than 250 function calls. They use simple primitives like points, lines, polygons, bitmaps and images to draw very complex three-dimensional scenes. A set of commands are provided to allow the users to specify whether the objects are in two dimensions or three dimensions. The primitives and the commands together control how the objects are rendered into the frame buffer.

OpenGL is operating system independent. Efficient implementations of OpenGL are available for platforms like Windows, Linux, Mac OS and many other Unix platforms. The event handling, color map operations, window management etc., is done by the windowing system. It was designed to be compatible with programming languages

like C and C++. But they also bind with other programming languages like FORTRAN, Java, ADA etc.

OpenGL provides the programmer with uniform APIs, thereby hiding the complexities of interfacing with 3D accelerators. As all the OpenGL implementations are required to support the full OpenGL feature set, it also hides the capabilities of different platforms.

OpenGL basically accepts primitives like points, lines, and polygons and are converted into pixels. The primitives are issued to the graphics pipeline by the OpenGL commands and are configured as to how the pipeline processes these primitives. Earlier, a fixed function was performed by each stage in the pipeline and they had very tight limits. But now in OpenGL 2.0, several stages can be programmed fully.

The OpenGL which is a procedural and low-level API requires the programmer to specify the exact steps required to render a scene. This is unlike other APIs', where the programmer describes a scene and the library manages the details of rendering it. OpenGL provides certain amount of freedom to implement rendering algorithms but it requires the programmers to have a good knowledge of the graphics pipeline. OpenGL is mainly designed to provide only rendering functions. It does not support any windowing concepts, input/output concept or devices, printing or any audio concepts. Though it seems restrictive, the main advantage is that, that it allows cross-platform development.

4.2.1 OpenGL Command Syntax

All the commands in OpenGL has the prefix ‘gl’ and use initial capital letters for each of the words that make up the command name such as, glMatrixMode(). All the OpenGL constants begin with ‘GL’, use underscores to separate each word and consists of all capital letters such as, GL_COLOR_BUFFER_BIT. In some of the OpenGL commands the type of parameters assigned to it are denoted by a number and have one, two or three letters at the end of the command. The number indicates the number of values which must be present in the command. The second character or character pair specifies the type of arguments: 8-bit integer, 16-bit integer, etc. If there is a last character present, it indicates that the command takes a pointer to an array of values. For example, in the command glColor3iv(), ‘3’ indicates that the command accepts 3 arguments, ‘i’ indicates that the arguments are integer values and ‘v’ indicates that the arguments are in vector format.

4.2.2 OpenGL Program Organization

The first call in any OpenGL program is to open a window into the frame buffer where the program will draw. Then, a GL context is allocated and the window associated with it. Now the programmer is free to issue any OpenGL commands.

```
main:
     find GL visual and create window
     initialize GL states (e.g. viewing, color, lighting)
     initialize display lists
     loop
          check for events (and process them)
          if   window event (window moved, exposed, etc.)
               modify viewport, if needed
               redraw
          else if mouse or keyboard
               do something, e.g., change states and redraw
redraw:
     clear screen (to background color)
     change state(s), if needed
     render some graphics
     change more states
     render some more graphics
     ……     .
 swap buffers
```

4.3 Cg Programming Language

Cg stands for C for Graphics. It is a high level shading language mainly for programming fragment and vertex shaders. It is created by NVIDIA. New data types have been added to and features modified in C programming language to make it more suitable for processing graphics, though its syntax is very similar to and based on C. This language resembles Microsoft's HLSL.

As the graphics cards have been advancing, 3D programming has been quite complex. In order to make it easier some features have been added to the graphics cards. Also the pixel and vertex shaders have the capability of modifying their rendering pipelines. Earlier the vertex and pixel shaders were programmed at the assembly language of the graphics card which, though provided more control over programming, was difficult to use. So Cg, a portable high level language for programming the GPU, was created to solve these problems. Its main advantage is the movement of the graphics related CPU load from the CPU to the GPU. Though Cg is a high level language, the Cg compiler can optimize the code and do low level tasks automatically, which are otherwise hard to do and prone to error.

Cg needs supporting programs to handle the rest of the rendering process as they are just fragment and vertex shaders. They are compatible with OpenGL and DirectX. Each of them have certain functions set to communicate with the Cg programs such as passing parameters and setting the shaders etc. The Cg compiler has the ability to compile the program during the run time of the supporting program. The concept of profiles was developed to hide the shader source code from the user. According to the

profiles, the shaders can be compiled to suit different hardware platforms. The profile also helps the select the most optimized shader to be loaded.

4.3.1 Vertex Shader

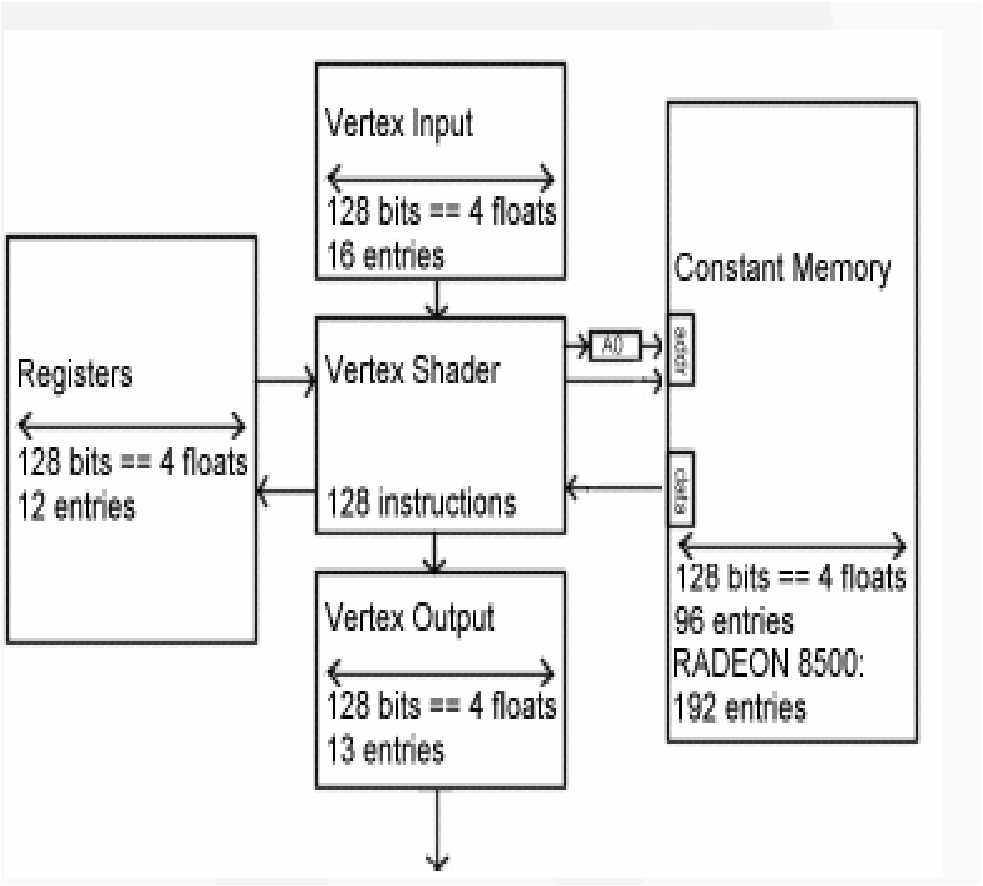
Manipulation of data that describes a vertex can be done using a vertex shader. It is a program that is run on the graphics processing unit, allowing the control of operations done on each vertex. It performs mathematical operations on the vertex's data in the 3D environment to add special effects to it. The type of data may depend on our application, for example it might be vertex's position, normal, texture co-ordinates for the polygon that should be textured or other lightning characteristics. These programs do not change the type of the data but they change the value of these data. This results in a vertex with altered color, position or texture values.

The vertex shader accepts a single vertex as the input, performs its assigned operation, then puts the resulting vertex back into the pipeline. This process is performed on every vertex in the scene. It does not operate on any primitives such as triangles or polygons. There is a way by which certain vertices can be selected or unselected, that is to load and unload the vertex program for every individual vertex or to group the vertices in two, one that has to undergo the vertex program and another that does not have to undergo the vertex program.

The vertex program is executed on the vertices before primitive assembly. There are certain predefined names such as POSITION, COLOR1, COLOR2, which are known as the binding semantics. These predefined names are mapped to certain hardware registers implicitly by OpenGL. The main program must supply values for the variables

declared. The position variable is a must in a vertex shader as it is used for rasterization. The basic vertex shader, at the very least, outputs the position of the vertex. The output of this program goes to a fragment shader. Even though a fragment shader is not implemented or activated explicitly, there is always a default fragment shader which is implemented in the hardware directly.

There are many effects that have been possible to create with the help of vertex shaders, of which no one even thought before. It has been possible to port these effects to consumer hardware with shaders. They are used to provide life and personality to characters and environments, which can intensify the graphics experience. An example is the animation of a joint with 32 bones in it, allowing them to move and flex convincingly [22].



[17] Figure 4.3.1 shows the architecture of a vertex shader:

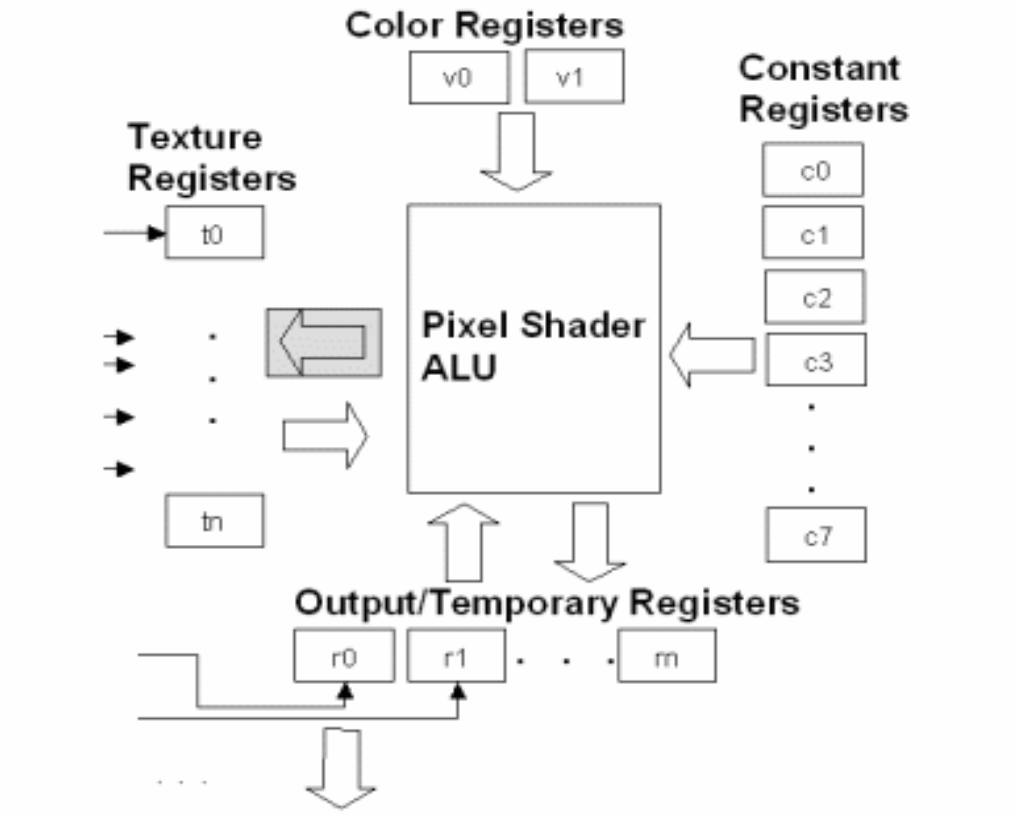
4.3.2 Fragment Shader

The data that describes a pixel can be manipulated using a program called the fragment shader, also known as the pixel shader. These are generally used to give effects to images. For example effects such as shadows, facial hair, realism, waves effect, explosion effect, golf balls with dimples can be applied to images. This is mainly done to mimic reality. A number of effects with their organic surfaces covers the computerized look with is very artificial.

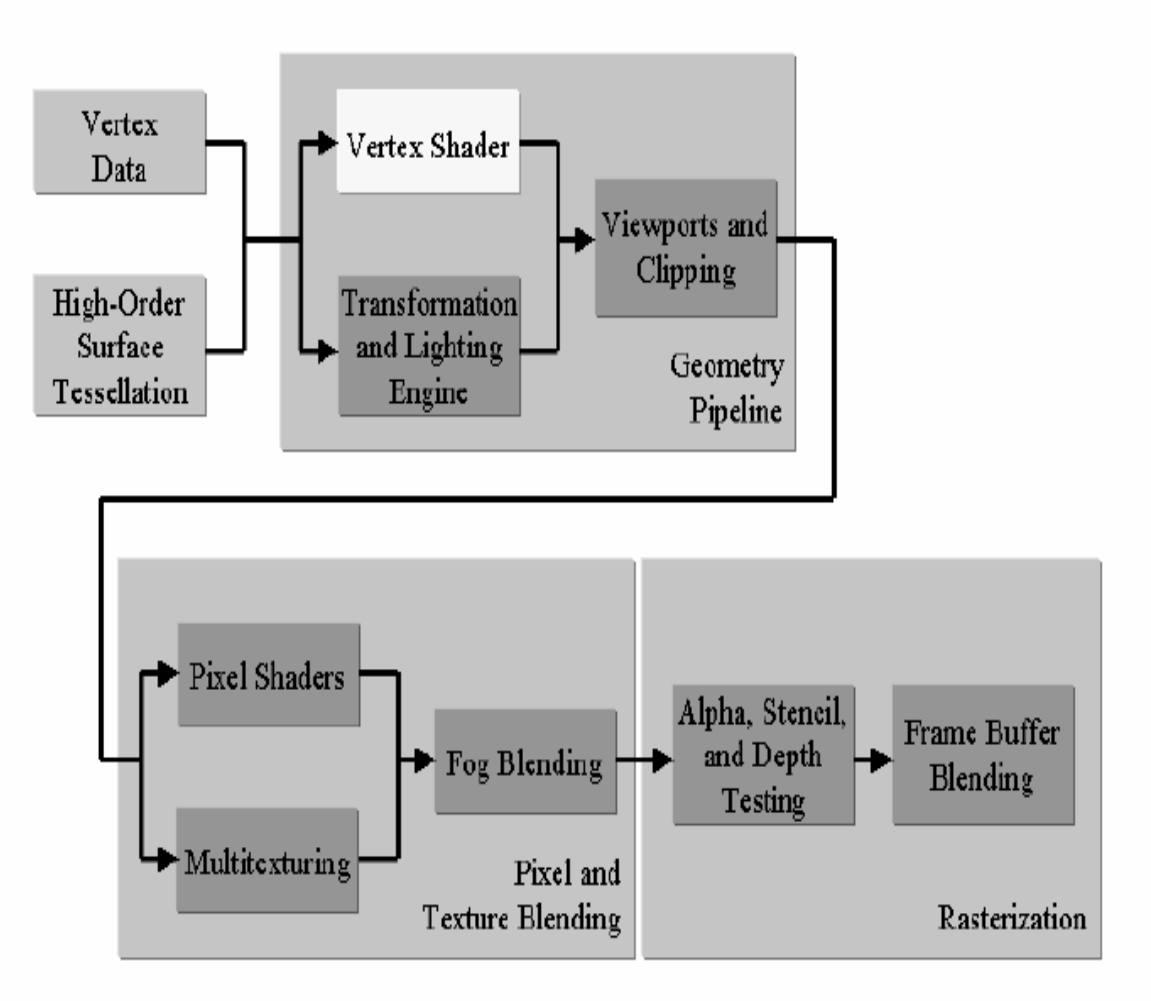
This program executes a graphic function for every pixel in the image. By altering the surface effects and lightning, colors, shapes and textures can be changed to render complex and real like images. Two million pixels may need to be rendered in excess for every frame, at 60 frames per second, depending upon the resolution. The fragment shader handles such huge computational load to bring the movie-style effects on PCs. Per-pixel shading can bring out a great level of surface detail. The fragment shader helps the programmer to put their creative vision to reality.

Thus, the programmable fragment shaders gives a lot of control to the programmer to create material effects, chosen from a preset palette of effects and also created on their own, to determine coloring, shading and lighting for each pixel. Also nowadays it is possible to achieve tremendous performances to enable pixel-level effects on common consumer platforms and PCs, which was previously impossible. By applying multiple textures in one pass yields a better performance that multiple passes. Multiple passes translate into multiple geometric transformations and also multiple Z-buffer computations, thus slowing the whole rendering process.

.



[18] Figure 4.3.2 (a) shows the architecture of a fragment shader



[19] Figure 4.3.2 (b) shows the data flow from the vertex shader to the fragment shader

5. GPU ASSISTED RENDERING

5.1 Concepts

5.1.1 Textures

A texture is a raw RGB image data which is used to create an illusion of color or grained details, which actually are not a part of the polygon, by mapping the image onto the polygon. This is similar to applying a patterned paper to a plain box. For example a texture map of the chess board coloration can be applied to a quad to create a chess board illusion. To create this, it would otherwise have taken many polygons. The color values of a texture are called as texels. Generally a texture image is 1D or 2D, but OpenGL 1.2 also supports 3D texture images. The extension SGIS_texture4D in OpenGL also supports 4D texture images. More than one image can be used at a time on a polygon and this is known as *multitexturing*.

Generally, standard 2D image files such as JPEG or TIFF are used as texture images. But a texture image can also be defined or generated. When the texture images are specified, the OpenGL's pixel transfer pipeline can process it. The reading and decompressing the image file is done by the OpenGL application. The OpenGL API offers no support for loading image files.

The height, width and the depth of the texture image should be powers of two. An additional one pixel border around the texture image is allowed. A border is a portion of the edge of the texture image. The border adds two texels in each dimension. The

coordinates have to be normalized to the range [0,1] by [0,1], independent of the actual dimension of the texture.

5.1.1.1 Texture Mapping

The steps in texture mapping are as follows:

- The first step in texture mapping is to create a texture object for each texture in use and associate textures for those objects. The texture maps and their associated texture parameter state are stored in the texture objects. The switching between textures can be done by a call to glBindTexture().

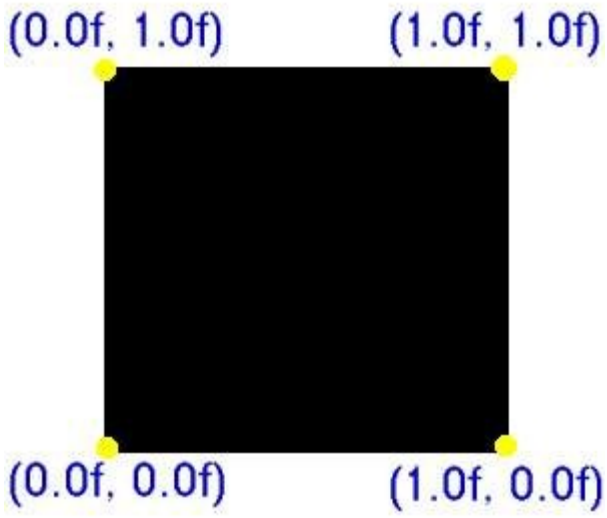
- It needs to be specified in what format the texture is supposed to be applied to the texel. There are four functions available for computing the final RGBA value from the fragment color and the texture-image data. One method is just to use the texture color as the final color. Here the texture color is applied on top of the fragment. Another possibility is to scale the fragment's color. This can be used to combine the lighting and texturing effects. Also the fragment color can be mixed with a constant color depending on the texture value.

- Texture mapping must be enabled before rendering the scene. This is done using the function calls glEnable() and glDisable(). The constant used for one- and two-dimensional texturing is GL_TEXTURE_1D and GL_TEXTURE_2D respectively.

- Finally the scene is drawn by specifying both the texture and object coordinates. For a two-dimensional texture map the texture coordinates range from 0.0 to 1.0 in both directions. But the object coordinates can be anything, so it should be indicated how the texture must be placed relative to the fragments to which it's to be applied. The code below is an example of how to texture a quad.

```
glBegin (GL_QUADS);
glTexCoord2f (0.0, 0.0);
glVertex3f (0.0, 0.0, 0.0);
glTexCoord2f (1.0, 0.0);
glVertex3f (10.0, 0.0, 0.0);
glTexCoord2f (1.0, 1.0);
glVertex3f (10.0, 10.0, 0.0);
glTexCoord2f (0.0, 1.0);
glVertex3f (0.0, 10.0, 0.0);
glEnd ();
```

The call to glTexCoord2f(x,y) places the texture coordinate at that place on the image. After the glEnd() is reached the quad which is formed by the texture coordinates is mapped onto the quad that is made up of the specified vertices. The figure below shows the texture coordinate system.



[23] Figure 5.1.1.1 shows the texture coordinate system

5.1.2 Frame Buffer Objects

OpenGL off-screen frame buffer is nothing but a collection of logical buffers. The frame buffer objects enables off-screen rendering by allowing the use of a frame buffer attachable image, which is a 2D array of pixels that can be attached to the frame buffer. This means that it defines an interface that allows the drawing to rendering destinations other than those provided by the window-system to the GL. Hence frame buffer-attachable images can be attached to the GL frame buffer as the standard GL logical buffers such as color, depth, and stencil. The frame buffer-attachable image attached to the frame buffer is used as the source and destination of fragment operations.

Furthermore, the images of a texture can be used as frame buffer-attachable images, thereby supporting ‘render to texture’. This allows the results of rendering to frame buffer to be directly used as textures, which avoids a copy from frame buffer to texture. This uses less memory as there will be only one copy of the image. This extension has similar semantics for the ‘render to texture’ as that of the traditional rendering to the frame buffer. But a call to CopyTexSubImage() follows it. Previous versions that allowed rendering to textures were complicated. But the EXT_framebuffer_\object extension is more efficient and simpler to use. A single frame buffer object can take care of the rendering to an unlimited number of texture objects.

A state that defines a traditional GL frame buffer state is contained in a frame buffer object, which also includes its set of images. Before, the window-system did all the definition and the collection of these images, by grouping them into a ‘drawable’. To bind a drawable to the GL context, the window-system also provided a function. Here, the GL provides the function BindFramebufferEXT to bind a frame buffer object to the context. The context can later bind back to the window-system provided frame buffer to display rendered content.

A new GL object type is also defined in this extension known as the “renderbuffer”. This encapsulates a single 2D pixel image. For general offscreen rendering, the renderbuffer image acts as a frame buffer-attachable image. Both a renderbuffer and a texture can be allocated independently and can be shared among many contexts, which saves memory.

Frame buffer is considered to be complete if all the attachments are consistent. If the status is not complete, then the glBegin or a call to DrawPixels() will generate the error INVALID_FRAMEBUFFER_OPERATION. Frame Buffer Object provides several ways of switching between different rendering destinations. A separate Frame Buffer Object for each texture to be rendered helps increase the performance [24]. In such a case, a switch can be done using the BindFrameBuffer() API. If a single Frame Buffer Object has multiple texture attachments, then all the images attached to it must have the same format, width and height, wherein a switch can be done using FrameBufferTexture() API. The textures can be attached to different color attachments. The glDrawBuffer() API can be used to switch rendering to different color attachments.

5.1.2.1 Framebuffer Object APIs

Creating and destroying framebuffer objects:

void GenFramebuffersEXT(sizei n, uint *framebuffers);

void DeleteFramebuffersEXT(sizei n, const uint framebuffers);

Binding an FBO:

void BindFramebufferEXT(enum target, uint framebuffer);

Attaching textures to FBOs:

void FramebufferTexture1DEXT(enum target, enum attachment, enum textarget , uint texture, int level);

void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget , uint texture, int level);

void FramebufferTexture3DEXT(enum target, enum attachment, enum textarget , uint texture, int level, int zoffset

Checking framebuffer status:

enum CheckFramebufferStatusEXT(enum target);

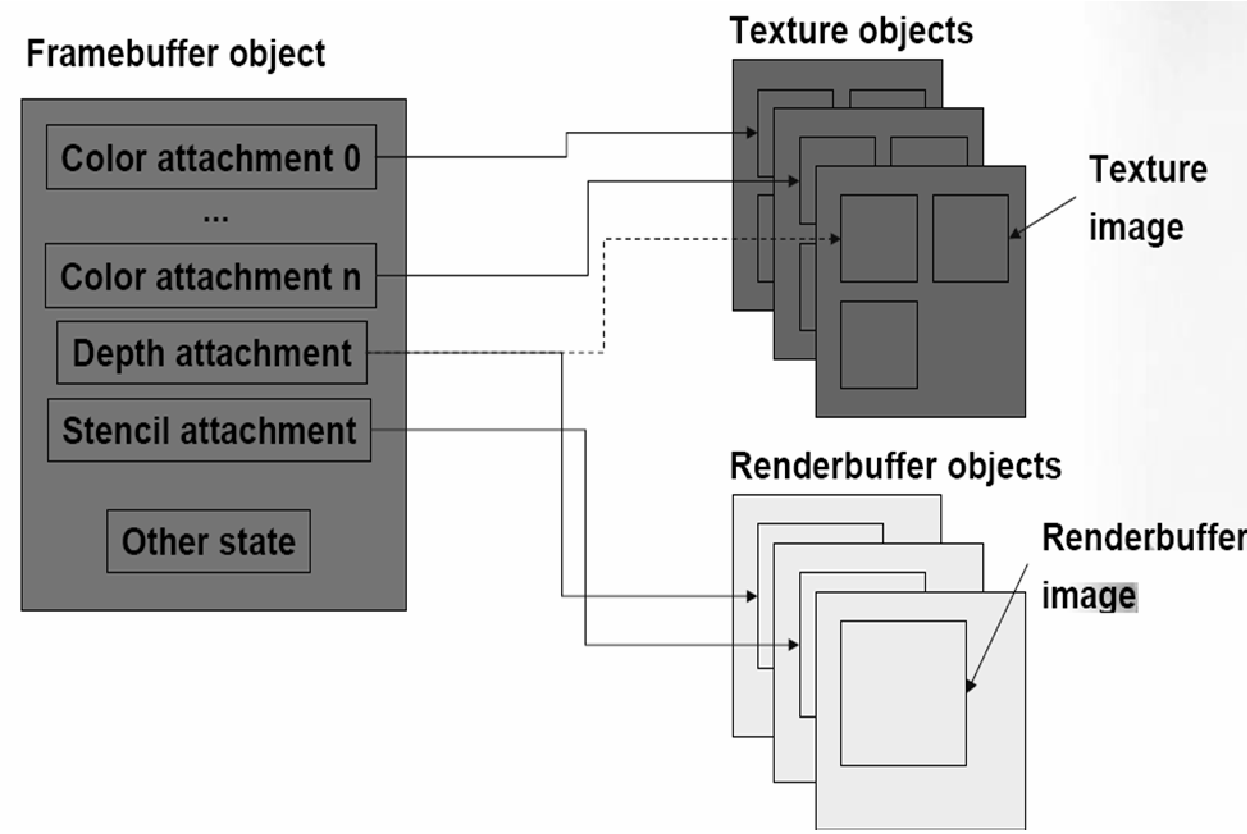
Other APIs:

boolean IsFramebufferEXT(uint framebuffer);

void FramebufferRenderbufferEXT(enum target, enum attachment, enum renderbuffertarget , uint renderbuffer);

void GetFramebufferAttachmentParameterivEXT(enum target, enum attachment, enum pname , int params);

void GenerateMipmapEXT(enum target);



[20] Figure 5.1.2 shows the architecture of frame buffer objects

5.1.3 Vertex Array

Per-vertex values contain information like color, position, normal etc. To render all these, more calls are needed. Calling many OpenGL calls is quite inefficient and requires more time. Vertex arrays are used to decrease these OpenGL calls. Vertex arrays let the access of a set of vertices and data by index. This makes it easier to many large scene rendering.

The vertex array accepts a pointer to the vertex data. The vertices to be rendered can be chosen and also how it has to be rendered can be specified. There are many ways by which OpenGL can know which vertices shall be composed to geometric primitives. With vertex arrays, vertices can be shared. GL_LINE_LOOP, GL_LINE_STRIP, GL_TRIANGLE_STRIP, GL_QUAD_STRIP allows the sharing of vertices between line segments, triangles and quadrilaterals. The sharing of vertices depends on the type of primitive chosen.

To turn on the vertex array functionality, a call to glEnableClientState() is necessary. The parameter to this call indicates whether a vertex array, normal array, color array, index array, texture array or an edge flag array has been enabled. Then a call to glVertexPointer, glNormalPointer, glColorPointer, glIndexPointer, glTexCoordPointer, glEdgeFlagPointer is made. The first parameter indicates whether it is a 2D, 3D or 4D vertices. The next parameter is used to indicate the data type, for example GL_FLOAT, GL_INT and so on. If the vertex array contains other data, such as the normals, the third parameter is used to signify this. The final parameter accepts a pointer to the vertex array.

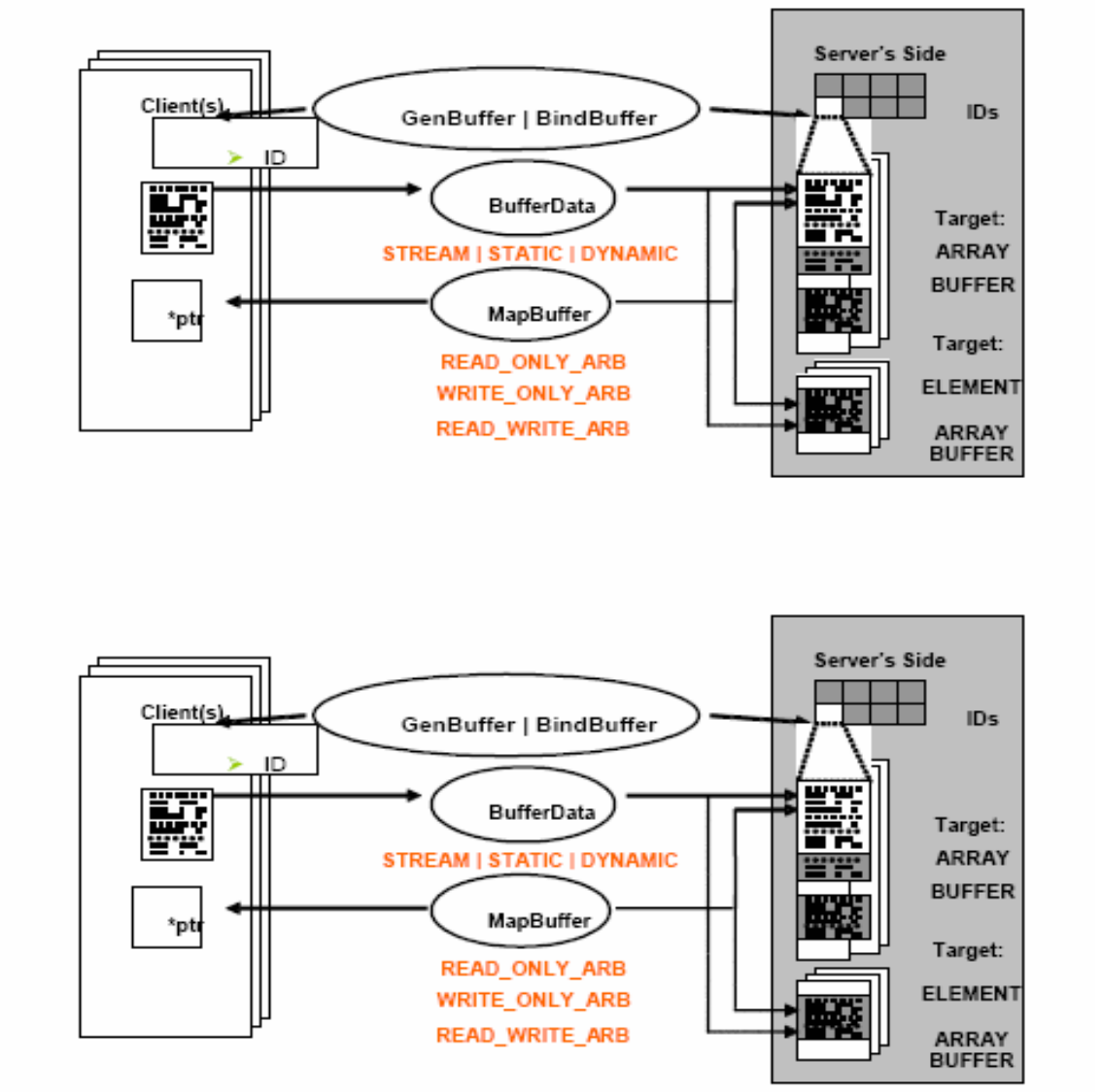
5.1.4 Vertex Buffer Objects

The vertex buffer object works just like the vertex array, but the difference between the vertex arrays and the vertex buffer objects is that, that the VBO use high-performance memory of the graphics card instead of the regular memory. This avoids the data transfer and hence reduces the rendering time.

Here some chunks of memory which can be accessed through identifiers are available. Just like in textures, buffers can be bound to them. After the buffer is bound, all the pointers in these functions become offsets. GL_ARB_Vertex_Buffer_Object extension in OpenGL provides this facility. VBO allows the sharing of the buffer objects with many clients. Multiple clients will be able to access the buffer, as it is on the server side.

The steps in creating the vertex buffer objects are, first creating a new buffer object by calling the API glGenBufferARB(). Next the buffer has to be bound with the call glBindBufferARB(). Copy the data into the buffer with the glBufferDataARB() API. Since vertex buffer objects are similar to vertex arrays, glBindBufferARB() is the only additional API required to draw the VBO.

Vertex buffer object also allows the data to be read and changed by the clients whereas the display lists does not allow this [21]. The data in the VBO can be updated by copying new data again into the bound buffer by using the glBufferDataARB(). After the usage of the VBO, it should be turned off, so as to allow normal vertex array operations.



[21] Figure 5.1.4 demonstrates the usage of VBOs

5.1.5 Pthread

As the regular code written is sequential, meaning that one instruction is executed after another even if more resources are available, the performance of an application can be degraded by this, especially if there is a blocking call in it. Hence to improve the performance, a thread can be implemented. A thread is nothing but a sequence of instructions which can be executed in parallel with other threads. They are not exactly processes, but a lightweight thread processes. They are called lightweight as they are a part of a currently running process.

Pthreads are represented by the type pthread_t. A thread is created by calling the API pthread_create with thread attributes defined in *thread_attributes.* A function *thread_function* is passed to the thread. This function contains the code for the thread execution. After the pthread_create is called, the function *thread_function* starts executing. To initialize the pthread attributes, condition variables and mutexes, the API pthread_XXXX_init() is used.

Sometimes different threads try to access and make changes to the same data. By this the consistency of the data is not maintained. Hence each thread has to access the data individually in order to preserve the data consistency. Therefore a thread has to acquire a lock on the data, make changes to it and then unlock it. To perform such locks and unlocks on these functions, APIs like pthread_mutex_lock(), pthread_mutex_trylock() and pthread_mutex_unlock() are available. The lock is a blocking function. If the thread cannot gain the lock, then the execution of the function is stopped until the lock is obtained. These functions are used when a globally shared

variable or data structure is being updated. The code between the lock and the unlock is the critical section.

5.2 The approach

The thesis consists of two parts. In both the parts the vascular tree dataset has been visualized using geometric primitives, but in each case a different primitive. In the first method the texture being mapped onto the blood vessel's wall is very detailed. This method is more suited where the results need to be enlarged and very small parts examined. To obtain results with this method, quite some time is required. As the time was not very promising, a different primitive was used in the second method, which is much simpler and consumes much lesser time to compute than the first method.

In the first part, the wall of each segment, which is a small part of a blood vessel, has been drawn with a number of triangles to make it look like a cylindrical structure. As the data being visualized might be very large, the sizes extending up to several gigabytes, this method takes some time to render all the segments in the dataset, as the amount of computation will be huge for large datasets. Hence another method was implemented to reduce the computation in order to make the application faster and more efficient.

In the second part, like the previous method a geometric primitive is used to render the segments of the dataset. All other parts in this approach are similar to the previous method except that the primitive, instead of consisting of several triangles which were being drawn as triangle stripes, now consists of only a single quad. For example, the number of vertices computed in the previous method was nearly 128 vertices per segment whereas this method requires as little as 4 vertices per segment. While visualizing huge

datasets this drastically reduces the amount of computation, thereby reducing the time being taken. Here a second fragment shader is implemented to apply coloring effects to it and give a much realistic look to the result.

5.2.1 Visualization using the first method

5.2.1.1 The Geometry

As mentioned earlier, the geometry used here is made up of triangle strips. Every blood vessel is made up of several segments. These segments are divided into a number of sectors. Depending upon the number of sectors the segment has been divided into, the number of triangles in the texture calculation increases. If the segment has more sectors then it looks more fine-grained and vice versa. Now triangles are drawn on the wall of the segment depending upon the numbers of sectors the segment has. The texture basically consists of two vertex values stored in an array, which is mapped at every pixel in the primitive and triangle strips drawn to join them forming the geometry. This computed geometry is used as the wall of the segments.

Other than the texture which contains the triangle strip values, another texture which contains the tree data is sent to the GPU. This texture basically contains all the information about the segments from the actual data, like the center points of the segment, the radius at each point and the spanning vectors. These values are put in an array and mapped as a texture.

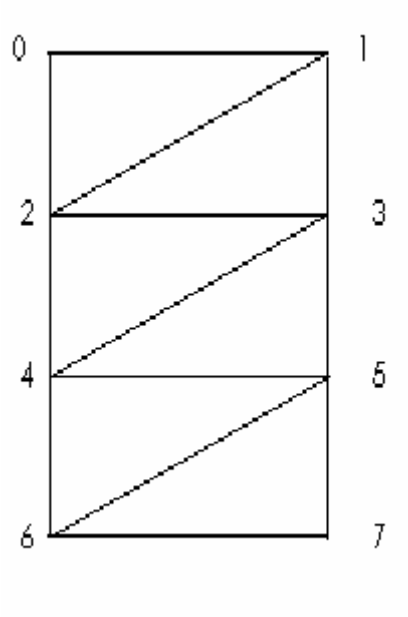
Both the textures have to be scaled before sending them to the GPU, so that their values range between 0 and 1. Each value is divided by its maximum possible value to reduce it to a smaller value. After extracting the values on the GPU, they are scaled back to get the actual value.

The code of the triangle strip texture looks as follows:

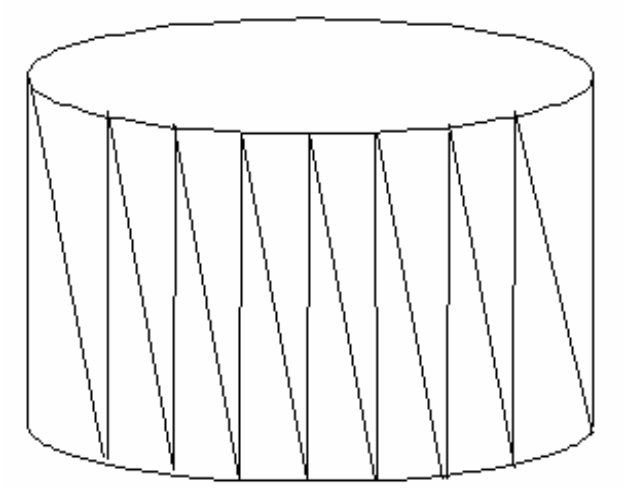
```
for(int i=0; i<sectors; i++) {

  sine = sin (i*2.0*PI/( sectors -1));
   cosine = cos (i*2.0*PI/( sectors -1));
   *pointer = sine;
   pointer++;
   *pointer = cosine;
   pointer++;
   *pointer = 0.0;
   pointer++;
   *pointer = 0.0;
   pointer++;
   *pointer = 0.0;
   pointer++;
   *pointer = 0.0;
   pointer++;
   *pointer = sine;
   pointer++;
```

```
  *pointer = cosine;
  pointer++;
}
```

The triangle strip texture looks like the figure 5.2.1.1 (a):



The texture after applying to the segment looks like the figure 5.2.1.1 (b):



5.2.1.2 CPU-GPU Data Transfer

There are two methods to transfer data between CPU's main memory and GPU's graphics memory. Our approach uses the conventional method which consists of converting a double precision array to a single precision array. A texture is previously allocated and the call to glTexSubImage2D() copies the data to it. The array data from the CPU is mapped row-wise onto the textures on the GPU. The values of the array before being mapped to the texture is scaled between 0 and 1, as that is the range of the texture color values.

The data read backs from graphics memory to the main memory in this method are the reverse of the above method, but not actually used in our approach. Here a texture attached to the FBO is used as the source of the read operations. The data from the texture is copied to a single precision temporary array on the CPU which can be converted back to double precision array.

Another way to transfer the data from CPU is transferred to GPU is by using PBO to accelerate the transfer speed. The array data from CPU will be transferred to GPU as textures just like the conventional method. Before transferring the data the texture has to be bound to a target using the call glBindBuffer(). The data to this buffer is initially passed as NULL to invalidate it and also the size of the buffer is declared. The array of data has to be dimensioned correctly and the array passed to the function as a pointer. After this the data is copied into a chunk of memory in the GPU. The call to glTexSubImage() API enables the shader to use the data in the texture by sourcing the

buffer which is still bound. Here an offset inside the buffer is passed on instead of a pointer to an array in the memory.

The method used in this thesis is the PBO-accelerated method. This method enables fast texture upload/download. In this case, the texture to be read is already in a FBO attachment, so we perform the technique to redirect the pointer as follows:

glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);

The call to glReadPixels() is used to copy the data from frame buffer object to the vertex buffer object. The data is blocked by the call until the data is available. Then the buffer object is unbound to return the memory control back to the driver.

5.2.1.3 Computation on the GPU

As mentioned earlier, the increasing programmability of the graphics processing unit and their high speed have made it possible for the GPU to be used as a co-processor to the CPU. In some cases they are even used as a substitute for the CPU. They can compute more than just the graphics for which they were designed. This is a result of the added programmable stages and higher precision arithmetic to the pipelines. By exploiting the GPU's architecture, the time taken by many applications can be decreased considerably. The goal of this thesis is to make use of the programmability of the graphics hardware to compute the texture coordinates using a fragment shader. No vertex shader has been used here, since the texture has to be applied to each and every pixel in

the primitive. As we don’t have to move the vertices here, per-pixel calculations are required and not per-vertex.

The fragment shader is executed for every pixel in the primitive. Here, the primitive used is a quadrilateral. The dimension of the quadrilateral is two in width and the number of sectors the cylinder has been divided into, the height. The triangle strip texture is mapped onto every pixel in the quadrilateral. Once the fragment shader goes through all the pixels in the segment, all the points for the texture is calculated and written into a frame buffer object.

The information about the vascular tree structure, that is, about the segments, is contained in another texture. The data corresponding to the segment being executed is extracted from this texture. After extracting, the triangle strip texture is calculated for this segment and the texture applied to it.

Both the textures are in the RGBA format, hence the color value extracted at any point in the texture returns four values, which is stored in a temporary vector. The function call tex2D() is used to extract the values from a texture. The first parameter passed to this function is the texture which contains the values and the second parameter is a vector of two values, mentioning the x and y coordinates at which the value has to be extracted from the texture. Once the values are obtained, they are scaled back to the original value.

5.2.1.4 Rendering the GPU-Computed Data

Now the vertex values calculated by the fragment shader for the texture are stored in the frame buffer object. This rendering to the frame buffer object is also known as off-screen rendering. These values are copied from the frame buffer object to the vertex buffer object. The glReadBuffer() call is used to direct the pointer to the corresponding frame buffer object. glBindBuffer() is used to bind the required vertex buffer object. The call to glReadPixels() actually copies the data into the vertex buffer object, which resides on the graphics card's memory. The length, breadth, the starting point and also the format of the data to be copied has to be specified for the data to be transferred correctly.

Once the data is transferred, the vertices in the vertex buffer object are rendered on to the screen. The rendering of the vertex buffer object is almost same as rendering of vertex arrays. Here the pointer to vertex array is an offset inside the bound buffer object. The call to glBindBufferARB() lets the usage of the buffer object. The client-side capabilities are by default disabled, it has to be enabled with a call to glEnableClientState(). Next the offset pointer and the data format has to be specified using the API glVertexPointer(). Finally the glDrawArrays() is used to draw the sequence of triangles from the specified vertex array. The vertex buffer object is turned off after its usage, in order to activate the normal vertex arrays. This is done with a call to glBindBufferARB() with 0 as the buffer object.

5.2.2 Visualization using the second method

We can achieve the same results using another method, but the only difference between the former method and this method is that, that this method uses a different geometric representation, which is explained in the below section. The rest of the approach remains the same. Even in this method a fragment shader is used to calculate the geometry, the results of which are rendered to a frame buffer object. From the frame buffer object the calculated vertices are copied to the vertex buffer object using the same steps as the previous method. Finally the vertices are displayed on the screen to visualize the vascular tree.

Here too, like the previous method, the tree data which contains the information about each segment like the center points and the radii, is mapped into a texture to be sent to the GPU.

5.2.2.1 The Geometry

Here the primitive used to draw the segment is just a quad. The calculation of the four vertices of the quad requires a lot less computation and time. Hence this method results in faster rendering.

The fragment shader is executed four times for every segment in the data set. Each time a vertex of the quad is calculated. To obtain the quad, an offset is calculated for the two center points of each segment. The offset is added to and subtracted from the center points to get the four vertices of the quad. The offset is the vector product of the center point and the camera coordinates.

5.2.3 Load Balancing

Load balancing is a technique to share work between many processes or processors. This is implemented in order more work done in the same time or to decrease the computation time and also to get the optimal resource utilization. The computational load can be balanced between hardware, software or its combination. This thesis uses a combination of both to achieve the higher performance.

In order to make the rendering process faster, the CPU capabilities has also been made use of. The entire load has been shared by the two processes to reduce the overhead on one processor. As the CPU stays idle while the GPU is processing the segments, to make efficient use of the available resources, a function is implemented which uses the CPU to process the segments. Thus the aim is to use all the computing power and resources available.

The load balancing has been implemented with the help of pthreads. A function has been implemented which processes the segments just like the fragment shader does, which uses the CPU to do the computation. A global pointer points to the segment which has to be processed next and the CPU or the GPU whichever finishes the processing of the previous segment takes up the next segment’s computation. The results from the CPU are stored in a list. After rendering the VBO containing the GPU results, the list is checked for updated segments, which are rendered on to the screen as vertex arrays.

6. RESULTS

6.1 Performance

The method was tested against the CPU version of the same algorithm. Both the versions were tested on a system with Intel Core 2 Duo processor with 2 GB of main memory. The system was equipped with Nvidia GeForce 7800 graphics card.

The geometry to visualize the entire vascular tree consisted of 11 million quads. Our method rendered at an average frame-rate of 0.54 fps. The CPU version rendered at an average frame-rate of 0.34 fps. Figure 6.1 shows the comparison between the frame-rates for the two versions obtained for the full data set.

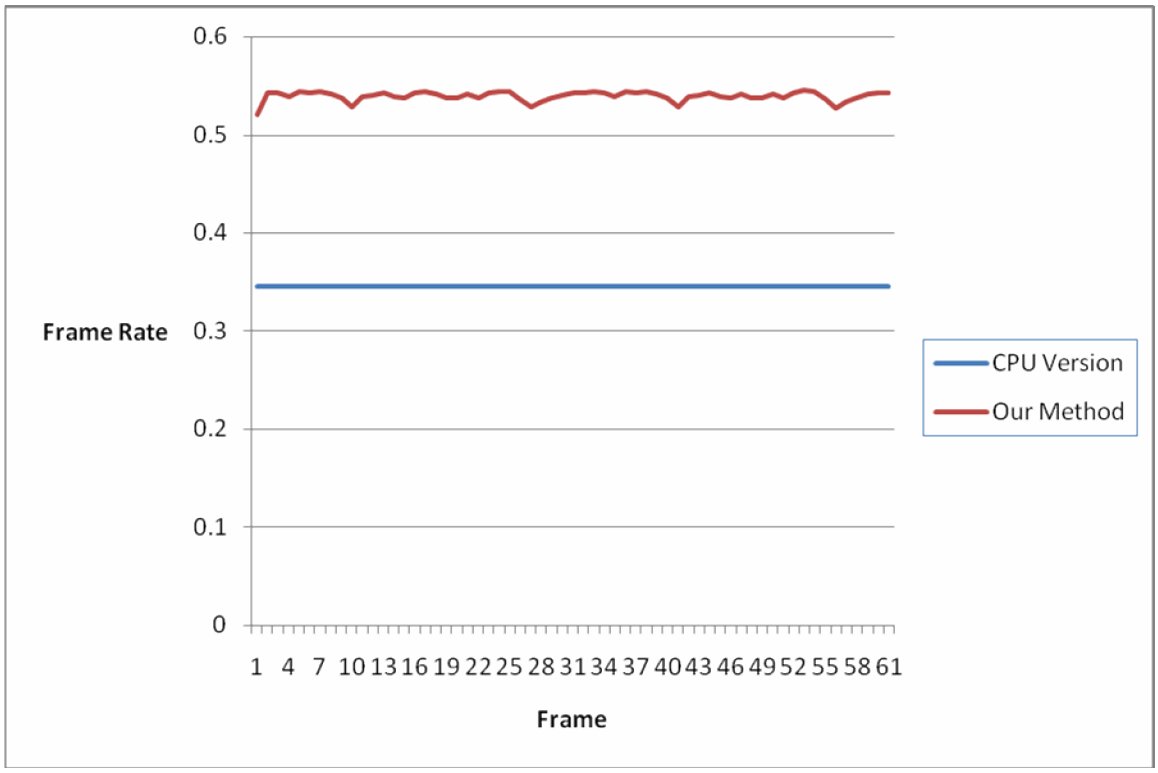


Figure 6.1 Frames per second for the two versions

6.2 Images

The figure 6.2(a) shows the complete vascular tree. It consists of three major branches. Figure 6.2(b) shows the close up view of the tree. The vessel segments in red color are computed by graphics processor and the vessel segments in white are computed by the CPU. The figures 6.2 (c) (d) and (e) shows the three major branches of the heart tree.

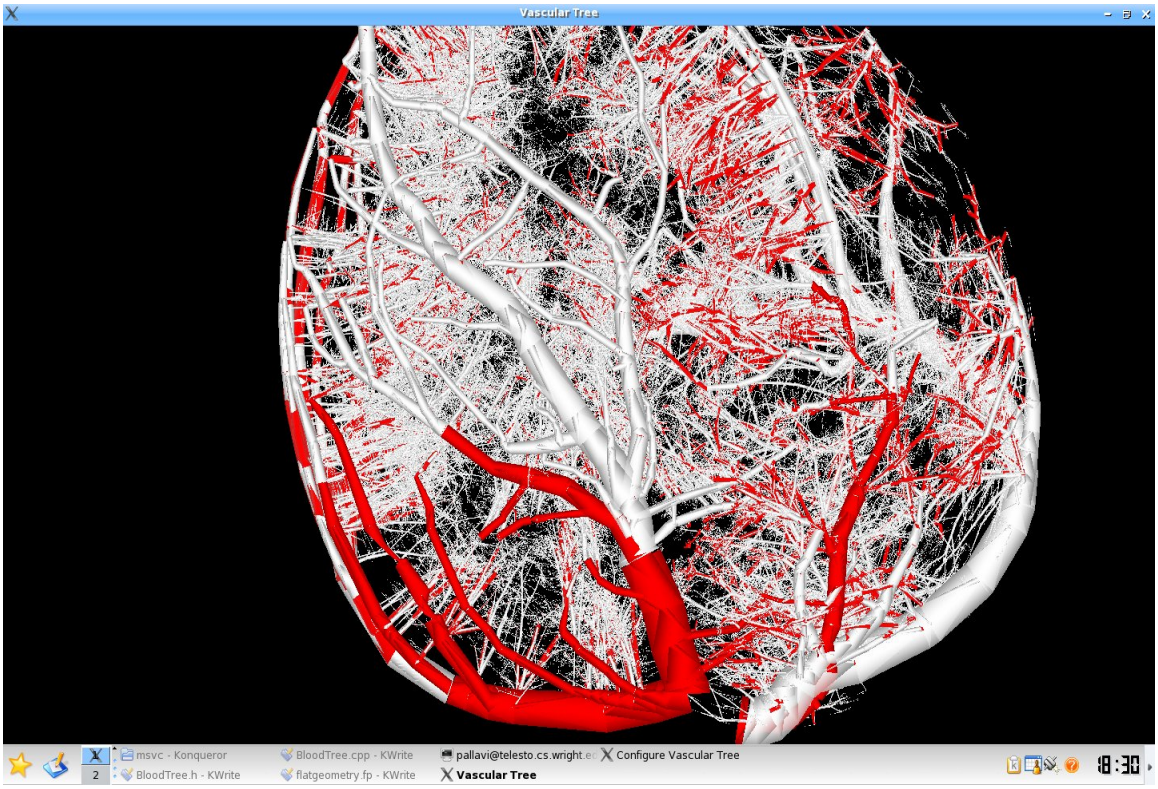


Figure 6.2(a). Complete representation of the vascular tree

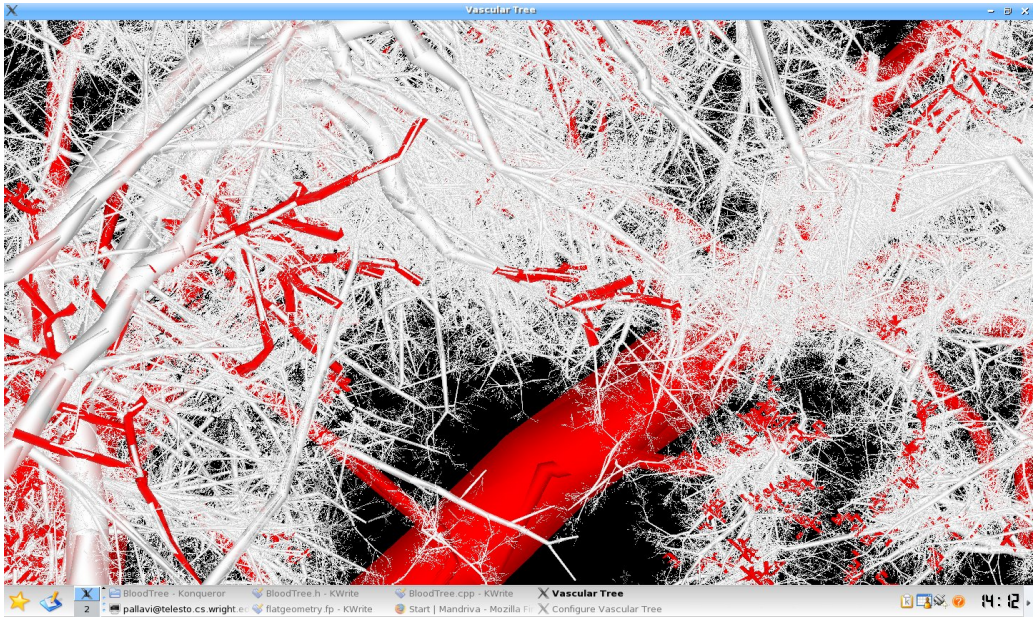


Figure 6.2(b). Shows the complexity of the model

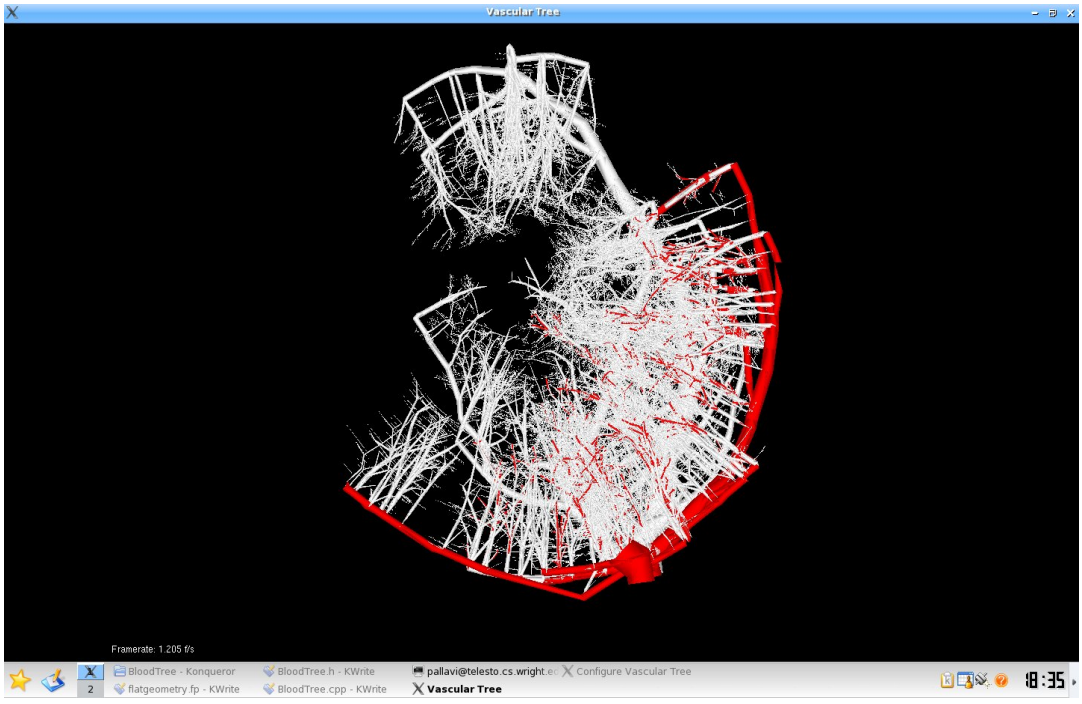


Figure 6.2(c) Visualization of LAD branch

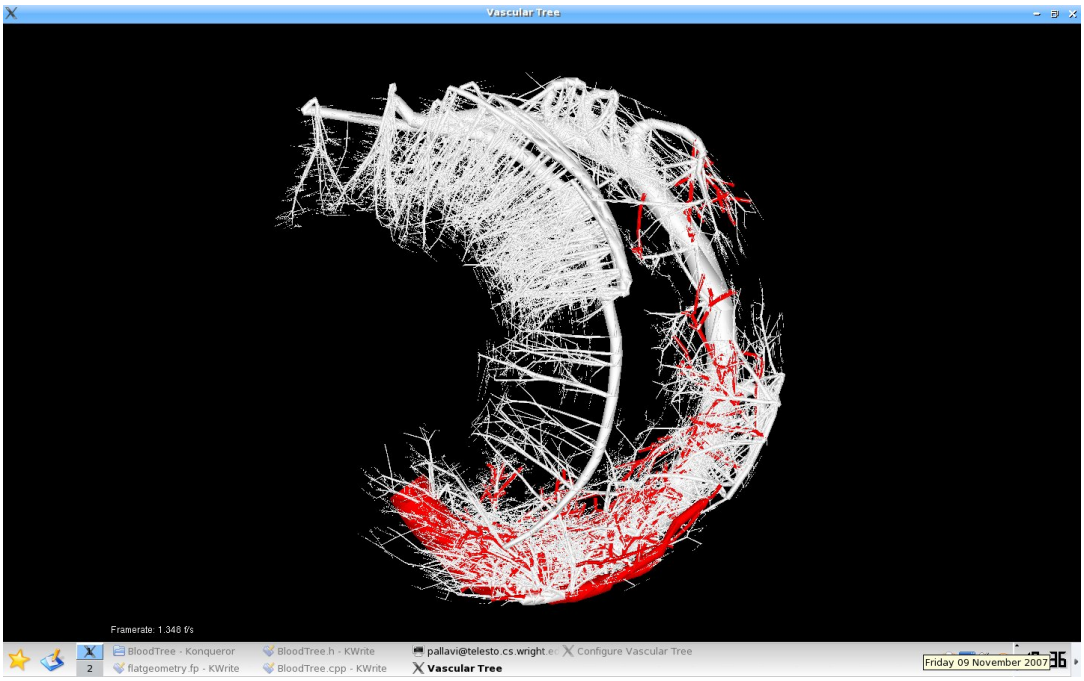


Figure 6.2(d) Visualization of RCA branch

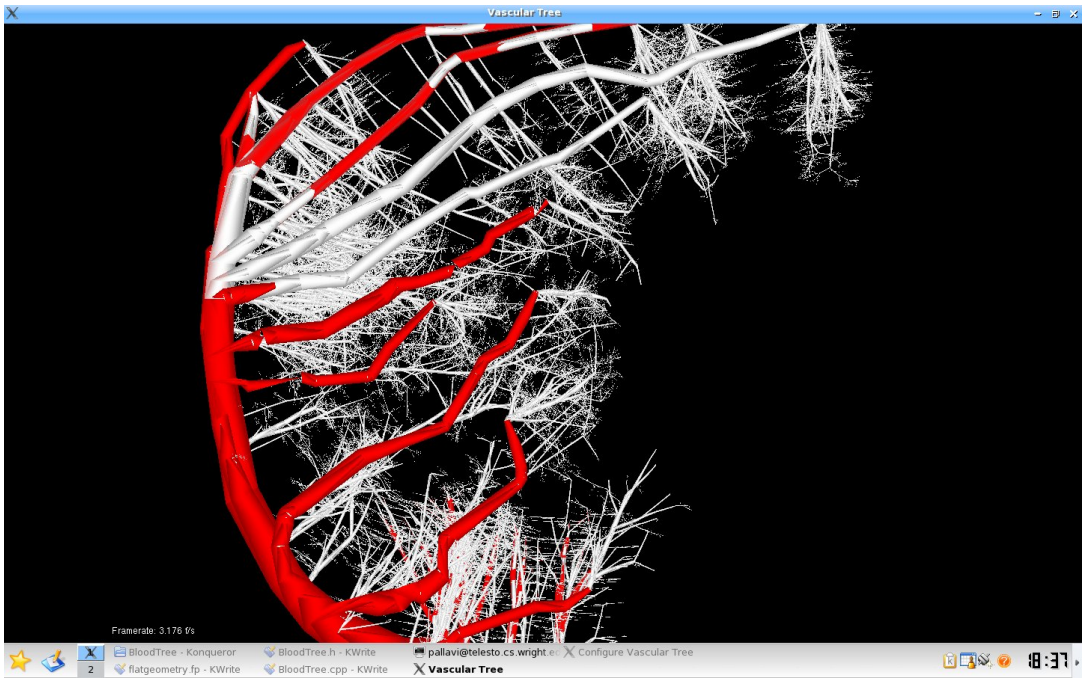


Figure 6.2(e) Visualization of LCX branch

The figure below shows the rendering of a tree data with method 1.

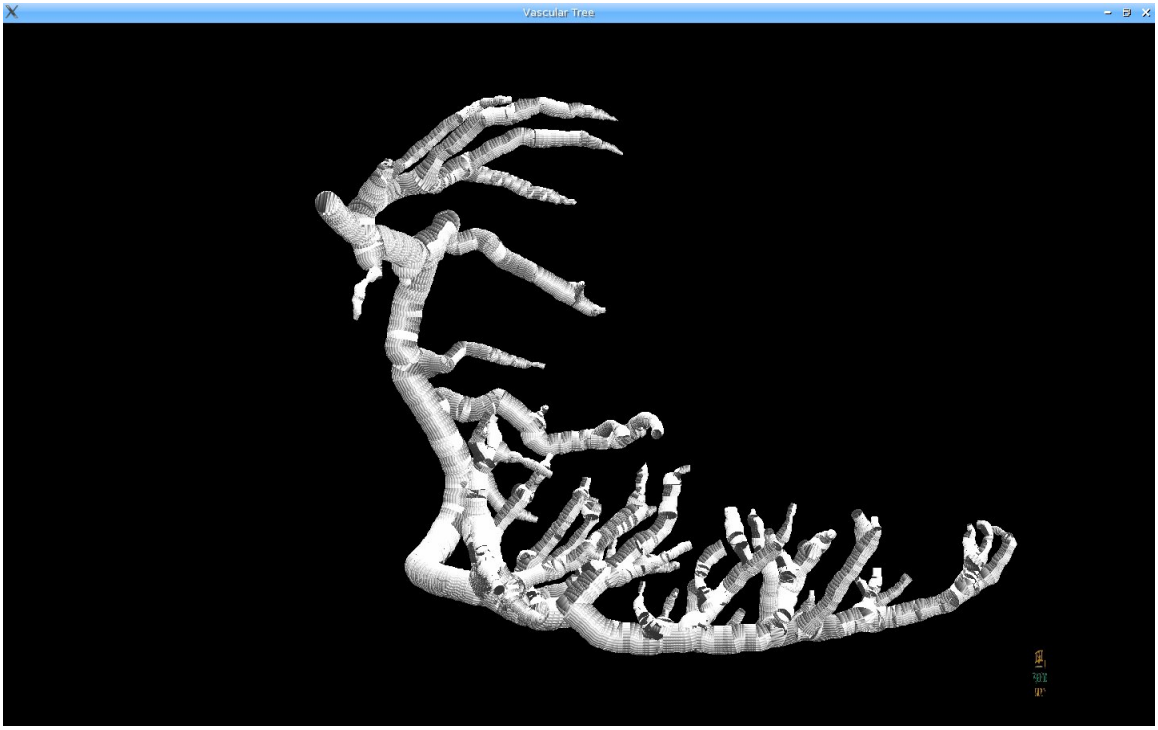


Figure 6.2(f). Rendering of a tree branch using the first method

7. CONCLUSION

A rendering method has been proposed which utilizes the programming capabilities of the graphics hardware along with the usage of the CPU to increase the speed of rendering. The graphics processor based implementation rendered the vasculature of the heart faster than the CPU-based implementation. The load-balancing technique enabled the usage of all the available resources on the system. Using this method the whole vascular tree data has been visualized which consisted of 11 million vessel segments.

REFERENCES

[1] T. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, 1993

[2] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. Proc. of the 1994 symposium on Volume visualization, pp. 91-98, 1994

[3] K. Akeley. Reality Engine Graphics. In Proc. SIGGRAPH'93, volume 27, pp. 109-116, 1993

[4] Kiefer, Gundolf; Lehmann, Helko; Weese, Juergen. Visualization of large medical data sets using memory-optimized CPU and GPU algorithms. 04/2005, Medical Imaging 2005: Visualization, Image-Guided Procedures, and Display.

[5] Yang Heng; Lixu Gu. GPU-based Volume Rendering for Medical Image Visualization**.** Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference.

[6] Qiqi Wang, Yinlong Sun and J. Paul Robinson, Perdue University. Published online Feb. 28, 2007. The International Society for Optical Engineering.

[7] Engek, K., and Ertl, T. Interactive high-quality volume rendering with flexible consumer graphics hardware. Eurographics 2002.

[8] Charles Loop and Jim Blinn. "Real-Time GPU Rendering of Piecewise Algebraic Surfaces". *SIGGRAPH Proceedings*, Boston, 2006

[9] Charles Loop and Jim Blinn. "Resolution Independent Curve Rendering using Programmable Graphics Hardware". *Siggraph Proceedings*, pages 1000-1010, Los Angeles, 2005

[10] Hui Zhan and Jae Choi “High Quality GPU Rendering with Displaced Pixel Shading” Society of Photo-Optical Instrumentation Engineers, Belligham, ETATS-UNIS (2001) (Revue)

[11] Jens Kruger, Ruduger Westermann, "Acceleration Techniques for GPU-based Volume Rendering”, Proceeding of IEEE Visualization 2003, 287-292, 2003

[12] K. Bürger, S. Hertel, J. Krüger, R. Westermann, “GPU Rendering of Secondary Effects” Conference on Vision, Modelling, and Visualization (VMV), *2007 (to appear)* [Bibtex]

[13] Durikovic, R. Ryou Kimura, “GPU Rendering of the Thin Films on Paints with Full Spectrum”, Information Visualization, 2006. IV 2006. Tenth International Conference on.

[14] John Hable and Jarek Rossignac, “Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes”, In Proceedings of Siggraph 2005.

[15] Ivan Viola and Armin Kanitsar, “GPU-based Frequency Domain Volume Rendering”, In proceedings of SCCG 2004

[16] http://prosjekt.ffi.no/unik-4660/lectures04/chapters/jpgfiles/Fragment.jpg

[17]http://images.google.com/imgres?imgurl=http://www.gamedev.net/columns/hardcore/dxshader1/vertex%2520shader.gif&imgrefurl=http://www.gamedev.net/columns/hardcore/dxshader1/page4.asp&h=240&w=374&sz=10&hl=en&start=1&um=1&tbnid=bB39Gz2PJUWIzM:&tbnh=78&tbnw=122&prev=/images%3Fq%3Dvertex%2Bshader%2Barchitecture%26svnum%3D10%26um%3D1%26hl%3Den%26sa%3DG

[18]http://images.google.com/imgres?imgurl=http://www.gamedev.net/columns/hardcore/dxshader3/pixelshaderdiagram.gif&imgrefurl=http://www.gamedev.net/reference/articles/article1820.asp&h=428&w=599&sz=35&hl=en&start=6&tbnid=0_ynPl5_0Xn6xM:&tbnh=96&tbnw=135&prev=/images%3Fq%3DFragment%2Bshader%2Barchitecture%26gbv%3D2%26svnum%3D10%26hl%3Den%26sa%3DG

[19] http://developer.nvidia.com/object/Intro_Vertex_Shaders.html

[20]http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf

[21] http://developer.nvidia.com/object/using_VBOs.html

[22] http://www.nvidia.com/object/feature_vertexshader.html

[23] http://www.gamedev.net/reference/articles/article947.asp[24]http://64.233.167.104/search?q=cache:6pEE6_HnRAUJ:http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf+frame+buffer+object&hl=en&ct=clnk&cd=2&gl=us

[25] Thomas Wischgoll, Joerg Meyer, Benjamin Kaimovitz, Yoram Lanir and Ghassan S. Kassab, “A Novel Method for Visualization of Entire Coronary Arterial Tree”, Annals of Biomedical Engineering 2007, 3 March 2007.

http://vergil.chemistry.gatech.edu/resources/programming/threads.html

http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html#arrays6

http://en.wikipedia.org/wiki/OpenGL

http://www.nvidia.com/object/feature_vertexshader.html

http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial3.html#conventionaldownload

http://www.devmaster.net/forums/showthread.php?t=7457

http://www.opengl.org/resources/faq/technical/displaylist.htm

http://www.codecolony.de/docs/VertexArrays.htm

http://www.songho.ca/opengl/gl_vbo.html#draw

http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=45