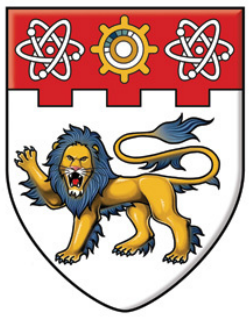


NANYANG TECHNOLOGICAL UNIVERSITY

VOLUME GRAPHICS SHADERS FOR GPU

MUHAMMAD MOBEEN MOVANIA

SCHOOL OF COMPUTER ENGINEERING

2012

VOLUME GRAPHICS SHADERS FOR GPU

MUHAMMAD MOBEEN MOVANIA

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2012

ACKNOWLEDGEMENT

"Coming together is a beginning. Keeping together is progress. Working together is success."

Henry Ford

We are living in a world of inter-dependencies. Every other living being is dependent on another one for its survival and it is this inter-dependency of the living beings that makes life go on. Same was the situation for my study. All praises be to Almighty ALLAH, who has helped me in emerging with flying colors in my struggle.

I am very fortunate to have Dr. Lin Feng as my advisor who has always been a great helper to me. He is always willing to help me whenever I need it. I would also like to thank Dr. Qian Kemao and Prof. Seah Hock Soon for their help and cooperation during the CelFI project.

My thanks and gratitude extends to my colleagues at the Emerging Research Lab, Ms. Cheong Lee Sing, Mr. Chiew Wei Ming, Mr. Cai Jianping, our lab technicians Ms. Thin Nander Soe and Ms. Ang Ah Giat Linda who have been instrumental throughout my research. I would also like to thank the team at National Cancer Center Singapore (NCCS) namely Dr. Patricia S. P. Thong, Dr. Ramaswamy Bhuvaneswari, Dr. Lucky, Prof. Malini Olivo and Prof. Khee-Chee Soo. Without this collaboration, this research study could not have been materialized. It was a wonderful experience working together with all of them. I thank them all for their support in one way or another.

TABLE OF CONTENTS

List of Tables

List of Figures

SUMMARY

Volume graphics attracts our research interest. On one hand, the challenging tasks in cyberspace applications are to model and render the objects and phenomena with complex properties such as the nonuniform and nonrigid materials in volumetric datasets. On the other hand, graphics algorithms have been developed mainly for rigid object representation, affine transformation and offline rendering. This limits their applications in time-constraint yet complex systems, for example, image-guided surgery and therapy systems. In response, this thesis presents a comprehensive study on modeling and rendering of volumetric graphical objects and the acceleration technologies for the new-generation Graphics Processing Unit (GPU). The research focus is on the novel hardware accelerated solutions by programming shaders on the GPU, including the vertex, tessellation, geometry and fragment shaders.

First, there is a strong demand for integrating the volume visualization and deformation processes in real-time systems. We are interested in an efficient method that can utilize the new features of the modern GPU. One such feature is transform feedback which is a special mode in which the GPU feedbacks the vertices in its own clock cycles. While this mode was initially used for dynamic tessellation and Level-Of-Detail (LOD) rendering, we have exploited this mode for a deformation pipeline implemented entirely on the GPU using transform feedback. Our experimental results reveal that the proposed pipeline outperforms other GPU implementation schemes. Moreover, we can couple such a deformation pipeline with the visualization pipeline seamlessly reducing the amount of data transfer out of the GPU core. Previous approaches suffer from an unbalanced utilization of the graphics pipeline; that is, they are either vertex shader bound or fragment shader bound. On the contrary, since our pipeline uses the vertex shader stage

for deformation and uses the fragment shader stage for volume rendering, it makes a balanced utilization of the programmable graphics pipeline. We apply the proposed pipeline first in a mesh based approach for large deformation and then extend it to a meshless approach for relatively small and accurately controlled deformation.

During deformation and volume rendering, intermediate data has to be obtained. This is usually carried out by volumetric interpolation. We propose an efficient method that uses the underlying gradient of the dataset. This allows us to capture nonlinear intensity variations for high quality volume rendering. As a case study, we applied the proposed interpolation scheme to the confocal laser endomicroscope (CLE) datasets. Experimental results reveal that the proposed interpolation scheme not only performs better (similar to hardware based trilinear interpolation scheme) but also preserves important features which otherwise would be lost when high-order interpolation schemes (such as cubic B-spline) are applied. We also provide an efficient implementation on the fragment shader so that the interpolation could be seamlessly integrated with our visualization system.

Equally important for artifact-free rendering is the accurate identification of features through assignment of transfer functions. Although there have been several approaches for transfer function designing, very few have focused on automated feature identification. We propose a robust and effective GPU based framework for automatic transfer function generation, using the statistical properties of the datasets and locally adaptive thresholding.

Now that high-performance computing systems can rely more on a cloud based infrastructure, it becomes much more important to have ubiquitous data processing and visualization capability. This will allow data sharing among numerous clients using shared data repositories through a secure web server. Thanks to the wide availability of GPU support in today's mobile devices such as smart phones and tablets, as well as the

recently published WebGL standard, pervasive computing for high-quality and real-time volume rendering may be realized on such high-performance platforms. We invented two high-performance volume renderers, namely, single-pass GPU ray caster and fast 3D texture slicer, for both mobile and desktop platforms. Rigorous experiments and performance assessments reveal that the proposed volume renderers outperform the existing WebGL volume renderers by almost 2x speedup.

Finally, we coalesce all of our proposed methods and apply them to an *in vivo* volume graphics tool for diagnostic imaging of the oral cavity using datasets acquired from the CLE. The developed prototype system, CelFI, is helpful in the diagnosis and prognosis of early-stage oral and mucosal cancers. The pilot studies using the prototype system were approved by the Centralized Institutional Review Board of Singapore Health Services Pte Ltd, which shows a great potential of the prototype system for virtual biopsies.

In short, this PhD study contributes in twofold: (a) The proposed modeling and rendering methods contribute to the disciplinary advancement of volume graphics; and (b) the GPU shaders developed in this project have a wide application domain in graphics industries: real-time applications such as 3D image registration in computer-aided surgery, scientific simulation and visualization, computer animation, interactive games and virtual reality. They rely on not only accurate modeling but also real-time rendering of the interactive processes.

1 INTRODUCTION

1.1 Objectives

The tasks of this PhD study are associated with our A-Star/SBIC research grant for "Creating an *In Vivo* Navigational Cellular Fluorescence Imaging System with Dynamically Optimized Endomicroscopy" jointly developed with National Cancer Centre Singapore. The *in vivo* cellular imaging system generates a volumetric dataset of mucosal tissues of live animal models and human volunteers. This requires a real-time 3D image reconstruction and visualization system to catch up with the rate of video stream. Upon obtaining the raw images in realtime, the most difficult part to surmount is the absence of real-time 4D (3D spatial and 1D temporal) visualization of the reconstructed images.

The offline solutions impede their acceptance and usage, as the results are difficult to replicate and archive. Therefore, they are inapplicable to applications that require real-time feedback of the results. Beyond a certain time frame, the 3D volumetric dataset will change due to physiologically based morphology. Thus, the deciding factor of whether to reconstruct certain slices into the 4^{th} dimension (temporal) as oppose to using it to refine the existing 3D spatial volume needs to be identified and considered.

Moreover, in the development of the 3D reconstruction and visualization system, we encountered the problems of nonrigid volumetric deformation inherently exhibited in the soft tissues, due to the physiologically based morphology and/or unstable handling of the hand-held endomicroscopy probe. Hence, for a more comprehensive study, we extend the scope of research to cover both modeling and rendering of objects with nonuniform materials and nonrigid deformation, which can be well fitted into the research area of volume graphics [1-3]. Volume graphics is an important sub-discipline of computer graphics, and has wide applicability in various graphics applications such as biomedical

imaging and visualization. It deals with volume modeling, voxelization, volumetric data interpolation and manipulation, volumetric feature enhancement, and volume rendering.

Lastly, in the past years of PhD study, we noticed that the trend in volume graphics study is towards manipulation and rendering of larger and larger dataset with strict time constraints. This is especially true in our *in vivo* and *in situ* cellular imaging and visualization system. The employed Confocal Laser Endomicroscope (CLE) continuously provides optical slices, that is, the cross-sectional images of the emission and absorption characteristics of the tissue. While an optical slice of the lesion obtained by CLE provides partial information of the microstructures, a concurrent 3D reconstruction would be much more helpful in extracting meaningful inferences.

The locality of different structures and their depth relations can be made easy to analyze and comprehend with the functions of real-time 3D reconstruction, which motivates us to develop a novel computing system that allows massively parallel computation for sophisticated algorithms. Therefore, we propose to study modeling and rendering of volumetric graphical objects and the acceleration technologies for the new-generation Graphics Processing Unit (GPU), especially the novel hardware accelerated solutions by programming shaders on the GPU, including the vertex, tessellation, geometry and fragment shaders.

Putting the above together, the objectives of this PhD study are to (a) propose innovative GPU based modeling and rendering methods to enhance the disciplinary study on volume graphics; and (b) to demonstrate the applications of the developed GPU shaders where the success relies on not only accurate modeling but also real-time rendering of the interactive processes.

1.2 Technical Challenges

The technical challenges rise mainly from the confliction between the need for fast modeling and rendering of complex volumetric datasets and the lack of efficient hardware accelerated graphics algorithms. The first question is how to seamlessly integrate the volume visualization and deformation processes in a real-time system. We would like to explore with the transform feedback mechanism in the modern GPU architecture to couple a deforming transformation pipeline with the volume visualization pipeline. Both mesh based approach for large deformation and meshless approach for accurately controlled small deformation should be studied.

Intermediate data has to be obtained by volumetric interpolation during deformation and volume rendering. Previous hardware based trilinear interpolation and high-order interpolation schemes such as cubic B-spline fail to preserve important volumetric features. We would like to utilize the fragment shader to compute the underlying gradient of the volumetric dataset so that nonlinear intensity variations can be captured for the data reconstruction.

It is also challenging to achieve artifact-free rendering for the accurate identification of features. Although there have been several approaches for the transfer function design, very few have explored with an automated feature identification algorithm. We would like to use the statistical properties of the datasets with locally adaptive thresholding, and to propose a robust and effective GPU based framework for automatic transfer function generation.

And lastly, ubiquitous data processing and visualization capability will allow data sharing among numerous clients using shared data repositories through a secure web server. Opportunities come with the wide availability of GPU support in nowadays mobile devices such as smart phones and tablets, but challenges exist in building a

pervasive computing system for high-quality and real-time volume rendering on such platforms. We would like to devise a high-performance GPU volume renderer for both mobile and desktop platforms compliant with the recently proposed WebGL standard.

In short, with the advent of the modern GPU [4], a good understanding of the GPU architecture may provide opportunities for acceleration of the time-consuming volume modeling and rendering algorithms. Great effort is needed to tackle the technical challenges to exploit the graphics hardware to speed up these algorithms.

1.3 Organization of the Thesis

This first chapter is an introduction to the thesis. It describes the objectives, the proposed approach and the technical challenges in this study. The following chapters are organized into a few specific topics, each with a brief introduction to the technical problems and analytical report on the previous methods. These are followed by our proposed formulations and strategies to tackle specific problems. Experimental results and assessments are provided and they are compared with previous methods in the literature.

Chapter 2 consolidates the background with an in-depth literature survey. A detailed review on volume graphics including volume modeling, interpolation, feature extraction and rendering is presented, and then GPU-based volume graphics is explored.

Chapter 3 details our GPU-based deformation pipeline that leverages the transform feedback mechanism of the modern GPU. We describe the shader designs and the data flow for large deformations on the GPU.

Chapter 4 extends the proposed deformation pipeline in Chapter 3 to a meshless FEM model for small and accurate volumetric deformation. We also integrate this pipeline into the GPU-based cell projection algorithm for seamless link between the deformation transformation and rendering processes.

Chapter 5 presents a GPU-based inter-slice directional interpolation technique. We introduce the technique and also report the experimental results for quality visualization of volumetric datasets.

Chapter 6 describes a novel GPU-based locally adaptive thresholding method for the detection of features in realtime. Problems with the previous methods are analyzed. The implementation details are presented and discussed.

Chapter 7 focuses on the design of ubiquitous volume renderer with WebGL. Two volume rendering approaches, namely, ray casting and 3D texture slicing are evaluated in detail. Our novel contribution for a function blended single-pass GPU-based ray casting algorithm is presented. Experiments and performance assessments on the shaders are given.

Chapter 8 reports our project on visualization of the *in vivo* cellular imaging. We discuss the system design, the pilot studies that were carried out on the system on both animal model and clinical trials on the human volunteers. Experimental results and discussions are presented.

Finally, **Chapter 9** concludes this PhD study and looks into the future research directions.

2 LITERATURE SURVEY

2.1 Overview

In this chapter, we aim at establishment of a framework for volume graphics on GPU through a systematic literature survey. Volume graphics encapsulates modeling, interpolation, feature extraction and rendering of volumetric datasets. Especially, modeling nonrigid and nonuniform volumetric objects is one of our interested areas.

To prepare for innovation on the GPU-based technologies, we investigate into major volume graphics algorithms, and the GPU approach with the programmable graphics pipeline. These algorithms include volume modeling, interpolation, deformation as well as volume rendering. Our focus is primarily on methods for fast and accurately coupled geometric transformation and visualization. Equipped with this knowledge, we will be able to identify the appropriate methods to establish the GPU-based real-time nonrigid volume graphics framework.

Section 2.2 gives a review on volume graphics for modeling, interpolation, feature extraction and rendering. It is then followed by section 2.3 which analyzes the emerging GPU architectures and how they can be used for volume graphics. Previous work on GPU-based algorithms for volumetric deformation, interpolation and rendering are given in section 2.4. And finally, section 2.5 summarizes our findings and suggests some general guidelines for our research and development.

2.2 Volume Graphics for Visualization

Visualization of 3D datasets has been an important research topic for scientific and medical disciplines. The raw data contains a lot of details which is invisible unless proper

visualization techniques are developed. Such visualization methods employ graphical primitives to give alternate views. These views extract detailed information from the raw data.

There are mainly two phases in this process, volume modeling and volume rendering [5, 6]. In volume modeling, the raw data is represented into an intermediate form for downstream visualization data processing. In volume rendering [7, 8], the represented data is then explored to generate an image that can be projected onto screen. In this section, we review the different methods and algorithms that have been proposed in each of these categories.

2.2.1 Volume Modeling

Typically, the volume dataset is represented as a density function $f(\mathbf{x}):R^3 \rightarrow R$**.** The scanning devices sample the volume using a finite number of planes. During modeling, these acquired densities are stored into temporary storage, or texture memory. During volume rendering, the underlying densities stored in the volume dataset are reconstructed from the discretized volume using interpolation [5-8].

There are other methods to represent volume, for example, storing the whole volume into a hierarchical data structure such as an octree [9], with the leaves containing the densities from the original volume dataset. First, the octree is constructed by extracting the bounding box of the input volume. Then, based on a division criterion such as the minimum node size or the average density value in the node, the current node is sub-divided. This division continues recursively until the whole volume is traversed. The advantage with this representation is that the locality of the different densities can be obtained quickly and thus sampling can be speed up during reconstruction as is usually needed in volume rendering. This incurs additional storage requirement. For nominal

datasets, this requirement is manageable. However, for large and out-of-core datasets, such large memory requirements quickly overflow the available memory.

Continuous volumetric models can also be generated from the polygonal data such as a 3D mesh [5, 10]. This is usually carried out by voxelization of the domain of the input mesh. The bounding box of the given polygonal object is extracted. Then, using a sampling box, the mesh is traversed. Based on the given density values inside the sampling box and the current criterion, the densities in the current box are added to the voxelized object. The advantage with this representation is that we can represent any polygonal surface into a 3D object and so can carry out volumetric operations on the voxelized mesh. The disadvantage is that the output of the voxelization is dependent on the size of the sampling box. A smaller sampling box size results in a smoother voxelization, however, the storage requirement increases tremendously [11, 12].

The modeling of volumes has also been represented in the frequency domain. Early approaches include the scheduled Fourier method [13] and the wavelet based morphing method [14], both of which decompose the volume into a set of frequency bands. These unfortunately suffer from high frequency artifacts that are introduced during interpolation.

The volumetric deformations can be modeled using ray deflectors [15, 16], whereby the viewing rays are transformed rather than the volume using translate, rotate, scale and discontinuous operators. The reported studies being limited to spheres, it is difficult to approximate more complex topologies without a very large number of spheres of varying radii.

With the introduction of the 3D texture mapping hardware, the texture mapping based volume modeling and deformation approach is introduced by [17] and [18]. The technique uses a tetrahedral representation as a proxy. An affine transformation is applied using a 4×4 matrix and several continuity constraints are also maintained on the proxy

[18]. This model unfortunately suffers from additional pre-processing time for extraction of the proxy tetrahedral mesh.

Non-physically based methods have also been proposed to represent volumes as chain rings [19]. The deformations in this case are restricted using constraints. However, the underlying model for deformation using this approach is non-physical. Alternatively, the skeleton trees model [20] uses the skeleton representation of the data to act as bones for deformation. This representation unfortunately adds additional preprocessing overhead.

For physically-based modeling, different methods have been proposed which may be broadly classified into mesh based methods and meshless methods [21, 22]. Mesh based methods include finite element methods (FEM), boundary element method (BEM), mass spring methods etc. Meshless methods include smoothed particle hydrodynamics (SPH) and other Lagrangian methods. Rather than listing all of the existing methods, we refer the reader to the survey in [22]. Especially in the context of the medical simulation, these can be broadly classified into mass spring models and FEM models.

2.2.2 Interpolation in Volumetric Datasets

An important step in volume modeling and reconstruction is interpolation whereby the missing data is approximated using known data. The early work is based on shape-based interpolation for multi-dimensional objects [23]. Being based on hard classification, this method is unfortunately not applicable to fuzzy datasets. These datasets are common in medical and scientific applications, for example, microscopic imaging. To reduce the interpolation artifacts, an algorithm for directional 3D interpolation using the local derivative information is developed [24]. Later, this approach is extended using higher order derivatives and the principle curvature by [25] for a layer-based visualization of iso-

surfaces in the volume data. The dependency on the derivatives makes this method very sensitive to noises; therefore, it requires a low-pass filtered dataset.

Later direction is incorporated into the shape-based interpolation [26] which is evaluated using the structure tensor obtained from the Eigen analysis. To minimize the global curvature, the Conjugate Gradient method is applied by [27]. This helps in smoothing out the interpolation, enabling accurate iso-surface extraction. All of the discussed approaches rely on shape-based interpolation one way or the other. However, the identification of shape may not always be feasible especially for the fluorescence datasets such as those acquired from the CLE.

Reconstruction of incomplete signals by nonlinear interpolation has been carried out in several disciplines. Initial approaches use genetic algorithms to extract Fourier coefficients [28]. Interpolation of B-splines is used for gradient estimation to estimate accurate gradients for shading [29] as well as for extraction of features using the adaptive bi-directional flow [30, 31]. Artifacts are removed by normal diffusion along the tangent direction and image features are preserved.

2.2.3 Feature Enhancement for Rendering

The extraction and enhancement of features has been studied in numerous domains particularly in the medical domain for cellular visualization systems [32, 33]. These early systems used graphics hardware for rendering of cellular datasets acquired by confocal fluorescence microscopy. They unfortunately lack feature extraction capability, thus classification of specific tissues is difficult.

To address this, [34] use texture mapping for visualization of laser scanning confocal microscopy (LSCM) datasets. They propose methods for interactive transfer function generation using the optical properties of the dataset. In a different way, [35] use contour

based surface rendering techniques to visualize the cervical cell nuclei using shape and size features. Unfortunately, this approach relies on hard classification making this approach unsuitable for fuzzy datasets. This is addressed by [36] who utilize the 3D texture mapping hardware for confocal image visualization and clipping using stencil buffers. However, determination of the optimum stencil buffer resolution is difficult and is highly dataset specific.

While prior discussed approaches lacked a framework for simultaneous feature extraction and visualization, this is addressed by [37]. They employ Voronoi tessellation to delineate cell boundaries and use 3D shape interpolation for nuclei shape detection. While screen space Voronoi tessellation is proven effective, it cannot be used as a general solution.

In volume visualization, specific objects are first classified and then assigned unique transfer functions [38]. This assignment is a cumbersome process and it requires a fair amount of knowledge about the dataset. Levoy [38] proposes to use gradient magnitudes based transfer functions to assist in the visualization of specific material boundaries. There are various approaches for automatic/semi-automatic transfer function assignment. These are categorized as image driven and data driven techniques.

Image Driven Techniques

These techniques involve assignment of transfer function using the rendering results as a feedback mechanism. The generated results from an initial assignment are used to guide the generation of the transfer function. Marks et al. [39] propose design galleries which is a semi-automatic image driven method. This method generates a large number of renderings by adjusting various parameters. The user then selects the images which serve

his/her purpose the most. A similar study using the stochastic search techniques is proposed by [40].

Another image based semi-automatic transfer function approach is proposed by [41]. The user is given a set of parameters to adjust. These parameters are input into the 3D image processing tool using which the transfer function is finally generated. The transfer functions are themselves represented as a set of intensity mappings using lookup tables and neighborhood operations. A similar approach is used by [42] but they incorporate multi-dimensional feature space. The image based techniques have a limited applicability due to their slow convergence. This makes their application difficult in a clinical setting.

Data Driven Techniques

One of the first data driven methods is attributed to [43]. They propose a semi-automatic way of using first and second order directional derivatives. Rezk-Salama et al. [44] use the dynamic programming approach to develop a template based automatic transfer function. They extend the approach of Kindlmann by using the directional derivatives along the gradient direction. A position function is then defined using the two gradient functions. This gives the average distance of a data voxel to the boundary. Application specific templates are then assigned and adjusted to specific requirements.

Hladuvka et al. [45] identified a new domain for transfer function by using the principle curvature magnitudes which helps to highlight specific shape classes. Their method of curvature estimation finds a set of planar curves. A similar study is carried out by [46]; however, they estimate curvature using the Hessian matrix with convolution based reconstruction and extensions for rendering silhouettes and contours.

While the above approaches relied on gradient and curvature estimates for transfer function estimation, none of these worked on colored cryosections images. Ebert et al.

[47] then propose to apply transfer function on colored cryosections. Since RGB color space suffers from nonlinearities due to gamma and hue issues, they use the CIE LUV color space to better capture the nonlinearity of the color space. Even with the CIE LUV color space, minute details are left out. Using logarithmic scale transfer functions [48], such details are easily highlighted. This improves the data centric user interfaces by allowing a fine grained control on the lower opacity ranges. In addition, it also approximates the physical model closely.

While all of the prior approaches relied on 1D transfer functions that related intensities to color and alpha, Kniss et al. introduce multi-dimensional transfer functions by relating the voxel intensities to the gradient magnitudes [49]. These multi-dimensional transfer functions prove effective in isolating structures that are otherwise impossible to extract. However, they lack spatial information, therefore, Roettger et al. [50] augment spatialized transfer functions into the multidimensional transfer functions. They incorporate the spatial information into the histogram volume to aid in classification of objects based on their spatial position. This highlights regions with material boundaries and suppresses relatively uniform regions.

Adding another dimension certainly introduce more problems since multi-dimensional transfer functions are difficult to tune and control. Tzeng et al. [51] provide a machine learning approach to create transfer functions automatically. The user paints relevant voxels on the 2D slices. The identified voxels are used in an iterative training process for classification of the whole volume.

An alternate approach uses distances instead of gradient magnitudes [52]. The distances are estimated from the distance maps, the calculation of which has considerable pre-processing overhead. The introduction of multi-dimensional transfer functions certainly paved way for several higher level interfaces for transfer function designing

including semantics based approach [53], styled transfer functions [54] and stroke based transfer functions [55] all of which simplify the designing of higher dimensional transfer functions.

2.2.4 Volume Rendering

Visualization of the volumetric datasets using polygonal rendering or sub-division approaches used to dominate the rendering techniques [10]. For polygonal rendering, the density field has to be discretized on a regular or irregular grid. This grid is then sampled to identify the iso-surface representing the underlying structure. This may work in some cases where the dataset does not have fuzzy boundaries. The iso-surface extraction is a Boolean operation which extracts features based on a condition. If the condition is true, the current voxel is considered part of the surface otherwise, it is rejected. This process continues until the entire volume is traversed [10].

In a visualization session, various rendering parameters should be adjusted at runtime. These include the assignment of color and opacity transfer functions. Such runtime modification of parameters is difficult in the case of polygonal rendering. To extract regions of interests (ROIs), the scalar field traversal has to be repeated with a different iso-value. Furthermore, the entire surface exists as one whole unit, which is assigned a common material property making identification of specific regions difficult. Volume rendering is attractive in this aspect as well. Transfer functions can be designed to highlight specific densities.

Volume rendering is such a tool within the visualization toolset that makes complex data visually meaningful. Most of the data is from the scientific experiments such as fluid flow simulation data, data arising from the finite element methods etc. In the biomedical imaging domain, data is mostly acquired using various cross-sectional imaging modalities

such as CT, MRI and confocal microscopy. Volume rendering has become more and more the method of choice alongside polygonal or sub-division algorithms. The reason for this is the highly accurate rendering results.

Volume rendering also possesses special features such as the ability to reveal intricate details that are difficult to visualize by any other method [3]. It enables detailed medical visualization and diagnosis, visualization of natural phenomena such as fire and smoke, as well as visualization of scientific data, data from a fluid simulation experiment or seismic data for prediction of earthquake etc. Volume rendering also provides the possibility of going into the data in order to select the best viewpoint. Polygonal data only shows the interface of the object from the viewpoints outside, missing the important details about the actual contents inside.

A sequence of operations is applied (in a specific order) on the acquired data to generate the volume rendering. This series of operations is in general termed as the *volume rendering pipeline* [1]. A typical pipeline is illustrated in Figure 2-1.

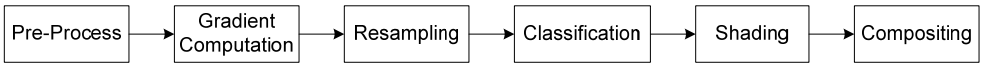


Figure 2-1. A typical volume rendering pipeline

The process starts by applying a pre-process on the acquired data which may include noise reduction with a low pass filter on the data. It may also involve segmentation to isolate regions of interest in the volume. Gradients of the volume data are usually computed using some finite difference methods. Following it, a resampling process may be applied to obtain the intensity values at intermediate positions within the dataset using some interpolation techniques. Then, classification may be carried out by applying

transfer functions to the dataset. The classified voxels are shaded using a desired shading model. Finally, the resulting voxels are composited in depth order to obtain the rendering.

There are two different ways to categorize the volume rendering techniques. The first way is based on which coordinate space the volume method is evaluated in, that is, the object-order (evaluated in the object space) or image-order (evaluated in the screen space) volume rendering. In the object-order approaches (also called feed-forward methods), each voxel is projected onto the screen; for example, cell projection, splatting, 2D/3D texture slicing and shear warp methods fall into this category. In the image-order algorithms, for each pixel on the screen, the contributing color is obtained through sampling a ray into the volume. The object properties at the intersection point are stored along the ray. Typical example of this method is ray casting.

Moreover, based on technical differences, the second classifies the volume techniques on how the volume is converted into the final representation. This can be either direct or indirect volume rendering [2, 3]. Direct volume rendering refers to a group of algorithms that modify the volume dataset directly into a render able form. One of such methods is ray casting whereby the contributions of individual voxels are composited to obtain the final rendition. Indirect volume rendering, in contrast, converts the volume dataset into an intermediate form (mostly polygons) and then performs the final rendition. Examples of this method include iso-surface extraction methods, cell projection and 3D texture slicing etc.

Direct Volume Rendering

While most of the previous literature on volume rendering relied on using offline visualization techniques with imaging modalities such as CT and MRI scans, real-time online volume rendering remains relatively untouched. In addition, almost all of the

previous work has focused on rigid volume graphics. With the emergence of the *in vivo* and *in situ* imaging technologies such as the CLE, there is a strong demand for innovative methods that are specifically geared towards real-time online nonrigid visualization of these datasets.

One of the earliest methods for generating images from 3D density functions / images is the ray tracing algorithm. Kajiya et al. [56] use ray tracing to approximate natural phenomena such as smoke and clouds using an absorption, emission and scattering model. While Kajiya et al. focused on density function, Garrity [57] develop an algorithm for ray tracing of convex cells with shared faces. Since the method uses linear interpolation to detect the intermediate samples, it compromises the quality of rendering. Therefore, [58] describe a simple and fast ray casting algorithm which projects convex polygons in screen space where the ray intersections are performed. Finally, color and opacity values are interpolated in screen space. Since this technique assumes a constant or near constant ray segment length for interpolation, it is unfortunately not applicable to nonlinear projections.

Realizing this, [59] introduce a two pass perspective ray casting approach that utilizes coherency between adjacent frames to optimize the renderings. Intermediate results for each segment are applied on rectangular proxy geometry. Later, the rectangles are alpha blended to approximate the volume integral on the hardware. Being a global image order approach, the performance remains dependent on the output resolution. To circumvent this, [60] describe a fast object order ray casting approach that projects the unstructured voxel mesh and then performs ray integration on the projected profiles. However, this method is only restricted to orthogonal projection.

Since ray casting is a very demanding process, prior approaches minimize the ray sampling overhead by reducing the problem to screen space using projection. This

unfortunately compromises the rendering quality. To sustain rendering quality along with fast sampling, Westermann et al. [61] introduce acceleration techniques for ray casting on the CPU using 2D textures and hybrid CPU/hardware based volume ray casting. Similarly, for efficient ray sampling, [62] describe an object order ray casting algorithm using hierarchical occlusion maps (HOM) so that needless sampling in occluded regions can be reduced. Relying on occlusion maps restricts the application of this technique to non pre-integrated algorithms only.

Indirect Volume Rendering

As discussed earlier, all volume rendering algorithms that use an intermediate representation for volume rendering come in this category. The notable examples in this class include 3D texture slicing and cell projection.

Texture slicing tries to approximate the volume rendering integral by applying texture maps onto the proxy geometry. Typically, these are rectangular [63, 64] or spherical slices [9]. In the case of rectangular slicing, these slices can be oriented either parallel to one of the principle axis (object aligned slicing) or perpendicular to the viewing direction (view-aligned slicing).

The earliest work is attributed to Culip et al. [63] who introduce object aligned slicing using a stack of 2D images. Unfortunately, the method is memory intensive and it suffers from popping artifacts when the stacks are switched based on the viewer's viewing direction. After the introduction of the 3D texturing capabilities, [64] refine this method by introducing view-aligned slicing which removed the popping artifacts. However, it lacked a lighting model, which is introduced by Van Gelder et al. [65] with quantized gradients using lookup tables to accelerate the rendering process. Later, Westermann et al.

[66] introduce clipping geometries with stencil buffers and non-polygonal iso-surface representation to 3D texture slicing.

With the introduction of new extensions for efficient multi-texturing support in hardware, Meissner et al. [67] use these to shade and classify the volume. Their method allows diffuse shading for semi-transparent volume rendering. Multi-texturing is also utilized by Rezk-Salama et al. [68] to generate intermediate slices on-the-fly for approximating the trilinear filtering.

While all of the discussed texture slicing algorithms use polygonal slices as proxy geometry, they suffer from slicing artifacts. These are reduced by using spherical shells [9] at the expanse of increased computational demands. To reduce which, multi-resolution schemes using octree data structure and the level of detail (LOD) rendering are introduced [9, 69]. After the introduction of multiple hardware texturing units, Kniss et al. [70] developed TREX, an interactive volume rendering system for large datasets. Coupled with multi-dimensional transfer function widgets, this system provides near interactive texture based volume rendering.

The most computationally challenging part of 3D texture slicing is the determination of the clipping points for the planes, to effectively slice the 3D volume perpendicular to the viewing direction. Rezk-Salama et al. [71] propose box plane intersection calculation for the vertex shader of Graphics Processing Unit (GPU). This helps to improve the performance of the application by offloading the CPU and, at the same time, balances the computation between the vertex and fragment stages within the GPU.

Even though slicing produces acceptable results, no matter how many slices are introduced, the slicing artifacts still remain. Some solutions include scaling of the slice opacities or using an additional interpolation in the fragment shader [71]. An alternate scheme is proposed by [72, 73] by reordering of the evaluation of the filter convolution

sum using B-splines and Catmull-Rom splines in a multi-pass approach. These reduce the rendering artifacts considerably.

One reason for artifacts in texture slicing is due to thin proxy geometry. If slabs are used instead and a pre-integration approach is used, such artifacts are removed [74]. Different variants have been proposed for both 2D and 3D pre-integration. The 2D pre-integration is restricted to structured grids whereas for unstructured grids, 3D pre-integration is needed. Guthe et al. [75] present a 2D multi-texture based approach to approximate the 3D pre-integration needed to render unstructured grids.

2.3 The Emerging GPU Technologies

The study on GPU-based algorithms would be incomplete without a brief review of the GPU architectures. We are not to elaborate all of the hardware architectures from both AMD and NVIDIA. Instead, we use the typical NVIDIA technologies to highlight hardware architectures that had significant impacts on the progress of the GPU development over the years. The key features among the earlier GPU architectures are elaborated in Table 2.1. For this study, we will elaborate two notable architectures, namely, the TESLA and Fermi architectures [4].

2.3.1 TESLA Architecture

This architecture was introduced with the GeForce 8 series GPUs from NVIDIA. The GeFroce 8 GPU (G80) is a fully-unified, heavily-threaded, self load-balancing (full time, agnostic of API) shading architecture. In a typical application setting, the G80 front end receives the data and prepares it for rendering. The thread controller then schedules the work to a specific shader cluster which has its own scheduler for data management. The hardware can render one triangle per clock.

Table 2.1. Comparison of various GPU architectures from NVIDIA

Release Year	Manufacturer	Model/Series (Code Name)	Features	Pixel Pipelines	Shortcomings
1992	3DFx	Voodoo Series	Texture mapping in hardware using (TMU)	1	No VGA Controller. No vertex processing
1998	NVIDIA	RIVA TNT	32 bit pixel format, 24-bit Z-Buffer, Improved Texture Filtering with trilinear Filtering, 16MB SDRAM	2	Hardware clocked at lower speed causing the card to heat up.
1999	NVIDIA	GeForce 256	First GPU with T&L support. 1 TMU.	4	Memory bandwidth constrained.
2000-2001	NVIDIA	GeForce 2 (NV15)	2 TMU, High clock rate 200 MHz producing increased fill rate, HDVP support, NSR introduced.	4	Memory bandwidth constrained, inefficient RAM controllers.
2001	NVIDIA	GeForce 3 (NV20)	Introduction of programmable vertex/pixel shaders,	4 x 2	Lower clock speed, low fill rates, no dual monitor support
2002	NVIDIA	GeForce 4 (NV17,18,19, 25,28)	Shader model 1.1 support, DVD playback, Hardware anti-aliasing.	4, 8	Overheating problems and driver issues.
2003	NVIDIA	GeForce FX (NV30)	Shader model 2.0 support, VPE introduced, improved anisotropic filtering,	4, 8	Narrow bus usage, Card could not be clocked higher.
2004	NVIDIA	GeForce 6 (NV40)	Shader Model 3.0, Support for DirectX 9.0 c and OpenGL 2.0, Pure Video functionality, SLI technology,	4, 8, 12,16	Driver issues with DirectX applications on AMD Athlon processors.
2005-2006	NVIDIA	GeForce 7 (NV47/G70)	Fully DirectX 9 vertex and pixel shader compliant, SLI support,	4 – 48	No HDR support on 7100 series
2006-2007	NVIDIA	GeForce 8 (G80, G92, G98)	Unified shader architecture (TESLA), DirectX 10 shader (Shader Model 4.0) OpenGL 2.1	8-128 stream processors (sp)	Only one kernel invocation allowed, cannot detect and correct errors, no double precision support
2008-2009	NVIDIA	GeForce 9/ GTX1xx/2xx/3xx (D9M,D9P,D9E)	Based on TESLA, DirectX 10.1 OpenGL 3.3	64-256 sp	Only one kernel invocation allowed, cannot detect and correct errors, double precision support introduced
2010-2011	NVIDIA	GTX 4xx/4xx/5xx	FERMI Architecture DirectX 11, OpenGL 4.2 (Shader Model 5)	up to 512 sp	Multiple kernel invocations allowed, supports error correcting code (ECC), full double precision support.
2012-2014	NVIDIA	GTX 6xx/7xx	Kepler/Maxwell, DirectX 11.1, PCI Express 3.0	up to 1536 sp	N/A

The shader core is a collection of floating point processors called stream processors (SPs). There are 128 such SPs grouped in clusters of 16. Each of these clusters is further grouped into two pairs of eight. The scheduler then runs the same instruction on each sub-cluster. Each cluster has dedicated register pool. It can also access the global register file and constant texture cache.

Each cluster can process three different thread types (vertex, geometry, and fragment) independently in the same clock cycle. No cluster is specifically tied to vertex, geometry or fragment data for the duration of the chip's operation. Any cluster can perform the necessary work whenever it is needed (the unified shading architecture). This unified shading architecture is what makes G80 GPU, a general purpose GPU (GPGPU). To summarize, here we list the notable features brought forth by the TESLA architecture:

- The first architecture that provides a high level C API (CUDA) for controlling the GPU
- A GPU architecture for single instruction multiple thread (SIMT) execution model
- The separate vertex and fragment pipelines replaced by a unified processor which can perform the vertex/geometry/fragment processing
- A scalar thread processor provided to avoid manually maintaining vector registers by the programmer
- The inter-thread communication, shared memory and barrier synchronization allowed

However, while this is the first step in the right direction, our study shows there are some notable shortcomings in the TESLA architecture. One of these is the lack of full double precision support which is absolutely crucial in scientific and experimental studies. In addition, there is no error correcting code (ECC) support. And also, a bottleneck in performance improvement is the lack of concurrent kernels, thus, at any point, only a single kernel can be run on the GPU. Thus, even if the current thread is waiting for some

data to arrive, the threads may remain idle. These might be the reasons that motivated the design of the Fermi architecture.

2.3.2 Fermi Architecture

After the TESLA architecture, the most significant innovation has been the Fermi architecture. The notable contributions introduced by Fermi architecture include improved double support with up to 4.2x double precision performance, support of ECC, ability to run multiple concurrent kernels, support for fast atomic operations and fast context switching as well as support for true cache hierarchy. In addition, a dual warp scheduler was introduced that allowed two independent warps to dispatch instructions. To summarize, here we list the notable advantages brought forth by the Fermi architecture:

- The stream multi-processors (SM) with 32 CUDA cores per SM (around 4x compared to their predecessors) introduced
- ECC memory support provided along with concurrent kernel execution and out-of-order thread block execution
- Full double precision support
- Flexible partitioning for shared memory and L1 cache for programmers
- Unified address space, fast atomic operations, faster thread context switch and dual overlapped memory transfer engine

In this thesis project, we intend to exploit the modern GPU architecture for acceleration of the modeling and rendering processes.

2.3.3 Paradigm for Parallel Computing

In the past years, NVIDIA introduced a collection of C API called CUDA for easier development of General-Purpose GPU, or GPGPU, applications. The general-purpose execution model of CUDA allows wide spread application of the parallel architecture,

from banking and finance to biomedical imaging and visualization [4]. When using the CUDA API, the GPU is in the *compute mode,* in which, instead of running graphics shaders, the GPU works as a general-purpose processor running tens of thousands of parallel threads.

The approach used by CUDA programming is different from the shader based approach. In the case of CUDA, the CPU program (also called the *host*) is required to call one or more *kernels* on the GPU (also called the *device*). The host has to first transfer the data to the device. This is carried out by several API functions provided by the CUDA API. The data is stored in the device memory which includes the constant memory, texture memory or CUDA arrays.

Once the data has been transferred, the execution configuration is given. This includes the number of blocks per grid and the number of threads per block which are used to define the total number of threads on which the kernel code will be executed. The execution configuration is given at the time of call of the kernel. After the kernel is executed, the modified data has to be copied from the device back to the host. This is again carried out by convenience functions provided by the CUDA API.

The invocation and running of the kernel is similar to the fragment shader in the graphics pipeline. In the shader based approach, however, we have no control of the number of execution units running the fragment shader. This fine-grained control provided by CUDA is a double edged sword. An efficient use of the threads and resources might give unprecedented performance while an inefficient use might even kill the performance advantage offered by CUDA. In addition, it is the responsibility of the programmer to maintain synchronization if there are any in the kernel. In the case of the shader based approach, this synchronization is handled implicitly by the shader compiler.

While there are many algorithms for volume rendering, the most suited for parallel implementation is the ray casting algorithm. Similar to how the volume ray casting is carried out in the shader based approach whereby a full screen quad is rendered to invoke the fragment shader. In the case of CUDA architecture, a similar strategy is used [76]. The device memory contains $N \times M$ pixels where N is the width and M is the height of the screen. The kernel is invoked by passing it the pointer to the device memory to store the output result.

In the kernel, using the built-in *threadIdx* and *blockIdx* registers, the linear index of the current thread is obtained. This is used to obtain the index of the output pixel where the final color will be written to. Using the *threadIdx* and *blockIdx* registers and the current camera position and projection parameters, the ray is generated from the camera position. This ray is then sampled at regular intervals into the volume (which is stored in a 3D texture) until the ray exits the volume or the alpha value of the accumulated color is saturated [76].

Since texture slicing and cell projection algorithms require support of the rasterizer hardware for rendering of the triangular primitives, these are unfortunately not implementable in CUDA unless a custom rasterizer is written. Therefore, the CUDA architecture cannot take advantage of the GPU rasterizer hardware.

We have tried to present the algorithms in a general way so that it should be easy to port the discussed algorithms to CUDA/OpenCL or any other compute API if required. During the course of this PhD study, several design decisions were made regarding the platform for development. GLSL and the modern programmable pipeline turn out to be an ideal tool for development. Several considerations are as follows:

- The algorithms we employed for volume rendering such as 3D texture slicing, ray casting and cell projection are more suited to the modern GPU pipeline instead of the

GPGPU approach of CUDA/OpenCL because all of these require the support of the rasterizer hardware which is unavailable in CUDA/OpenCL.

- For nonrigid transformation, we employ a meshless model which can only be rendered as an unstructured mesh. The fastest method for rendering such meshes is the GPU-based cell projection using the hardware accelerated projected tetrahedra (HAPT) algorithm. As discussed by Maximo et al. [80], for optimum performance, it is better to use CUDA/OpenCL for preprocessing of datasets and use GLSL-based cell projection renderer for the rendering because the amount of time required for pushing data to CUDA/OpenCL amortizes its speed advantage.
- Our meshless model uses the deformation pipeline which leverages the transform feedback mode of the modern GPU. This mode is unfortunately unavailable in CUDA/OpenCL.
- When writing to buffer objects memory through CUDA, we have to map the buffer object to CUDA/OpenCL and then, after the kernel is executed, the buffer is unmapped again. This incurs additional time whereby the OpenGL interop has to synchronize with the compute device. Our experimental results suggested that while CUDA/OpenCL is better at running parallel code, the time required to transfer the data from/to the OpenGL client code to/from the host is prohibitively long.
- We intend to develop a ubiquitous renderer that could be run on all devices with GPUs, for example, smart phones and tablets. Using the shader based approach with the WebGL architecture provides just the right tools.

Based on these considerations, we opt for the GLSL-based approach, as the proposed algorithms will be detailed in the rest of this thesis.

2.4 GPU Technologies in Volume Graphics

In this section, we closely look into the recent advances in GPU-based volume graphics research. Specifically, volumetric deformation, volumetric interpolation and volumetric rendering are discussed.

2.4.1 Acceleration for Deformable Volume Modeling

Initial approaches in volumetric deformation rely on an elastic model using a mass spring system. One of the earliest mass spring system for deformation is attributed to [21] which developed a basic framework for physically-based volumetric deformation. Since the method uses explicit integration, for convergence, very small time step values are required. Moreover, extracting the proper stiffness and damping values for the mesh are a challenging task. In addition, this method does not conserve volume. Therefore, Vassilev et al. [77] introduce support springs from the center of the volume mesh to its surface for volume conservation.

With the popularity of GPU, mass spring methods were implemented on GPU. One of the first mass spring methods for large deformation on the GPU for surgical simulators is attributed to Mosegaard et al. [78]. While the elastic model used is similar to [21], they use Verlet integration scheme implemented in the fragment shader. Using the same GPGPU technique, Georgii et al. [79, 80] implement the mass spring system for soft bodies.

The approach by Mosegaard et al. required transfer of positions at each iteration. Georgii et al. focused on how to minimize this transfer by exploiting the ATI superbuffers extension. This extension enabled a chunk of memory to be used both as a texture and a render target. In addition, they describe two approaches; an edge centric approach (ECA) and a point centric approach (PCA). While the PCA requires less memory and force

accumulation enjoys high floating point precision, the force is calculated twice for each spring. On the contrary, the ECA is independent of the mesh valence. Only 4 passes are required (first three are for force calculation) while the last pass is for time integration. This unfortunately requires more memory and it relies on the low precision alpha blending hardware for force accumulation.

Yunhe Shen et al. [81] addressed the volume conservation problem for GPU based real-time surgery simulation of soft bodies. To achieve this, they use a pressure based volumetric model. Recently, a CUDA based mass spring model is proposed [82]. All of the mass spring models and methods discussed earlier use explicit integration schemes which are only conditionally stable. For unconditional stability, implicit integration can be used, as demonstrated for GPU-based deformation by Tejada et al. [83]. The formulation uses implicit Euler integration coupled with a preconditioned Conjugate Gradient (CG) solver.

Now we will look into various FEM methods that have been proposed for both surgical simulation systems and animation systems. One of the first work to model deformations of curves, surfaces and solids using the linear elastic forces given in terms of the displacement is attributed to [84]. A similar study is carried out by Bro-Nielsen and Cotin [85] who apply the linear FEM to the surgical simulation environment and by James and Pai in an animation system (ArtDefo) [86].

All of the above methods assume linear elasticity and therefore can only handle small displacements. Moreover, these approaches rely on pre-computed responses based on FEM which give runtime speedup. However, these suffer from stability problems in the case of cutting simulation where the pre-computations have to be recomputed. In such applications, linearized strain tensors such as the Cauchy strain tensor are used which are not invariant to rotation and so they introduce artificial increase of the element volume.

In addition, the small strain assumption produces wrong results unless the co-rotational formulation is used [87]. This isolates the per-element rotation matrix when computing the strain. The co-rotational formulation has been applied in real-time systems [88-90].

With the introduction of the increasing computational power, nonlinear FE methods (taking both material and geometric nonlinearities into consideration) have been implemented by [91-95]. The fastest numerical methods for solving FEM systems for deformable bodies are based on the multi-grid scheme [96, 97] with approaches in medical applications for both the tetrahedral [98] and the hexahedral FEM [99].

In addition to the above approaches, explicit nonlinear methods have been proposed using the Lagrangian explicit dynamics [100] which are especially ideal for real-time simulations. This scheme does not require the global stiffness assembly for the whole mesh. Instead, a single stiffness matrix is reused for the entire mesh. Especially with the introduction of the CUDA architecture, the Lagrangian explicit formulation has been applied for both the tetrahedral FEM and the hexahedral FEM [101]. The benefit obtained from this integration scheme is that the equations of motion can be solved explicitly for each degree of freedom separately. This make such schemes suitable for a parallel implementation [102].

The conditional stability of the explicit integration schemes requires the time step value to be very small. In addition, such integration schemes may not be suitable during complex interactions as in surgery simulations and during topological changes. Allard et al. [103] circumvent these cons by proposing an implicitly integrated GPU-based nonlinear co-rotational model for laparoscopic surgery simulator. They use the pre-conditioned Conjugate Gradient (CG) solver for solving the FEM. However, ill-conditioned elements that may be generated during cutting or tearing produce numerical instabilities.

All of the discussed deformation approaches are mesh based. As we will show in chapter 4, such approaches may not be appropriate for a coupled physically based deformation and rendering system since a very large number of finite elements are needed to approximate a volumetric dataset. Moreover, corotated formulation would be required which puts an additional computational burden. On the contrary, the meshless approach seems a good choice in this case due to its flexibility for representing an unstructured mesh. Therefore, we have applied a novel deformation algorithm using the meshless formulation and coupled it to the state-of-the-art volume rendering algorithm. The details of our algorithm are given in chapter 4.

2.4.2 Acceleration for Data Interpolation

Data interpolation is a crucial step during the volume reconstruction stage. Various approaches have been proposed for both linear and nonlinear interpolation. Early nonlinear interpolation methods on GPU use edge directed interpolation [104]. The method tries to direct interpolation in the direction of the edges to maintain edges by using the first and second order derivatives estimated using finite differences similar to the edge detectors of Canny and Marr-Hildreth [104]. This work is later extended to CUDA [105].

A fast nonlinear interpolation algorithm called Nohalo is introduced to reduce the interpolation artifacts [106]. This method first finds the nonlinear gradient of the image. Then, it generates a double density image from the original image by super sampling it by entering new samples mid way between the existing samples. Finally, using the double density image and the gradient, it estimates the new sample values by bilinear interpolation [106]. The main weakness of this method is that it is more suited to natural

images which have continuous gradients otherwise the method boils down to plain bilinear interpolation and hence generates visible artifacts.

The default hardware supported filters (nearest and linear) are insufficient; therefore, [107] and [108] show how to do fast cubic B-spline interpolation on the graphics hardware. It is shown by [109] that a better method to interpolate a cubic B-spline function is to go backwards and extract the B-spline coefficients for a given intensity value using prefiltering and a recursive application of infinite impulse response (IIR) filter. This way, the output is always guaranteed to contain the intensities from the input dataset.

Recently, a CUDA implementation of the IIR filter is proposed by [110, 111] who use prefilter not only to improve the interpolation performance but also to reduce the excessive smoothing especially when higher order interpolation scheme is used. Since most of the extracted coefficients from the prefilter are zero, it is shown by [112] that the IIR filter can be replaced by a finite impulse response (FIR) filter without compromising quality significantly. Thus, the first 30 coefficients extracted from the prefilter are enough to produce a comparable output to an IIR filter.

2.4.3 Acceleration for Volume Rendering

In this section, we review GPU-based techniques in various volume rendering approaches, namely ray casting, 2D/3D texture slicing and cell projection. Equipped with this knowledge, we will be able to formulate a robust framework for nonrigid volume graphics. As discussed in an earlier section, the volume rendering methods may be categorized into direct and indirect volume rendering algorithms.

One volume rendering algorithm in the latter category which is suitable for unstructured datasets is the cell projection algorithm. The principle method, the projected

tetrahedra (PT) is pioneered by Shirley [113] and extended by Roettger et al. [74] by combining iso-surface rendering on the GPU. They add cell rendering using 3D textures and use pre-integrated 2D textures for rendering of multiple shaded iso-surfaces.

The underlying model of PT relies on classification to determine the class of the convex polygon. For efficient classification, Wylie et al. [114] introduce a basis graph analogy to implement PT algorithm on the GPU in their algorithm named GPU Accelerated Tetrahedra Renderer (GATOR). Their method performs classification, decomposition and depth sorting of tetrahedral vertices on the GPU and it relies on the view direction for correct sorting of tetrahedra. To remove the view dependency, Weiler et al. [115] introduce view independent cell projection on the GPU using two approaches, one using the vertex shader and another using the fragment shader. Their approach uses pre-integration similar to that in [75].

While prior cell projection methods pay little attention to the quality of rendering for perspective correct interpolation, [116] refine the PT algorithm by introducing logarithmic scale to the pre-integration table on the GPU. This improves the rendering quality at the expanse of increased storage requirement for the pre-integration table, the size of which is dependent on the classification case. Therefore, [117] introduce partial pre-integration on the GPU to generate a pre-integration table independent of the classification case used.

Another multi-pass technique is introduced, called Hardware Accelerated Ray-Casting (HARC) [118], which uses ray casting on unstructured grids for volume rendering. The accumulation of colors is carried out by rendering intermediate results to a set of textures. Since the integration of the tetrahedra required correct visibility sorting, therefore, for fast sorting, [119] introduce Hardware Assisted Visibility Sorting (HAVS) algorithm. This algorithm is based on a hybrid CPU-GPU scheme that uses the centroid of the tetrahedra

in object space for sorting.

With the introduction of the DirectX 10 hardware, [120] introduce sampling of the tetrahedral elements in local barycentric coordinates using a feed forward pipeline. This minimizes the amount of fragments generated and also utilizes the rasterizer efficiently. The method assumes correct visibility order of the tetrahedral mesh therefore, with every view change, the elements require sorting. To reduce such unnecessary re-sorting, [121] introduce a hybrid GPU-based PT implementation called Projected Tetrahedra with Partial Pre-integration (PTINT). It extends the GATOR algorithm by removing the basis graph redundancy and better approximates the final color. In addition, the Phong shaded iso-surfaces are also introduced using a two-pass approach, which does not require any topological information and it minimizes the CPU-GPU transfer which is a bottleneck in performance critical applications.

With the introduction of the geometry shader pipeline, Maximo et al. propose the hardware accelerated projected tetrahedra (HAPT) algorithm [122]. The algorithm first converts the tetrahedra into point primitives. Then, in the geometry shader, the triangles are generated based on the classification case. Finally, the thick and thin vertices are determined and the fragment shader is invoked. To date, this is the fastest volume rendering algorithm for unstructured datasets.

We have analyzed the fastest volume rendering algorithms, compared their performance on CPU and GPU, and identified the shortcomings of each of these. All of the discussed approaches have their pros and cons. While the underlying principle of all of these techniques is same (that is, to sample the volume dataset), exactly how they do this varies from one method to another. Texture slicing slices the volume using proxy geometry and then samples it by applying a texture map on the proxy geometry. Ray

casting on the other hand samples the volume directly using a ray that probes through the volume dataset.

Cell projection is the method of choice for unstructured datasets. Especially, during nonrigid transformation, such datasets may be produced. Therefore, efficient rendering of unstructured dataset can be carried out by coupling the deforming dataset with the GPU-based cell projection. We will analyze how this coupling can be carried out in chapter 4 where we detail the nonrigid transformation pipeline based on the meshless FEM formulation.

For 3D texture slicing, the view-aligned 3D texture slicing [64], the multi-texturing extensions [68] and for GPU-based slicing on vertex shader, the approach by Rezk-Salama et al. [71] are widely used methods. For GPU-based cell projection, the HAPT algorithm [122] is preferred. For GPU-based ray casting, there are two notable approaches, the multi-pass approach [123] and the single-pass approach [124].

2.5 Summary

In this chapter, we have conducted an analytical review on volume graphics research, specifically, volume modeling, volume interpolation, volume feature enhancement and volume rendering. We studied the relevant works highlighting the pros and cons of the methods. Upon the relevant literature review, we then extended our discussion to the emerging GPU architectures and the current trends in GPU-based volume graphics algorithm development.

Specifically, for all the GPU based volume rendering algorithms, they require the volumetric dataset to be stored in a 3D texture in the GPU texture memory so that the hardware filtering can be used to efficiently sample the volume dataset in the shaders. Due to the way the memory is structured, for efficient texture fetch performance, the data

has to be aligned to the byte boundary in memory. Therefore, additional padding is appended by the graphics driver if the data is not properly aligned to the byte boundary which incurs a performance penalty. Hence, the general rule of thumb is to align the dataset to the nearest byte boundary using zero padding where necessary.

The volume rendering methods have different texture access requirements which vary depending on the sampling rate used in the volume rendering algorithm. In texture slicing, this is the step size used to slice the volume; in ray casting, this is the ray step size; and in cell projection, this is the blending step size. The texture access requirements also vary depending on the rendering mode used, for example, the composite rendering mode takes contributions from more samples for output as compared to the iso-surface mode.

3 A VERTEX SHADER BASED DEFORMATION PIPELINE

3.1 Introduction

We start with the problem of low update rate in rendering deformable models, especially in dealing with a large volumetric dataset. For an interactive visualization experience with these deformable models, efficient algorithms have to be developed. A reasonable strategy is to exploit the massive parallelism exposed by the GPUs to maximum. While prior approaches have focused on using the fragment shader based approach, it becomes difficult to couple real-time visualization with deformation as is usually required in, for example, biomedical imaging systems.

We propose a vertex shader based approach that uses the transform feedback mechanism of the modern GPU. In particular, we are interested in a new method for efficient modeling of deformation processes for the mesh based model. In the transform feedback mode, the GPU feedbacks the transformed vertices in its own clock cycles through the vertex shader. A major advantage with this approach is that the fragment shader can then be used for other tasks such as volume rendering. Therefore, the whole graphics pipeline is efficiently organized. Such a deformation pipeline is particularly suitable for large deformations where geometric parameters have to be updated for maintaining the topological structures of the dataset.

The rest of this chapter is organized as follows. Mathematics of the physical model for a mesh based transformation is presented in section 3.2. The hardware support of transform feedback is detailed in section 3.3. A detailed technical description of our deformation pipeline based on the transform feedback mechanism is given in section 3.4. Relevant shaders for collision detection and response are described in section 3.5.

Experimental results and performance assessments are presented in section 3.6. And finally, section 3.7 concludes this chapter.

3.2 Formulation of Deforming Transformation

There is a common issue between different physical models for deforming transformation, that is, to represent the properties of elasticity and viscoelasticity displayed in the deformable model. Such a model can be represented as a 3D mesh of virtual masses which are linked through mass-less springs. We define three types of topology for the springs (as shown in Figure 3-1):

i. Structural springs that link the node to its immediate neighbor in x, y and z axis only,

ii. Shear springs that connect the remaining neighbors including all of the diagonal links, and

iii. Flexion springs that are structural springs connected to the nodes one node away.

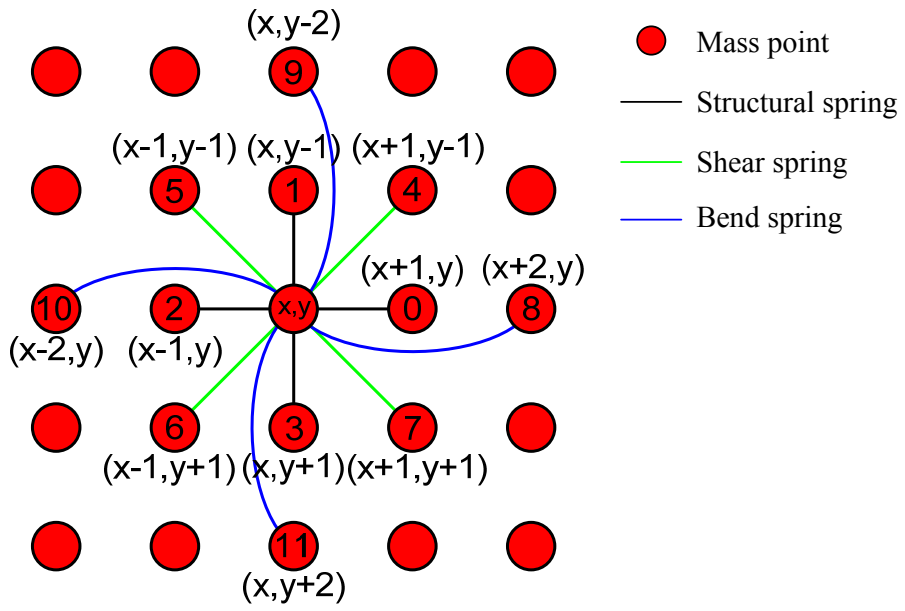


Figure 3-1. Different spring types used in a mass spring system

Each of these springs has different internal and external constraints. For example, shear springs are constrained under pure stress, structural springs under pure compression/traction stress and flexion springs under pure flexion/bending. The voxel mesh is stabilized to equilibrium due to the constant interaction among these springs. In a physical simulation, the points are assigned several physical properties which include mass (m), position (x), velocity (v) and acceleration (a). At any point in time, the system is governed by the following second-order ODE:

$$m\ddot{x} = -c\dot{x} + \sum (f_{\text{int}} + f_{ext}) \tag{3.1}$$

where, c is the damping coefficient, f_{int} is the internal (spring) force and f_{ext} is the external force which may be due to the user's intervention, wind or gravity force, or collision force due to collision of the object with other objects. The spring force f_{int} can be defined as

$$f_{\text{int}} = k_i(\left\|x_i(t) - x_j(t)\right\| - l_i)\frac{x_i(t) - x_j(t)}{\left\|x_i(t) - x_j(t)\right\|}$$

where, k_i is the spring's stiffness, l_i is the resting length of the spring, $\boldsymbol{x_i}$ is the spring's position and $\boldsymbol{x_j}$ is the position of its neighbor.

The system in Eq. (3.1) may be solved using any of the numerical integration schemes. We may use either the explicit integration schemes [21, 79] or the implicit integration schemes. Typical explicit integration schemes include Euler integration, mid-point method (2nd order Runge Kutta), Verlet integration and 4th order Runge Kutta integration; and a widely used implicit integration scheme is the implicit Euler integration [83, 125]. Whichever integration scheme is used, the acceleration (a) can be calculated using the Newton's second law of motion.

$$a_i(t) = \frac{f_i(t)}{m_i}$$

Contrary to the conventional physically based systems, we store per vertex attributes to represent the physical properties. Therefore, each vertex in the mesh stores these properties so that an efficient iterative implementation may be carried out on the GPU using the vertex shader. Our proposed iterative algorithm uses the per vertex attributes to estimate the new positions and velocities. For example, in the case of the explicit Euler integration, the iteration can be formulated as follows:

$$v_i(t+\Delta t) = v_i(t) + \Delta t a_i(t)$$
$$x_i(t+\Delta t) = x_i(t) + \Delta t v_i(t)$$

For the Verlet integration, there is no need to calculate velocity since it can be implicitly obtained from the current and previous position using:

$$x_i(t+\Delta t) = 2x_i(t) - x_i(t-\Delta t) + a_i(t)\Delta t^2 \tag{3.2}$$

For the mid-point Euler method, both acceleration and velocity are evaluated at mid-point, thus the new velocity and positions can be obtained as follows:

$$\begin{aligned} v_i(t+\Delta t) &= v_i(t) + \Delta t a_i(t+\frac{\Delta t}{2}) \\ x_i(t+\Delta t) &= x_i(t) + \Delta t v_i(t+\frac{\Delta t}{2}) \end{aligned} \tag{3.3}$$

For the 4th order RK method, the set of computations are as follows:

$$v_i(t+\Delta t) = v_i(t) + \frac{1}{6}(F_1 + 2(F_2 + F_3) + F_4)$$
$$F_1 = \frac{\Delta t}{2} a_i(t)$$
$$F_2 = \frac{\Delta t}{2}\frac{F_1}{m_i}$$
$$F_3 = \Delta t \frac{F_2}{m_i}$$
$$F_4 = \Delta t \frac{F_3}{m_i}$$

$$x_i(t+\Delta t) = x_i(t) + \frac{1}{6}(K_1 + 2(K_2 + K_3) + K_4)$$
$$K_1 = \frac{\Delta t}{2} a_i(t)$$
$$K_2 = \frac{\Delta t}{2} K_1$$
$$K_3 = \Delta t K_2$$
$$K_4 = \Delta t K_3$$

While all of the explicit integration schemes are straightforward for implementation, they suffer from the problems of instability and require the time step value to be very small. This is because the velocity and position evaluation is carried out explicitly without taking care of the wildly changing derivatives.

On the contrary, the implicit integration schemes are unconditionally stable because the system is solved as a coupled unit. It starts backward in time to find a new position given an output state. The implicit Euler integration can be represented by

$$\Delta x = \Delta t(v_0 + \Delta v)$$
$$\Delta v = \Delta t(M^{-1}F)$$
$$F = f_0 + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial v}\Delta v$$

where, M is the diagonal mass matrix, F is the total force, $\partial f/\partial v$ and $\partial f/\partial x$ are force derivatives with respect to velocity and position respectively. The integration can then be solved using any iterative solver such as the Newton Raphson (Newton) method or the Conjugate Gradient (CG) method.

In general, we can implement both implicit and explicit integration schemes in our deformation transformation pipeline. In this chapter, to concentrate on the architectural design of the GPU pipeline, we use the Verlet integration scheme for explanation. Not only is this scheme second-order accurate but also it does not require estimation of velocity explicitly since it represents velocity implicitly in the formulation, using the current and previous positions.

Similar to the strategy employed for the explicit integration, the implicit integration can be efficiently implemented in the proposed pipeline. First, the force derivatives with respect to the position and velocity are estimated. Then, using an iterative solver such as the Conjugate Gradient (CG) method, the new change in velocity (*dv*) is obtained. This change in velocity is then added to the current velocity to get the new velocity. The new velocity is then integrated to obtain the new position.

Typically in an implicit integration scheme, convergence is determined by comparing the relative difference (distance between the current estimated position and the last position) within an epsilon range. The value of epsilon used in most simulation system is 0.001 [126]. Since implicit integration dampens the simulation excessively, the overall system energy diminishes and thus, the solution converges [126]. Irrespective of the integration scheme used, the physically based deformation simulation proceeds as shown in the flowchart in Figure 3-2.

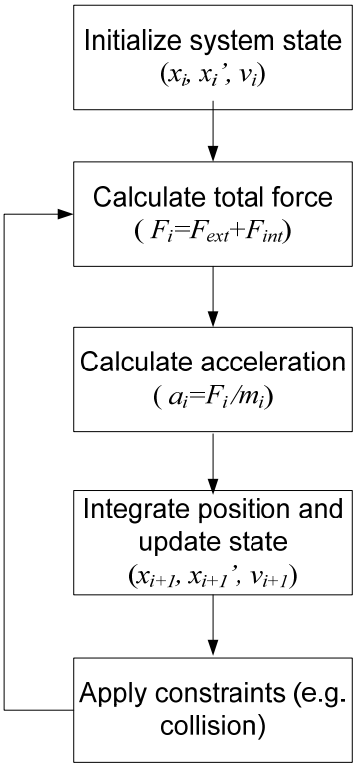


Figure 3-2. Dataflow of the physically based simulation

3.3 Establishing the Transform Feedback

3.3.1 Exploration for the Modern GPU Processing Pipeline

The transform feedback in the modern GPU architecture has been used for dynamic tessellation/level-of-detail (LOD) rendering for terrain visualization with detail rendering controls [127]. We propose to use this mode for an efficient real-time transformation pipeline.

The vertex shader usually transforms input vertex positions from object space to clip space. This is carried out by multiplying the current object space vertex position with the combined modelview projection matrix. However, the modern GPU architectures allow the vertex shader to circulate its result in a loop to perform iterative tasks on the input. The advantage is that the data remains on the GPU and it is not transferred back to the CPU memory via the low bandwidth bus. This feature is called *transform feedback*. Using this feature, the output values from a vertex or geometry shader can be stored back into a buffer object. These buffer objects are called *transform feedback buffers*. The recorded data may be read back on the CPU for interaction with the data or it may be visualized directly.

Technically, the transform feedback mechanism was introduced by NVIDIA as a vendor specific extension GL_NV_transform_feedback in OpenGL 3.0 [128]. The initial extension lacked support for fine grained control. Therefore it was later extended in the form of GL_NV_transform_feedback2 which gave way to four extensions in OpenGL 4.0, GL_ARB_transform_feedback2, GL_ARB_transform_feedback3, GL_ARB_draw_indirect (partially) and GL_ARB_gpu_shader5 (partially) [129].

With GL_ARB_transform_feedback2, a transform feedback object is defined similar to the other OpenGL objects. In addition, it introduces the capability of pausing and

resuming transform feedback so that multiple transform feedback objects can record their attributes. Moreover, the earlier extensions require a hardware query to obtain the total number of primitives. The query took additional clock cycles. Therefore, GL_ARB_transform_feedback2 adds support to draw the transform feedback primitives directly through glDrawTransformFeedback function [129].

One drawback with GL_ARB_transform_feedback2 and earlier extensions is that they require the vertex attributes to be specified separately in separate buffers. Therefore, it is impossible to interleave attributes in a single large buffer. This feature was introduced in GL_ARB_transform_feedback3 [130]. In addition, this extension also introduces vertex streams so that vertex data can be streamed directly to one or more transform feedback buffers in realtime.

To better understand the mechanism of our feedback formation based processes, Figure 3-3 gives an explanatory diagram of the hardware architecture including streams in the transform feedback stage of the modern GPU pipeline. In the current architecture, up to four transform feedback buffer streams could be run simultaneously. Each of these streams can be connected to a number of attributes from the vertex shader. This number is a platform specific constant. For the Verlet integration example in this chapter, two attributes are used, that is, the current position (x_i) and the previous position (x_i') which are output from the vertex shader. More attributes may be connected to the streams as desired.

Earlier transform feedback extensions did not allow instanced rendering; thus, it was impossible to draw instances from a transform feedback buffer without querying the output primitive count. In OpenGL 4.2, GL_ARB_transform_feedback_instanced solves this problem through two functions, glDrawTransformFeedbackInstanced and glDrawTransformFeedbackStreamInstanced.

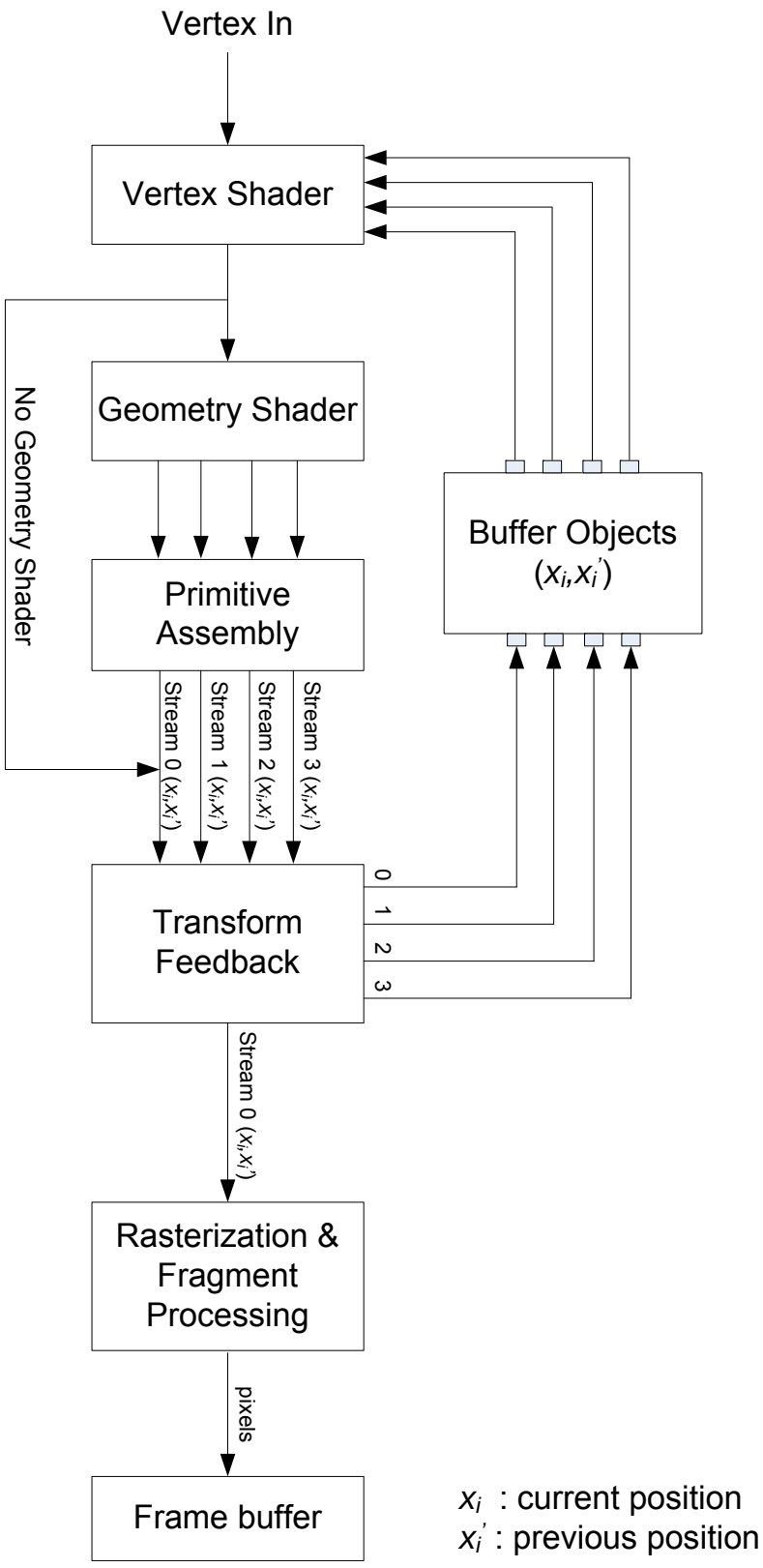


Figure 3-3. Architectural design for the formation of transform feedback

The transform feedback mechanism is available on a wide range of hardware from both NVIDIA and ATI/AMD. The OpenGL 3.x transform feedback and OpenGL 4.x transform feedback has been supported on Radeon 2000 series by ATI/AMD via ARB_transform_feedback2 and ARB_transform_feedback3. On NVIDIA hardware, OpenGL 3.x transform feedback is supported from GeForce 8 series whereas, OpenGL 4.x ARB_transform_feedback2 is supported from GeForce GTX 200 series and OpenGL 4.x ARB_transform_feedback3 is supported from GeForce 400 series [128-130].

3.3.2 Software Architecture Based on OpenGL API

We can now form the processing pipeline based on the modern GPU architecture. First, with OpenGL 4.0 and above, we can create the transform feedback object by calling the glGenTransformFeedbacks function. This object encapsulates transform feedback state. Once the object is used, it can be deleted by calling glDeleteTransformFeedbacks.

After creation of the transform feedback object, it should be bound to the current OpenGL context. This can be done by issuing a call to glBindTransformFeedback. The vertex attributes that are needed for recording in the buffer object using transform feedback should be registered. This can be done by issuing a call to glTransformFeedbackVaryings. The first parameter is the name of the program object which will output the attributes; the second is the number of output attributes that will be recorded using transform feedback; the third is the array of C-style strings containing the names of the output attributes; and the last identifies the mode of recording. This mode can be either GL_INTERLEAVED_ATTRIBS (if the attributes are recorded into a single buffer) or GL_SEPARATE_ATTRIBS (if the attributes are recorded into their own separate buffers). After specifying the transform feedback varyings, the shader must be linked again.

The buffer objects where the outputs from the vertex shader will be stored must also be identified. This can be done by issuing a call to glBindBufferBase. There is a platform-specific limit on the maximum number of transform feedback buffers that can be bound simultaneously. This also depends on the number of attributes being output from the vertex or geometry shader. To initiate transform feedback, the glBeginTransformFeedback function should be invoked. The only parameter is the type of primitive which is being feedback. Next, the drawing call is issued using glDraw to draw the required primitives. Finally, the transform feedback should be terminated by issuing a call to glEndTransformFeedback.

In OpenGL 4.0, the transform feedback buffer can be drawn directly, by calling glDrawTransformFeedback and passing it the type of primitive required. This is very convenient since there is no need to query the number of primitives output from transform feedback as was required in the previous extensions. In this way, the transform feedback mechanism can be exploited to implement a real-time deformation pipeline entirely on the GPU. Our real-time deformation pipeline highlighting the transform feedback stage is given in Figure 3-4.

3.4 Shader for Deformation Transformation

With reference to the transform feedback stage in Figure 3-4, this mode feedbacks the vertices to the vertex shader after the primitive assembly but before rasterization. Usually, during the deformation loop, the rasterizer is disabled so that the unnecessary computations may be avoided.

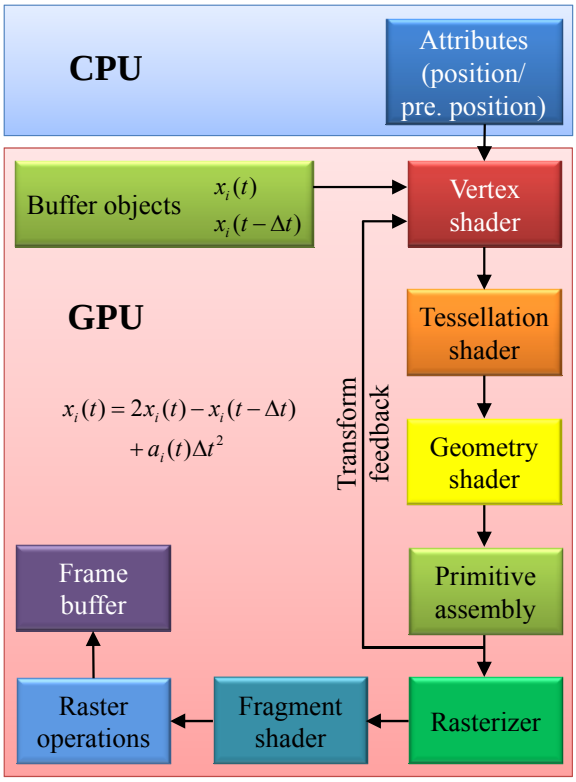


Figure 3-4. The transform feedback stage

To understand how the different steps of the algorithm work, for the rest of this discussion, we will discuss the steps needed to implement the Verlet integration as an example. To give a bird's eye view, we first perform the integration calculation in the vertex shader. Then, using transform feedback, we direct the new and previous positions to a set of vertex array objects containing a set of vertex buffer objects. The choice of using the Verlet integration is motivated by a set of experiments. In the first experiment, the motion of a projectile with initial position and velocity is estimated numerically. In the second experiment, the numerical evaluation of the sin function is carried out. We have used an oscillatory function here which may approximate the simple harmonic motion of a mass spring system.

We compared the numerical accuracy of the Verlet integration and the explicit Euler integration (which is the fastest integration scheme for numerical simulation) against the

exact solution. The initial conditions of the projectile include an initial velocity of 0 m/s, the acceleration due to gravity of 9.8m/s^2 and the fixed time step value of 0.15 seconds. The results are given in Figure 3-5.

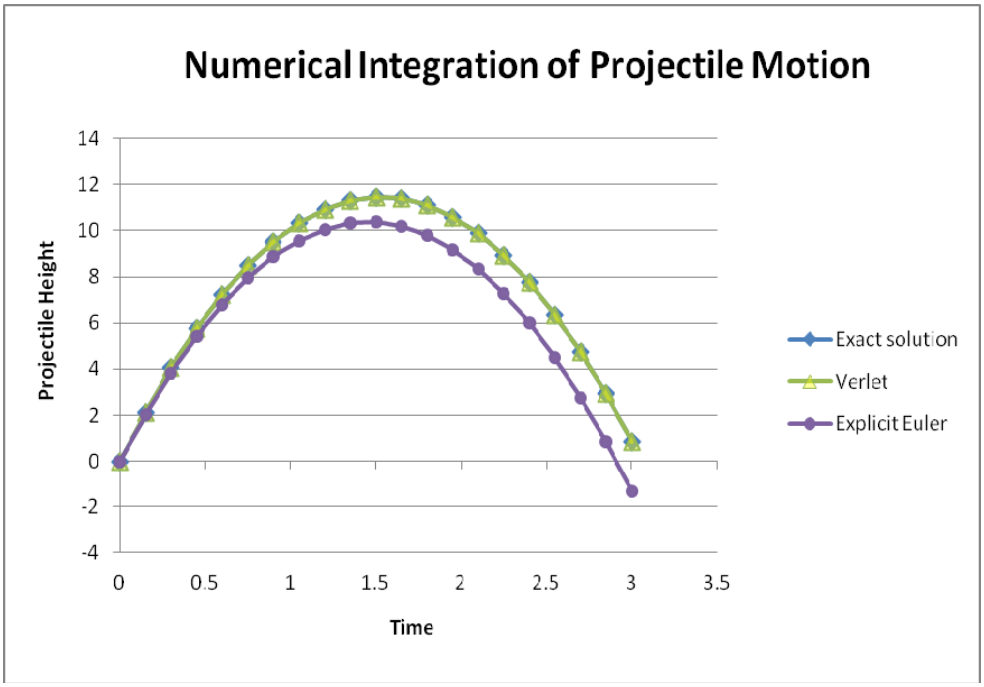


Figure 3-5. Comparison of two integration schemes for estimation of the projectile motion

As can be seen, the explicit Euler integration does not follow the exact solution as compared to the Verlet scheme because Verlet integration is second order accurate since it calculates the velocity implicitly. Explicit Euler on the other hand takes the velocity at the end of the time step. In case of fixed time step, the Verlet integration converges to the exact solution.

In the second experiment, we numerically evaluate the sin function using explicit Euler and Verlet integration schemes. For this experiment, a variable time step value is used. The results are given in Figure 3-6. As can be seen, Verlet is more accurate as compared to explicit Euler for variable time step values. The explicit Euler matches the Verlet integration only for small time step values but diverges for larger time step values.

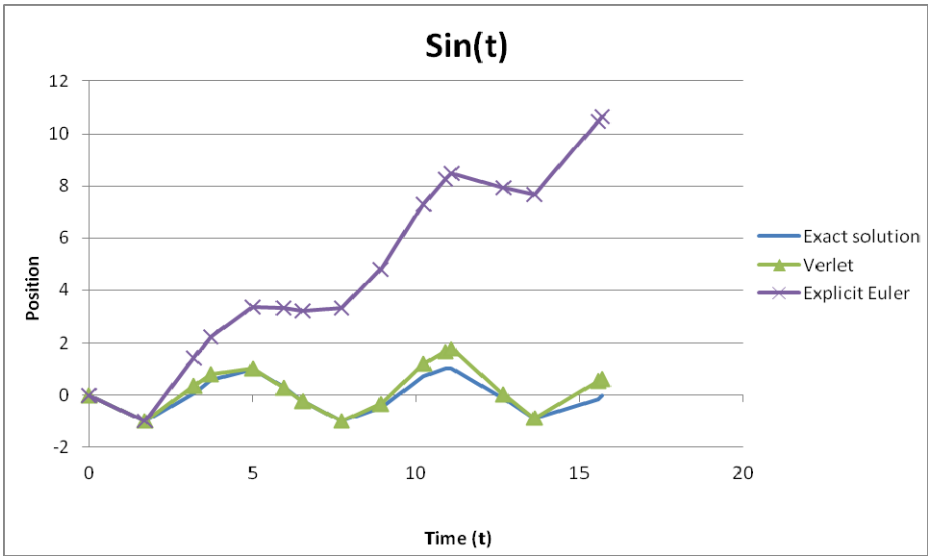


Figure 3-6. Numerical approximation of the sin function with variable time step

3.4.1 The Verlet Integration Vertex Shader

The most important part in our implementation is the Verlet integration vertex shader. We first store the current and previous positions of the mass points into a 2D grid. These are stored on the GPU into a pair of buffer objects. In order to efficiently obtain the neighborhood information, we attach the current and previous position buffer objects to the texture buffer target. This allows us to fetch the neighbor's current and previous positions in the vertex shader using the corresponding *samplerBuffer*.

The vertex shader starts with extracting the current position, the previous position and the current velocity. Next, the index of the current vertex is determined using the built-in register (**gl_VertexID**). Using this global index, the x/y index into the 2D grid is obtained. This is used to extract the correct neighbor from the *samplerBuffer*. The Verlet integration vertex shader is given in List 3.1.

In each rendering cycle, we swap between the two buffers to alternate the read/write pathways. Before the transform feedback can proceed, we need to bind the update vertex array objects to setup the appropriate vertex buffer object for writing data. At the same

time, the appropriate vertex buffer objects are bound to the current transform feedback buffer (for reading the current and previous positions).

```
for each vertex
   vec3 vel = ( pos - pos_old ) / dt;
   float ks = 0, kd =0;
   int index = gl_VertexID,
          ix = index % texsize_x,
          iy = index / texsize_x;
   vec3 F    = gravity * m + ( DEFAULT_DAMPING * vel );
   for each neighbor k
      ivec2 coord = getNextNeighbor (k, ks , kd);
      int j       = coord.x;
      int i       = coord.y;
      if !IsValidIndex(i,j)
         continue;
      end if
      int   index_neigh  = (iy + i) * texsize_x + ix + j;
      vec3  p2           = getPos(index_neigh);
      vec3  p2_last      = getLastPos(index_neigh);
      vec2  coord_neigh  = vec2 (ix + j, iy + i)* step;
      float rest_length  = length(coord * inv_cloth_size );
      vec3  v2           = (p2 - p2_last )/dt;
      vec3  deltaP       = pos - p2;
      vec3  deltaV       = vel - v2;
      float dist         = length(deltaP);
      float left    = -ks*(dist - rest_length );
      float right   = kd*(dot(deltaV , deltaP )/ dist );
      vec3  f_i     = (left + right)*normalize(deltaP);
      F += f_i;
   end for
   position = getPos(F,m);
end for
```

List 3.1. The Verlet integration vertex shader

3.4.2 Registering Attributes to Transform Feedback

The outputs from the Verlet integration vertex shader, that is, the current position and previous positions, must be registered to the transform feedback object. This is done by issuing a call to glTransformFeedbackVaryings, passing it the names of the output attributes. To make this binding effective immediately, the shader must be relinked. After registering, the modified attributes are written to the registered buffer object/s.

3.4.3 The Array Buffer and Buffer Object Setup

The application pushes a set of positions (current and previous positions) to the GPU. Each element is a *vec4* with x, y and z in the first three components and mass in the fourth component. We use a set of buffer objects for positions so that we may use the ping pong strategy to read from a set of positions while we write to another set using the transform feedback approach. We cannot write to a transform feedback attribute when we are reading from it.

We have two array objects for updating the physical simulation and two more array objects for rendering of the resulting positions. Each array object stores a set of buffer objects for current and previous position. The usage flags for the position buffer objects are set as dynamic (GL_DYNAMIC_COPY in OpenGL) since the data will be dynamically modified using the shaders. This gives an additional indication to the GPU so that it may put the buffers in the fastest accessible memory.

In each rendering cycle, we swap between the two buffers to alternate the read/write pathways. Before the transform feedback can proceed, we need to bind the update array object so that the appropriate buffer objects can be setup for writing data. We bind the appropriate buffer object for reading the current and previous positions to the current transform feedback buffer base by issuing a call to glBindBufferBase.

The rasterizer is disabled to prevent the execution of the rest of the programmable pipeline. The call for drawing point is issued to allow writing vertices to the buffer object. The transform feedback is then disabled. Following the transform feedback, the rasterizer is enabled and then the points are drawn. This time, the render array object is bound. This renders the deformed points on screen.

3.5 Shader for Collision Detection and Response

3.5.1 Collision with Sphere and Arbitrarily Oriented Ellipsoid

Collision detection and response can be incorporated into the proposed pipeline by directly adding constraints on the modified positions in the Verlet integration vertex shader. In an interactive graphics application, an ellipsoids hierarchy can be implemented to model the interaction of a humanoid with the virtual world for collision detection and response. More complex geometries may also be approximated by a combination of spheres and ellipsoids for fast and efficient collision detection.

To illustrate the shader in such a pipeline, we show how to do collision detection and response for a sphere and an arbitrarily oriented ellipsoid. A similar strategy can be used for doing collision against an arbitrary polygonal mesh. Consider a sphere with a center (C) and a radius (r). We have a mass at position (x_i), and it is transformed to a new position (x_{i+1}). The collision constraint of an arbitrary mesh with a sphere can be given by

$$x_{i+1} = \begin{cases} C + \dfrac{(x_i - C).r}{|x_i - C|} & if\,|x_i - C| < r \\ x_i & else \end{cases}$$

The above formulation can be translated directly into the shader code and integrated into the Verlet integration vertex shader. We first check the distance of the current mass point to the center of the sphere. If it is less than the radius of the sphere, we encounter a collision; therefore, the mass point is kept on the sphere's surface to prevent penetration into the sphere.

For an arbitrarily oriented ellipsoid, it is easier to resolve collisions in the ellipsoid's object space. For that reason, we first calculate the inverse transform of the ellipsoid's transform. In this space, the ellipsoid is a unit sphere. Next, we transform the current vertex position into the ellipsoid's object space (by multiplying with the inverse of the

ellipsoid's transform). If the distance between the current position and the centroid of the ellipsoid is less than one, we encounter a collision.

Once a collision is detected, we can find the penetration depth and then move the current point's position such that it is out of the unit sphere in the transformed ellipsoid's space. For an ellipsoid with a center (*ellipsoid_center*), radius (*ellipsoid_radius*), transform (T), and inverse transform (T_{inv}), the shader of the whole collision detection and response is given in List 3.2.

```
vec4 x0 = Tinv * vec4(pos,1);
vec3 delta0 = x0.xyz-ellipsoid_center;
float dist = dot(delta0, delta0);

if(dist < 1) {
   delta0 = (ellipsoid_radius - dist) * delta0 / dist;

   vec3 delta;
   vec3 transformInv = vec3(T.row0);
   transformInv /= dot(transformInv, transformInv);
   delta.x = dot(delta0, transformInv);

   transformInv = vec3(T.row1);
   transformInv /= dot(transformInv, transformInv);
   delta.y = dot(delta0, transformInv);

   transformInv = vec3(T.row2);
   transformInv /= dot(transformInv, transformInv);
   delta.z = dot(delta0, transformInv);

   pos += delta;
   pos_old = pos;
}
```

List 3.2. Shader for collision of a mesh with an oriented ellipsoid

3.5.2 On-the-fly Modification of the Parameters for Deforming Models

In the case of user interaction during deformation, we need to obtain the pointer to data to modify the parameters for the deforming model on demand. This is done in the current deformation pipeline by first mapping the appropriate buffer object (by calling the glMapBuffer OpenGL function). This call returns immediately with the pointer to the GPU memory. Once the data has been modified, the new values are assigned to the pointer and then the buffer is unmapped to release the pointer to the GPU memory.

3.6 Experimental Results and Performance Assessment

We would like to assess the performance of a 2D mesh (shown in Figure 3-7) with different resolutions ranging from 64×64 mass points to 2048×2048 mass points on a Dell Precision T7500 desktop with an Intel Xeon E5507 @ 2.27 MHz CPU and an NVIDIA Quadro FX 5800 GPU card.

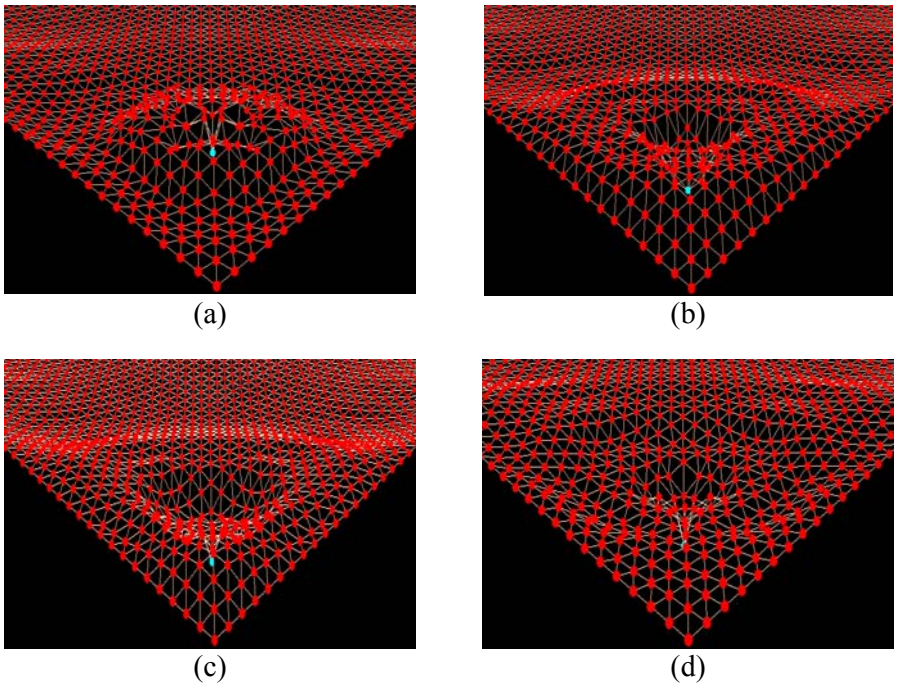


Figure 3-7. Real-time renderings of deformation using transform feedback. The user plucks a point up and down (a-d) and the deformation is propagated in the mesh

In the first experiment, we compare the performance of our deformation pipeline using transform feedback with the OpenCL GPU implementation in the bullet sdk v 2.80. For fair comparison, the setup for both cases is identical. The OpenCL code failed to run for mesh size larger than 512×512. The possible cause of this is the unavailability of aligned memory required by the GPU deformation demo in the Bullet physics sdk. In our transform feedback codes, we allocate the memory once at initialization using the vertex buffer object and that is reused for both deformation and rendering. These results are presented in Table 3.1.

Table 3.1. Performance comparison between GPU codes using the OpenCL demo in the bullet physics sdk and our transform feedback pipeline

Grid size	Time for rendering a single frame (in msecs).		
	OpenCL (a)	GPU TF (b)	Speed up (a/b)
64×64	1.75	0.91	1.92
128×128	2.01	1.20	1.69
256×256	3.95	2.25	1.76
512×512	8.98	6.33	1.42

As can be seen, our proposed pipeline outperforms the OpenCL code by about 1.4x. The reason for this speedup is that the proposed pipeline does not require any read back since the data remains in the GPU memory throughout the simulation. In the case of OpenCL however, the data has to be mapped from the OpenCL device memory to the OpenGL buffer object memory. This needs additional runtime and hence the performance suffers.

In the second experiment, to assess the robustness and effectiveness of the transform feedback based shader, we compare the performance of the different integration schemes. These results are presented in Table 3.2. The times noted include the time for both deformation and rendering. As can be seen, the overall performances of these integration solvers are close to each other in the transform feedback pipeline. This ensures the stability and scalability of the new algorithm in applications.

Accuracy of a physical simulation is usually determined by carrying out the same simulation on the CPU. We have given a quantitative evaluation in this section because we focused on the performance aspect. We have shared our demo implementation on the book's (The OpenGL Insights) accompanying web site (www.openglinsights.com) which includes this comparison. In the demo application, the GPU/CPU/CPU optimized modes may be toggled but the output remains indistinguishable between the three modes.

Table 3.2. Performance of different integration schemes

Grid size	Time for different integration schemes (in msecs)				
	Explicit Euler	**Semi Implicit Euler**	**Mid-point Euler**	**4th Order R.K.**	**Verlet**
64×64	1.07	0.867	0.905	1.151	0.853
128×128	1.47	1.817	1.806	1.895	1.793
256×256	3.34	3.409	3.464	3.678	3.107
512×512	11.65	11.588	11.088	11.957	11.027
1024×1024	77.94	54.288	54.436	55.157	53.705
2048×2048	251.26	274.725	271.739	327.868	258.397

The Verlet integration has reduced memory requirement (no velocity storage required), and performs well; it is also stable, with an approximation error on the order of $O(n^2)$. From the point of view of stability, the 4th order Runge Kutta method is found the best whereas the explicit Euler method is the worst.

Currently, the shaders cannot write to arbitrary memory regions directly (the scattered writes). If such features are exposed, the performance of the algorithms can be improved further. In the new OpenGL 4.2 API, there are new extensions such as the image-load-store extension that may allow reading and writing to arbitrary memory in the shaders. This extension is unfortunately unsupported on the hardware that was used in these experiments. We can expect a better performance from this extension. However, this requires a more rigorous experimentation. Thanks to the flexibility of the proposed pipeline, we have enjoyed minimal efforts to add support for all of the integration schemes in the proposed deformation pipeline.

3.7 Summary

We have presented a novel GPU-based deformation pipeline to perform real-time mesh based deformation transformation as well as to perform collision detection and response. Our approach is based on the mechanism of transform feedback available in the modern GPU architectures. Our experiments reveal that the proposed deformation pipeline outperforms the OpenCL GPU implementation by a speedup of 1.4x.

To the best of our knowledge, this is the first work on using transform feedback for deformation entirely on the GPU. We are confident on the results obtained from the algorithm and would like to expand the model to address specific areas such as biomedical modeling and simulation [5-6].

On the other hand, while the proposed mesh spring model is suitable for large deformations, it lacks accuracy for handling small local deformations [22]. Such intricate details can be well captured by the meshless FEM approach. Therefore, in the next chapter, we will extend this pipeline to a meshless FEM model for approximating non-rigid behavior in a dynamic 3D volumetric dataset, to complement our study on high-performance volumetric modeling.

4 NONRIGID VOLUMETRIC TRANSFORMATION WITH MESHLESS FEM

4.1 Introduction

Nonrigid transformation plays a significant role in portraying complex interactions between graphical objects. Such subtle movements are absolutely necessary in some domains, for example, surgical simulation in which the surgeon's training experience is directly related to the feedback he/she gets from the training system. If the system models the soft tissue deformations accurately, the training session is much more effective as compared to a system without such effects. While the previous chapter focused on handling of large deformations, often in real-time simulations, small intricate deformations arise and the mesh based approach cannot effectively handle such cases.

Prior to the advent of the GPUs, such simulations were only restricted to offline renderers. However, now that the computational power of GPUs is enormous, these interactions can be implemented with accurate physically-based deformable models in realtime [22]. While the models for physically-based deformation range from highly accurate finite elements model (FEM) to less accurate mass spring models, the choice of the model largely depends on the required accuracy and fidelity of deformation; for example, in some real-time games, we may compromise on quality for speed of interaction, whereas in a surgical simulation system, accuracy is required and the quality should not be compromised.

Meshless FEM models have been proposed in the literature for deformation in computer graphics [131-133]. An efficient method for real-time meshless deformation is proposed in [131]. The method first extracts an octree from the point cloud representation of the reference or rest dataset. Then, it generates the implicit surface and deformation

nodes (for meshless simulation). To speed up the process, it precomputes the system matrices (stiffness and mass matrices) based on the rest shape and solves it using an Eigen value problem. This gives the model displacements which are used for the meshless simulation of the moving shape.

Following the evaluation of displacements, the deformation of implicit surface (which are extracted using the octree and distance map) is then carried out. This may work for surfaces however, for a volume, the stiffness matrices would have to be re-evaluated since the volume will be changing continuously. This will also invalidate the octree and the distance maps and they would have to be recomputed. These steps have an $O(N^3)$ complexity with N being the dimension of the dataset on one axis which would largely reduce the runtime performance of the method.

Another meshless approach was proposed in the context of frame-by-frame animation [132]. It uses the meshless model to determine the intermediate pose as is typically required in a frame-by-frame animation system. In this method, nodes are picked up iteratively to generate an approximate representation. The points obtained are then used to deform the given mesh. While the technique can be applied to polygonal data, it may be difficult to apply it to volumetric datasets because volumetric datasets store a distribution of densities. Thus, it is difficult to identify a unique surface as a representative of the whole volume. Due to the same reason, using their formulation, it is impossible to find the approximate representation that could be used to generate for the deformation.

Frame based animation was extended to an anisotropic material model by [133]. They augment the frame based deformation with material based kernels for distributing anisotropic response. The computation of the ideal shape functions for 2D requires a solution of a $2N \times 2N$ equation system with (N) being the number of independent points. The worst case complexity is $O(N^3)$. On the other hand for a 3D voxel grid an

approximate result may be obtained by using a 3D distance field with O(N*log*N) complexity. However, there are infinite paths from one point to another for stress and body force distribution in 3D hence the stress is non-uniform and thus the simplification assumed in their approach cannot be applied. Another problem is the number of deformation modes. In higher dimensions, several stretching and shearing modes exists therefore, it is impossible to represent these values with a single scalar as assumed in the formulation. Based on the above justification, this method cannot be applied to deformation of 3D volumes.

In Chapter 2, we have reviewed a wide range of models for physically-based deformation. It seems to us that the existing tetrahedral FEM models are unsuitable for real-time volumetric deformation. Highlighted below are the main concerns:

- Approximating a volumetric dataset requires a large number of finite tetrahedral elements. Numerical solution of such a large system would require a large stiffness matrix assembly. This makes the model unsuitable for real-time volumetric deformation. In addition, the corotated formulation is needed which increases the computational burden even further.
- The solution of the tetrahedral FEM requires an iterative implicit solver; for example, Newton Raphson (Newton) or Conjugate Gradient (CG) method. These methods converge slowly. Moreover, the implicit integration solvers reduce the overall energy of the system causing the physical simulation to dampen excessively.
- Even though multi-grid schemes are fast, they have to update the deformation parameters across the different grid hierarchy levels. This requires considerable computation. Moreover, the number of grid levels required is subjective to the dataset at hand and there is no rule to follow for accurate results.

We have noticed that the meshless FEM approach has not been applied for volumetric

deformation in the literature. We are interested in its use for volumetric deformation coupled with simultaneous GPU-based real-time visualization. Initial study shows that the meshless formulation possesses a few advantages:

- It supports deformation without the need for stiffness warping (the corotated formulation).
- The solution of meshless FEM is based on a semi-implicit integration scheme which not only is stable but also converges faster as compared to the implicit integration required by the tetrahedral FEM solver. In addition, it does not introduce artificial damping.
- It does not require an iterative solver such as the conjugate gradient (CG) method which is required for conventional tetrahedral FEM.

Therefore, in the following sections, we describe the meshless FEM approach and the formulation of the nonrigid transformation model.

The rest of this chapter is organized as follows. Mathematical details about the meshless FEM are described in section 4.2. We then present the coupling between our transformation pipeline and the GPU-based volume rendering in section 4.3. Rendering of the deformed volume using the cell projection pipeline is described in section 4.4. Experimental results and performance assessment are presented in section 4.5. And finally, section 4.6 concludes this chapter.

4.2 The Meshless FEM Approach

4.2.1 Nonrigid Volume Modeling

Before continuing with volumetric deformation and rendering, we need to come up with an efficient data representation. The volume dataset is usually specified on a rectilinear grid (for a structured dataset) or on a curvilinear or other nonlinear grid (for an

unstructured dataset). No matter which category the dataset belongs to, it can be represented as a mapping $f(\mathbf{x}):R^3 \rightarrow R$. In order to perform any transformation T on the volume, a simple representation is to work in the forward direction that is

$$y(\vec{x}) = f(T(\vec{x}))$$

where, y is the transformed (output) dataset and the f is the untransformed (input) dataset. This representation unfortunately results in holes because the transformation T might result in a new position that is not in the domain of the input dataset. Therefore, a much better approach is to go backwards that is

$$y(T(\vec{x})) = f(\vec{x})$$

Now each element is accessed on the transformed location in the output dataset. Since all elements in the input dataset are accessed, we always get a sample value. If the transformed location $T(\mathbf{x})$ is outside the output dataset's domain, it can be ignored safely. In nonrigid volume modeling, the transformation (T) may include one of the various types given in Table 4.1. In the transformations given in Table 4.1, A is an affine 4×4 matrix, c is the center of rotation, t is a translation vector, N_x is the set of all neighbors of the current voxel, β are the B-spline basis, and G are the thin plate spline basis functions.

Certainly, the transformations given in Table 4.1 are not the only nonrigid transformations available. Higher order transforms such as the cubic B-spline and thin plate spline generally require a collection of control points for deformation. Each control point has a weight associated with it which dictates the deformation behavior. Using more control points not only increases the computational burden, it also smoothes out the underlying volume densities due to higher order interpolation to obtain the off-grid values.

It was shown by Thevanes et al. [109] that a better method to interpolate a cubic B-spline function is to go backwards and extract the B-spline coefficients for a given

intensity value using prefiltering and a recursive application of infinite impulse response (IIR) filter. This way, the output is always guaranteed to contain the intensities from the input dataset.

Table 4.1. Various nonrigid volume transformations

Nonrigid Transform	Formulation
Affine	$T(\vec{x}) = A(\vec{x} - \vec{c}) + t + c$
B-spline	$T(\vec{x}) = \vec{x} + \sum_{j \in N_x} p_j \beta^3 (\frac{\vec{x} - \vec{x}_j}{\sigma})$
Thin plate spline	$T(\vec{x}) = \vec{x} + A\vec{x} + t + \sum_{j \in N_x} c_j G(\vec{x} - \vec{x}_j)$

The solution of elasticity equations require the interior of the object to be modeled. Among some of the models for simulating the volumetric deformable objects are the particle-based model (such as the mass spring method) and the continuum mechanics model such as FEM [22].

4.2.2 Continuum Elasticity

In order to understand the continuum elasticity in general and FEM in particular, basic quantities such as stress, strain and displacement play an important role. Strain (ε) is defined as the relative elongation of the element. Assuming an element undergoing a displacement (ΔL) having length (l), the strain may be given as:

$$\varepsilon = \frac{\Delta L}{l}$$

With displacement field (u) inside an element, for a one dimensional problem, the strain at location (x) for an infinitesimal element of length (dx) may be approximated as

$$\varepsilon(x) = \frac{u(x+dx) - u(x)}{dx} \approx \frac{d}{dx} u(x)$$

For a three-dimensional problem, the strain (ε) is represented as a symmetric 3×3 tensor:

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix}$$

There are two popular choices for the strain tensor in computer graphics, the linear Cauchy strain tensor which is given as

$$\varepsilon_{Cauchy} = \frac{1}{2}(\nabla U + [\nabla U]^T) \tag{4.1}$$

and the nonlinear Green strain tensor which is given as

$$\varepsilon_{Green} = \frac{1}{2}(\nabla U + [\nabla U]^T + [\nabla U]^T \nabla U) \tag{4.2}$$

In Eq. (4.1) and (4.2), the ∇U is the gradient of the displacement field U given as

$$\nabla U = \begin{bmatrix} \frac{\partial U_x}{\partial x} & \frac{\partial U_x}{\partial y} & \frac{\partial U_x}{\partial z} \\ \frac{\partial U_y}{\partial x} & \frac{\partial U_y}{\partial y} & \frac{\partial U_y}{\partial z} \\ \frac{\partial U_z}{\partial x} & \frac{\partial U_z}{\partial y} & \frac{\partial U_z}{\partial z} \end{bmatrix}$$

Similar to the strain, in the three dimensional problem, the stress tensor (σ) is also given as a 3×3 tensor

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}$$

Assuming that the material under consideration is isotropic and it undergoes small deformations (*geometric linearity*), the stress and strain may be linearly related (*material linearity*) using Hooke's law, given as

$$\sigma = D\varepsilon \tag{4.3}$$

Since stress and strain are symmetric matrices, there are six independent elements in each

of them. This reduces the isotropic elasticity matrix (D) to a 6×6 matrix as follows:

$$D = B\begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}$$

$$B = \frac{E}{(1+\nu)(1-2\nu)}$$

where, E is the Young's modulus of the material which controls the material's resistance to stretching and ν is the Poisson's ratio which controls how much a material contracts in the direction transverse to stretching.

In a finite element simulation, we try to estimate the amount of displacement due to application of forces. There are three forces to consider:

- Stress force due to stress (σ) which is an internal force,
- Nodal force (q) which is an external force applied to each finite element node, and
- Loading force (t) which is an external force applied to the boundary or surface of the finite element

For the finite element to be in static equilibrium, the amount of work done by the external forces must be equal to the amount of work done by the internal forces given as

$$\int_{V^e} W_\sigma dV = W_q + \int_{A^e} W_t dA \tag{4.4}$$

where, W_σ is the internal work done per unit volume by stress σ, W_q is the external work done by the nodal force q on the element's node and W_t is the external work done by the loading force t on the element per unit area. V_e is the volume and A_e is the area of the finite element e. W_σ is given as

$$W_\sigma = (\delta\varepsilon)^T \sigma$$

where, $\delta\varepsilon$ is the strain produced by the stress σ. Similarly, W_q is given as

$$W_q = \left(\delta u^e\right)^T q^e$$

where, δu^e is the displacement of the finite element e produced by the force q^e. W_t is given as

$$W_t = \left(\delta u\right)^T t$$

Substituting these in Eq. (4.4), we get

$$\int_{V^e} \left(\delta\varepsilon\right)^T \sigma dV = \left(\delta u^e\right)^T q^e + \int_{A^e} \left(\delta u\right)^T t dA \tag{4.5}$$

Since δu^e provides the displacement of node e at vertices only, to get the displacement at any point within the finite element, we can interpolate it with the shape function N

$$\delta u = N\delta u^e$$

Likewise, we may apply a differential operator $\boldsymbol{S}$ to get the change in strain ($\delta\varepsilon$) as

$$\delta\varepsilon = SN\delta u^e$$

The matrix product ($\boldsymbol{SN}$) can be replaced by $\boldsymbol{B}$

$$\delta\varepsilon = B\delta u^e$$

Substituting in Eq. (4.5), we get

$$\int_{V^e} \left(B\delta u^e\right)^T \sigma dV = \left(\delta u^e\right)^T q^e + \int_{A^e} \left(N\delta u^e\right)^T t dA$$

Simplification and substitution using Eq. (4.3) gives

$$\int_{V^e} (B^T D\varepsilon) dV = q^e + \int_{A^e} (N^T t) dA$$

Taking constant terms out of the equation and solving the left side integral gives

$$B^T DBu^e V^e = q^e + \int_{A^e} (N^T t) dA$$

The left side is replaced by the element stiffness matrix ($K^e=B^TDBV^e$) and the right side by the element surface force (f^e), which gives us the stiffness matrix assembly equation:

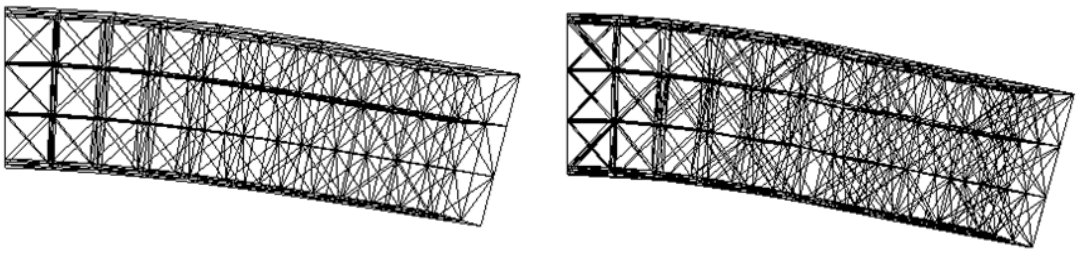
$$K^e u^e = q^e + f^e$$
$$f^e = \int_{A^e} (N^T t) dA$$

4.2.3 Meshless FEM

The conventional FEM methods discretize the whole body into a set of finite elements. Calculation of the element stiffness matrix requires the volume of the body which is usually represented as the sum of the finite elements' volume. For instance, the widely used corotated linear FEM [88] has to construct the global stiffness matrix for each deformation frame, and then the matrix is solved using an iterative solver such as the Conjugate Gradients (CG) method. This makes the implementation inefficient for a large volume.

In our meshless FEM, the whole body is sampled at finite number of points. The typical simulation quantities such as the position (x), velocity (v) and density (ρ) are all stored with the points. The displacement field is estimated from the volume of the point and its mass distribution in the local neighborhood. The gradient of the displacement field is then estimated to obtain the Jacobian. Finally, the Jacobian is used to calculate the stresses and strains. These, in turn, allow us to obtain the internal forces. Since the meshless FEM uses the moving least square approximation, it does not require the stiffness matrix assembly enabling a much better execution performance.

To ascertain that our proposed meshless FEM is able to produce the same deformation as that in the conventional FEM such as the corotated linear FEM, we conducted a computational experiment on a horizontal beam as shown in Figure 4-1. The two results show a horizontal beam having Young's modulus of 500,000 psi and the Poisson ratio of 0.33. The dimensions of the two beams are the same.



(a) (b)

Figure 4-1. Comparison of deformation of a horizontal beam using (a) linear FEM and (b) meshless FEM

The beam in Figure 4-1 (a) contains 450 tetrahedra for the corotated linear FEM whereas its equivalent one in Figure 4-1 (b) contains 176 points for the meshless FEM. While the two computations yield the same deformation under the given load, our meshless FEM has a significantly improved execution performance: 40 msecs per frame by the corotated linear FEM, compared to 1.25 msecs per frame with the meshless FEM. These timings include both the deformation and rendering time.

In a dynamic simulation, we are to solve the following system:

$$m\ddot{x} = -c\dot{x} + \sum \left(f_{\text{int}} + f_{ext} \right) \tag{4.6}$$

The first term on the right is the velocity damping term with c being the damping coefficient. For an infinitesimal element, mass is approximated using density (ρ). This changes Eq. (4.6) to

$$\rho\ddot{x} = -c\dot{x} + \sum \left(f_{\text{int}} + f_{ext} \right) \tag{4.7}$$

The external forces (f_{ext}) are due to gravity, wind, collision and others. Since our system assumes geometric and material linearity, Eq. (4.7) becomes a linear PDE that may be solved by discretizing the domain of the input dataset using finite differences over finite elements. This system may be solved using either explicit or implicit integration schemes.

4.2.4 Smoothing Kernel

In the conventional FEM, the volume of the body is estimated from the volume of its constituent finite elements. In the case of the meshless FEM, it is approximated from the point's neighborhood. For each point, its mass is distributed into its neighborhood by using a smoothing kernel (w)

$$w(r,h)=\begin{cases}\dfrac{313}{64\pi h^9}(h^2-r^2)^3 & if\,(r<h)\\ 0 & else\end{cases} \tag{4.8}$$

where, r is the distance between the current particle and its neighbor, and h is the kernel support radius. The density is approximated by summing the product of the current point's mass with the kernel.

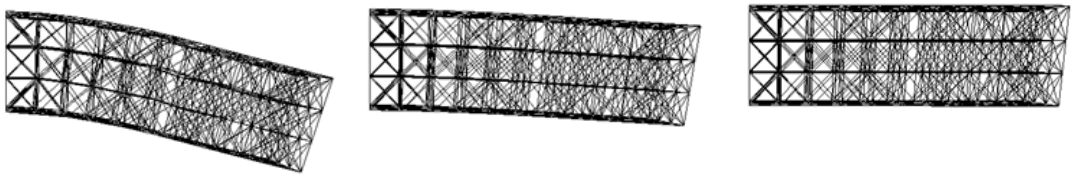
We analyzed the effect of varying the smoothing kernel [134]. These kernels include the normal smoothing kernel (as given in Eq. (4.8)), the spiky kernel, given as

$$w(r,h)=\begin{cases}\dfrac{15}{\pi h^6}(h-\|r\|)^3 & 0\le\|r\|\le h\\ 0 & \|r\|>h\end{cases} \tag{4.9}$$

and the blobby kernel, given as

$$w(r,h)=\begin{cases}\dfrac{15}{2\pi h^3}(-\dfrac{r^3}{2h^3}+\dfrac{r^2}{h^2}+\dfrac{h}{2r}-1) & 0\le\|r\|\le h\\ 0 & \|r\|>h\end{cases} \tag{4.10}$$

The deformation results on a horizontal beam containing 176 points are shown in Figure 4-2. Note that for all of the beams shown in Figure 4-2, the Young's modulus of 500,000 psi and the Poisson ratio of 0.33 are used.



(a) (b) (c)

Figure 4-2. Effect of different smoothing kernels on the deformation: (a) the normal smoothing kernel (Eq. 4.8), (b) the spiky kernel (Eq. 4.9) and (c) the blobby kernel (Eq. 4.10)

As can be seen, changing the smoothing kernel alters the stiffness of the soft body. This is because each kernel has a distinct support radius under which it influences the neighboring points. Moreover, each of these kernels has a different falloff (or, different derivative) which gives a different deformation result even though the rest of the simulation parameters are the same.

4.2.5 Propagation of Transforms

For propagating the stress, strain and body forces in the meshless FEM, we compute the gradient of the displacement field (U) by a moving least square interpolation between the displacement values at the current point (u_i) and its neighbor (u_j)

$$e = \sum_i (u_j - u_i)^2 w_{ij} \tag{4.11}$$

where, (w_{ij}) is the kernel function given in Eq. (4.8). The displacement values (u_j) are given using the spatial derivatives approximated at point (i) as

$$u_j = u_i + \nabla u.(x_j - x_i)$$

We want to minimize the error (e) in Eq. (4.11) so we differentiate e with respect to x, y and z and set the derivative equal to zero. This gives us three equations for three unknowns

$$\nabla u\,|_x = A^{-1}\left(\sum_i (u_j - u_i)(x_j - x_i)w_{ij}\right)$$

where, $A = \Sigma_i(x_j\text{-}x_i)(x_j\text{-}x_i)^T w_{ij}$ is the moment matrix that can be pre-calculated since it is independent of the current position and displacement. Once ∇u is obtained, the strain (ε) is obtained using Eq. (4.2). Using strain, the stress (σ) may be obtained using Eq. (4.3). The internal forces (f_{int}) in Eq. (4.7) are calculated as the divergence of the strain energy which is a function of the particle's volume

$$U_i = v_i \frac{1}{2}(\varepsilon_i.\sigma_i)$$

where, v_i is the volume of the particle. The force acting on neighboring particle (j) due to particle (i) is given as

$$f_j = -\nabla U_i = -v_i \frac{1}{2}\nabla\varepsilon_i.\sigma_i$$

To sum up, the internal forces acting on the particles i and j may be given as

$$\begin{aligned} f_i &= -2v_i J\sigma_i d_i \\ f_j &= -2v_j J\sigma_j d_j \end{aligned} \tag{4.12}$$

where, $d_i = M^{-1}(\Sigma_i(x_j\text{-}x_i)w_{ij})$ and $d_j = M^{-1}(x_j\text{-}x_i)w_{ij}$, J is the Jacobian, v is the volume of the point and σ is the stress at the given point. Note that in the case of the point masses, the volume measure does not make intuitive sense. However, it may be calculated by first approximating the mass of the particle. The mass (m_i) of the particle (i) is calculated using

$$m_i = sr_i^3\rho \tag{4.13}$$

where, r_i is the average distance between the mass point and its neighbors, ρ is the material density and s is a scaling constant. The scaling constant is calculated as

$$\beta_j = \frac{1}{\sum_j r_j^3 w_j}$$
$$s = \frac{\sum_i \beta_i}{n} \tag{4.14}$$

where, n is the total number of points. Once the mass is obtained, the per-point density is then obtained by summing the product of the mass of its neighbor (m_j) with the kernel evaluated at the neighbor j (w_j)

$$\rho_i = \sum_j m_j w_j \tag{4.15}$$

This density can then be used to obtain the volume of the point which is given as

$$vol_i = \frac{m_i}{\rho_i} \tag{4.16}$$

During the force evaluation, the internal forces are scattered in the neighborhood of the point (Eq. 4.12). The meshless FEM algorithm using scattering is given in Figure 4-3 (a). Since scattering cannot be implemented in a GPU shader program, for parallel processing, we convert the scatter operation into a gather operation by reformulation [135]. First, the sum of force matrices (F_e and F_v) is obtained

$$sumF_i = (F_e + F_v)$$

Instead of calculating the force on the point i as given in Eq. (4.12), we multiply the matrix sum ($sumF_i$) with d_i

$$F_i = F_i + sumF_i d_i$$

where, d_i is obtained as in Eq. (4.12). The internal force due to the neighbor points is then given as

$$F_i = F_i - sumF_i d_i$$

where, F_i is the net internal force, j loops for each neighbor of the current point i and d_j is obtained as in Eq. (4.12). This allows us to run the program in parallel on all points simultaneously. The new algorithm with gathering for an efficient parallel implementation of our model is given in Figure 4-3 (b).

Algorithm 1: Meshless FEM pseudocode with scatter

```
1  Initalize force: F = gravity * m_i - K_d * V_i;
2  Calculate displacement: U_i = X_i - X_i^0;
3  Calculate Jacobian:
4     B = mat33();
5     delU = mat33();
6     J = mat33();
7     Minv = fetch(texMinv, vertexID);
8     for each neighbor n
9        U_ji = U_j - U_i;
10       B = B + outerProduct(U_ji, n.rdist * n.w);
11    end for
12    ∇U = Minv * B^T;
13    J = ∇U^T + I;
14 Calculate stress and strain:
15    ε = J^T * J - I;
16    σ = D * ε;
17 Calculate internal force:
18    Fe = -2 * vol_i * J * σ;
19    J_u = row0(J);
20    J_v = row1(J);
21    J_w = row2(J);
22    M = mat33(J_v × J_w, J_w × J_u, J_u × J_v)^T;
23    Fv = -vol_i * K_v * (|J| - 1) * M;
24    for each neighbor j
25       F_j = F_j + (Fe + Fv) * dj;
26    end for
27    F_i = F_i + (Fe + Fv) * di;
28 Integrate position and velocity using leapfrog
   integration:
29    a_1 = F_i / m_i;
30    X = X + VΔt + (a_1/2)Δt^2;
31    F_new = CalcForces();
32    a_2 = F_new / m_i;
33    V = V + ((a_1 + a_2)/2)Δt;
```

(a)

Algorithm 2: Meshless FEM pseudocode with gather

```
1  Initalize force: F = gravity * m_i - K_d * V_i;
2  Calculate displacement: U_i = X_i - X_i^0;
3  Calculate Jacobian:
4     B = mat33();
5     delU = mat33();
6     J = mat33();
7     Minv = fetch(texMinv, vertexID);
8     for each neighbor n
9        U_ji = U_j - U_i;
10       B = B + outerProduct(U_ji, n.rdist * n.w);
11    end for
12    ∇U = Minv * B^T;
13    J = ∇U^T + I;
14 Calculate stress and strain:
15    ε = J^T * J - I;
16    σ = D * ε;
17 Calculate internal force:
18    Fe = -2 * vol_i * J * σ;
19    J_u = row0(J);
20    J_v = row1(J);
21    J_w = row2(J);
22    M = mat33(J_v × J_w, J_w × J_u, J_u × J_v)^T;
23    Fv = -vol_i * K_v * (|J| - 1) * M;
24    sumF_i = Fe + Fv;
25    F_i = F_i + sumF_i * d_i;
26    for each neighbor j
27       F_i = F_i - sumF_j * d_j;
28    end for
   Leap frog integration
29    a_1 = F_i / m_i;
30    X = X + VΔt + (a_1/2)Δt^2;
31    F_new = CalcForces();
32    a_2 = F_new / m_i;
33    V = V + ((a_1 + a_2)/2)Δt;
```

(b)

Figure 4-3. Algorithm for Meshless FEM using (a) scatter and (b) gather

4.3 Coupling between Volumetric Transformation and Rendering

Prior rendering algorithms [77-79] have resorted to the GPGPU-based techniques for evaluating the position and/or velocity integration on the fragment shader. This involves rendering a screen sized quad with the appropriate textures setup and then the fragment shader is invoked to solve the integration for each fragment. The output from the fragment shader is written to another texture.

On the contrary, we adopt a different approach (see Figure 4-4). We implement the meshless deformation by using the transform feedback mechanism of the modern GPU. This mechanism allows us to push as many vertices as the GPU may handle for deformation. The original unstructured mesh vertices are used directly in our implementation. We have two vertex array objects for updating the physics and two more vertex array objects for rendering of the resulting deformation. Both vertex array object pair share the same pair of position and velocity vertex buffer objects.

The reason we use a set of vertex buffer objects for positions and velocities is so that we may use the ping pong strategy to read from a set of position/velocity while we write to another set using the transform feedback approach since we may not write to a transform feedback attribute when we are reading from it.

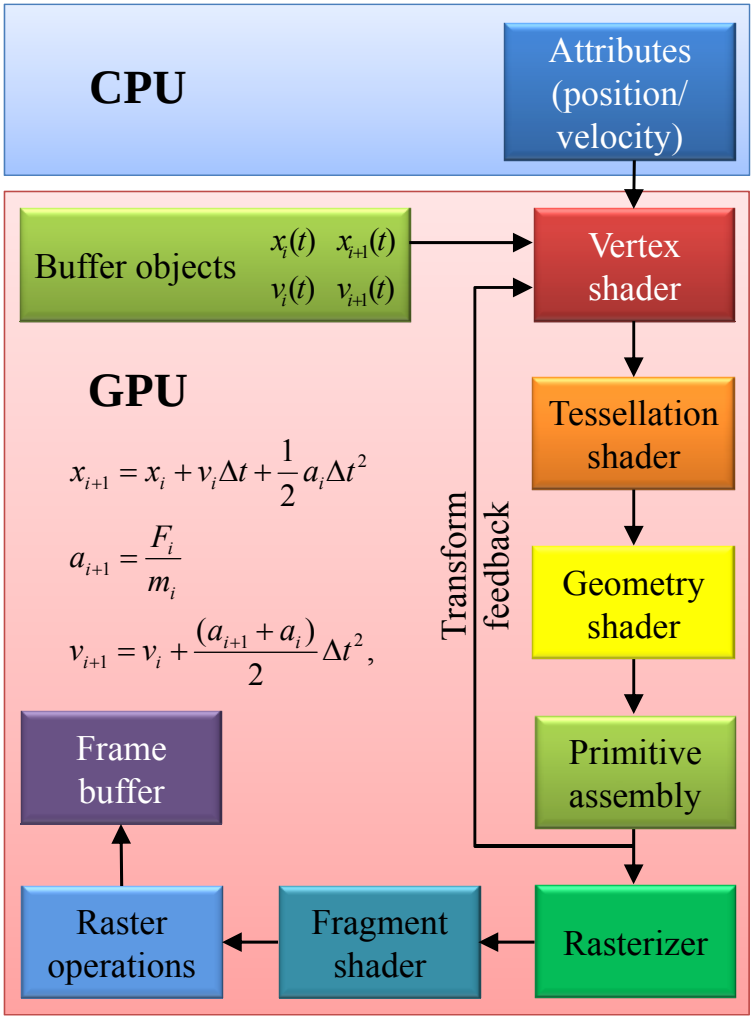


Figure 4-4. Proposed deformation pipeline using transform feedback

Referring to Figure 4-5 for the followings, each vertex array object stores a set of vertex buffer objects for storing per vertex position and velocity. Thus, at any time, one attribute is read from one array object, it is written using another array object. The usage flags for the position and velocity vertex buffer objects are set as dynamic (GL_DYNAMIC_COPY in OpenGL) since the data will be dynamically modified using the shaders. This gives an additional hint to the GPU so that it may put the buffers in the fastest accessible memory.

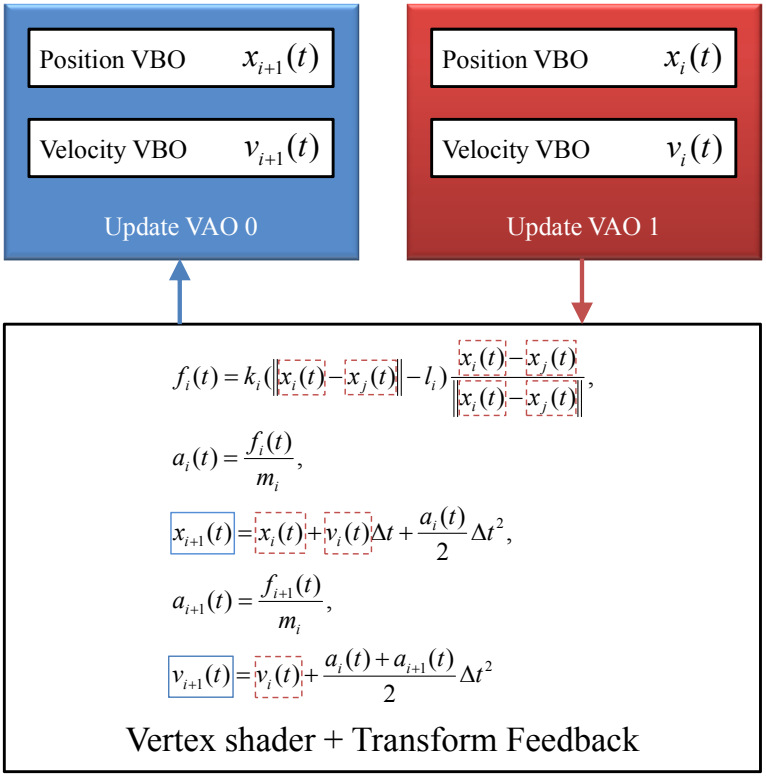


Figure 4-5. The vertex array object and vertex buffer object setup for transform feedback: the blue/solid rectangles show the attributes written to a vertex array object; the red/dashed rectangles show the attributes being read simultaneously from another vertex array object

4.3.1 Transform Feedback Attribute Setup

All of the per particle attributes such as the current position (x), previous position (x_p) and velocity (v) are passed to the transform feedback vertex shader as per vertex attributes. The moment matrices (M_{inv}), the vector (d_i) calculated in Eq. (4.12) are pre-computed and stored into a *constantTexture*. The particle neighbor distance (r_i), support radius (h_i), the particle mass (m_i) and volume (vol_i) are pre-computed and stored into a set of textures.

The position and velocity vertex buffer objects are also bound to texture buffer target to enable easy access in the vertex shader. At the same time, another pair of position and velocity vertex buffer objects is bound as a transfer feedback buffer. This enables the vertex shader to output the results to a vertex buffer object directly.

The application pushes a set of point positions (each element is a *vec4* with position coordinates in the first three components and mass in the fourth component) and velocities (each element is a *vec4* containing velocities in xyz components and volume of the point in the fourth component). In addition, the neighborhood information is stored in a pair of textures. Each element of the first neighborhood texture (*vec4*) contains the displacement of the current point to its neighbor (in xyz coordinates) and its global index in the fourth component w. The second neighborhood texture contains the vector d_i (see Eq. (4.12)) and the kernel weight for the neighbor (w_{ij}) (see Eq. (4.8)).

4.3.2 Transform Feedback Dataflow

Referring to Figure 4-6, in each iteration, we swap between the two buffers to alternate the read/write pathways. At initialization, the update vertex array object is bound for writing. At the same time, we bind the appropriate vertex buffer objects for reading the current positions and velocities to the current transform feedback buffer (by issuing a call to glBindBufferBase OpenGL function).

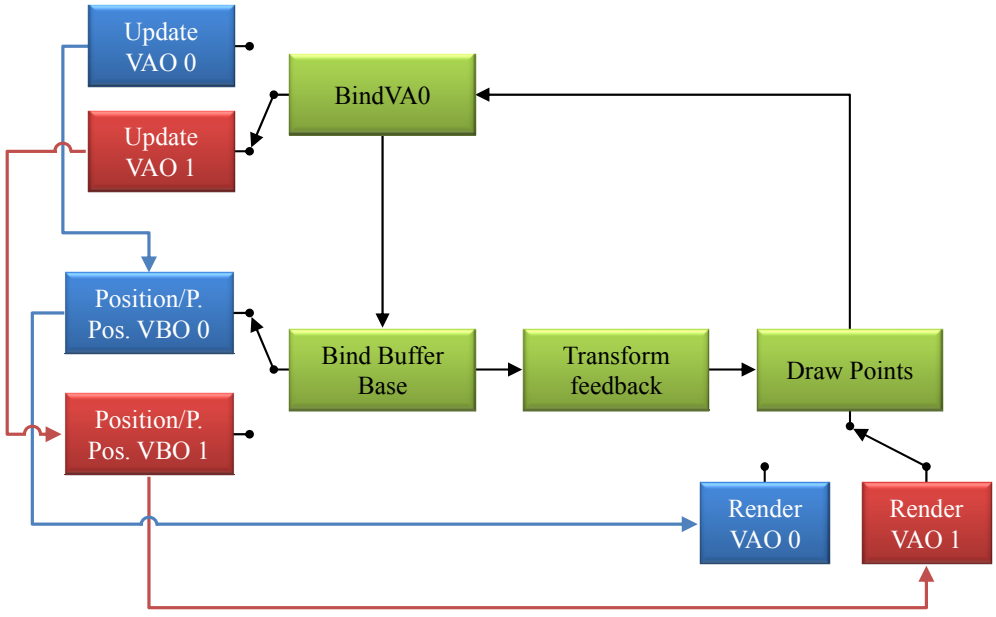


Figure 4-6. The transform feedback dataflow between the update and render cycle

The rasterizer is disabled to prevent the execution of the rest of the programmable pipeline. The draw point call is issued to allow writing vertices to the vertex buffer object. The transform feedback is then disabled. The amount of primitives transformed could be queried by issuing a query. Following the transform feedback, the rasterizer is enabled and then the points are drawn. This time, the render vertex array objects are bound. This renders the deformed points on screen.

A recent extension GL_ARB_transform_feedback2, in the core OpenGL 4.0 has eased the transform feedback along with the handling of its buffer object/s. It provides specific states that allow transform feedback to store references to the vertex buffer object. The extension also adds some functions that allow rendering of primitives without the need to query the number of primitives written through the transform feedback.

4.3.3 Evaluation of Forces

The forces are evaluated per-vertex. Rather than storing the isotropic elasticity matrix (D) as a 6×6 matrix, we store it into a single *vec3* attribute containing the three non-zero entries. The inverse mass matrix is pre-calculated at initialization on the CPU and then transferred to the GPU as a per vertex attribute.

Prior to invoking the vertex shader, we calculate the displacement field on the CPU using $u_i=(x_i - x^0_i)$ where, x_i is the current position of the point mass and x^0_i is the initial undeformed position. These displacements are then stored into a texture to be read back later in the vertex shader.

The first thing that we calculate on the vertex shader is the external force such as the gravity force and the velocity damping force. Once these forces are calculated, we then calculate the Jacobians, the stresses and the internal forces. This calculation requires the fetching of the neighbor node displacements. For efficient fetching of the neighbor

indices, we keep just a single 2D texture neighborListTexture (each element of which is a *vec4* containing the index of the neighbor, the distances (*r* and *h*) and the kernel weight (*w*)). The neighbor index is used to fetch the displacements (for Jacobian calculation) and later to obtain the internal forces (see Section 4.2).

4.3.4 Numerical Integration

Following the calculation of the forces, we perform the leap-frog integration. The leap-frog integration works by evaluating the velocities at ½ time step offset from the position [136]. Mathematically, we may give the leap-frog integration as:

$$\begin{aligned} x_{i+1} &= x_i + v_i \Delta t + a_i \frac{\Delta t^2}{2} \\ v_{i+1} &= v_i + \frac{a_i + a_{i+1}}{2} \Delta t \end{aligned} \tag{4.17}$$

The advantage that we obtain from this integration scheme is that it conserves the energy of the system. The expression given in Eq. (4.17) may be converted directly into shader statements. The current position (x_i) and velocity (v_i) are passed in as per vertex attributes whereas the next position (x_{i+1}) and velocity (v_{i+1}) are written to using the transform feedback mechanism.

4.4 Cell Projection of Transformed Volume

The cell projection involves algorithms that render the volume by first approximating it using a polygonal representation. Each of these cells has some transparency assigned to it based on its scalar value. Then, this polygonal representation is projected and finally blended on the screen. One of the most common methods is the projected tetrahedra (PT) algorithm [113].

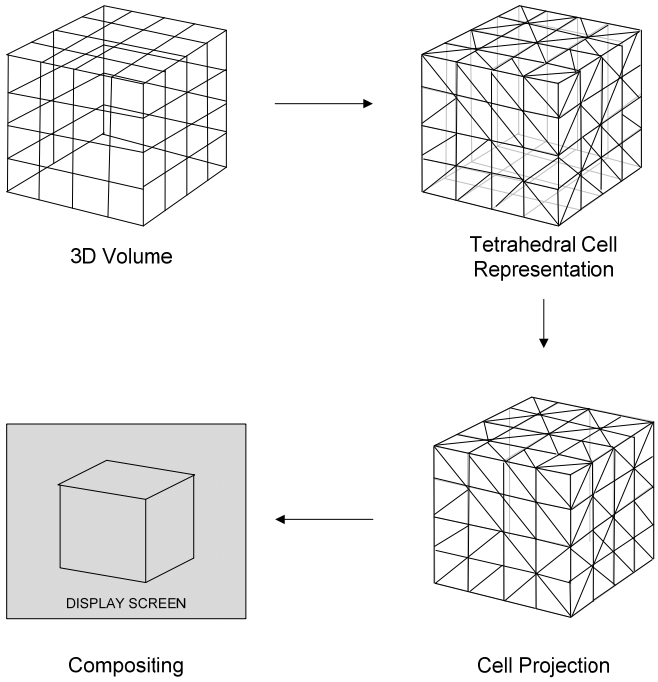


Figure 4-7. The cell projection pipeline

This algorithm first decomposes the volume into tetrahedral cells. These tetrahedra are then classified based on the eye position. Then, they are projected on the screen. Triangles are then constructed from these tetrahedral projections. Finally, color and transparency are linearly interpolated in screen space.

Figure 4-7 shows the cell projection pipeline. In order to generate tetrahedra in screen space, four possible cases (see Figure 4-8) are defined. The determination of the possible case to use is based on the dot product between the face normal and the viewpoint. Care must be taken to avoid overlapping edges which generate artifacts. The screen space triangles are composited using the average color of the two segments given as

$$C(t_1) = \alpha \frac{C_p(t_0) + C_p(t_1)}{2} + (1-\alpha)C(t_0)$$

where, $C(t)$ is the color value at segment t, $C_p(t)$ is the particle color at segment t, t_0 is the start segment, t_1 is the end segment and α is the opacity of the current segment.

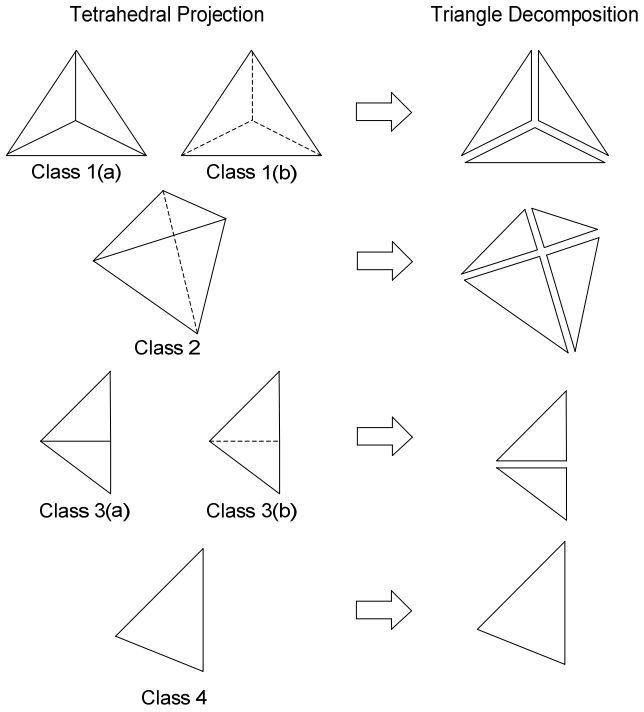


Figure 4-8. All possible tetrahedral configurations and their associated triangulations

In the PT algorithm, the render time is independent of the output image size. Moreover, since triangles can be rendered very fast in hardware, interactive rendering performance can be achieved. Voxels can be processed independently which makes this algorithm an ideal candidate for GPU implementation.

The down side of the algorithm is that it needs to generate the tetrahedral cells from the volume and then sort them in back-to-front order based on the direction of the viewer. The linear interpolation produces artifacts when the dataset or the transfer function contains high frequencies.

The color of the current segment can be approximated using the average color of the current and the next segments. This approximation does not take the nonlinearity of projection into consideration which could be approximated by assigning more weight to the front segment and less weight to the rear segment. This method is ideal for datasets containing moderate number of voxels along with relatively smooth variations.

Some artifacts can be reduced using hardware assisted texture mapping for interpolation of color and opacities [137]. The reason for artifacts in the rendering is because linear interpolation in screen space produces quadratic error term and that can be rectified by approximating the correct opacity term (α) given as

$$\alpha = 1 - \exp(-\tau l)$$

where, τ is the extinction coefficient, and l is the integration segment length.

Williams et al. [138] generalized the cell projection to linear as well as quadratic tetrahedra. In their high accuracy volume rendering (HIAC) system, they provide mixed cell types which include tetrahedra, prisms, wedges etc. In volume regions with rapid changes, they introduce smaller voxels using adaptive refinement and approximate the volume integral in those regions using splatting with piecewise linear transfer function.

In view of the various aspects of modeling and rendering requirements, for fast rendering of transformed points, we use the hardware assisted projected tetrahedra (HAPT) algorithm [122]. The deformation pipeline outputs a pair of buffer objects. These are used directly as positions for cell projection. Thus, we do not need to transfer the deformation results to CPU. The nonrigid transformation is incorporated in the HAPT pipeline by streaming our deformed points directly, as shown in Figure 4-9.

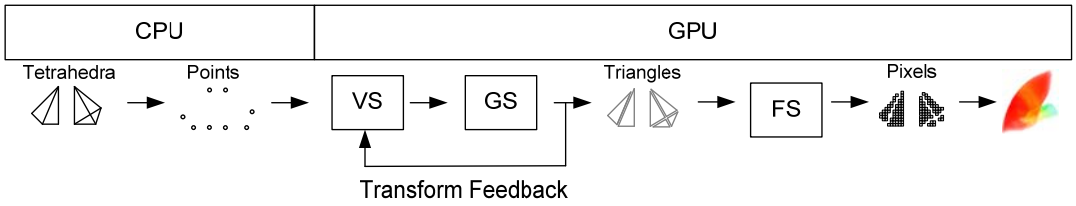


Figure 4-9. Augmenting our novel deformation pipeline in the HAPT algorithm

The HAPT algorithm stores positions in texture objects. In our implementation, we reuse the buffer objects output from our deformation pipeline directly. This involves no CPU read back and thus, the data can be visualized directly.

4.5 Experimental Results and Performance Assessment

Now that we have described how the whole deformation and rendering pipeline is setup, we can examine some of the deformation results obtained. These experiments were conducted on our test machine, a Dell Precision T7500 workstation Windows 7 64-bit with a 2.27 GHz Intel Xeon CPU with 4 GB of RAM. The machine is equipped with an NVIDIA Quadro FX 5800 GPU card with 4096 MB of dedicated video memory.

The output resolution for all of the experiments is 1024×1024 pixels. We applied the meshless FEM to two volumetric datasets, the spx dataset (containing 12936 tetrahedra) and liver dataset (containing 3912 tetrahedra). We allowed the spx dataset to fall under gravity while the liver dataset was manipulated by the user. The output result with deformation and rendering are shown in Figure 4-10 and 4-11 respectively. The results clearly indicate that the meshless FEM is suitable for volumetric deformation. Thanks to the convenience of our proposed deformation pipeline, we can integrate it directly into the HAPT algorithm.

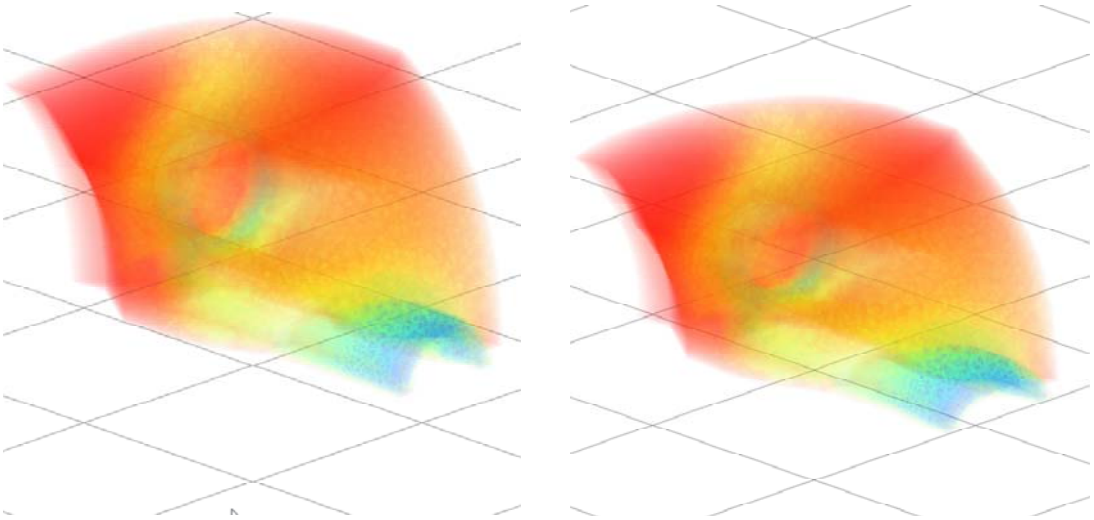


Figure 4-10. Two frames of deformation of the spx dataset falling due to gravity onto the floor, generated by our GPU-based meshless deformation pipeline using transform feedback

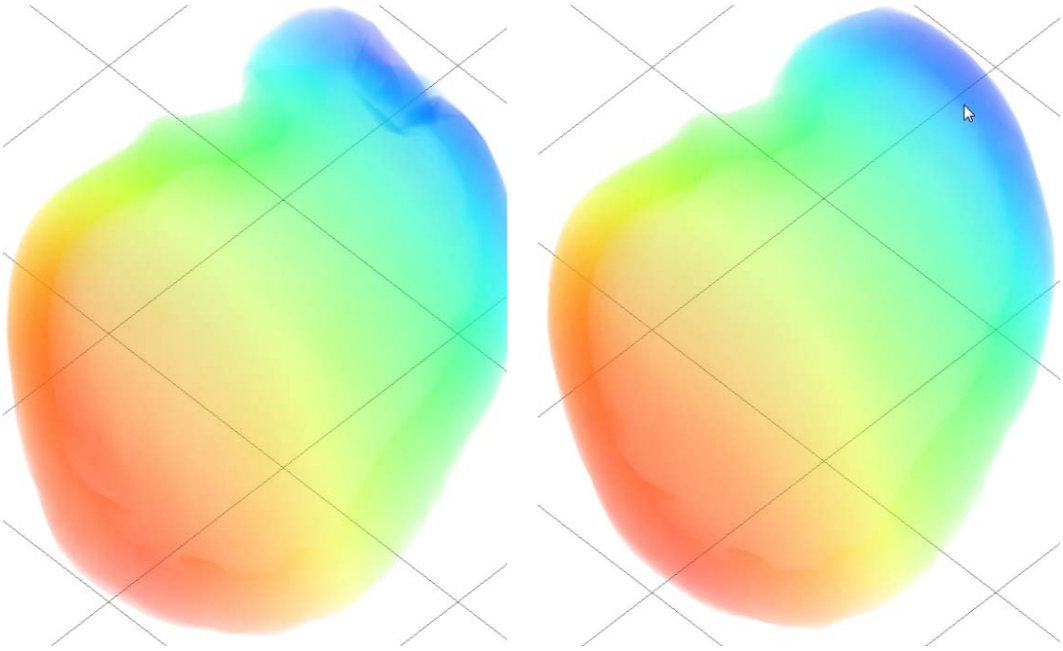


Figure 4-11. Two frames of deformation of the liver dataset with user interaction, generated by our meshless deformation pipeline using transform feedback

To assess the performance of the proposed meshless deformation, we conduct another experiment whereby we compare our GPU-based meshless FEM with an already optimized CPU implementation. For this experiment, the bar model containing varying number of points (from 250 to 10000) were used. These timings only include the time for deformation and they do not include the time for rendering. These results are given in Figure 4-12.

These clearly show that the performance of the meshless FEM scales up well with the large datasets and we gain a speed up of up to 3.3 times compared to an optimized CPU implementation. From this graph, it is clear that the runtime for CPU would increase exponentially for larger datasets and the performance gap between the CPU and the GPU would widen further.

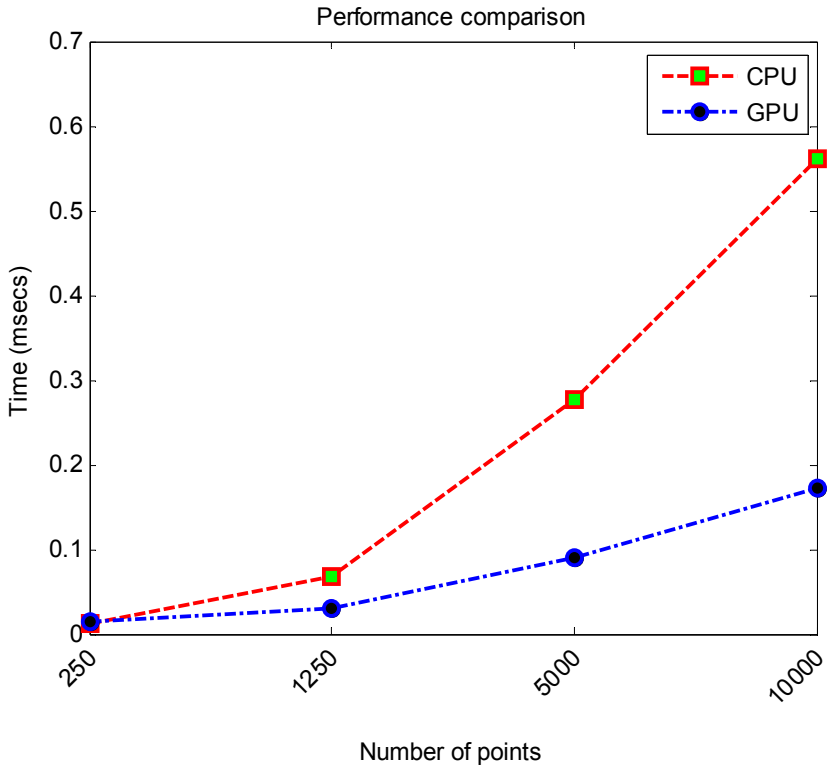


Figure 4-12. Performance of meshless FEM on GPU and CPU

It is the first time a meshless FEM model is applied to unstructured volumetric datasets. Our method uses the leap-frog integration which is semi-implicit whereas the existing tetrahedral FEM approaches use implicit integration schemes. The amount of time required for convergence in the case of implicit integration schemes is much more as compared to semi-implicit integration.

Moreover, the simulation system built using such formulation requires the stiffness matrix assembly which is then solved using an iterative solver such as the Newton Raphson method or Conjugate Gradients (CG) method. Such stiffness assemblies are not required in meshless FEM. Therefore, ours converges much faster. Nevertheless, for completeness, in the third experiment, we compare the performance of meshless FEM with an implicit tetrahedral FEM [103] program. These results are presented in Table 4.2.

Table 4.2. Comparison of meshless FEM against implicit tetrahedral FEM solver

Dataset	Total Tetrahedra	Frame rate (in frames per second)	
		Implicit FEM	Meshless FEM
liver	3912	118.50-178.70	249.50-331.01
spx	12936	76.70-81.90	124.80-128.23
raptor	19409	40.30-43.80	71.22-71.71

As expected, the performance of meshless FEM is better as compared with an implicit tetrahedral FEM. Our meshless FEM is based on a semi-implicit integration scheme which does not require an iterative solver as is required for the implicit tetrahedral FEM.

Qualitative Assessment

In all of the deformation approaches (including meshless deformation approaches), the quality of deformation and validation is typically given by comparing the deformation of a uniform beam against the deformation produced by using the linear FEM as given in Figure 4-1. We compare the deformation produced by linear FEM (Figure 4-1 (a)) to the deformation produced by our meshless FEM approach (Figure 4-1(b)).

We extend the experiments to evaluate the quality/accuracy of deformation on the number of points used in the meshless deformation. For this experiment, two horizontal beams were used with 11×4×4=176 points (the red colored beam) and 19×6×6=684 points (the blue colored beam). The rest of the simulation configurations are same for both of the beams i.e. the Young modulus of 220000 psi, the Poisson ratio of 0.4 and the neighborhood size of 10 points. These results are given in Figure 4-13. Since the density of the bar is a function of the total number of points, the overall weight of the bar increases as the number of points increase. This causes the bar to bend more as the number of points increase.

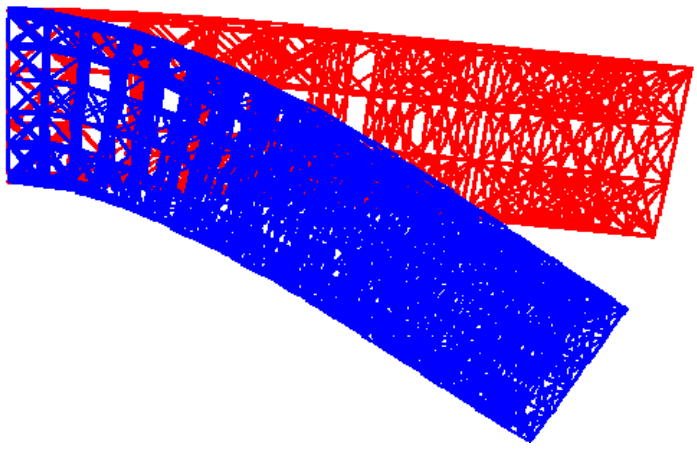


Figure 4-13. Comparison of deformation of a horizontal beam with 2 different resolutions of points showing the point resolution of 11×4×4=176 points in red and 19×6×6=684 points in blue.

In addition to changing the number of points, in another experiment, we altered the neighborhood size from 10 to 20 to see its effect on the quality and accuracy of deformation. These results are given in Figure 4-14. As can be seen, the output remains the same as in Figure 4-13.

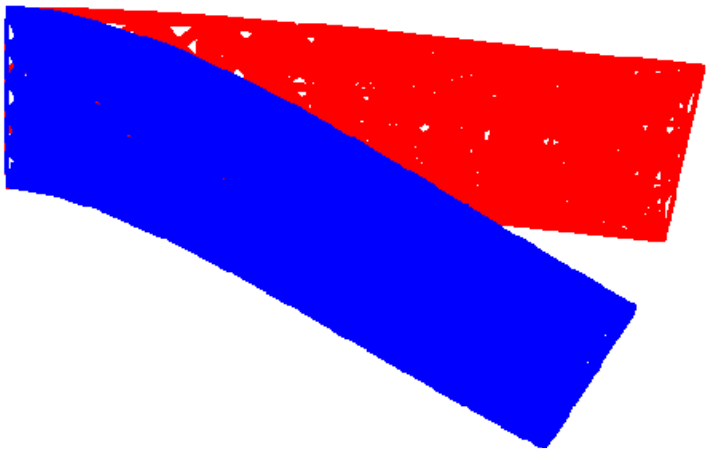


Figure 4-14. Comparison of deformation of a horizontal bar with the same point set resolution as in Figure 4-13 but with different neighborhood size.

4.6 Summary

We have applied meshless FEM to nonrigid volumetric deformation, leveraging on our novel GPU-based deformation pipeline. Using the transform feedback mechanism for deformation entirely on the GPU, interactive visualization of deformation on large volumetric dataset is made possible. We may summarize our main contributions as follows:

- We have applied meshless FEM for volumetric deformation.
- We have integrated the meshless FEM into a novel deformation pipeline, taking advantages of the transform feedback mechanism.
- We have integrated our novel deformation pipeline into the HAPT algorithm.

There are some considerations on the meshless FEM. First, the deformation result is directly dependent on the number of mesh points used. More points generally give a better approximation and vice versa. Moreover, since this method uses least square approximation, it cannot work with collinear points and thus, it needs a 3D sparse point set. We presented two implementations: (i) using scatter operations which performs in place updates for better performance and (ii) using gather operations. Unfortunately scattering is unavailable in the shader APIs and therefore the performance suffers specially in the case of high-resolution datasets.

To tackle such a performance loss, we can think of two strategies:

i. In modern OpenGL 4.2, there is a new image load-store extension which allows a shader program to read from and write to arbitrary GPU memory location. This may be used to do scattered writes. Since the hardware used in this study did not support OpenGL 4.2, this approach could not be verified.

ii. We may use a CUDA kernel to do scattered writes, alongside a GLSL shader. In this way, CUDA may be used for scattered data writes as well as for processing the more computationally demanding steps. This hybrid scheme however requires a more rigorous treatment and that will possibly be a future research direction.

5 NONLINEAR DIRECTIONAL VOLUMETRIC DATA INTERPOLATION

5.1 Introduction

While modeling and simulation technologies are studied for nonrigid volumetric transformation in the previous chapters, advanced interpolation algorithms should also be explored for quality data generation. As was concluded in the review chapter, all of the volume rendering algorithms differ in how they do sampling of the 3D volume data. The quality of sampling is largely dependent on the dataset resolution and the interpolation technique used to generate the in-between data.

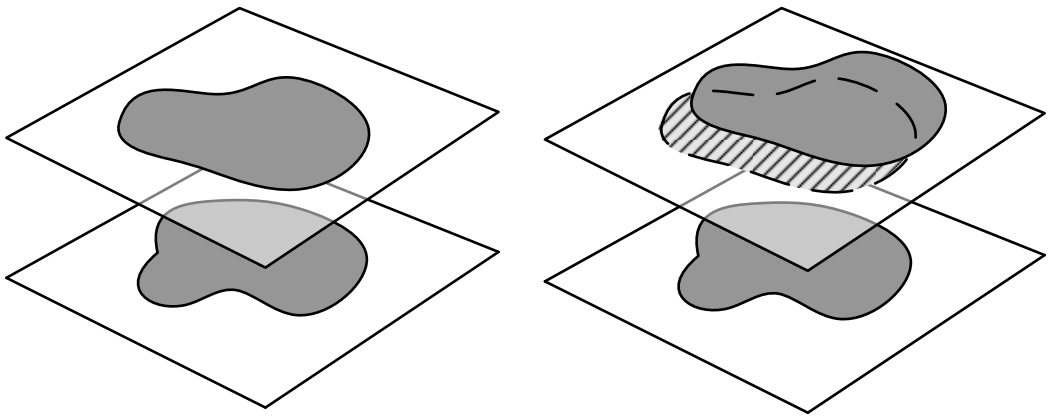
Data interpolation becomes much more challenging for deformable image acquisition. For example, in our joint research with National Cancer Centre Singapore, Confocal Laser Endomicroscopy (CLE) is used as an *in vivo* 3D cellular imaging tool. The image datasets acquired from CLE are generally with low z-depth resolution, which causes artifacts in volume rendering. Intermediate data between consecutive cross-sectional images needs to be generated. In the prior work, assuming no deformation or linear deformation in imaging processes, one uses shape-based trilinear interpolation which reduces the artifacts to some extent, but blocky artifacts are clearly visible especially in the areas of low spatial resolution.

One of the reasons for such artifacts is that trilinear interpolation does not take into account the underlying gradient of volume for interpolation. When handling deformable and nonuniform subjects such as human tissues, these shape-based interpolation techniques cannot effectively remove such artifacts since they rely on hard classification for identification of shape features. Some cubic and B-spline interpolation techniques have been proposed [67-68], but they require additional memory and increased

computational time. In addition, earlier methods also rely on multi-pass approaches with CPU intervention which unfortunately reduces the performance of the algorithms.

CLE represents a typical imaging process in which nonrigid deformation happens during the optical slicing of the live tissues. As illustrated in Figure 5-1, first, intra-slice image deformation is caused by physiological morphing of the subjects (see Figure 5-1 (a)); and secondly, inter-slice volumetric deformation is caused by both physiological morphing and unstable handheld probe during the z-depth scanning (see Figure 5-1 (b)) The dashed region demarcates the place where the upper slice should have been acquired if there were no probe movement. In practice, slight movement of the handheld probe produces significant movement in the acquired slice since the image acquisition takes place in a region of around $512\times512\times250\ \mu m^3$.

The conventional interpolation methods can hardly deal with the nonrigid deformation in the volumetric datasets. To solve this problem, we propose a novel nonlinear directional interpolation method, which uses the underlying gradient orientation to obtain the nonuniform interpolation directions. This allows the inter-slice image data to be generated accurately, even in the regions of low spatial resolution.



(a) (b)

Figure 5-1. Intra-slice and inter-slice deformation due to (a) physiological morphing and (b) unstable handheld probe

The rest of this chapter is organized as follows. Section 5.2 describes the concepts and algorithm, especially our model for capturing the nonlinear intensity variation. The mapping of our model to GPU shaders is presented in Section 5.3. Experimental results and performance assessments are given in Section 5.4. And finally, Section 5.5 concludes this chapter.

5.2 The Nonlinear Interpolation Approach

In order to estimate the direction of interpolation, we identify the plane of interpolation. This is carried out by finding the gradient at the current voxel location. Once the gradient is obtained, the orthonormal pair of vectors is identified so that any point on the interpolation plane could be estimated. The obtained gradient and the gradient magnitudes are used to guide the interpolation direction.

5.2.1 Identification of Interpolation Plane

Let $f(x, y, z) : R^3 \rightarrow R$ be a function representing the density at (*x, y, z*). The gradient of this function can be given by the following partial derivatives:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right]^T$$

This can be approximated numerically using finite differences such as centered finite difference given by

$$\nabla f \approx \left[\begin{array}{c} \frac{f(x+1, y, z) - f(x-1, y, z)}{2} \\ \frac{f(x, y+1, z) - f(x, y-1, z)}{2} \\ \frac{f(x, y, z+1) - f(x, y, z-1)}{2} \end{array} \right]^T \tag{5.1}$$

The local gradient obtained by using Eq. (5.1) gives the orientation vector (***N***), which will be used from here onward as the surface normal.

Once the normal vector ***N*** is obtained, we can use it to compute the interpolation plane. To get any point on this plane, we need a pair of orthonormal vectors (***U*** and ***V***) perpendicular to the normal vector ***N***. The orthonormal pair may be obtained by first projecting the 3 components using the maximum component of the normal vector. This allows us to reduce the calculation from 3D space to 2D space.

Since there are three possible outcomes, we have three conditions for each axis. The axis on which the projection is done is set to 0. The perpendicular vector to another vector in 2D can be obtained by interchanging the components and changing the sign of one of the components. This is repeated for each component depending on the projection outcomes, mathematically given as follows:

$$\vec{V} = \begin{cases} (0, -N.z, N.y) \Leftrightarrow |N.x| < |N.y| and |N.x| < |N.z| \\ (-N.z, 0, N.x) \Leftrightarrow |N.y| < |N.x| and |N.y| < |N.z| \\ (-N.y, N.x, 0) \Leftrightarrow |N.z| < |N.x| and |N.z| < |N.y| \end{cases}$$

$$\hat{V} = \frac{\vec{V}}{|\vec{V}|}$$

$$\hat{U} = \hat{V} \times \hat{N}$$

where $\boldsymbol{N}.x$, $\boldsymbol{N}.y$ and $\boldsymbol{N}.z$ are the three components of the normal vector ***N***. The two orthonormal vectors (***U*** and ***V***) can be used to obtain any point on the plane of interpolation.

5.2.2 Extracting Missing Intensities

After the plane of interpolation is defined, we can obtain the intersection point of this plane with the dataset slices. As illustrated in Figure 5-2, P_0 is the current location, A' is the projection of point A on the slice B, and we need to obtain the locations of points A and B on slices A and B respectively. To do this, we first find out the distance (H_{sd}) between points A and P_0, which can be computed by geometrical interpretation and trigonometry as follows:

$$\frac{S_d}{2} = H_{sd}\cos(\theta)$$
$$\theta = a\sin(N \cdot Z)$$

where, S_d is the distance between slice A and slice B, θ is the angle that the interpolation plane makes with the ***Z*** axis, and ***Z*** is the unit vector pointing in the positive ***Z*** axis. Using the distance $\boldsymbol{H_{sd}}$, we can obtain the two point locations A and B using

$$A = P_0 + H_{sd}.V$$
$$B = P_0 - H_{sd}.V \qquad (5.2)$$

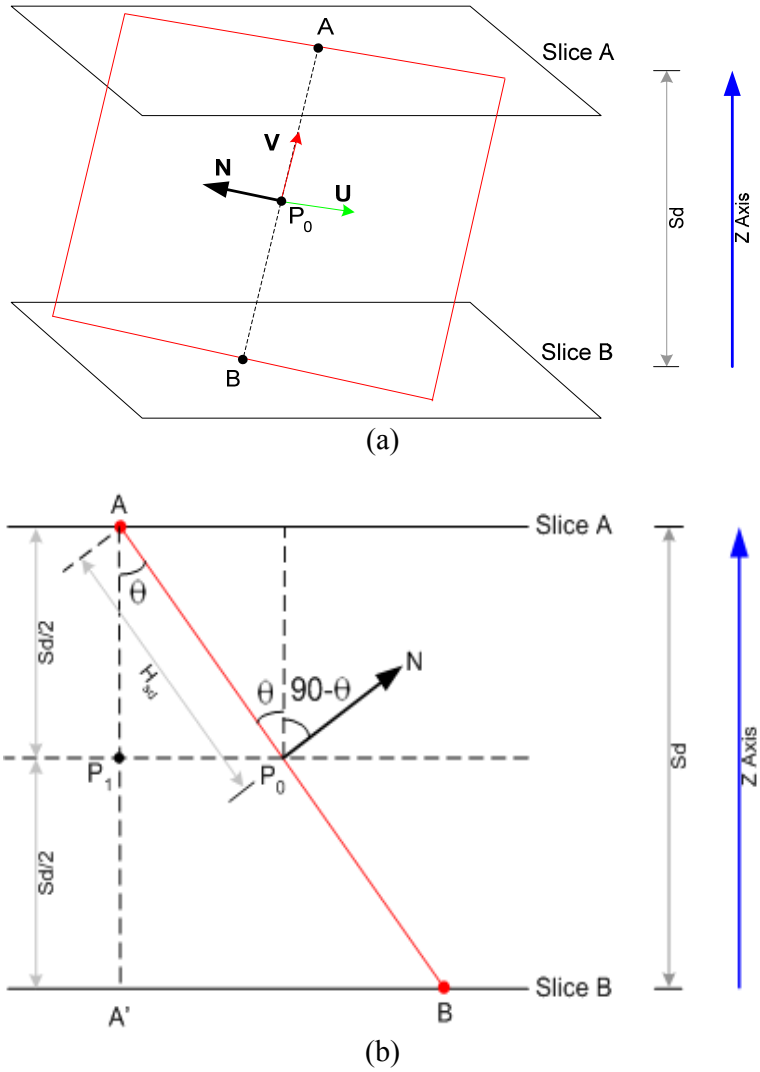


Figure 5-2. Inter-slice interpolation plane: (a) 3D view and (b) side view

Once we have the point locations on the slices, we can use these locations as a lookup into the slices to obtain the intensity values I_A and I_B. The intensity value for the current location (P_0) is obtained by linear interpolation of the two values (I_A and I_B). While the formula given in Eq. (5.2) assumes that the ***V*** axis is the interpolation axis, this may not always be the case. Instead, we need to calculate the amount of overlap between the ***V*** axis and the ***Z*** axis (the default interpolation axis). This can be evaluated using a dot product between the two unit vectors and multiplying this scalar to the ***Z*** axis.

5.2.3 Edge Characterization and Nonlinear Intensity Variation

For a low depth resolution dataset such as the CLE, it is important to estimate the nonlinear variation in intensities to properly capture the missing details. This can be carried out by formulating a function based on the gradient magnitude of the underlying volume intensities. Thus, the interpolation direction will follow the high frequency features and revert to linear interpolation in the case of low frequency features especially in relatively homogeneous regions.

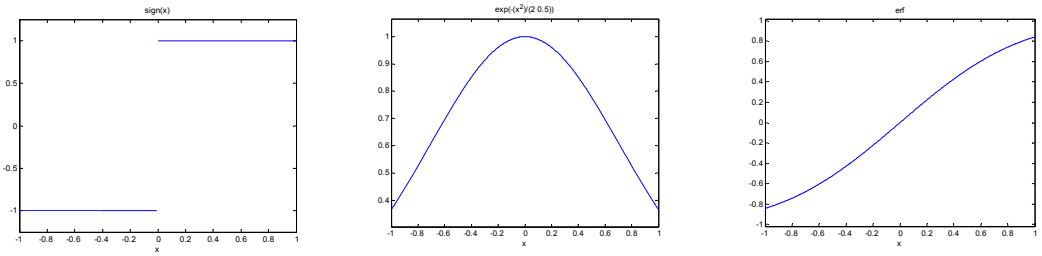
Using nonlinear interpolation, we are able to preserve the ideal edge without blurring and ringing artifacts. An ideal edge is usually given by the step function as

$$H(x) = \int_{-\infty}^{x} \delta(t)dt$$

where, δ is the Dirac delta function. During reconstruction, the function can be calculated by a convolution with a Gaussian function (G) as given by

$$g(x) = \int_{-\infty}^{\infty} H(t)G(x-t)dt$$

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



(a) (b) (c)

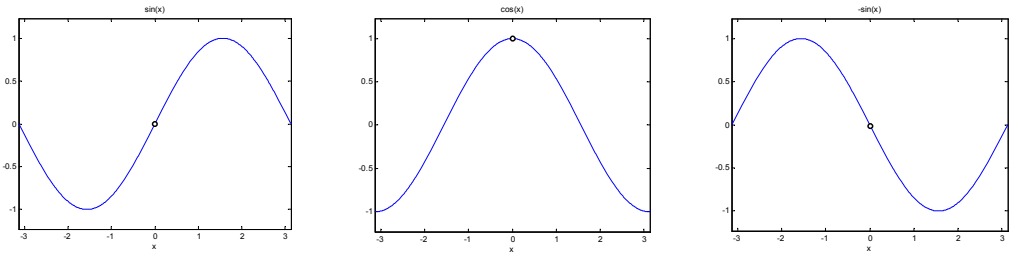
Figure 5-3. Edge characterization showing the ideal edge (a) using the sign function (b) convolved with the Gaussian function, and (c) producing the error function (erf)

The resulting function (g) from this convolution is the error function (*erf*) given as

$$g(x) = \frac{2}{\sqrt{x}} \int_0^x e^{-t^2} dt$$

This result is shown in Figure 5-3.

We need to identify the position of the ideal edge. For a one dimensional function f, this can be obtained by looking at the maximum of f' and zero crossing of f''. We show that the position of the edge indeed corresponds to the maximum of f' and zero crossing of f'' by looking at the example of the *sin* function. This is shown in Figure 5-4. The position of the edge is shown by the point. Note that the position of edge shown in (a) corresponds to the maximum of first derivative and zero crossing of second derivative.



(a) (b) (c)

Figure 5-4. The edge characterization of (a) the sin function for identifying the position of the ideal edge. The first derivative (b) and the second derivatives (c) are also plotted

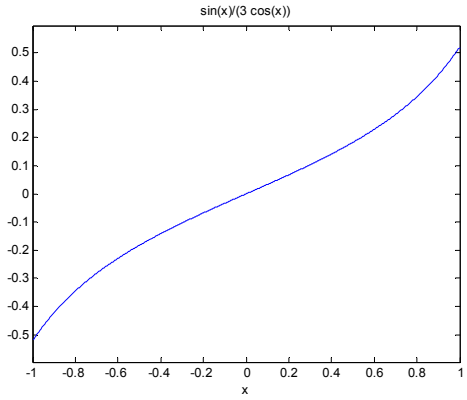


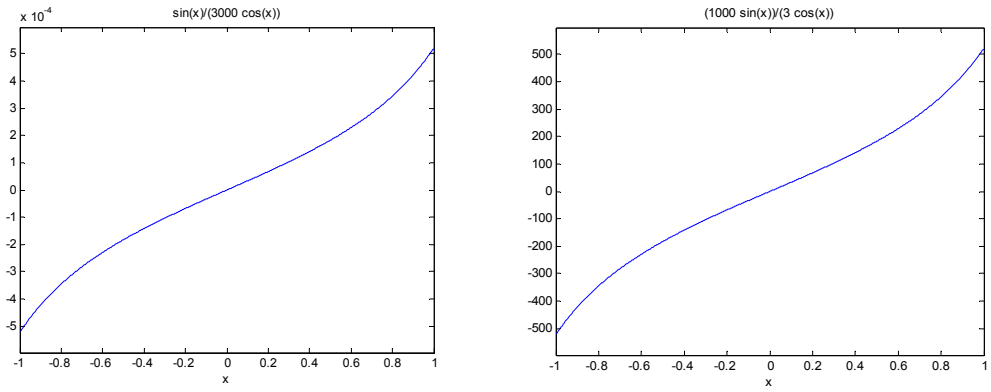
Figure 5-5. Edge characterization using the proposed function

We might be tempted to use the first derivative (*f'*) and the second derivative (*f''*) of the one dimensional function (*f*) to identify the transitional region of the ideal edge. However, the scale of the derivatives depends on the range of the function. If on the other hand the ratio of the two derivatives is used, we get a scale invariant term. Therefore, to capture the nonlinear variation in intensities across the plane of interpolation, we introduce the following scale invariant function:

$$m(x) = \frac{-|\nabla f| f''(x)}{f'(x)}$$

where, *f* is the input dataset, $|\nabla f|$ is the gradient magnitude, *f'* is the first derivative and *f''* is the second derivative (Laplacian) evaluated at *x*.

To demonstrate, we apply this to the *sin* function example used earlier, using the gradient magnitude as a parameter and change its value from -1 to 1 (see Figure 5-5). Note that the range of the function (*m*) is scaled from -0.5 to 0.5 in Figure 5-5 to match the range of the erf function in Figure 5-3 (c). To show that the function (*m*) is indeed scale invariant, we multiply and then divide the function by 1000. This is given in Figure 5-6.



(a) (b)

Figure 5-6. Scaling the proposed function by a scale of (a) 1/1000 and (b) 1000

As can be seen, the function remains invariant irrespective of the scale used. Calculation of the function (*m)* requires two things: the first and the second derivative. To avoid any discontinuities, the input dataset must be filtered with a low-pass filter. Assuming a low pass filtered two dimensional function $\tilde{f}(x,y): R^2 \to R$ for example a low pass filtered image, the first derivative is approximated using the finite difference approximation.

The second derivative may be obtained by evaluating the Laplacian of the image at the current sample point. In the low pass filtered image, we may obtain the Laplacian by convolution with the Barlett filter kernel [104] given as:

$$\Delta f \approx \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix} \otimes \tilde{f}$$

where, $\tilde{f}$ is the low pass filtered image and $\otimes$ is the convolution operator. For a three-dimensional dataset, the convolution may be evaluated on-the-fly in the 3x3x3 neighborhood.

Once we have the Laplacian and the derivatives estimated, we may then obtain the function (*m)*. Incorporating function (*m*) changes Eq. (5.2) to the followings:

$$\begin{aligned} A &= P_0 + H_{sd}.V.m \\ B &= P_0 - H_{sd}.V.m \end{aligned} \tag{5.3}$$

The function (*m)* not only approximates a convolved ideal edge, but also modulates the interpolation direction with the gradient magnitude at the current position.

Hence, if the current point is in a homogeneous region, the gradient magnitude will be close to 0, and the interpolation falls back to the linear interpolation; otherwise, the point is in non-homogeneous region, and this function extracts the nonlinear distribution which takes into consideration the local gradient magnitude and direction.

5.3 Shaders for Parallel Interpolation Processes

The proposed inter-slice directional interpolation algorithm is intensively used in rendering of nonrigid volumetric datasets, which requires the algorithm to be executed in realtime. The slice-by-slice acquisition from the CLE allows us to push data incrementally into the GPU texture memory. Once the data has been transferred to the GPU, it can be resampled directly in the fragment shader. Since the fragment shader is a highly parallelized processor, this enables real-time processing of the 3D image datasets, such as the CLE images used in our case, at video rate.

5.3.1 Asynchronous Transfer to GPU Memory

When a slice is obtained for reconstruction of the volumetric dataset, it is pushed into the GPU texture memory. For asynchronous transfer through the direct memory access (DMA), we use the pixel buffer objects (PBO). For efficient and fast transfers, we use a pair of PBOs and then ping pong between them as each new slice is scanned.

There are two methods to update the 3D texture memory in OpenGL. The first requires recreating a new texture every time a new slice comes in, while the other updates an existing texture using convenience function provided by the graphics API. For example, OpenGL provides glTexSubImage3D and glCopyTexSubImage3D functions to update a portion of the 3D texture memory. This however only works if the total size of the volume is known. We target at a real-time imaging and 3D reconstruction system in which the total number of scans in the current session is unknown, hence, we opt for the first method of recreating the 3D texture whenever a new slice is acquired.

5.3.2 GPU Mapping

During interpolation, our algorithm estimates the Laplacian and the local gradient in the local neighborhood through texture lookups from the GPU texture memory. Since there is a strong coherency between data samples, there are more cache hits than cache misses. In addition, for better performance, we cache the samples used to obtain the Laplacian to estimate the gradient. This results in an efficient and high performance GPU implementation on the fragment shader. Figure 5-7 gives the mapping of our algorithm on the GPU.

To make an efficient use of the single-instruction-multiple-data (SIMD) architecture of the modern GPU, we invent a vectorization process. We obtain the gradient information on demand by doing texture lookups in the fragment shader. This saves a lot of memory space because we do not require an additional 3D texture for gradient. List 5.1 details the shader for calculating the gradient on demand.

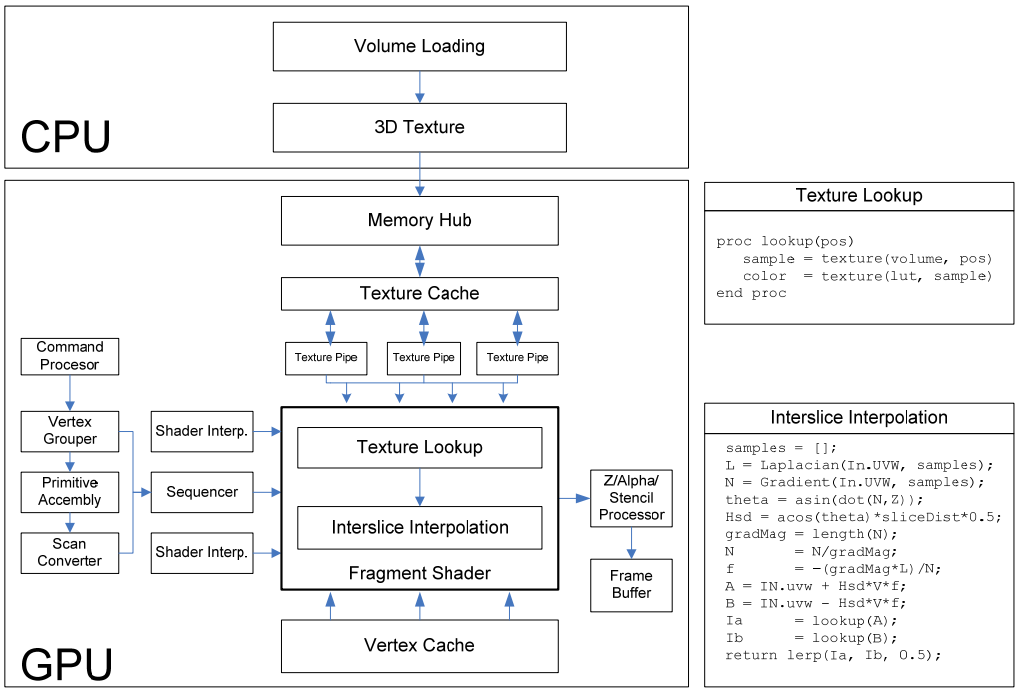


Figure 5-7. Mapping of the nonlinear interpolation on GPU

```
#define DELTA 0.01
proc GetGradient(volume:sampler3D, x:vec3)
      var s1: vec3;
      var s2: vec3;

      s1.x = texture(volume, x-vec3(DELTA,0,0));
      s2.x = texture(volume, x+vec3(DELTA,0,0));
      s1.y = texture(volume, x-vec3(0,DELTA,0));
      s2.y = texture(volume, x+vec3(0,DELTA,0));
      s1.z = texture(volume, x-vec3(0,0,DELTA));
      s2.z = texture(volume, x+vec3(0,0,DELTA));

      return (s2-s1)/2.0;
end proc
```

List 5.1. The shader for calculation of the gradient on demand

5.3.3 Resampling Streaming Data

Our proposed algorithm does not require the entire dataset to be in the GPU texture memory because as soon as the second slice comes in, we can estimate the in-between data using the two slices. This resampling requires some prior information such as the distance between individual scans. Since this information is known, (for example 4μm in our scanning device), we can estimate the exact position the slice is in realtime.

Therefore, every new slice being acquired is pushed into the GPU texture memory, it is resampled using the proposed nonlinear interpolation scheme and then visualized at the same time. This real-time resampling and visualization is possible due to tight coupling between the two components which are part of a common fragment shader. List 5.2 details the implementation of our algorithm using the fragment shader.

```
for each fragment do
  uvw = In.UVW;
  samples = [];
  L = GetLaplacian(uvw, samples);
  N = GetGradient(uvw, samples);
  Z = [0,0,1]
  theta = asin(dot(N,Z));
  Hsd   = acos(theta)*sliceDist*0.5;
  absN  = abs(N);

  if (absN.x<absN.y && absN.x < absN.z)
     V = [0,-N.z, N.y];
  else if (absN.y < absN.x && absN.y < absN.z)
     V = [-N.z,0,N.y];
  else if (absN.z < absN.x && absN.z < absN.y)
     V = [-N.y,N.x,0];

  gradMag = length(N);
  N       = N/gradMag;
  m       = -(gradMag*L)/N;

  sampleA = texture(volume,(uvw – Hsd*V*m));
  sampleB = texture(volume,(uvw + Hsd*V*m));
  Ia      = texture(lut, sampleA);
  Ib      = texture(lut, sampleB);

  return lerp(Ia, Ib, 0.5);
end for
```

List 5.2. The shader for nonrigid volumetric interpolation

We first estimate the Laplacian by looking up the local neighborhood of the current voxel. The samples that were used to obtain the Laplacian are cached for calculating the gradient. The orthonormal vectors on the interpolation plane are then identified. Finally, the dataset slices are sampled using the proposed nonlinear function.

Once the sample values are obtained, they are classified using the transfer function. The classified values are then interpolated to obtain the new density values. Since the proposed approach is implemented entirely on the GPU, it enjoys 16-bit floating point

precision in all the steps of the pipeline, which improves the rendering quality. Moreover, since the dataset is only transferred when a new slice comes in, the performance is improved as is evident from the statistics given in the next section.

5.4 Experimental Results and Performance Assessment

All of the rendering results described in this chapter are generated on an Intel Pentium 4, 3 GHz CPU with 1 GB of RAM. The PC is equipped with the NVIDIA GeForce 7800 GTX GPU with 256 MB of memory. To evaluate the performance of our algorithm, we first conducted experiments on several datasets. The initial tests were carried out on a set of twenty 16-bit images, acquired from a confocal fluorescence endomicroscopy system (OptiScan Five 1), which is the dorsal surface of the pig tongue CLE dataset (courtesy of our collaborators at National Cancer Centre Singapore).

Each slice is captured by z-depth scanning with a resolution of 1024×1024 pixels. For all of the datasets, GPU-based 3D texture slicing technique is used to generate real-time direct volume rendering and the opacity transfer function is applied to highlight the visualization results. Especially, for the CLE datasets, the pseudo-color transfer function (as shown in Figure 5-8) is used to demarcate different regions.

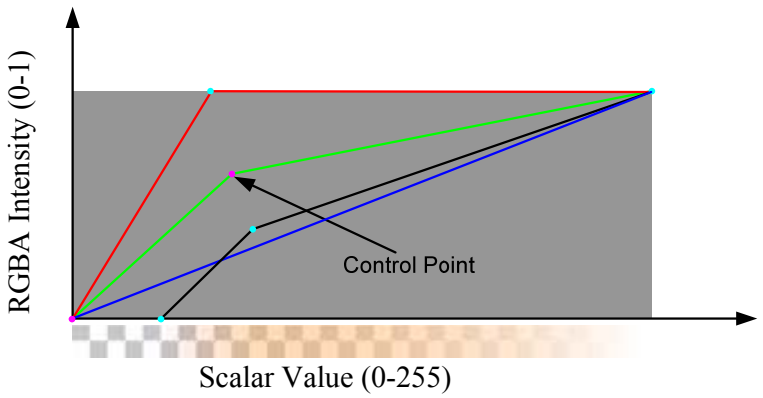
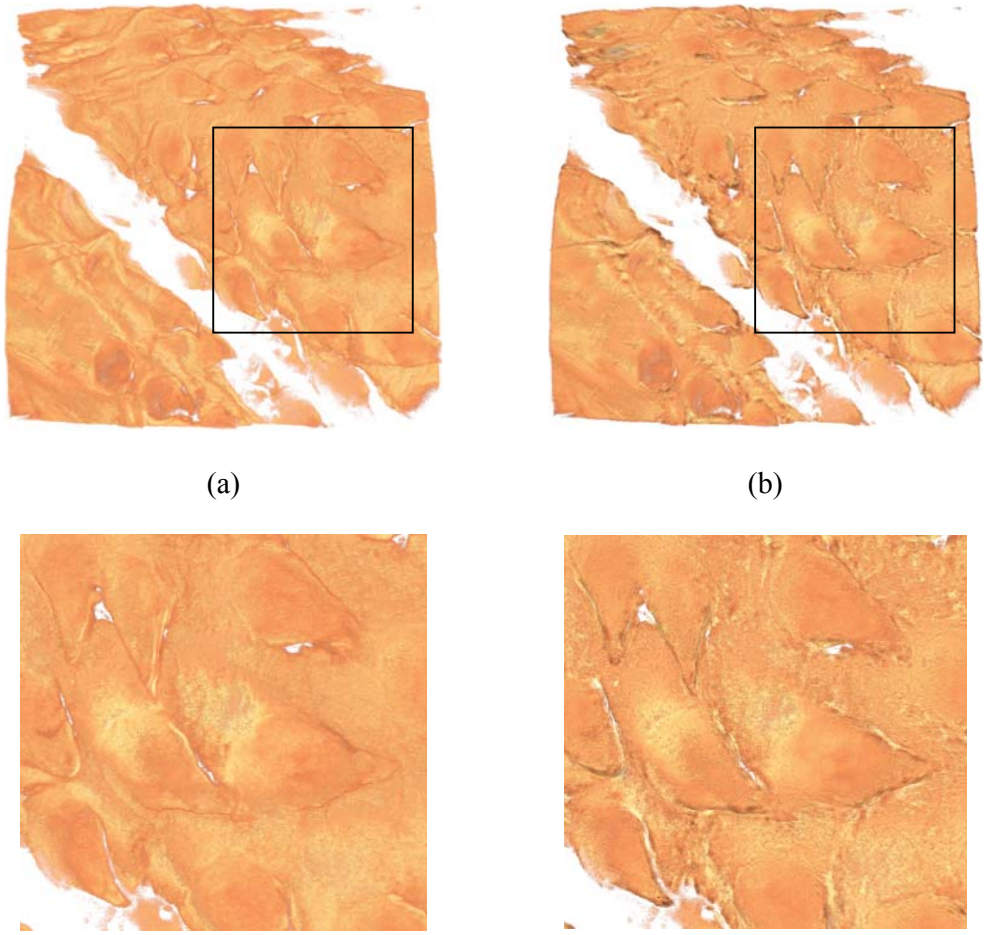


Figure 5-8. Transfer function used in the volume rendering

The first experiment was carried out on the pig tongue dataset. There were 20 slices obtained from the Optiscan Five 1 scanning device courtesy of our collaborators. These slices show the dorsal surface of the pig tongue. The most significant feature in this dataset is the nodular extensions called the *filiform papillae*. The results after application of the proposed interpolation scheme are shown in Figure 5-9.



(a) (b)

(c) (d)

Figure 5-9. Volume rendering showing the dorsal surface of the pig tongue CLE dataset with (a,c) trilinear interpolation (b,d) inter-slice directional interpolation on the GPU (Note that the surface detail is visible in (b,d) that is invisible in (a,c))

As illustrated in Figure 5-9, our method reconstructs the tissue details better as compared with the trilinear interpolation (especially, note the zoom-in rectangle regions where both the edges and the inner lining of the nodular extension are more detailed and visible). When compared with the trilinear interpolation result Figure 5-9 (a, c), the surface detail as well as the filiform papillae are distinctly identifiable in Figure 5-9 (b, d).

The next experiment was carried out to examine the execution time of our algorithm to compare with existing interpolation methods. We implemented various interpolation methods for comparisons, using two additional 8-bit datasets the visible male head (120×120×130) and Bonsai (256×256×128). These results are given in Table 5.1.

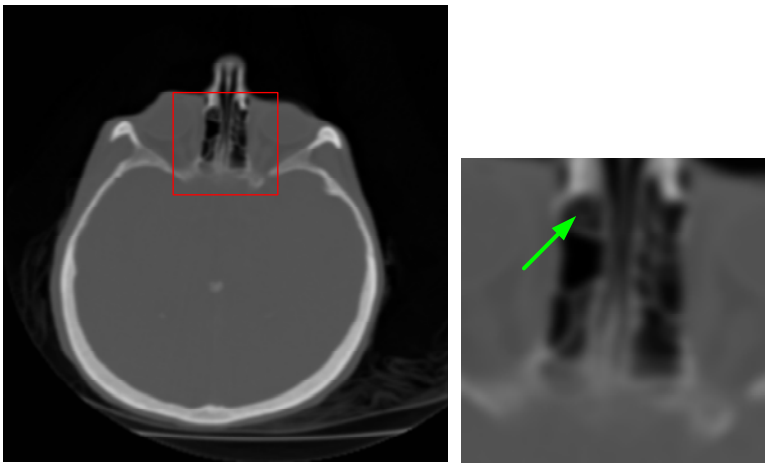
Table 5.1. Performance comparison of different interpolation schemes

Interpolation Method	**Time (secs.)**	
	Visible Male Head	**Bonsai**
Trilinear	0.005	0.022
Hermite	0.240	1.170
Blackman	0.270	1.200
Hamming	0.240	1.210
Gaussian	0.340	2.060
Cubic B-spline	0.440	2.090
Catmull Rom	0.430	2.120
Truncated Sinc	1.080	5.700
Our surface oriented interslice directional interpolation	0.018	0.027

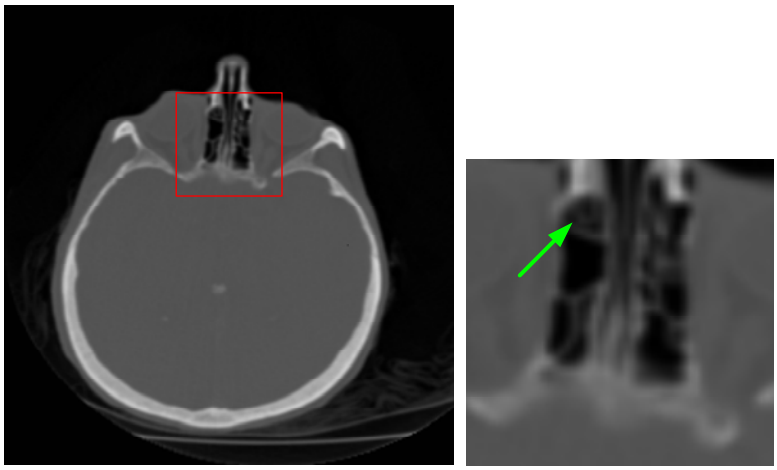
Instead of calculating the gradient for the whole volume, we calculated the gradient on demand in a fragment shader as described in section 5.3. For an efficient GPU implementation, we introduce vectorization of the calculations so that GPU SIMD advantages could be exploited. Furthermore, instead of invoking a component by component calculation, we opted for dot products that are translated into a single GPU

instruction. The results show that these steps help us to obtain up to 40 times faster rendering times as compared with the other interpolation methods.

To do a quantitative assessment of quality of the proposed method, we extended the experiment to a 16-bit CTHead dataset (512×512×106). We resampled the dataset using three different interpolation schemes namely the linear interpolation, our proposed inter-slice directional 3D interpolation and the cubic B-spline interpolation scheme. The resampling results comparing our proposed scheme to cubic B-spline interpolation scheme are shown in Figure 5-10.



(a)



(b)

Figure 5-10. Comparison of interpolation quality showing in (a) cubic B-spline interpolation and in (b) our proposed inter-slice directional interpolation (Note that cubic B-spline interpolation smoothes out intricate details as indicated by the green arrow)

When compared to the interpolation results of the cubic B-spline method, our method produces sharper images. The reason for this is because cubic B-spline method uses a very large neighborhood to estimate the interpolated value. This smoothes out intricate detail as shown in Figure 5-10 (a). In contrast, our approach uses a smaller neighborhood therefore, it preserves intricate details and the resampled output is sharper as shown in Figure 5-10 (b).

A quantitative error analysis is required to see how well our interpolation scheme performs. We compared the mean squared error (MSE) and the peak signal to noise ratio (PSNR) for the three filtering methods, cubic B-spline, our proposed interslice directional method and the trilinear interpolation method. These results are charted in Figure 5-11.

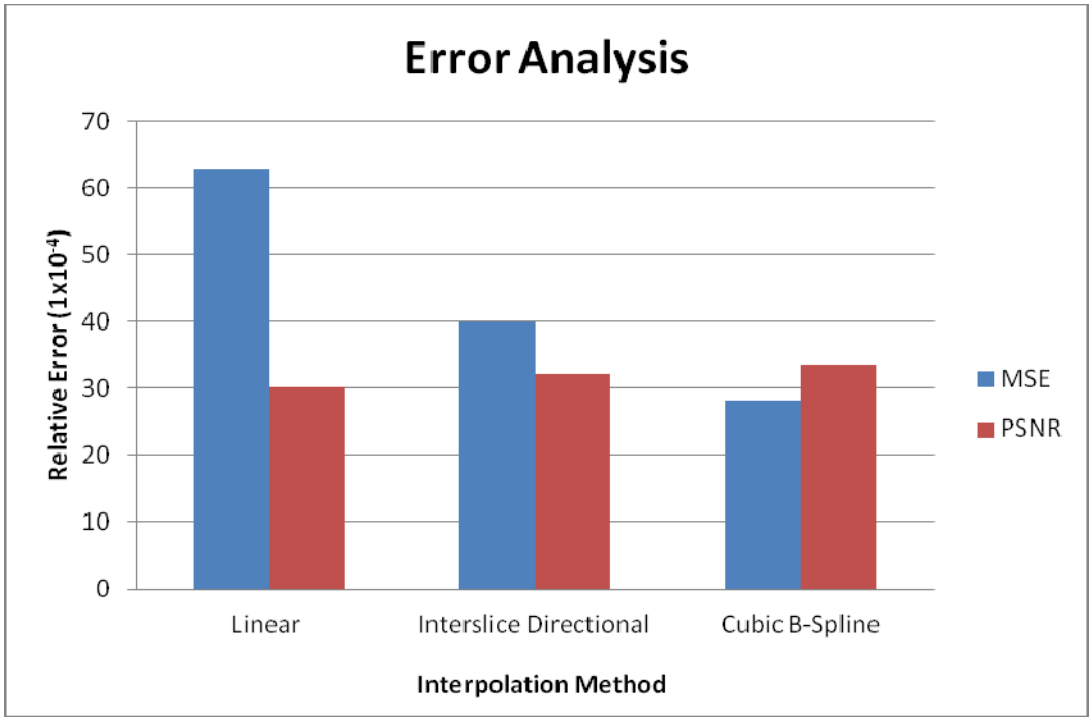


Figure 5-11. Error analysis of various interpolation schemes

As can be seen, for our proposed inter-slice directional interpolation, the MSE (the lower the better) is significantly lower as compared to linear interpolation. At the same time the PSNR (the higher the better) is higher compared to linear interpolation and is very close to cubic B-spline interpolation. Since higher order interpolation schemes such as the cubic B-spline require a large neighborhood, their relative error is the lowest whereas their PSNR is the highest. Our method on the other hand serves as a good compromise between the simple trilinear interpolation and the cubic B-spline interpolation method.

Since all of the previous nonlinear methods have been applied to 2D images, it is difficult to generate rigorous experiments for a 3D datasets since the formulations of the cubic B-spline interpolation [107, 108], the edge directed interpolation [104], and Nohalo [106] require some pre-processing steps which would unfortunately increase the processing time for a 3D dataset and, therefore, the total pre-processing time will be biased. Instead, to do a fair comparison, we compare the existing nonlinear interpolation schemes on the GPU using three metrics, the total texture access requirements, the requirement for multiple passes and the requirement for linear filtering. These results are given in Table 5.2.

The cubic B-spline interpolation method of [107, 108] using prefiltering requires 8 texture fetches (2D case) and 16 texture fetches (3D case) for each pixel for interpolation and it assumes that the texture is filtered using linear filtering. If using unfiltered texture, it requires 16 texture fetches (2D case) and 64 texture fetches (3D case). The edge directed interpolation method [104] requires 14 texture access in all for 2D case (9 for Laplacian, 4 for gradient and 1 for sampling). However, there is no requirement of linear filtering.

Table 5.2. Comparison of our method to existing nonlinear interpolation methods

Method	Texture Accesses		Pre-processing Required	Multi-pass	Linear Filtering Required
	2D	3D			
Cubic B-spline (Prefilter) [108]	8	16	Yes	Yes	Yes
Cubic B-spline (No Prefilter)	16	64	No	No	No
Edge directed [104]	14	35	No	No	No
Nohalo [106]	17	21	Yes	Yes	No
Our method (linear using Eq. 5.2)	6	8	No	No	No
Our method (nonlinear using Eq. 5.3)	11	29	No	No	No

For the 3D case, the cubic B-spline method requires 35 texture fetches (27 for Laplacian, 6 for gradient and 2 for sampling). However, there is no requirement of linear filtering. The Nohalo method requires 17 texture accesses for 2D case (5 for gradient, and 12 for double density sampling). For the 3D case, it needs 21 texture accesses (9 for gradient, and 12 for double density sampling). This method needs multiple passes for estimating the gradients followed by the double density image.

For the 2D case, our method requires 6 texture accesses for linear case, that is, using Eq. 5.2 (4 for gradient and 2 for sampling) and 11 for nonlinear case, that is, using Eq. 5.3 (9 accesses for Laplacian and 2 for sampling). For the 3D case, our method requires 8 texture accesses for linear (6 for gradient and 2 for sampling) and 29 for nonlinear interpolation (27 accesses for Laplacian and 2 for sampling) since we reuse the samples that were used to calculate the Laplacian, for gradient calculation.

Our method works in a single-pass and there is no requirement of linear filtering as is required for cubic B-spline filtering with prefilter. Moreover, there is no pre-processing requirement for our method as is needed to calculate the prefilter coefficients for cubic B-

spline interpolation [108]. Of all of the existing nonlinear interpolation methods, our method requires the least number of texture accesses. Not only is our method efficient in terms of the total texture accesses it requires, it also works in a single-pass without any texture filtering or pre-processing requirement.

5.5 Summary

We have presented a nonlinear directional interpolation technique that takes the local gradient information into account using a scale invariant function. Compared with the shape-based techniques, our method requires neither pre-processing of the original data nor hard classification. Thus, it is suitable for nonuniform and deformable datasets. In addition, we also exploit the SIMD architecture of the modern GPU by vectorizing most of our calculations. We also utilize the efficient dot product instruction in favor of a component by component multiplication. The benefit of this shows up in the form of an unprecedented performance.

Both qualitative and quantitative assessments have been conducted. First, our method performs better than not only the trilinear interpolation but also the other nonlinear interpolation schemes such as the edge directed interpolation and the Nohalo method. Secondly, our method does not have any filtering requirement, in contrast to the cubic B-spline method. And lastly, our algorithm works in a single-pass, unlike the Nohalo method which requires multiple passes.

We have given an example implementation using the fragment shaders because it allowed us to integrate the proposed interpolation scheme in our CLE volume rendering system directly. With a few platform specific modifications, the same algorithm works for other parallel programming paradigms such as OpenCL and CUDA.

6 LOCALLY ADAPTIVE THRESHOLDING FOR FEATURE ENHANCEMENT

6.1 Introduction

In volume rendering, color and opacity values are assigned to give different material characteristics to the voxels using the transfer function [1-3]. The volume rendering result is dependent on this transfer function assignment. A good transfer function will highlight the targeted features whereas an improper one will not.

As an example, in an *in vivo* visualization of cellular images acquired from the CLE system [139], key issues include accurate identification of volumes of interest (VOI) in realtime and proper rendering with correct transfer functions. The intensity distributions in different datasets vary, and thus, there is no single transfer function that would highlight features in all datasets. Often the designing of the transfer function is a trial and error process and domain knowledge is crucial for good transfer function design [53].

We have conducted a comprehensive survey on the respective technologies and concluded that there is a need for algorithms to identify features in realtime for effective visualization. The main ideas of our algorithm come from the fact that statistical information contained in the dataset is sufficient to automate the design of a robust transfer function. This may then be refined by the user as needed. This helps to reduce the trial and error process significantly.

In section 6.2, we give a detailed description of our GPU-based feature extraction pipeline. The feature rendering pipeline is presented in section 6.3. Experimental results and performance assessment are given in section 6.4. And finally, section 6.5 concludes this chapter.

6.2 Shaders for Real-time Feature Detection

For feature detection, we define a set of morphological operations. We first employ median filter to remove the noise in the dataset which is usually introduced during the scanning process. After median filtering, adaptive local thresholding is applied. This operation is given by,

$$\mu = \sum_{j=1}^{n}\sum_{i=1}^{n}\frac{f(x_i, y_j)}{n^2}$$
$$T = \mu - C$$
$$y = \begin{bmatrix} 1 \Leftrightarrow f(x,y) > T \\ 0 \Leftrightarrow f(x,y) \le T \end{bmatrix}$$

where, $f(x_i,y_j)$ is the input image, μ is the mean value in an n×n neighborhood, T is the threshold value, and C is a constant.

Local adaptive thresholding is introduced for the CLE datasets in which illumination conditions change across the whole image and they greatly affect the downstream operations. With thresholding applied, the resulting image turns into a binary image. Following this, the erosion filter is applied on the image. Consider a binary image I and a structural element S in $\boldsymbol{Z}^2$. The erosion operation of image I with the structure element S is given by,

$$I \ominus S = \{z \mid (S)_z \subseteq I\} \tag{6.1}$$

Putting in words, Eq. (6.1) says that the erosion of an image I with structural element S is a set z such that S translated by z is a subset of I. For our case, we use a disc structure element of radius 3 pixels. The erosion operation removes spurious disconnected pixels. Then, this image is used as the subtraction mask which is applied to the original image to extract the cells. The complete process is illustrated in Figure 6-1.

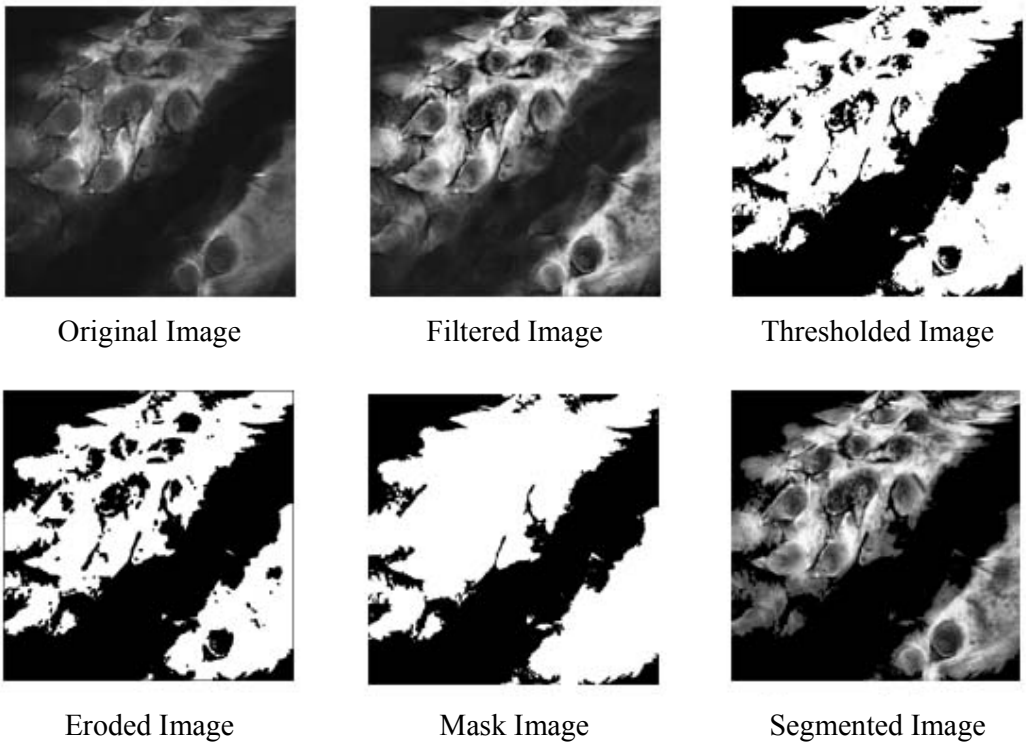


Figure 6-1. Feature detection process applied on the tongue dataset

6.2.1 The GPU Pipeline with the Fragment Shader

In our system, the feature extraction process is performed using the GPU in a pre-processing step on the CLE datasets. Every slice is loaded and a multi-pass approach is applied (using multiple render-to-texture functionality). We develop fragment shaders to accomplish most of the tasks. The pipeline is shown in Figure 6-2.

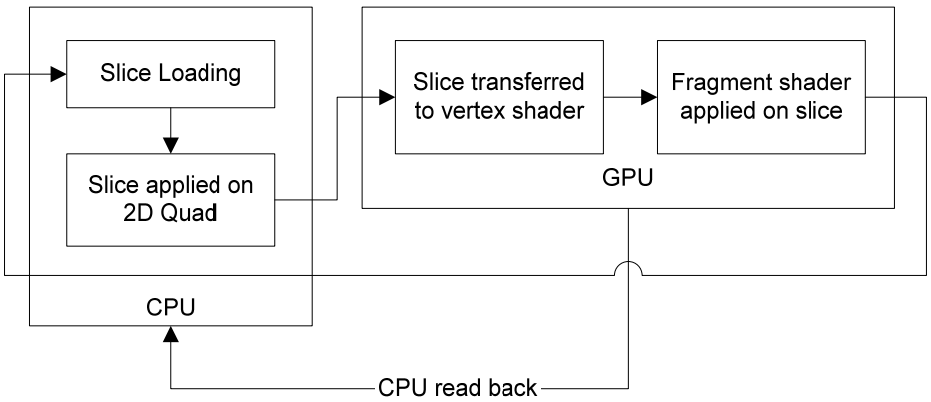


Figure 6-2. The GPU-based feature detection pipeline

The process starts on the CPU where we load the slices. After loading, the slices are applied on a 2D Quad of the same dimensions as the slice itself. Then, the fragment shaders are applied in multiple passes. Finally, the filtered slices are downloaded to the CPU where two operations are performed:

i. Feature database is created from the filtered slice

ii. The slice is stored in a 3D texture for volume rendering.

Note that all of this process apart from the slice loading is occurring on the GPU.

6.2.2 Transfer Function Assignment

Once the slice is filtered, the characterizing features such as the gradient magnitude, the feature centroid, and the local minimum and maximum intensities are identified and stored with a unique ID for each identified region. These features will be used during the creation of transfer functions in the volume rendering pipeline, as discussed in a later section.

For volume rendering of cells, a unique transfer function should be assigned to each identified region. From the feature database, the minimum and maximum intensities are used to demarcate a range of intensities. Only intensities in this range are highlighted by assigning opacities scaled by the gradient magnitudes and pseudo-colors. This process is illustrated in Figure 6-3.

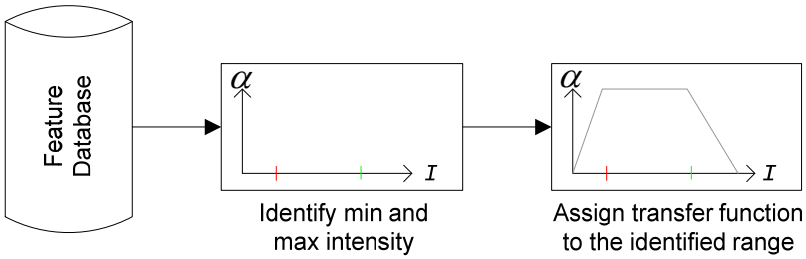


Figure 6-3. The transfer function assignment

6.3 Rendering of Features

While the GPU-based volume rendering algorithms have been reported in literature, we developed a new framework which combines 3D texture slicing with pre-integration. As a result, the speed of the algorithm has been significantly increased, enabling the video rate for processing of live cell imaging. We implement the Newton Cotes integration formulas in the fragment shader to remove the slicing artifacts and improve the rendering quality. There are two major stages of our pipeline.

6.3.1 The Pre-processing Stage

The OpenGL API provides lightweight frame buffer objects (FBO) for render-to-texture functionality [140]. Each FBO has multiple color attachments, each of which can have a color buffer attached to it. This enables writing at the same location in multiple textures at the same time on the GPU. This is a very powerful feature that enables multiple render targets. The pre-processing pipeline is described in pseudo code in List 6.1.

```
Attach the FBO to current rendering context;
for each slice s {
   for each shader sh {
      Set the Shader for sh;
      drawFullScreenQuad();
  }
}
Detach the FBO from the current rendering context;
Read GPU memory using DMA;
Extract the regional properties;
Generate the transfer function;
```

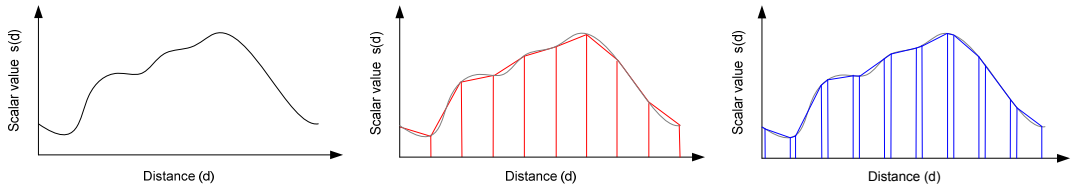
List 6.1. The pre-processing pipeline

Upon the completion of processing, the slice is read back from the GPU. For fast and asynchronous read back from the GPU, we utilize the pixel buffer object (PBO) that uses direct memory access (DMA) for data transfer. The transferred data is then used to create a 3D texture for direct volume rendering. The features such as the gradient magnitudes

are extracted from each slice and stored in a table. These features are then used to construct a transfer function.

6.3.2 Pre-Integrated Volume Rendering and GPU Implementations

Numerous texture slicing approaches tried to solve the sampling problem, however, none was a perfect solution. The fundamental problem of sampling the volume using a thin slice was replaced by pre-integration and slab based rendering. The principle behind pre-integrated volume rendering is to approximate the volume rendering integral using piecewise linear approximations [141]. This is shown in Figure 6-4.



(a) (b) (c)

Figure 6-4. Approximation of the volume density integral with distance showing in (a) the original density function, (b) the normal density sampling without pre-integration, and (c) the pre-integrated volume sampling

When pre-integrated classification is employed, slabs rather than slices are used for sampling the 3D volume using a piecewise linear approximation. The volume integral is then computed using a single table lookup for each segment. The parameters for the table lookup include the sample value at the start of the segment S_s, the sample value at the end of the segment S_e, and the distance d (the thickness of the slab) [141]. If we represent the volume as a function $f(\boldsymbol{X}):R^3 \rightarrow R$, then these values can be given as follows,

$$S_s = f(x_i.d)$$
$$S_e = f(x_{i+1}.d)$$

The opacities and associated colors for the ray integral become a function of S_f, S_b and d, given as

$$\alpha_i \approx 1 - e^{-\int_{i.d}^{(i+1)d} \kappa(f(x(l)))dl}$$

$$c_i \approx \int_0^1 q((1-t)S_f + tS_b).e^{(-\int_0^t \kappa((1-t)S_f + tS_b)dt')dt} \tag{6.2}$$

Eqs. (6.2) may be discretized by using Riemann sums and simplification [141] into the following form:

$$I \approx \sum_{i=0}^{n} c_i \prod_{j=0}^{i-1} (1-\alpha_i) \tag{6.3}$$

where, c_i are the colors and α_j are the opacities. This method utilizes multi-texturing and advanced texture fetch operations in the fragment shader. The transfer function is stored in a 2D texture and later looked up in the fragment shader.

The output produced by the pre-integrated rendering is much better especially when nonlinear transfer functions are applied. This is because the pre-integration table contains RGBA values for all possible scalar values in the dataset. Interpolation can therefore be carried out between the two adjacent slices to get the intermediate slice values rather than assigning constant values.

Using pre-integrated classification, a continuous scalar field can be sampled without increasing the sampling rate even if a nonlinear transfer function is applied. There are some problems with this approach. It assumes a uniform ray segment length for integration which fails for perspective projection especially extreme perspectives. Moreover, in the case of semi-transparent iso-surfaces, some of the pixels are rendered twice especially if the surface intersects the viewing ray at the front slice.

Another short coming is that since the pre-integration table is created as a pre-process, with every modification of the transfer function, the pre-integration texture has to be recalculated. To address this problem, we use integral functions to approximate Eq. (6.2)

and Eq. (6.3). We know that we can use combinations of integral functions to calculate integrals with arbitrary limits. If we have two functions *F(x)* and *G(t)* given as:

$$F(x) = \int_a^b f(x)dx$$

$$G(t) = \int_0^\infty f(t)dt$$

then, we can approximate *F(x)* using *G(t)* as follows

$$F(x) = G(b) - G(a)$$

This will require us to evaluate *G(t)* for all possible values. Using the integral functions, we can rewrite Eq. (6.2) as

$$\alpha_i \approx 1 - e^{-\int_0^1 \kappa((1-\omega)S_f + \omega S_b)dd\omega}$$

$$\alpha_i \approx 1 - e^{-\frac{d}{S_b - S_f}\int_{S_f}^{S_b} \kappa(s)ds}$$

$$\alpha_i \approx 1 - e^{-\frac{d}{S_b - S_f}(\tau(S_b) - \tau(S_f))}$$

$$\tau(s) = \int_0^s \kappa(s')ds'$$

In a similar fashion, color value c_i can be given as

$$c_i \approx \int_0^1 q((1-\omega)S_f + \omega S_b)dd\omega$$

$$c_i = \frac{d}{S_b - S_f}\int_{S_f}^{S_b} q(s)ds$$

$$c_i = \frac{d}{S_b - S_f}\int_{S_f}^{S_b} R(s_b) - R(s_f)$$

$$R(s) = \int_0^s q(s')ds'$$

The pre-integration table is calculated on the GPU to allow real-time modification of the transfer function. For 3D texture based pre-integrated volume rendering, a pre-integration table is generated using the transfer function extracted in the previous step. An OpenGL based widget is constructed that can be tweaked to directly modify the transfer function at

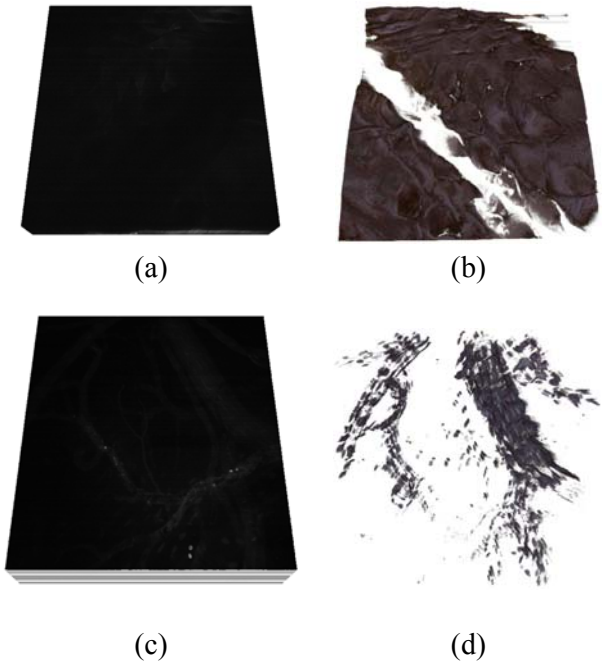
run-time. A point light source is also added to show the shaded iso-surface.

6.4 Experimental Results and Performance Assessment

All of the renderings described in this chapter were generated on an Intel Pentium 4, 3 GHz CPU with 1 GB of RAM. The PC was equipped with the NVIDIA GeForce 7800 GTX GPU with 256 MB of memory. We conducted tests on two datasets acquired from the Optiscan Five 1 endomicroscopy system, courtesy of our collaborators at National Cancer Centre Singapore.

The first dataset is the 16-bit tongue dataset consisting of 20 depth scans each with a resolution of 1024×1024 pixels. The second dataset is the brain blood vessel dataset [131, 132] consisting of 57 scans each with a resolution of 1024×1024 pixels. Figure 6-5 shows the rendering results using our rendering framework. The transfer functions used for the rendering are shown in Figure 6-6. In Figure 6-5 (a) and (c), linear transfer functions are assigned while in Figure 6-5 (b) and (d), automatic feature detection using the pipeline discussed previously is used. Note that the interesting features in the dataset are automatically highlighted by assigning unique opacity and pseudo-color transfer functions.

Clinical assessment of the acquired cellular images relies on two major classifications, namely, separation of the tissues from the background, and segmentation of specific regions (in our case, to highlight fluorescent cells) from the others within the dataset. As can be clearly seen in Figure 6-5 (b) and (d), these two functions have been successfully realized.

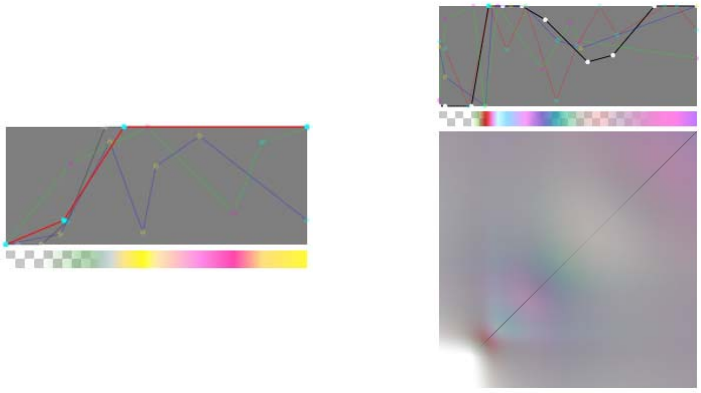


(a) (b)

(c) (d)

Figure 6-5. Comparison of rendering results (a,c) Tongue/mouse brain blood vessel dataset rendered without feature-based assignment, (b,d) Tongue/mouse brain blood vessel dataset rendered using our feature-detected volume rendering method. Note that the feature detection pipeline automatically removes air and identifies features instantly

In another series of experiments, we compared our approach to the position function based approach of Rezk-Salama et al. [44]. The position function tells us how far the current data value is from the edge so it may assist us in identifying a good transfer function. The calculation of the position function requires some pre-processing time which is dependent on the dataset resolution. On the other hand since our approach uses the GPU efficiently, the pre-processing time of our proposed pipeline is significantly less (see Table 6.1).



(a) (b)

Figure 6-6. The transfer functions assigned to the datasets in Figure 6-5

The adaptive thresholding shader is given in List 6.2. The process starts with a texture lookup to obtain the color value at the current fragment position. Once this color value is obtained, the mean value is determined in a neighborhood of 7x7 pixels. This neighborhood size was determined by considerable experimentation. The mean value is then used to threshold the current fragment color. The GPU mapping of the algorithm is given in Figure 6-7.

Table 6.1. Comparison of preprocessing times

Dataset	**Technique**	**Preprocessing Time (secs)**	**Speedup**
Brain blood vessel	Position function	5.719	100.3
	Our approach	0.057	
Pig Tongue	Position function	1.984	99.2
	Our approach	0.02	

```
proc adaptiveThreshold( uv: textureCoord,
                        textureMap:sampler,
                        C: single,
                        width:integer,
                        height:integer)
  var mean;
  var x = uv.x-1.0/width,
      y = uv.y-1.0/height;
  var col = sample(textureMap, (x,y));
  int i=0, j=0;

  for(i=-hsize;i<=hsize;i++) {
    for(j=-hsize;j<=hsize;j++) {
      var newuv = (x+(i/width), y+(j/height));
      mean += sample(textureMap, newuv);
    }
  }
  mean /= (size*size);
  return (col>(mean-C));
end proc
```

List 6.2. The shader for adaptive thresholding

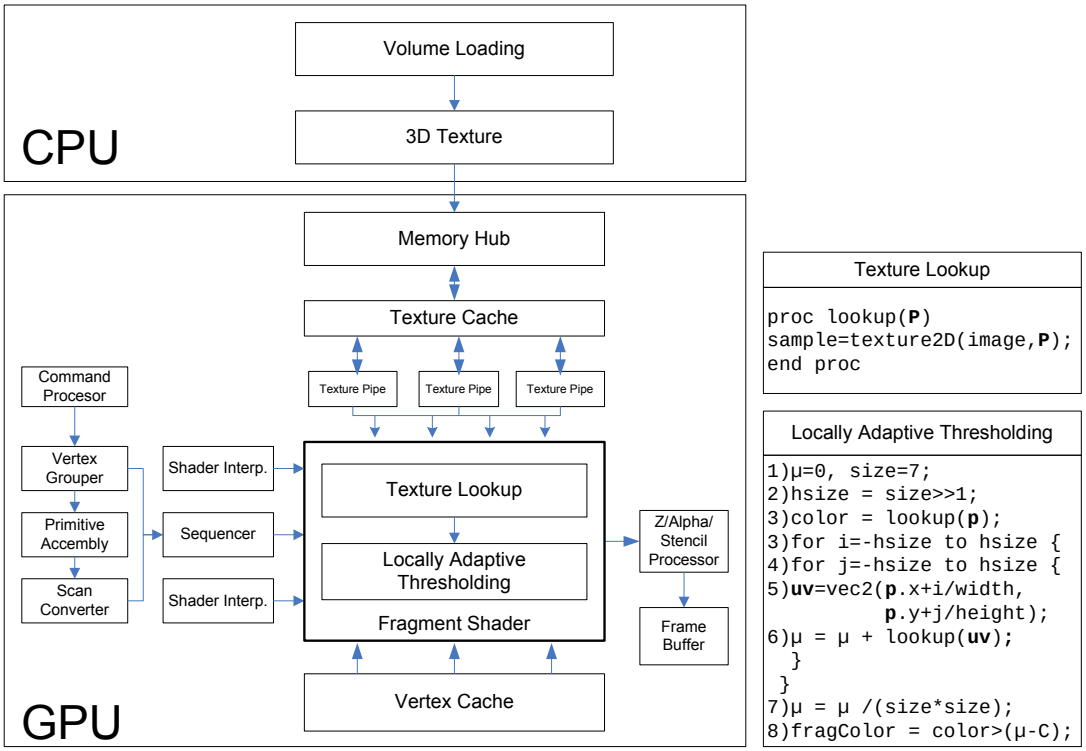


Figure 6-7. GPU mapping of locally adaptive thresholding

6.5 Summary

We have investigated into the hardware accelerated algorithms to carry out real-time feature detection and volume rendering, to match the video rate of image acquisition in endomicroscopy. We defined morphological operations coupled with local adaptive thresholding. Being implemented on the GPU, our method enjoys high precision floating point arithmetic in all stages resulting in highly accurate feature computation. The identified features are then stored for use later during the transfer function construction. This method allows visualization of features without any user intervention because features are automatically highlighted with a pseudo-color assignment. The pre-integrated volume rendering with second-order approximation provides un-surpassed quality of the rendered images.

The transfer functions (shown in Figure 6-6), are still globally assigned but spatial transfer functions [50] can be incorporated by modifying the transfer function stage to utilize the identified gradient magnitudes and VOI positions. The current method relies on the region's local minimum and maximum intensities and the resulting rendering is subjective to the segmentation result. We are considering incorporating more sophisticated segmentation methods such as the level sets into our pipeline and use them to formulate the transfer functions.

7 UBIQUITOUS RENDERING ON THE WEBGL ARCHITECTURE

7.1 Introduction

In volume rendering, the volume sampling is carried out to identify density values and their contributions are composited on screen. Up to now, sophisticated volume rendering and feature visualization algorithms have mainly been developed for stand-alone workstations. On the other hand, it becomes more and more important to enable ubiquitous data processing and visualization on mobile platforms such that the data repositories could be shared among all clients through a secure web server. An example is the mobile volume rendering of medical images from a repository of the Radiological Information System. In such a system, data will be stored on a secure server and the client will fetch and process the data on demand, anywhere and anytime.

Thanks to the wide availability of GPU support in nowadays mobile devices such as smart phones and tablets, ubiquitous computing for high-quality and real-time volume rendering is becoming possible. Therefore, we focus on designing volume rendering algorithms which can be efficiently implemented on both stand-alone and mobile platforms, for example, GPU shaders for on-the-fly blending of the classified densities with other rendering modes.

This chapter is organized into specific topics identifying the key innovations for a ubiquitous volume renderer on the WebGL platform. Section 7.2 introduces WebGL and the system architecture. Technical details on the novel ray function blending ray caster as well as fast 3D texture slicer are presented in section 7.3. In section 7.4, performances of the proposed algorithms on both mobile and stand-alone platforms are extensively evaluated. And finally, section 7.5 summarizes the work.

7.2 The WebGL Supported System Architecture

WebGL is proposed as a new standard for plugin-less graphics computing from within a web browser. It provides a cross-platform, immediate mode and royalty-free 3D graphics API. Since WebGL is based on the OpenGL ES 2.0 API which is a subset of OpenGL for embedded devices, it uses the same shader language framework as OpenGL.

Before WebGL was introduced, most of the 3D contents were available only through plugins for the web browsers which often had compatibility issues; and therefore, the programs were not a write-once-run-everywhere solution. Moreover, such plugins had to be manually installed before any 3D contents could be rendered. Supported by WebGL, applications can now have native access to the graphics hardware through the well established OpenGL ES API without any plugin [142].

From the implementation point of view, WebGL is an extension of the HTML5 canvas element. This element provides fast and high performance graphics constructs for rendering high-quality graphics in a web-browser window. For 2D graphics, this is usually accomplished by using a 2D rendering context called the CanvasRenderingContext2D. For 3D rendering, the canvas element provides a rendering context, called the WebGLRenderingContext which provides the OpenGL ES 2.0 functionalities. Both of these APIs are controlled through javascript.

Internally, the OpenGL context makes calls to the graphics hardware directly through the graphics driver (see Figure 7-1). Therefore, there have been several debates on the security issues and vulnerabilities of the WebGL API. While such a low-level access is necessary for high-performance graphics, this also allows the client program to have access to the hardware directly which could potentially be used for an attack. Nevertheless, such security issues have been addressed by the new specifications and more security features are being introduced with each new specification update.

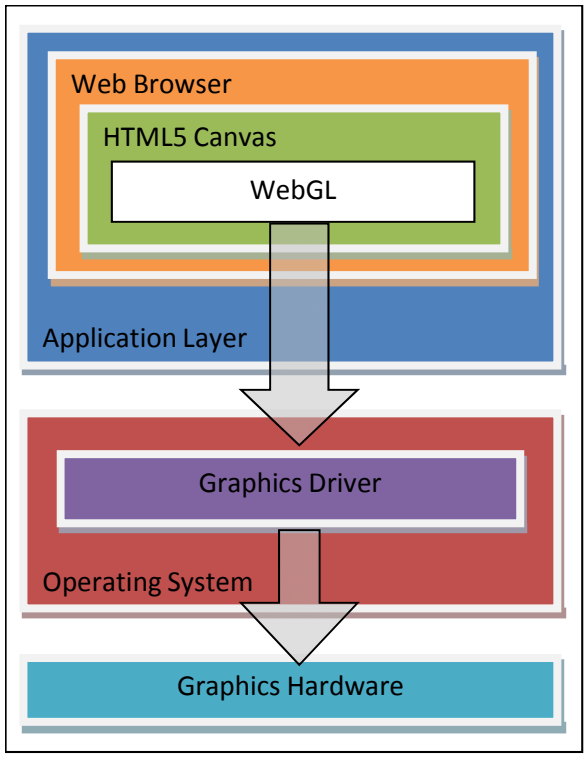


Figure 7-1. System architecture supported by WebGL

In a WebGL supported application, a rendering context is obtained from the html5 canvas element. This is usually accomplished using javascript code. In the new specifications v 1.0.1, the implementations are required to provide a pure non-experimental WebGL context. The returned gl context can then be used to call any OpenGL ES 2.0 API function.

With the type-less javascript API, the object type and their memory management is handled by WebGL. This has some performance implications, due to the strongly typed nature of OpenGL. WebGL has introduced special typed arrays that correspond to the native types used by OpenGL. This allows proper argument passing to the native OpenGL call from WebGL. With newer browser release, their javascript engine performance is improved and it is now comparable to a native call as in C/C++.

7.3 WebGL Compliant Shaders for Volume Rendering

The framework for our WebGL based single-pass volume rendering algorithm is shown in Figure 7-2. A fundamental requirement for volume rendering is the support of 3D textures. The volume dataset is first stored into the GPU texture memory. Usually, this is carried out by using 3D textures.

Since WebGL is based on the OpenGL ES 2.0 API which is a restricted subset of the desktop OpenGL, there are some functionalities missing; in particular, there is no 3D texture which is a crucial requirement for volume rendering. As shown in Figure 7-2, in our design, this limitation is circumvented by tiling each slice of the 3D texture into a 2D flat texture layout.

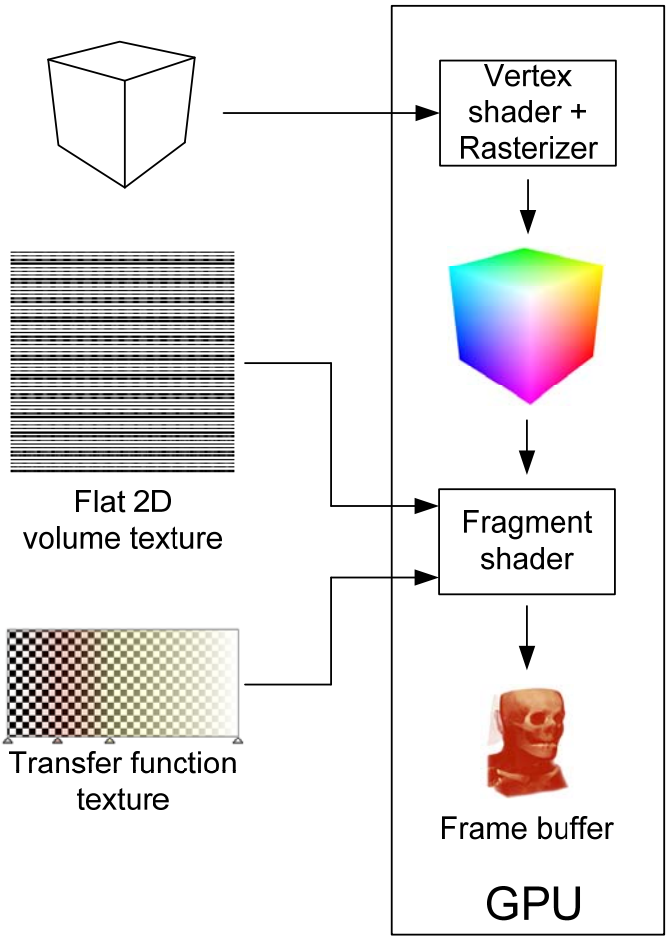


Figure 7-2. Framework of our single-pass volume rendering supported by WebGL

7.3.1 The Single-pass Shader for Ray Casting

Previous GPU-based algorithms for WebGL used the fragment shader in a multi-pass pipeline [143]. The multi-pass algorithm renders front and back faces of a unit color cube and then generates rays. It was introduced when the support of loops was limited and sampling had to be implemented by multiple passes using the front and back textures. This approach has also been applied in the previous WebGL supported volume rendering algorithms. However, our study shows that each additional texture access considerably reduces the rendering performance, especially on the mobile platforms.

Hence, catering to the GPU functionality accessed via the WebGL API, we propose a single-pass ray casting approach for the on-the-fly volume rendering requirement in real-time applications. This approach is based on an optical model to approximate the volume rendering integral [144], to model the emission, absorption and transmission properties that map the physical attributes stored in the 3D density function.

As a foundation, we try to approximate an optical model given as

$$I(D,t) = F(s_0, D, t)(I_0 + \int_{s_0}^{D} q(s)ds)$$
$$F(a,b,t) = e^{-\int_a^b \kappa(t)dt} \tag{7.1}$$

where,

t ranges from 0 at the eye position to ∞,

I_0 is the background light intensity,

F is the transparency function,

$q(s)$ is the source term (the illumination model used),

$I(D)$ is the intensity leaving the volume,

$\kappa(t)$ is the extinction coefficient, and

D is the distance of the viewer from the volume.

Typically, the volume data is rendered in a back-to-front order with the depth buffer disabled. Since alpha test is performed before depth test, some fragments are discarded; however, the depth buffer contains the depth value of the closest fragment, which can be utilized by our algorithm for estimating the object space positions. Finally, the world space coordinates are generated. Mathematically, this can be given as

$$\begin{aligned} p_w &= S(M_{MVP})^{-1} V . p_s \\ M_{MVP} &= P M_{MV} \\ p_s &= [x_s, y_s, z_s] \\ p_w &= [x_w, y_w, z_w] \end{aligned} \tag{7.2}$$

where, p_w is the world space position, p_s is the screen space position, M_{MV} is the modelview matrix, P is the projection matrix, M_{MVP} is the combined model view projection matrix, V is the viewport transformation matrix, and S is the scaling matrix. The scaling matrix (S) in Eq. (7.2) scales the volume by half the size in each dimension and translates the volume by half its length.

Once the positions are obtained, rays are cast into the volume. These rays are sampled and adjacent material values are used to reconstruct the original data, typically using trilinear interpolation. For shading calculation, the centered finite difference approximation is used.

This single-pass approach is illustrated in Figure 7-3. First, a full screen quad is rendered on screen in order to invoke the fragment shader. Then, the ray casting fragment shader is applied. Using the assigned texture coordinates, the ray directions for sampling of volume are determined. Finally, the volume is traversed front-to-back.

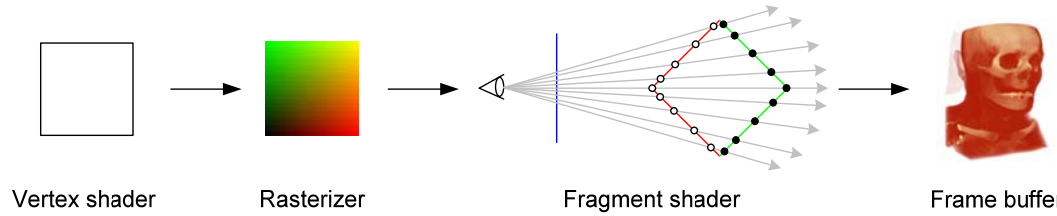


Figure 7-3. Texture coordinates assignment in the WebGL compliant single-pass ray casting shader

With such a design, the ray functions can be efficiently implemented in both front-to-back and back-to-front variants. In fact, they can be modified on-the-fly using the fragment shader. This helps to accommodate various ray functions such as maximum intensity projection (MIP), maximum intensity difference accumulation (MIDA) [145], composite accumulation, psuedo-isosurface rendering etc. for running directly on the mobile devices without reconfiguration.

The vertex shader projects the unit cube's vertices to clip space by multiplying them with the modelview matrix. It also calculates the 3D texture coordinates for sampling and stores the object space positions for shading calculation. The bulk of the work for our single-pass volume rendering takes place in the fragment shader. The current camera position (which is the eye ray's origin) is passed in as a shader uniform variable. Using this eye ray's origin and the cube's interpolated object space position, the ray directions are obtained.

Before starting the costly sampling loop, we introduce an optimization process which checks the current ray against the bounding box of the unit cube. Only if the ray intersects the bounding box, the sampling loop is initiated; otherwise, the current fragment is discarded. This gives significant performance boost especially on the mobile platforms. Once a valid ray is found, it is then cast from the interpolated object space position.

Depending on the current step size used and the total number of sampling points, a loop is initiated.

In each iteration, the sample value is obtained by a lookup from the volume dataset. To the interpolated value, the transfer function is applied and then the classified value is accumulated using the current compositing scheme. Finally, the ray is stepped forward in the current viewing direction. The whole process is repeated until the ray exits the whole volume dataset. After the loop, the final composited color is returned as the current fragment color. List 7.1 presents the fragment shader for the single-pass ray casting.

```
for each fragment
   var ray_start:vec3;
   var   ray_dir:vec3;
   var accum_col:vec4;

   ray_start = IN.uvw;
   ray_step  = normalze(IN.pos - eye_pos) * step_size;
   accum_col = background_color;

   for all samples along the ray
      if ray_start within the dataset bounds
         sample = texture(volume, ray_start);
         color  = texture(tf_sampler, sample);

         prev_alpha     = color.a – (color.a*accum_col.a);
         accum_col.rgb += prev_alpha * color.rgb;
         accum_col.a   += prev_alpha;
      else
         break;
      end if
      ray_start += ray_step;
   end for
   return accum_col;
end for
```

List 7.1. Fragment shader for the single-pass ray casting

7.3.2 Ray Function Blending

Based on the single-pass ray casting pipeline, we further developed a new hybrid rendering mode. This mode allows us to combine the result of multiple ray functions. For instance, we can combine the iso-surface ray function to the composite ray function. The ray function blended ray casting shader is given in List 7.2.

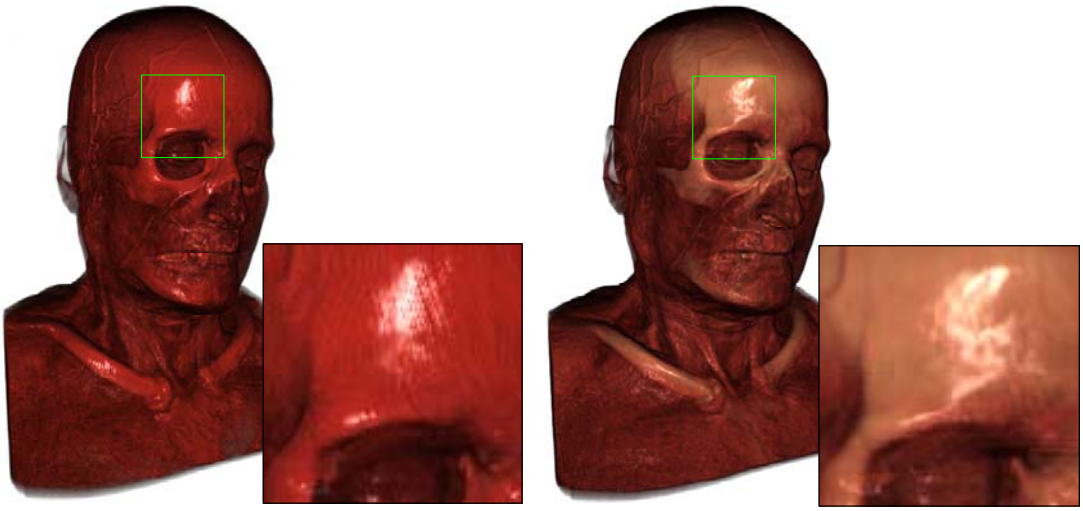
```
for each fragment
   var ray_start:vec3;
   var   ray_dir:vec3;
   var accum_col:vec4;

   ray_start = IN.uvw;
    ray_step = normalze(IN.pos - eye_pos) * step_size;
   accum_col = background_color;

   for all samples along the ray
      increment ray position using ray direction and step amount
      if ray within the dataset bounds
         sample = texture(volume, ray_start);
         color1 = texture(tf_sampler, sample);

         prev_alpha  = color1.a – (color1.a*accum_col.a);
         accum_col.rgb += prev_alpha * color1.rgb;
         accum_col.a += prev_alpha;

         sample2 = texture(tf_iso_sampler, sample);
         color2 = isosurface_shade(sample2);

         prev_alpha  = color2.a – (color2.a*accum_col.a);
         accum_col.rgb += prev_alpha * color2.rgb;
         accum_col.a += prev_alpha;
      else
         break;
      end if
   end for
   return accum_col;
end for
```

List 7.2. Fragment shader for the function blended single-pass ray casting

The conventional iso-surfaces are evaluated using a condition to test whether the current sample value is closer to the required iso-value. This results in an image containing a lot of sampling artifacts especially in regions where the iso-surface blends with the background color, as shown by a rendering experiment in Figure 7-4 (a). Moreover, since the iso-surface is combined with the background color, the colors bleed on the iso-surface, making its identification difficult.

In our solution, we eliminate the condition test. Instead, we use blended iso-surfaces which are generated using an alpha transfer function. This allows the iso-surfaces to be blended naturally with the existing ray function. This avoids flickering artifacts in iso-surface evaluation. The flickering artifacts would otherwise be generated when the next ray step composites the current sample value, and the iso-surface loses its color. In

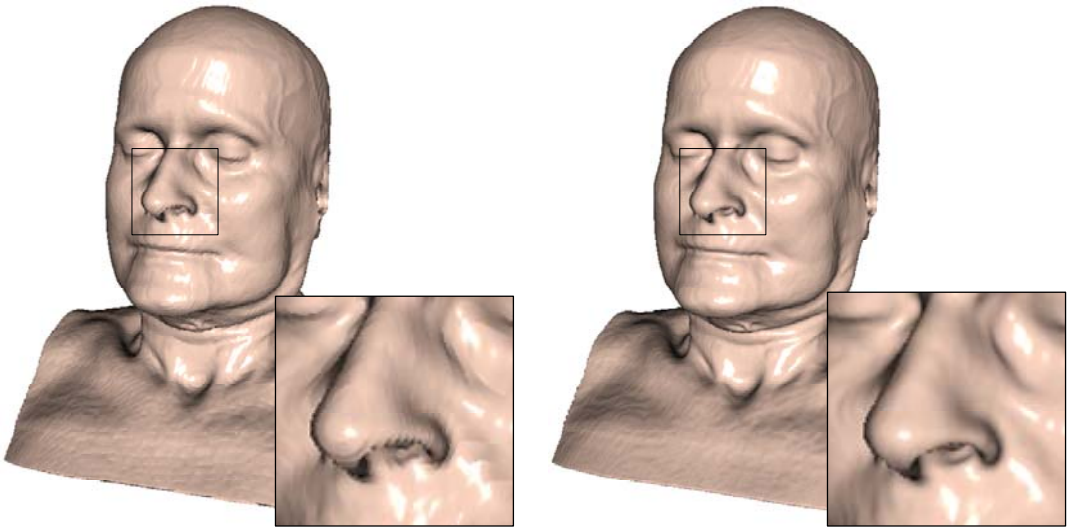
addition, the resulting iso-surfaces will not suffer from color bleeding artifacts, and therefore the iso-surface color is sustained, as can been verified by the rendering experiment in Figure 7-4 (b).



(a) (b)

Figure 7-4. Ray casting for the Manix dataset: (a) without ray function blending, and (b) with ray function blending. (Note that both renderings used the same transfer function, but the color bleeding and sampling artifacts are removed in our ray function blended single-pass ray casting.)

Aliasing artifacts also appear in the conventional iso-surface ray function due to fixed sampling step size, as shown by a rendering experiment in Figure 7-5 (a). To solve this problem, in our single-pass shader, the ray step size is adapted on the underlying surface function using the adaptive refinement technique: Our refinement technique works by first finding two candidate voxels, called left and right samples, across which the iso-surface exists. Then, a loop is initiated. In each iteration, the midpoint between the two samples is obtained. The voxel value is checked for the iso-surface. If the sample value is less than the given iso-value, the iteration continues in the right half which redefines the bisection limits; otherwise, the left half is used. This iterative process yields the correct iso-surfaces without rendering artifacts, as can be verified by the rendering experiment in Figure 7-5 (b).



(a) (b)

Figure 7-5. Ray casting with the iso-surface ray function: (a) with fix ray step size, and (b) with adaptive ray step size

The method adapts the ray step size iteratively such that the correct sample is used for iso-surface rendering. This results in a smooth and continuous iso-surface without rendering artifacts. As can be seen in Figure 7-5, even for a much lower sampling rate, the iso-surface rendering quality is much improved. List 7.3 gives the shader for the refined single-pass ray casting with adaptive iso-surface refinement.

7.3.3 3D Texture Slicing

In the current mobile devices, the dynamic loops in a fragment shader are capped at a maximum number of iterations. Thus, in the case of very large volumetric datasets, rendering quality using the single-pass ray casting shaders on these platforms is still not satisfactory, although the executive performance is significantly improved. Moreover, the variable loop size in a fragment shader requires a compile time constant. Therefore, the sampling rate cannot be changed without recompilation on the mobile platforms.

```
proc bisection(left, right, iso)
   for i=0 to 5
      mid = (left + right)/2.0;
      val = texture(volume, mid);
      if(val < iso)   left  = mid;
      else            right = mid;
      end if
   end for
   return (left+right)/2.0;
end proc

for each fragment
   get ray start position using the input texture coordinates
   determine ray direction and step amount
   set fragment_color to background color
   for all samples along the ray
      increment ray position using ray direction and step amount
      if ray within the dataset bounds
         sample1 = texture(volume, ray_position)
         sample2 = texture(volume, ray_position+ray_step)
         if (sample1-iso_value)<0 && (sample2-iso_value)>=0
            left  = ray_position;
            right = ray_position + ray_step;
            tc    = bisection(left,right, iso_value);
            fragment_color = Phong_Shading(L,N,V, shininess,
                                           diffuse_color);
            break
         end if
      end if
   end for
   return fragment_color
end for
```

List 7.3. Shader for adaptive iso-surface refinement in single-pass ray casting

As an alternative solution for WebGL platform, we propose a rasterization based volume rendering algorithm, 3D texture slicing, making use of the hardware support for rasterization on the mobile devices. Since texture slicing is only limited by the mobile device's memory, we are able to implement advanced volumetric shading using the dynamic lookups to extract the gradient at the current sampling point.

Technically, texture slicing also approximates the volume rendering integral given in Eq. (7.1), but it takes a different approach by applying texture maps to the proxy geometry. Typically, rectangular [63, 64] or spherical slices [9] are used. In the case of rectangular slicing, these slices can be oriented either parallel to one of the principle axis

(object aligned slicing [63]) or perpendicular to the viewing direction (view-aligned slicing [64]).

The view-aligned slicing works by estimating the intersection of a unit cube with the plane perpendicular to the viewing ray. The viewing ray becomes the plane normal. There are six possible cases for intersection between the plane and the unit cube. These are obtained by looping through all three edges of the cube's vertex and finding the ray intersection point.

Considering two points, P_0 and P_1, any point on the edge (P_1-P_0) may be given using the following parametric equation

$$P(t) = P_0 + t*(P_1 - P_0) \tag{7.3}$$

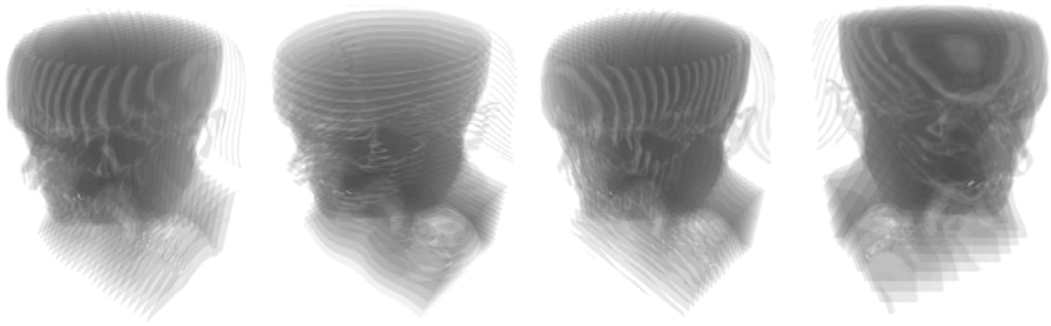
where, t=[0,1]. For a plane with a normal $\boldsymbol{N}$, we can define any point on it using the following equation

$$N \bullet P + d = 0 \tag{7.4}$$

Combining Eq. (7.4) and Eq. (7.3), and solving for t gives

$$t = \frac{-d - N \bullet P_0}{(P_1 - P_0)}$$

Defining the parameter t value in Eq. (7.3) gives us the intersection point. We iterate over all edges from the current vertex and obtain all intersection points. These points are then used to generate the proxy polygon for sampling of volume. This process is repeated for all eight cube vertices going front-to-back in the viewing direction changing the plane distance in each step, until the whole volume is traversed. We implemented both 2D and the view-aligned 3D texture slicing on the WebGL platform, as illustrated by the intermediate results in our rendering experiments in Figure 7-6.



(a) (b) (c) (d)

Figure 7-6. 2D texture slicing showing axis based proxy geometry selected based on the principle viewing direction (a) X-Axis, (b) Y-Axis, and (c) Z-Axis; 3D texture slicing showing (d) view-aligned proxy geometry. (Note that the polygons always remain perpendicular to the viewing direction.)

In OpenGL 3.3 and above, the buffer objects are used to maintain the proxy geometry. For comparison, we first implemented a CPU program as a texture slicer (the pseudo code is given in List 7.4). We determine the intersections of the viewer with the volume and then use this information to determine the intersection case. This is then used to push the relevant proxy geometry onto the GPU. For the WebGL compliant implementation, the shaders for texture slicing on GPU include both the vertex shader and the fragment shader. The volume is sliced on the vertex shader while the shading is handled by the fragment shader, as detailed in List 7.5.

```
proc Texture_Slicer_CPU()
   initialize geometry list;
   loop through all cube vertices and find the max and min distances;
   find the abs maximum of the view direction;
   determine the slicing plane and the slice plane increments;
   loop over all cube edges to find the intersections and increments;

   for i=0 to TOTAL_SLICES
      determine the current intersection point;
      determine the intersection case;
      for all intersections found
         add the intersected vertices to the geometry list;
      end for
   end for
end proc
```

List 7.4. Pseudo code for view-aligned 3D texture slicing on CPU

```
proc Texture_Slicer_GPU()
   determine the slicing increment as plane_increment;
   determine the nearest intersection from the view point to volume;
   set the nearest intersection to the volume as plane_start;

   setup slicing shaders;
   pass vertex/edge lists, plane start/increment to shader;
   set other shader uniforms like MVP matrices;
   for i=0 to TOTAL_SLICES
      draw a six sided polygon at plane_start position;
      increment start position by plane_increment;
   end for
   remove slicing shaders;
end proc

proc Texture_Slicer_Vertex_Shader()
   plane_dist = plane_start + slice_index*plane_increment;
   for all edges e
      v1 = vertices[e.index0];
      v2 = vertices[e.index1];
      vec_start = v1 + vec_translate;
      vec_dir = v2 – v1;

      denom = dot(vec_dir, view_dir);
      t = (denom!=0)?(plane_dist-dot(vec_start,view_dir))/denom:-1;
      if (t>=0 && t<=1)
         position = vec_start + t*vec_dir;
         break;
      end if
   end for
   OUT.vertex = MVP*position;
   OUT.uvw = 0.5*position+0.5;
end proc

proc Texture_Slicer_Fragment_Shader()
   sample = texture(volume, IN.uvw);
   color  = texture(tf_sampler, sample);
   return color;
end proc
```

List 7.5. Shaders for view-aligned 3D texture slicing on GPU

Since each vertex passes its index to the vertex shader when the geometry is pushed onto GPU, the case is determined in the vertex shader using that index. Using the plane increment and plane intersection to the volume, the intersected vertex is returned. The shading is then handled by the fragment shader. The process is repeated for all proxy slices and finally, the slices are blended back-to-front as shown in List 7.5.

7.4 Experiments and Performance Assessment

The WebGL compliant single-pass volume rendering algorithms have been developed in this project. Extensive experiments were carried out on both mobile and desktop platforms, as shown in Figure 7-7. The performances of the single-pass ray caster and 3D texture slicer were evaluated. The findings from these experiments are detailed in this section.

Figure 7-7. WebGL compliant volume rendering of 3D medical dataset by our single-pass GPU ray caster implemented on the stand-alone desktop (left) and two mobile platforms: ACER Iconia A500 tablet (middle) and the Samsung Galaxy SII GT-I9100 mobile phone (right)

For effective visualization, we provide a transfer function widget that can modify the current transfer function as required in realtime (as shown in Figure 7-8). The user can adjust the transfer function color and the current sample's alpha value by assigning color and alpha values to specific keys. The widget is implemented in javascript for a web browser. In Figure 7-8, the X-axis corresponds to the scalar value. In our case, it varies from 0 to 255 as we use 8-bit image datasets. The Y-axis corresponds to the transparency value which varies in the range of [0,1].

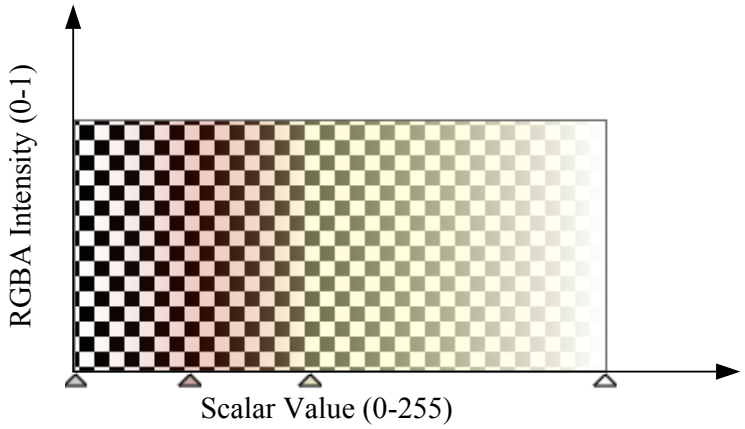


Figure 7-8. The transfer function widget for a web browser

7.4.1 WebGL Compliant Ray Casting

We first evaluated the major WebGL compliant web browsers. We looked at Opera Next v12.00 Alpha, Google Chrome v 17.0.963.56m and Mozilla Firefox v 11.00 beta. Based on our extensive experiments, we conclude that the WebGL implementation on Google Chrome is the most stable and scalable compared with the other WebGL compliant web browsers. Hence, we use Google Chrome with the Angle backend disabled.

We used four datasets in these experiments, namely Aorta, Skull, CTHead and Visible Male Head datasets all of which were stored into a 2D flat texture layout. The Aorta, CTHead and Skull datasets are with an image resolution of 4096×4096; whereas, the Visible Male Head dataset is with a resolution of 1920×1920. The Aorta dataset contains 96 slices; the CTHead and Skull datasets contains 256 slices; and the Visible Male Head dataset contains 128 slices. With the image resolution of 4096×4096, we were able to safely accommodate a 256^3 dataset in a 16×16 flat texture layout.

Two mobile platforms were used: One is ACER Iconia A500 Tablet (referred as MOBILE1), 1GHz dual-core Cortex A9 processor with an NVIDIA Tegra 2 GPU; and the other is Samsung Galaxy SII GT-I9100 (referred as MOBILE2), dual-core 1.2 GHz Cortex-A9 mobile phone with Mali-400MP GPU.

Both of these mobile platforms run the Google Android operating system. Using pseudo-color transfer function, the WebGL rendering results from the proposed single-pass ray caster are given in Figure 7-9. The first set of experiments was carried out on the mobile platforms, for estimating the performance of the proposed single-pass ray casting algorithm with the previous multi-pass algorithm on the WebGL platform. In this experiment, composite function without transfer function is used.

(a) (b) (c)

Figure 7-9. Rendering results from the proposed GPU-based single-pass ray casting on WebGL showing (a) the Aorta dataset, (b) the CTHead dataset, and (c) the Skull dataset. We used 256 sampling steps and a pseudo-color transfer function by using a custom transfer function widget shown in Figure 7-8

In the initial tests, 50 sampling steps were used (Table 7.1). Later, the sampling steps were increased to 100 (Table 7.2) because we found out that the maximum steps for a variable loop supported in the Tablet were 100. Any loop size larger than this value timed out the shader compiler and the tablet could not run the shader. The experimental results on the mobile platforms are given in Table 7.1 and Table 7.2. As can be seen, our single-pass ray caster out performs the multi-pass ray caster by approximately 2x. We get much better performance with the same rendering settings. We expect the upcoming hardware to relax the variable length loop limit further which would improve the performance of this algorithm considerably.

Table 7.1. Comparison of performance of single-pass WebGL ray caster and that of multi-pass ray caster for a sampling rate of 0.02 (50 sample points) on the mobile platforms

Dataset	Hardware	Frame rate (in frames per second)			
		Multi-pass ray caster [141]		Our proposed single-pass ray caster	
		512×512	1024×1024	512×512	1024×1024
Aorta	MOBILE1	0.9-1.0	0.2-0.3	1.8-1.9	0.5-0.6
	MOBILE2	2.7-3.3	0.9-1.0	8.4-9.0	1.9-2.1
CTHead	MOBILE1	0.9-1.0	0.2	1.8-1.9	0.4-0.5
	MOBILE2	2.4-3.2	0.8-0.9	5.7-7.0	1.6-1.8
Skull	MOBILE1	0.9-1.0	0.2-0.3	1.8-1.9	0.4-0.5
	MOBILE2	2.3-2.5	0.8	5.9-6.4	1.6-1.8

Table 7.2. Comparison of performance of single-pass WebGL ray caster and that of multi-pass ray caster for a sampling rate of 0.01 (100 sample points) on the mobile platforms

Dataset	Hardware	Frame rate (in frames per second)			
		Multi-pass ray caster [141]		Our proposed single-pass ray caster	
		512×512	1024×1024	512×512	1024×1024
Aorta	MOBILE1	0.5	0.1	0.9-1.0	0.2-0.3
	MOBILE2	1.5-1.8	0.5	4.4-4.6	1.3-1.6
CTHead	MOBILE1	0.5	0.1-0.2	1.0	0.2-0.3
	MOBILE2	1.8-1.9	0.5	3.3-3.6	1.0-1.1
Skull	MOBILE1	0.5	0.1-0.2	1.0	0.2-0.3
	MOBILE2	1.4-1.5	0.4-0.5	3.3-3.5	0.9-1.0

In order to remove any biases in a specific WebGL implementation, it is necessary to estimate the performance of the WebGL based algorithm on not only mobile platforms but also stand-alone desktop platforms. Therefore, we also evaluated the performance of the proposed algorithm on two desktop platforms. The first platform is the Dell Precision T7500 workstation (referred as SYSTEM1) with a 2.27 GHz Intel Xeon CPU with 4 GB of RAM. This machine is equipped with an NVIDIA Quadro FX 5800 GPU with 4096 MB of dedicated video memory. The second platform is the Dell Alienware M15x laptop (referred as SYSTEM2) with an Intel Core i7 CPU and an NVIDIA GeForce 260M GPU. The experiment was conducted on both SYSTEM1 and SYSTEM2 to compare the performance of our single-pass ray caster against the multi-pass ray caster [143].

In this experiment, we use the composite rendering mode with transfer function. All of the rendering and startup settings (for example, the distance of the volume from the camera) are the same for both of the WebGL compliant ray casters. The tests were carried out on the canvas resolutions of 1024×1024 pixels. The ray sampling steps for this experiment are 100. These results are presented in Table 7.3. The performance trend we observed in Table 7.1 and 7.2 is followed in this case also. Our single-pass ray caster outperforms the multi-pass ray caster by almost 2x.

The primary reason for this speedup in our proposed algorithm is the significantly less number of texture fetches and more efficient ray traversal. Since our single-pass algorithm does not require any additional lookup whereas the multi-pass approach requires the front and back textures for extracting the ray direction, our algorithm's performance remains consistent even as the output size increases, that is, it exhibits an excellent scalability.

Table 7.3. Comparison of performance of our proposed single-pass ray caster and that of multi-pass ray caster on the desktop platforms

Dataset	**Hardware**	**Frame rate (in frames per second)**	
		Multi-pass ray caster [143]	**Our proposed single-pass ray caster**
Aorta	SYSTEM1	72.3-92.7	90.8-90.9
	SYSTEM2	32.0-46.4	77.8-80.8
CTHead	SYSTEM1	52.3-80.7	89.4-90.9
	SYSTEM2	31.6-45.0	73.1-77.5
Skull	SYSTEM1	54.5-81.2	90.7-90.8
	SYSTEM2	31.0-44.7	75.6-77.8

7.4.2 Adaptation of Ray Function at Runtime

The versatility of the single-pass ray casting algorithm makes it easy to change the ray function at runtime. A set of experiments was conducted to examine the rendering results with various ray functions on CTHead dataset, as shown in Figure 7-10. The average ray function uses the mean of all the samples along the ray, resulting in an X-ray like scan (see Figure 7-10 (a)).

The maximum ray function generates the maximum intensity projection (MIP) (see Figure 7-10 (b)), which helps in angiography and bone studies; however, the resulting image loses its spatial locality. If the difference between the current sample from the maximum value encountered so far is taken [145], the spatial locality is restored (see Figure 7-10 (c)).

Composite ray function allows us to use a transfer function to assign color and opacity values to individual voxels so that these can be classified. The composite with iso-surface blending is shown in Figure 7-10 (d, e). In Figure 7-10 (f), we use a pseudo-color transfer function to highlight the various density values in the CT data.

As can be seen, the different densities are much more clearly identifiable in the pseudo-color rendering mode. It is often required to generate iso-surface rendering from the volumetric dataset. This mode can be added easily in the single-pass ray caster thanks to the versatility of the proposed algorithm.

In addition, since the iso-surfaces are generated dynamically, they can be adapted in our shader; that is, the iso-values can be modified at runtime. This allows the user to identify the relevant densities in the dataset easily by dynamically adjusting the iso-value as required. Moreover, since the iso-surface is determined from evaluation of a density function, they have infinite resolution. This ensures high rendering quality for the extracted iso-surface (see Figure 7-10 (g) and (h)).

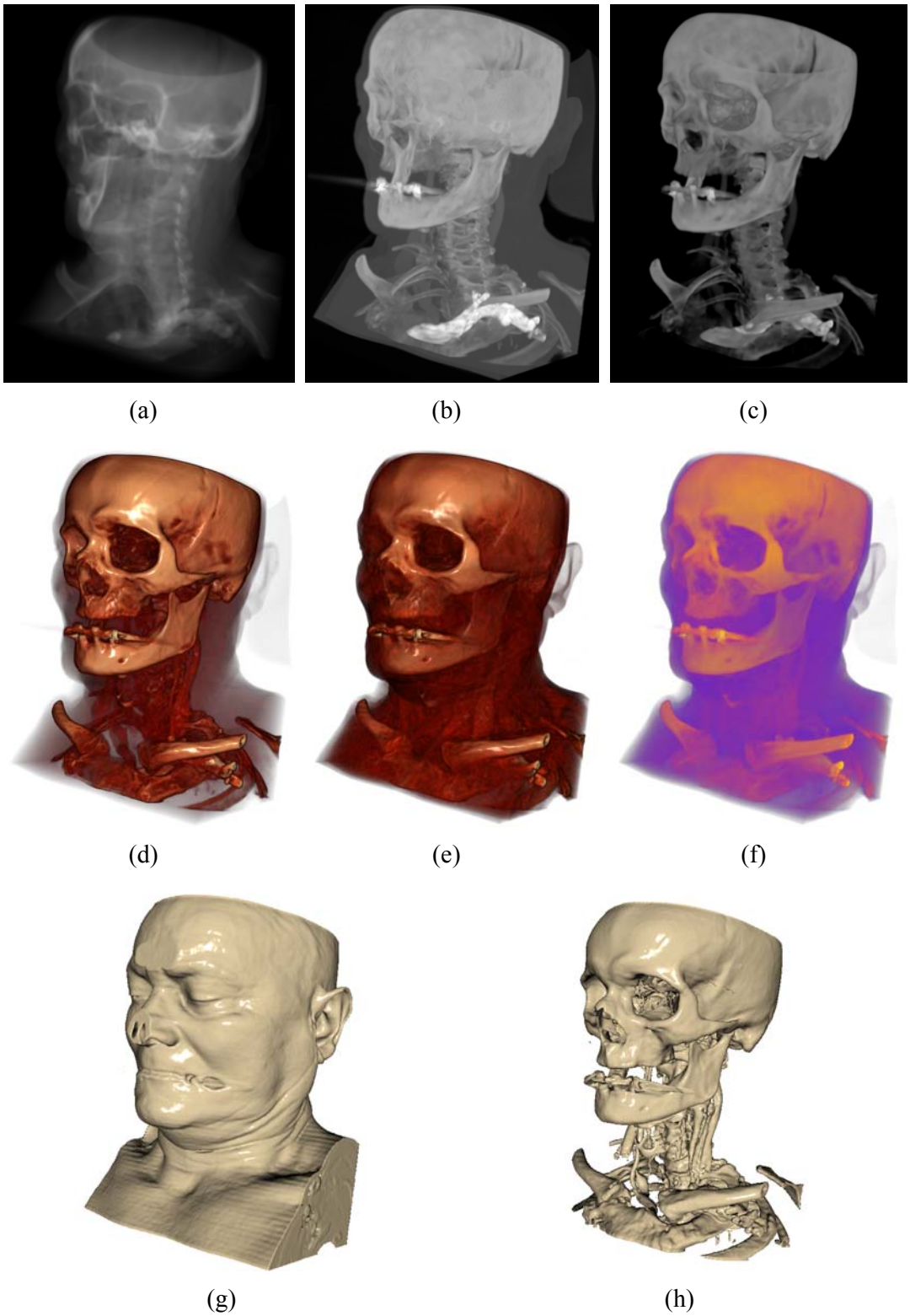
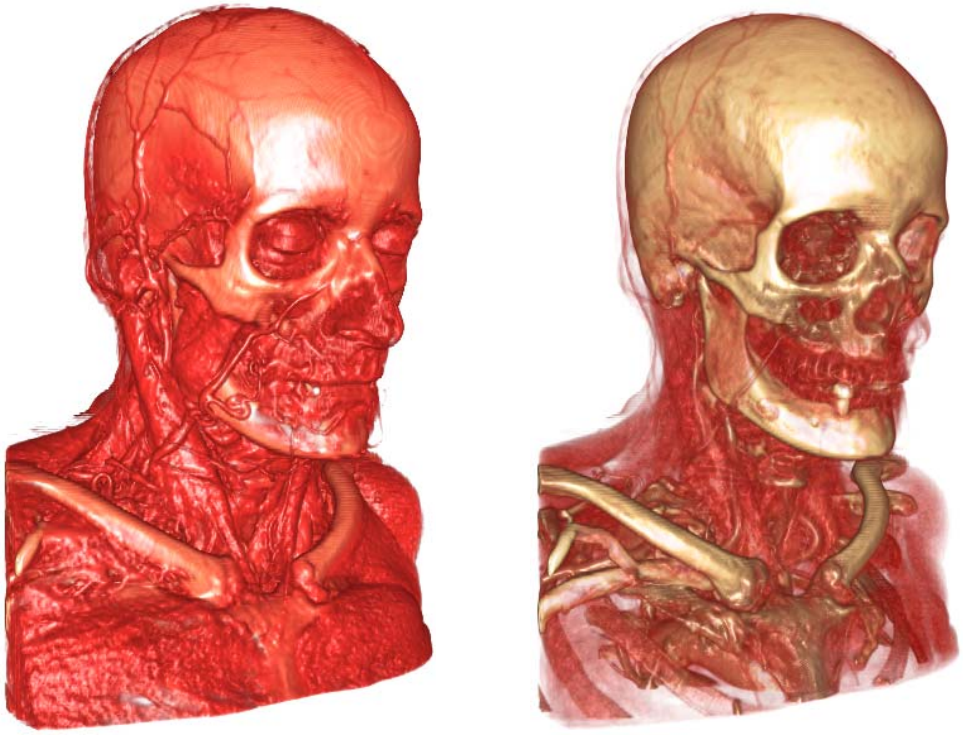


Figure 7-10. Rendering results from the single-pass GPU ray caster using different ray functions showing (a) average, (b) MIP, (c) MIDA, (d,e) composite with shading, (f) composite with pseudo-color assignment, (g) iso-surface with (iso-value=80) and (h) iso-surface with (iso-value=56)

7.4.3 WebGL Compliant 3D Texture Slicing

The current WebGL implementations have a limit on the maximum number of steps a variable loop can take in a fragment shader. Moreover, since the variable loop size in a fragment shader is a compile time constant, we cannot change the sampling rate at runtime especially on the mobile platforms. On the contrary, 3D texture slicing does not suffer from these restrictions because it relies only on the rasterization hardware.

Therefore, a set of experiments was conducted to evaluate the performance of 3D texture slicing with composite ray function and shading. Since the support of the rasterizer hardware is ubiquitous on both desktop and mobile platforms, we can provide a pervasive volume renderer that is guaranteed to provide high performance volume rendering on all platforms. We rendered the Manix dataset using gradient based 3D texture slicing, as shown in Figure 7-11.



(a) (b)

Figure 7-11. The shaded 3D texture slicing of the Manix dataset showing (a) the muscle and soft tissue and (b) bone and vasculature

There are different WebGL implementations in different browsers. To show that our proposed algorithm gives high performance on all platforms, we carried out a rigorous evaluation of 3D texture slicing on all supported WebGL platforms. The results for a sampling rate of 0.0039 (256 sample points) are given in Table 7.4 whereas, the result with half of this sampling rate that is 0.00195 (512 sample points) are given in Table 7.5.

Two rendering modes were used in this experiment, the composite mode and the composite with shading mode which calculated the normal for shading by on demand gradient estimation using centered finite difference. We can see that there are differences in the performances on different browsers. In our experiments, Google Chrome and Firefox performed the best for all of the experiments. Nevertheless, our proposed 3D texture slicer performs consistently well (with the frame rates required for real-time interaction) on all WebGL supported web browsers.

Table 7.4. The performance results for WebGL compliant 3D texture slicing for a sampling rate of 0.0039 (256 sample points) on SYSTEM1

Dataset	**Web browser**	**Frame rate (in frames per second)**	
		Composite	**Composite with Shading**
Aorta	Opera	80.1-100.0	66.9-74.6
	Chrome GL	96.3-97.5	60.6-70.9
	Chrome DX	95.8-97.7	60.8-70.9
	Firefox GL	54.8-59.8	38.3-43.3
	Firefox DX	94.9-100.0	65.2-75.2
CTHead	Opera	74.0-76.1	24.9-26.8
	Chrome GL	68.7-71.1	26.2-30.7
	Chrome DX	68.9-71.4	26.7-30.1
	Firefox GL	41.7-43.5	21.6-24.0
	Firefox DX	76.9-82.3	28.5-31.2
Skull	Opera	73.5-80.1	24.2-26.9
	Chrome GL	71.6-74.4	26.7-31.4
	Chrome DX	70.9-74.8	26.9-31.7
	Firefox GL	32.4-43.4	21.1-24.8
	Firefox DX	80.8-85.5	28.3-33.1
Visible Male Head	Opera	97.0-100.0	80.6-89.0
	Chrome GL	96.6-97.7	96.9-97.6
	Chrome DX	95.6-97.7	95.8-97.6
	Firefox GL	63.3-65.7	52.9-57.3
	Firefox DX	97.0-100.1	96.9-99.0

Table 7.5. The performance results for WebGL compliant 3D texture slicing for a sampling rate of 0.00195 (512 sample points) on SYSTEM1

Dataset	Web browser	Frame rate (in frames per second)	
		Composite	Composite with Shading
Aorta	Opera	78.6-84.7	34.7-42.2
	Chrome GL	61.4-72.2	30.7-39.2
	Chrome DX	64.5-72.2	31.4-41.6
	Firefox GL	37.6-41.1	23.4-30.0
	Firefox DX	67.4-75.5	31.8-42.1
CTHead	Opera	38.1-41.8	11.8-13.3
	Chrome GL	35.9-39.2	13.7-15.2
	Chrome DX	38.2-44.1	13.6-16.7
	Firefox GL	27.7-28.9	11.9-13.2
	Firefox DX	38.3-44.3	14.2-17.9
Skull	Opera	38.3-41.7	11.8-13.3
	Chrome GL	38.6-40.2	13.6-15.2
	Chrome DX	40.9-46.1	14.6-16.1
	Firefox GL	27.6-29.6	12.0-14.5
	Firefox DX	40.2-46.0	14.2-17.2
Visible Male Head	Opera	79.6-97.2	64.2-69.2
	Chrome GL	90.8-93.0	57.2-62.1
	Chrome DX	96.1-97.7	58.5-62.3
	Firefox GL	45.9-50.1	36.3-40.3
	Firefox DX	95.8-98.0	59.0-67.1

7.5 Summary

We have presented a first-of-the-kind single-pass GPU ray caster for WebGL. In contrast to the previous GPU volume rendering on WebGL, our implementation is able to handle dynamic transfer functions on mobile platforms. We also proposed an optimization to cast rays only in the volume dataset and discard all rays that do not intersect the volume's bounding box. A quantitative evaluation revealed that the proposed algorithm outperforms the existing WebGL ray caster by up to 2x on both stand-alone and mobile platforms. Such high performance is helpful especially on the mobile platforms where the texture accesses are too costly. With the new mobile devices, we expect more support in the upcoming WebGL implementations for more advanced shaders.

Currently, due to the limitation in the loop iteration for dynamic loops on the mobile platforms, it is not possible to implement advanced shaders. Moreover, since the variable

loop size in a fragment shader is a compile time constant, we cannot change the sampling rate at runtime. Therefore, to have high performance volume rendering especially on the mobile platform, we devised a 3D texture slicer for WebGL. This allows us to render volumetric images with higher sampling rates on the mobile platforms. The experimental results have shown that, performance-wise, 3D texture slicer is with better frame rates. However, it suffers from depth fighting artifacts. The single-pass ray caster yields high quality renderings without such depth fighting artifacts. While both of them are suitable for WebGL implementation, the choice is subject to whether performance or quality is in priority.

Although, the limited support for variable loops in the current WebGL implementations on the mobile platforms prevents us from introducing more optimization in the single-pass ray caster, we expect that the upcoming hardware will relax such restrictions. This will enable the proposed single-pass ray caster to perform better on the upcoming mobile platforms.

In conclusion, thanks to the wide availability of the WebGL architecture, we have successfully developed a ubiquitous volume renderer for visualization of the datasets directly on the mobile platforms. With new and improved hardware features in the upcoming mobile devices, we expect these advanced features to be exposed through WebGL. This would allow a richer interactive visualization experience. Moreover, with new and improved mobile GPUs in the coming generations, we expect our algorithms to perform even better on these newer devices. WebGL is indeed a promising platform for high-quality mobile applications.

8 AN INTEGRATED SYSTEM FOR *IN VIVO* CELLULAR VISUALIZATION

8.1 Introduction

Parts of this PhD study are related to our joint project with National Cancer Centre Singapore on "Creating an *In Vivo* Navigational Cellular Fluorescence Imaging System with Dynamically Optimized Endomicroscopy" in which *in vivo* cellular imaging system acquires a volumetric dataset of mucosal tissues of live animal models and human volunteers.

The background problem is that lesions of the oral cavity, larynx and esophagus have been diagnosed mainly using white light endoscopy followed by histopathological examination of biopsy samples. The disadvantages of this imaging modality include: (i) early oral lesions are often flat, making it difficult to distinguish between benign and malignant lesions under white light illumination; (ii) histopathology is a time-consuming and subjective procedure, and there is a risk of complications to patients; and (iii) multiple biopsies are frequently required to determine the margins of oral lesions.

Therefore, minimally invasive imaging techniques have been explored to provide accurate diagnosis of oral lesions in the clinic. One such technique is *in vivo* fluorescence confocal laser endomicroscopy (CLE), with the use of suitable fluorescent dyes. Possible fluorescent dyes include hypericin, a photosensitizer extracted from the plant that is commonly known as St John's wort, and the fluorescent pre-cursor, 5-aminolevulinic acid (5-ALA), that metabolizes into the fluorescent compound protoporphyrin IX (PpIX). Hypericin and 5-ALA are more selectively taken up by abnormal cells, and may thus enable fluorescence diagnostic imaging with higher specificity. In this way, CLE enables *in vivo* and *in situ* imaging of tissue and cellular structures at microscopic resolution of

about 1 μm laterally, and can also perform optical or virtual biopsies for deeper subsurface layers.

While a cross-sectional image of the lesion obtained by the CLE system provides partial information of the microstructures, a concurrent 3D reconstruction of the acquired dataset would be much more helpful in extracting meaningful inferences. The locality of different structures and their depth relations can be made easy to analyze and comprehend. Unfortunately, currently available CLE systems lack the function of real-time 3D reconstruction, which motivates us to develop a novel solution to meet the needs in the diagnosis and prognosis of oral mucosal lesions.

We would like to integrate our hardware accelerated computing technologies in the previous chapters to build an *in vivo* imaging and visualization system. A pilot study using animal models and a clinical study using human volunteers has been carried out by our team. Technical challenge for concurrent 3D visualization of the acquired stack of confocal images is the real-time image reconstruction. We propose to exploit the graphics hardware to speed up these algorithms.

The rest of this chapter is organized as follows. In section 8.2, we introduce the clinic protocol including the animal model, the human model and the diagnostic standards used in our study. The fluorescence confocal scanning system and the prototype visualization system are also described. Experimental results for on-the-fly volumetric data interpolation, as well as feature enhancement of both the animal models and the human volunteers are presented in section 8.3. And finally, section 8.4 summarizes and concludes this chapter.

8.2 Clinical Protocol and System Building

8.2.1 Fluorescent Dyes

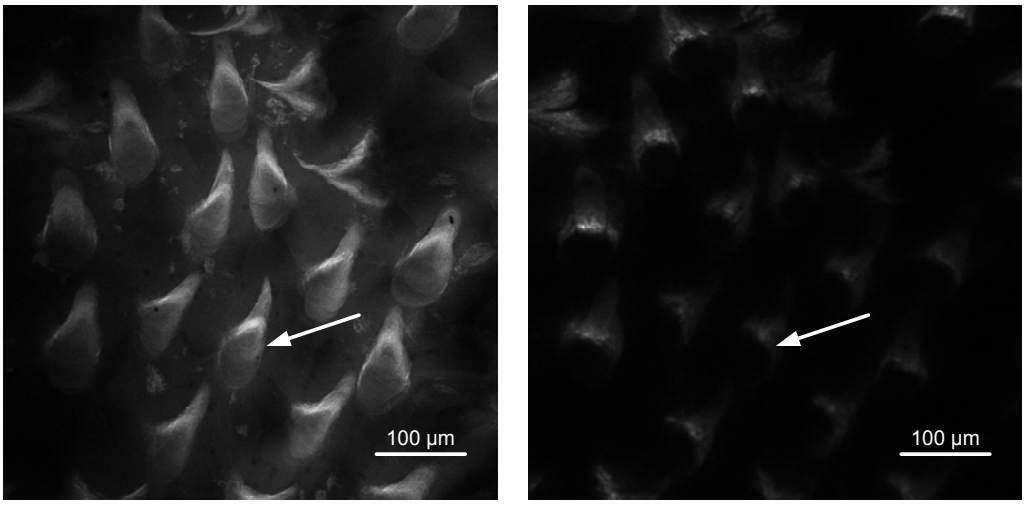
Specific contrast agents are required for fluorescence imaging using CLE. The laser scanning source used in this study is the Optiscan Five 1 confocal scanner with a 488 nm laser source. With the use of suitable fluorescent dyes, confocal fluorescence imaging can be carried out.

In our implementation, a prototype confocal endomicroscope with a rigid handheld probe is used with 5-ALA, fluorescein sodium and hypericin as contrast agents. Hypericin is used in mouse models while 5-ALA and fluorescein sodium is used in rat models. Fluorescence images of the normal rat tongue are compared to those from carcinogen-induced models of oral squamous cell carcinoma (SCC). Fluorescein sodium (Novartis Pharma AG, Switzerland) is freshly prepared as a 1% solution while hypericin (Molecular Probes, USA) is prepared as a 0.004% solution. Topical application to the murine oral cavity is carried out by the insertion of cotton buds soaked in the freshly prepared fluorescein or hypericin solutions for 5 to 10 minutes. Following an incubation period of 30 minutes, the mice are sacrificed and the tongues are excised for imaging. Excised tissue of the mouse tongue is sectioned and processed for haematoxylin and eosin (H&E) staining.

In our pilot clinical studies, 5-ALA is topically applied to the oral cavities of healthy volunteers and an oral SCC patient to compare ALA-induced PpIX fluorescence images from the normal and SCC human tongue. Hypericin (Molecular Probes, USA) is freshly prepared in 1% serum albumin in PBS and diluted in saline to give an 8 μm instillation solution just before administration. Fluorescein (Novartis Pharma AG, Switzerland) is freshly diluted in PBS to obtain a 0.1% solution. The fluorescein or hypericin solutions

are filtered and topically administered to the volunteers by oral rinsing using 100 ml of the solution over 30 minutes. Following an incubation period of at least 45 minutes, *in vivo* fluorescence 3D imaging of the oral cavity is carried out. In the patient trial, hypericin is topically applied prior to fluorescence 3D imaging.

We conducted the tests using Balb/c mouse models. Freshly prepared fluorescein sodium is topically applied to the murine oral cavity, 30 min prior to fluorescence imaging. Figure 8-1 shows fluorescein images from a confocal image stack acquired from the dorsal surface of the mouse tongue. Figure 8-1 (a) is captured from a single focal plane near the surface while Figure 8-1 (b) is from a deeper plane approximately 30 μm below the surface. Both images show filiform papillae (arrows), but the fluorescence intensity decreases as the imaging plane gets deeper.

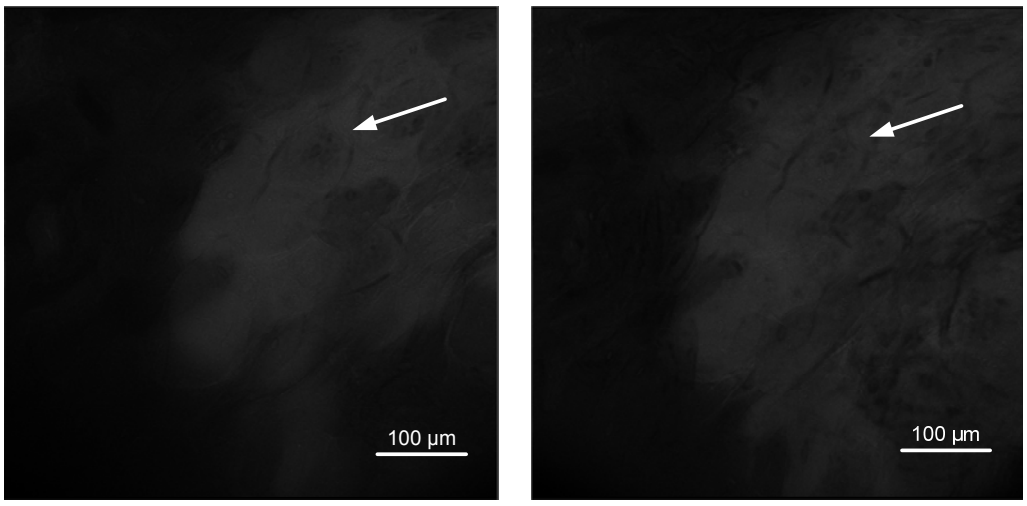


(a) (b)

Figure 8-1. The first (a) and the last image (b) from a stack of murine oral cavity images obtained from the confocal fluorescence endomicroscope

Figure 8-2 shows fluorescein images from a confocal image stack acquired from the dorsal surface of the human tongue using hypericin as the contrast agent. Figure 8-2 (a) is captured from a single focal plane near the surface while Figure 8-2 (b) is from a deeper

plane approximately 16 μm below the surface. Both images show filiform papillae (arrows). Since hypericin is not spectrally matched to our laser scanning source, the acquired stack of images is not as sharp as the stack obtained using fluorescein sodium.



(a) (b)

Figure 8-2. The first (a) and the last image (b) from a stack of human tongue images obtained from the confocal fluorescence endomicroscope

8.2.2 Confocal Endomicroscopic Imaging and 3D Visualization

A conventional confocal laser endomicroscope (CLE) captures and displays images from one single focal plane at a time. In order to develop a real-time 3D fluorescence imaging system and eventually move towards "virtual" biopsy, we interfaced a CLE with an embedded computing system. It is a specialized system to perform dedicated functions with real-time constraints. In our system, we used a Field Programmable Gated Array (FPGA) board as a reconfigurable platform.

The FPGA board has the required interfaces such as dual video support for DVI, TFT flat panel display, PS2 keyboard and mouse ports, a 4-line by 16-character LCD display, 8 white user-programmable LEDs, and general I/O pins. Other peripherals such as a

keyboard can also be used for user interface. In this study, we describe the development of the endomicroscope-embedded computing system for real-time 3D fluorescence imaging of the oral cavity.

The scanning machine is an Optiscan Five 1 confocal laser endomicroscope with a short hand held probe for oral cavity imaging. The laser source has an excitation wavelength of 488nm. This is coupled with a single fiber to aid in the confocal imaging. The scanning machine captures and displays images from a single focal plane. Figure 8-3 shows the different components of the system.

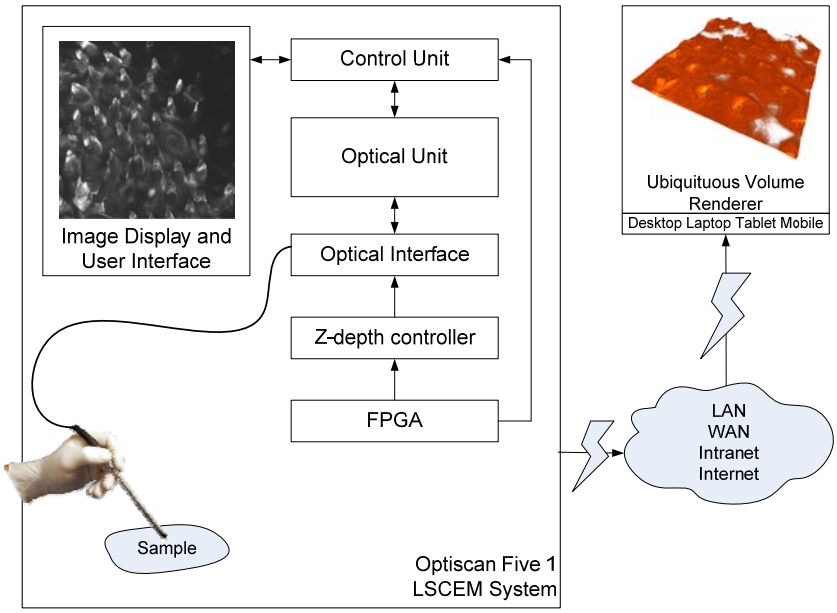


Figure 8-3. The overall real-time 3D endomicroscopy system design

The lateral resolution of the imaging probe is about 0.7 μm. The rigid probe houses the miniaturized components of the X-Y scanning mechanism, allowing images to be captured with a field of view of 475μm × 475μm. The laser power can be adjusted between 0 and a maximum of 1000 μW at the distal tip of the probe in contact with the tissue sample. Fluorescence signals are collected via a 505–750 nm emission filter.

Under the normal mode of operation, z-depth sectioning is achieved via a footswitch that controls the imaging depth from the surface down to deeper planes with a nominal

step size of about 4 μm between consecutive slices. In biological samples, the maximum imaging depth is about 250 μm below the surface, depending on the tissue's optical properties and the concentration of the contrast agent. The FPGA board acts as the z-depth controller and enables synchronized cross-sectional imaging between adjacent scans. With this programmed controller, the operator pushes one button for the FPGA board to start capturing a confocal image stack from the initial imaging depth until the desired imaging depth has been reached or the operator initiates a stop signal. This automated process speeds up the rate of image acquisition and effectively minimizes the chances of movement between consecutive images.

The acquired dataset is a series of images at various depths that are reconstructed into a 3D volume. The registered and mosaicked image stack is then illuminated from the viewing angle using algorithms such as Phong illumination algorithm and rendered using volume rendering algorithms. We applied our proposed nonlinear inter-slice directional interpolation scheme to the datasets acquired from the CLE scanner. Since our interpolation and visualization pipelines are tightly coupled, we can integrate the proposed techniques into the volume rendering system. The integrated imaging and visualization system, CelFI, is shown in Figure 8-4.

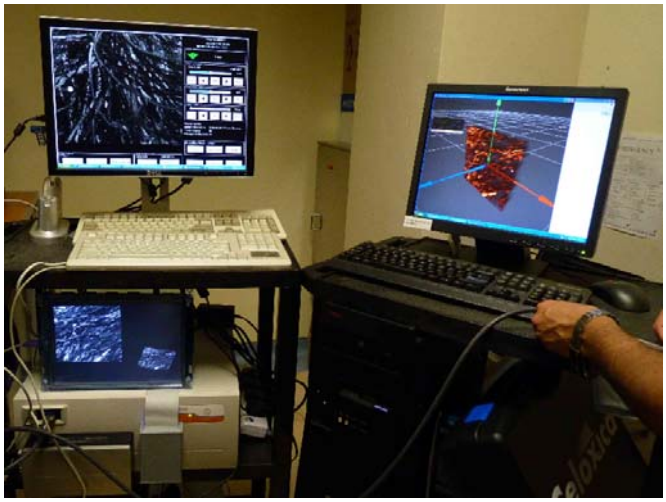


Figure 8-4. The integrated imaging and visualization system

8.3 Experimental Results

8.3.1 On-the-fly Nonlinear Interpolation for Volume Rendering

The scanned slices are pushed to the GPU texture memory, and the volume rendering is carried out on the CelFI system. In the fragment shader, our nonlinear interslice directional interpolation scheme is applied on-the-fly to obtain the density values from the volume texture containing the scanned slices, as detailed in chapter 5. To the obtained densities values, a transfer function is applied so that the features could be extracted.

Initial experiments were carried out on the murine tongue. Figure 8-5 shows the rendering results with the inter-slice directional interpolation scheme. As have been evaluated by our clinical partners, our proposed interpolation scheme can extract meaningful surface details for the diagnostic purpose. Despite the thin stack size, our proposed method extracts features that otherwise are invisible (see the texture features of the zoomed filiform papillae in Figure 8-5).

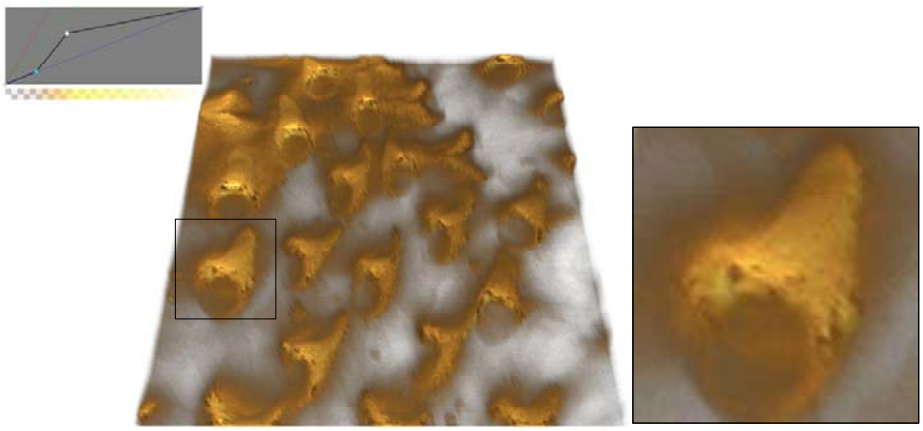


Figure 8-5. GPU volume rendering of the murine tongue obtained with nonlinear directional interpolation

8.3.2 Feature Enhancement for Volume Rendering

In the CelFI system, a customized widget was designed for assignment of the transfer function based on techniques detailed in chapter 6. This widget generates piecewise linear transfer functions and enables the user to create custom mappings for each individual color channel (alpha, red, green and blue) by plotting control points. The generated textures are streamed in realtime to the GPU texture memory so that the features could be enhanced instantly. Figure 8-6 shows the 3D rendering result obtained using the GPU-based volume rendering system for the entire confocal image stack.

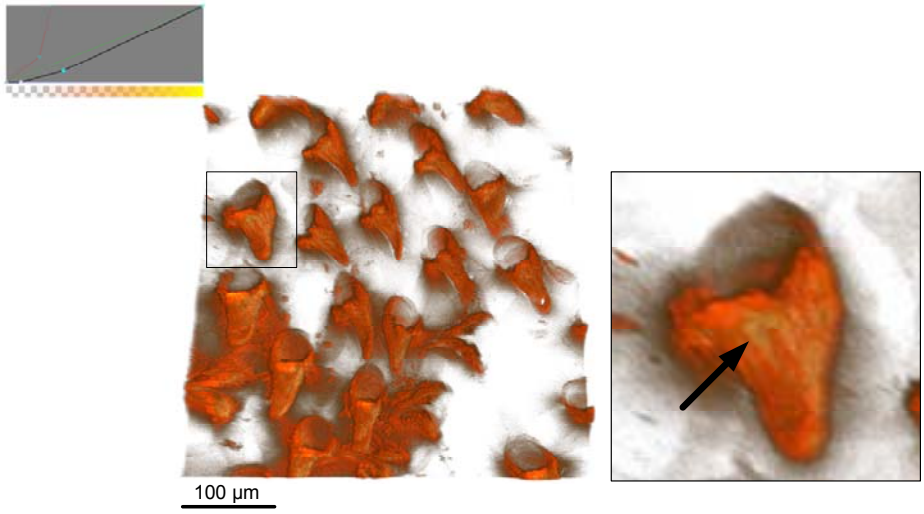


Figure 8-6. GPU volume rendering of the murine tongue obtained from confocal endomicroscope after application of fluorescein sodium, using the Blinn Phong shading

The conical shapes of the filiform papillae are well rendered in this 3D rendering that is displayed in pseudo-colors. The 3D image highlights depth and topographical information of the surface of the murine tongue that is not easily visualized from the individual 2D images obtained from the confocal endomicroscope. Such a 3D display demonstrates one step closer to the aim of achieving a real-time 3D “virtual” biopsy tool that can be used to visualize depth information in addition to morphology. The transfer function that is used for rendering is displayed in the top-left corner.

The Blinn Phong lighting model helps to highlight surface roughness and texture features that otherwise are invisible. Note that the surface roughness of the filiform papillae (the black arrow) is clearly visible. More surface details become visible after introduction of shading (see the zoomed filiform papillae in Figure 8-6).

The second set of experiments was conducted using hypericin. These results are presented in Figure 8-7 which shows the rendering results for a confocal image stack acquired from the dorsal surface of a mouse tongue following topical administration of hypericin. In addition to the normal composite rendering with classification in Figure 8-7 (a), we also used an omni directional light source to illuminate surface details as shown in Figure 8-7 (b).

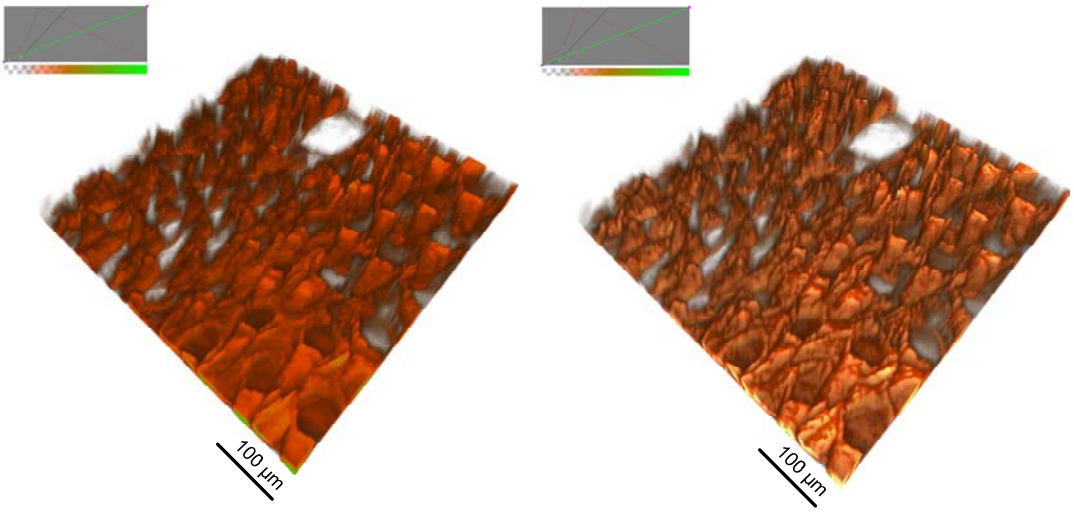


Figure 8-7. GPU volume rendering results for a confocal image stack acquired from the dorsal surface of a mouse tongue following topical administration of hypericin. The composite rendering result (a) without lighting and (b) with lighting

Confocal fluorescence image stacks of various sites in the human oral cavity were acquired *in vivo* from healthy volunteers following topical administration of fluorescein sodium or hypericin. The sites investigated include the dorsal surface of the tongue, base

of tongue, floor of mouth, the buccal mucosa and the lips. Real-time volume renderings of the acquired image stacks were achieved using the CelFI system.

Figure 8-8 (a, b) shows confocal images acquired from the dorsal surface of the human tongue following topical administration of fluorescein sodium. Both images are from the same dataset with image (a) captured from a focal plane near the surface and image (b) from a deeper layer approximately 30 μm below the surface. The individual fluorescence images show filiform papillae (solid arrows in Figure 8-8 (a)) and cellular structures (dotted arrows in Figure 8-8 (b)). The GPU volume rendering result of the entire dataset is shown in pseudo-color in Figure 8-8 (c) and it shows the depth relation of the filiform papillae with respect to the cellular structures below the surface. Such information on depth relation is not easily visualized from the individual 2D confocal images.

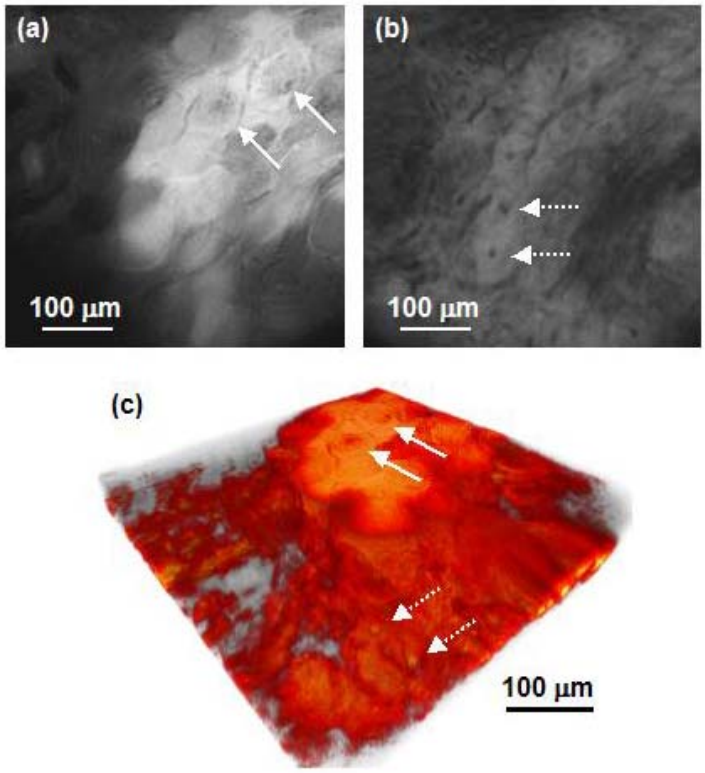


Figure 8-8. Confocal images acquired from the dorsal surface of the human tongue following topical administration of fluorescein sodium and the volume rendering result

Figure 8-9 shows confocal images obtained from the buccal mucosa of a healthy volunteer following topical application of fluorescein sodium. Both images in Figure 8-9 (a) and Figure 8-9 (b) belong to the same dataset. Figure 8-9 (a) is captured from a single focal plane near the surface and Figure 8-9 (b) is from a deeper layer approximately 30 μm below the surface. Cellular structures near the surface (solid arrow in (a)) and below the surface (dotted arrow in (b)) are observable from the individual images. The GPU volume rendering result of the entire dataset is displayed in pseudo-colors in (c) and shows the depth relation between the cellular structures from different layers.

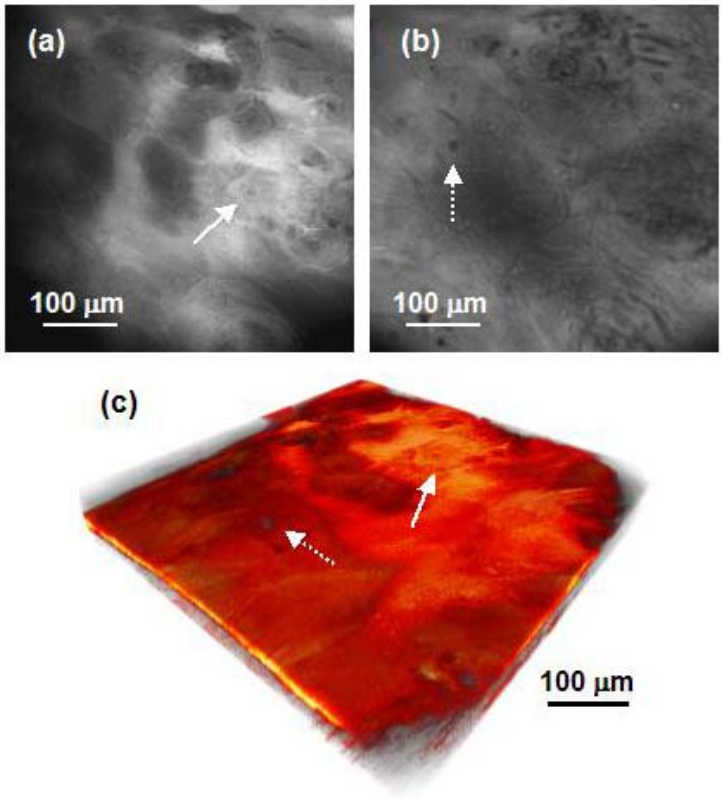


Figure 8-9. Confocal images acquired from the human buccal mucosa following topical administration of fluorescein sodium and the volume rendering result

Figure 8-10 shows images acquired from the human buccal mucosa following topical administration of hypericin. Both images in Figure 8-10 (a) and Figure 8-10 (b) belong to the same dataset. Figure 8-10 (a) is captured from a single focal plane near the surface and Figure 8-10 (b) is from a deeper layer approximately 15 μm below the surface. Figures 8-10 (a) and (b) show cellular structures near the surface (solid arrow) and below the surface (dotted arrow) while the GPU volume rendering result of the entire dataset is displayed in pseudo-colors in Figure 8-10 (c). It is noted that the volume rendering result from the hypericin dataset is shallow compared to the previous result from a fluorescein dataset and does not provide much depth information. This is due to weaker signals of hypericin fluorescence compared to fluorescein and as a consequence, shallower maximum imaging depth is achieved with topically applied hypericin.

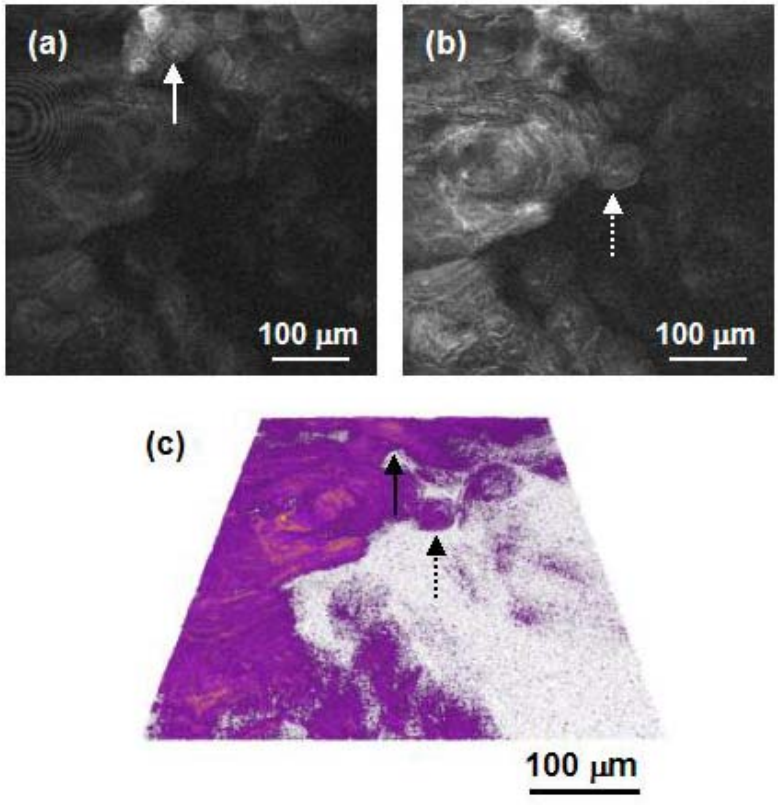
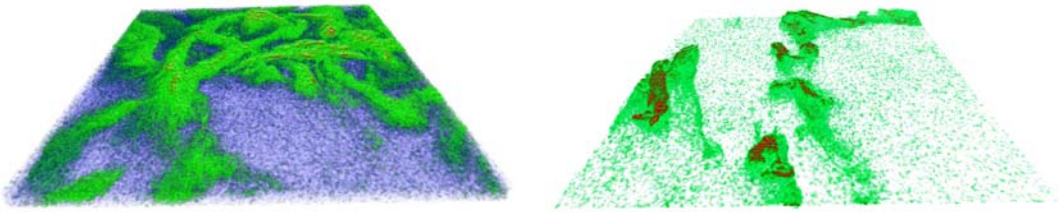


Figure 8-10. Confocal images acquired from the human buccal mucosa following topical administration of hypericin and the volume rendering result

Figure 8-11 shows the rendering result, displayed in pseudo-colors of a confocal image stack acquired from the human tongue. The assigned transfer function uses the feature enhancement pipeline discussed in chapter 6. The 3D image illustrates the topography and morphology of the filiform papillae on the surface of the tongue. Using the feature enhancement pipeline, the filiform papillae are visible instantly with little intervention by the user.



(a) (b)

Figure 8-11. Rendering results of human tongue datasets obtained using the GPU, showing the filiform papillae from (a) the dorsal surface of the tongue, structurally different from (b) the base of tongue

8.4 Summary

The success in development of our integrated system is a first step towards real-time 3D virtual biopsy. The prototype system enables automated acquisition of confocal image stacks. This automation minimizes the image acquisition time and reduces the chances of misalignment. Images of the normal rat tongue showed regularly arranged filiform papillae while the SCC rat tongue showed more irregular architecture. The results demonstrate that the CLE can be used to differentiate between normal and SCC tongue by morphology.

The 3D rendering results highlight topographical and depth information which is invisible with the conventional 2D confocal images. With further studies, 3D images may

be useful for detection of abnormalities based on altered architecture and morphology [146]. Depth information may also be important for assessment of the depth of neoplastic changes or invasive carcinoma.

In our pilot testing, we tried out both fluorescein sodium and hypericin as the fluorescent contrast agent. The excitation spectrum of fluorescein sodium spectrally matched the 488 nm laser in our endomicroscope system; therefore, images with strong fluorescence intensities were obtained. On the contrary, since the excitation spectrum of hypericin is not spectrally matched, the fluorescence signals were not as strong, especially in the normal tissue oral cavity used in our pilot tests. It is anticipated that hypericin may be selectively retained by abnormal tissue and hence offer enhanced contrast between normal and neoplastic tissue in the case of clinical cancer detection and image guided surgical or biopsy procedures [146].

The results from the pilot studies are encouraging and our prototype system proves that we may be able to use such a system in a clinical setting for early diagnosis of malignancies. The pilot studies were approved by the Centralized Institutional Review Board of the Singapore Health Services Pte Ltd, which we understand is a positive consideration on the prototype system for virtual biopsies. This will relieve the patients from the expensive and painful biopsies that are unfortunately unavoidable for early and middle stage cancer diagnosis.

9 CONCLUSION

9.1 Major Contributions and Research Impacts

This thesis presents a comprehensive study on modeling and rendering of volumetric graphical objects and the acceleration technologies for the new-generation GPU, focused on the novel hardware accelerated solutions by programming shaders on the GPU. The proposed modeling and rendering methods contribute to the disciplinary advancement of volume graphics. The GPU shaders developed in this project have a wide application domain in graphics industries, such as real-time 3D imaging and visualization.

First, we have proposed a novel GPU-based deformation pipeline and its extension to a meshless FEM model. The proposed deformation pipeline exploits the transform feedback mechanism of the modern GPU and can achieve real-time deforming transformation on a large dataset. We have applied the meshless FEM to nonrigid volumetric deformation, leveraging on our novel GPU-based deformation pipeline. And we have also integrated the volumetric deformation model with the HAPT algorithm and realized seamless volumetric deformation and rendering in a programmable graphics pipeline. Using the transform feedback mechanism for deformation entirely on the GPU, interactive visualization of deformation on large volumetric dataset is made possible. (The meshless FEM module has been extracted into an open source project (OpenCloth) which can be downloaded from http://code.google.com/p/opencloth to facilitate other researchers with our library.)

Secondly, we have devised a nonlinear directional volumetric data interpolation algorithm which can be efficiently implemented by the GPU fragment shader. While previous algorithms use the shape-based techniques which require pre-processing and hard classification, ours uses the underlying gradient of the dataset as direction for

interpolation and takes the local gradient information into account using a scale invariant function. Compared with the shape-based techniques, our method requires neither pre-processing of the original data nor hard classification thus is suitable for nonuniform and deformable datasets. In addition, we also exploit the SIMD architecture of the modern GPU by vectorizing most of our calculations. We also utilize the efficient dot product instruction in favor of a component-by-component multiplication. Implemented on the fragment shader, the high-quality interpolation can be efficiently incorporated into the on-the-fly volume rendering system.

Thirdly, we have invented a GPU accelerated algorithm for real-time volumetric feature detection, which can match the video rate of image acquisition, reconstruction and visualization. With the proposed morphological operations and local adaptive thresholding techniques, the automatically identified features are used for the transfer function construction, therefore, the algorithm allows feature-enhanced rendering without any user intervention. The pre-integrated volume rendering with second-order approximation provides unsurpassed quality of the rendered images. Being implemented by the GPU shader, the program enjoys high precision floating point arithmetic in all stages, enabling highly accurate feature computation in realtime.

Fourthly, we have explored the pervasive computing technology for high-quality and real-time volume rendering on the WebGL compliant platforms. We have presented a first-of-the-kind single-pass GPU ray caster and a new 3D texture slicer for WebGL. In contrast to the previous GPU volume rendering on WebGL, our implementation is able to handle dynamic transfer functions on mobile platforms. We also proposed an optimization to cast rays only in the volume dataset and discard all rays that do not intersect the volume's bounding box. A quantitative evaluation revealed that the proposed algorithm outperforms the existing WebGL ray caster on both stand-alone and mobile

platforms. Such high performance is helpful especially on the mobile platforms where the texture accesses are too costly.

Finally, as parts of our *in vivo* cellular imaging project with National Cancer Centre Singapore, the proposed algorithms have been integrated into our prototype CelFI system. The research work has been applied to diagnostic imaging of the oral cavity, and the 3D rendering results highlight topographical and depth information that is otherwise invisible with the conventional 2D confocal images. Depth information may also be important for assessment of the depth of neoplastic changes or invasive carcinoma.

9.2 Future Work

First of all, we would like to enhance our proposed volumetric deformation models. In the current meshless FEM implementation, the deformation result is directly dependent on the number of mesh points used. More points generally give a better approximation and vice versa. Moreover, since this method uses least square approximation, it cannot work with collinear points and thus it needs a 3D sparse point set. We tried using scatter operations and gather operations. Unfortunately scattering is unavailable in the shader APIs.

In the future work, we are thinking of two strategies: (i) In OpenGL v 4.2, the new image load-store extension allows a shader program to read from and write to arbitrary GPU memory location. This may be used to do scattered writes. (ii) We may consider a CUDA kernel for scattered writes, alongside a GLSL shader. In this way, CUDA may be used for scattered data writes as well as for processing the more computationally demanding steps. This hybrid scheme however requires a more rigorous treatment.

There are also rooms for improvement of the proposed nonlinear volumetric interpolation and feature detection algorithms. In our current work, for integration of the

interpolation processes in our CLE volume rendering system, we used the fragment shaders. This makes the method fragment-bound, therefore it becomes difficult to stream more fragment processing pipelines without a multi-pass strategy. We can avoid this by streaming the output from the fragment shader to the GPU memory and then chaining the CUDA/OpenCL kernel for more processing. This hybrid scheme will allow additional computationally expensive processing to be offloaded to the compute pipeline.

Another issue in the current volume rendering is the globally assigned transfer functions. We have been thinking of spatial transfer functions which can be incorporated by modifying the transfer function stage to utilize the identified gradient magnitudes and VOI positions. The current method relies on the region's local minimum and maximum intensities and the resulting rendering is subjective to the segmentation result. We are considering incorporating more sophisticated segmentation methods such as the level sets into our pipeline and use them to formulate new transfer functions.

Besides the algorithmic level considerations, with the new GPU embedded mobile devices, we expect more support in the upcoming WebGL implementations for more advanced shaders. Currently, due to the limitation in the loop iteration for dynamic loops on the mobile platforms, it is not possible to implement advanced shaders. Moreover, since the variable loop size in a fragment shader is a compile time constant, we cannot change the sampling rate at runtime. Our 3D texture slicer for WebGL allows rendering volume images with higher sampling rates on the mobile platforms and, performance-wise, it is with better frame rates. However, it suffers from depth fighting artifacts. The single-pass ray caster yields high quality renderings without such depth fighting artifacts, but the limited support for variable loops in the current WebGL implementations on the mobile platforms prevents us from introducing more optimization in the single-pass ray caster. We expect that the upcoming hardware will relax such restrictions. This will

enable the proposed single-pass ray caster to perform better on the upcoming mobile platforms.

As an application, our volume graphics shaders have been applied to the *in vivo* fluorescence endomicroscopy imaging system for detecting early-stage mucosal cancer cells in superficial tissues of live animal models and human volunteers. The CLE system combined with the real-time computer visualization system can provide an innovative approach to non-invasive cancer diagnosis. An exciting finding from our research is that such an online cellular imaging system can be further developed into a photodynamic therapy (PDT) tool by using therapeutic laser sources and instant feedback of their toxic effects can be visualized. It will open a door to minimally invasive, targeted PDT for all types of mucosal cancer, which will largely increase the survival rate of cancer patients.

Nevertheless, our initial study shows great challenge in development of an effective clinic protocol of the proposed PDT tool. Understanding the fluorescence images and online assessing therapeutic margins requires the radiochemist and endomicroscopy based surgeon to well establish the relationship between genotype and phenotype of the cancerous cells, the morphology of the specific cells and tissues under examination, the toxic effects on the targeted cancerous cells (and the surrounding tissues), and the cellular response to the therapeutic laser. Acquisition of this capability is through clinic education and training; and for obvious reasons, simulation based training is the most acceptable approach. Based on our algorithmic studies on the volume graphics shaders, we have successfully secured a research fund for “An Immersive Virtual Endomicroscopic Environment” from the University to further develop our *in vivo* imaging and visualization system into a real-time physiologically-based simulation system.

AUTHOR'S PUBLICATIONS

Movania Muhammad Mobeen, Lin Feng, Qian Kemao, Chiew Wei-Ming and Seah Hock-Soon, "Coupling between Meshless FEM Modeling and Rendering on GPU for Real-time Physically-based Volumetric Deformation", *Journal of WSCG*, Vol. 20, No. 1, pp:1-10, ISSN 1213-6972, Union Agency, 2012.

Movania Muhammad Mobeen and Lin Feng, "A Novel GPU-based Deformation Pipeline," *ISRN Computer Graphics*, Vol. 2012, Article ID: 936315, 8 pages, 2012. doi:10.5402/2012/936315.

Movania Muhammad Mobeen and Lin Feng, "Real-time Physically-based Deformation on the GPU using Transform Feedback," Chapter 17, *The OpenGL Insights*, AK Peters/CRC press, pp:233-248, 2012.

Movania Muhammad Mobeen, Lin Feng, Qian Kemao, Chiew Wei-Ming and Seah Hock-Soon, "Coupling between Meshless FEM Modeling and Rendering on GPU for Real-time Physically-based Volumetric Deformation", in *Proceedings of the EUROGRAPHICS International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2012)*, Prague Czech Republic, 25-28 June 2012.

Movania Muhammad Mobeen and Lin Feng, "High-Performance Volume Rendering on the Ubiquitous WebGL Platform," in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communication (HPCC 2012)*, Liverpool UK, 25-28 June 2012.

Movania Muhammad Mobeen and Lin Feng, "Ubiquitous Medical Volume Rendering on Mobile Devices," in *Proceedings of the 3rd IEEE International Conference on Information Society (i-Society 2012)*, pp: 93-98, London, UK, 25-28 June 2012.

Patricia S. P. Thong, Stephanus S. Tandjung, **Movania Muhammad Mobeen**, Wei-Ming Chiew, Malini Olivo, Ramaswamy Bhuvaneswari, Lin Feng, Qian Kemao, Seah Hock-Soon, Khee-Chee Soo, "Towards Real-time Virtual Biopsy of Oral Lesions using Confocal Laser Endomicroscopy Interfaced with Embedded Computing," *Journal of Biomedical Optics*, Vol. 7 No. 5, pp:056009, 2012.

Patricia S. P. Thong, Malini Olivo, Stephanus S. Tandjung, **Movania Muhammad Mobeen**, Lin Feng, Qian Kemao, Seah Hock-Soon, Khee-Chee Soo, "Review of Confocal Fluorescence Endomicroscopy for Cancer Detection," *IEEE Photonics Society (IPS) Journal of Selected Topics in Quantum Electronics*, Volume PP(99), pp:1-12, 2011.

Patricia S. P. Thong, Malini Olivo, **Movania Muhammad Mobeen**, Stephanus S. Tandjung, Seah Hock-Soon, Lin Feng, Qian Kemao and Khee-Chee Soo, "Hypericin Fluorescence Imaging of Oral Cancer: From Endoscopy to Real-time 3-Dimensional Endomicroscopy," *Journal of Medical Imaging Health Informatics*, Vol 1(1), pp: 139-143 (2011).

Patricia S. P. Thong, Ramaswamy Bhuvaneswari, Malini Olivo, **Movania Muhammad Mobeen**, Stephanus S. Tandjung, Seah Hock-Soon, Lin Feng, Qian Kemao and Khee-Chee Soo, "Toward 3-dimensional virtual biopsy of oral lesions through the development of a confocal endomicroscope interfaced with embedded computing," in *Proceedings of the SPIE-OSA Biomedical Optics*, SPIE Vol. 8086, pp:80860W, Munich, Germany, 2011.

Movania Muhammad Mobeen, Cheong Lee Sing, Zhao Feng, Lin Feng, Qian Kemao, Seah Hock-Soon, "GPU-based Surface Oriented Interslice Directional Interpolation for Volume Visualization," Proceedings of *Isabel 2009, 2nd International Symposium on Applied Sciences in Biomedical and Communication Technologies*, Bratislava, Slovak Republic, pp:1-5, 2009.

Movania Muhammad Mobeen, Lin Feng, Qian Kemao, Seah Hock-Soon, "Automated Local Adaptive Thresholding for Real-time Feature Detection and Rendering of 3D Endomicroscopy Images on GPU," in *Proceedings of the World Comp, Computer Graphics and Virtual Reality (CGVR'09)*, *World Comp'09*, Las Vegas, US, July 13-16, 2009, pp:10-16.

Movania Muhammad Mobeen, Lin Feng, Qian Kemao, Seah Hock-Soon, "On-Demand Volumetric Feature Rendering through GPU Acceleration", in *Proceedings of the 3rd Asian Conference on Computer Aided Surgery (ACCASS 2007)*, Singapore, 1-2 December 2007.

REFERENCES

[1] B. Lichtenbelt, R. Crane, and S. Naqvi, *Introduction to volume rendering*: Prentice-Hall, Inc., 1998.

[2] K. Engel, M. Hadwiger, J. M. Kniss, A. Lefohn, C. R. Salama, and D. Weiskopf, *Real-time volume graphics*. Los Angeles, CA: ACM, 2004.

[3] B. Preim and D. Bartz, *Visualization in Medicine* Elsevier Inc, 2007.

[4] NVIDIA. *Architecture documents, Available online: http://developer.nvidia.com/object/arch_docs_by_date.html accessed in 2012*.

[5] G. M. Nielson, "On Marching Cubes," *IEEE Trans. Vis. Comput. Graphics,* vol. 9(3), pp. 283-297, 2003.

[6] G. M. Nielson, G. Graf, A. Huang, M. Phliepp, and R. Holmes, "Shrouds: Optimal Separating Surface for Enumerated Volumes," in *IEEE TCVG Symp. Vis.*, 2003, pp. 75-84.

[7] W. Wang, D. Zhou, and E. Wu, "Accelerating Techniques in Volume Rendering of Irregular Data," *Comput. Graph.,* vol. 21(3), pp. 289-295, 1997.

[8] W. Wang and E. Wu, "Adaptable Splatting for Irregular Volume Rendering," *Comput. Graph. Forum,* vol. 18(4), pp. 213-222, 1999.

[9] E. C. Lamar, B. Hamann, and K. I. Joy, "Multiresolution techniques for interactive texture-based volume visualization," in *Proceedings of the 10th IEEE Visualization Conference (VIS '99)*, 1999.

[10] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *Computer Graphics (Proceedings of the ACM SIGGRAPH '87),* vol. 21(4), pp. 163-169, 1987.

[11] F. Lin, H. S. Seah, and Y. T. Lee, "Deformable volumetric model and isosurface: Exploring a new approach for surface construction," *Computers and Graphics,* vol. 20(1), pp. 33-40, 1996.

[12] F. Lin, H. S. Seah, Z. Wu, and D. Ma, "Voxelisation and fabrication of freeform models," *Virtual and Physical Prototyping,* vol. 2(2), pp. 65-73, 2007.

[13] J. F. Hughes, "Scheduled fourier volume morphing," in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '92)* 1992, pp. 43-46.

[14] T. He, S. Wang, and A. Kaufman, "Wavelet based volume morphing," in *Proceedings of the IEEE Visualization '94 (VIS'94)*, 1994, pp. 85-92.

[15] Y. Kurzion and R. Yagel, "Space deformation using ray deflectors," in *Proceedings of the sixth Eurographics Workshop on Rendering (Rendering Techniques'95)*, 1995, pp. 21-30.

[16] Y. Kurzion and R. Yagel, "Interactive space deformation with hardware assisted rendering," *IEEE Computer Graphics and Applications,* pp. 66-77, 1997.

[17] S. Fang, S. Huang, S. Rajagopalan, and R. Raghavan, "Deformable volume rendering by 3D texture mapping and octree encoding," in *Proceedings of IEEE Visualization '96*, 1996.

[18] R. Westermann and C. Rezk-Salama, "Real-time volume deformation," in *Proceedings of Eurographics*, 2001.

[19] S. F. Gibson, "3D ChainMail: A fast algorithm for deforming volumetric objects," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics (I3D '97)* 1997, pp. 149-154.

[20] G. Nikhil, D. Kenchammana-Hosekote, and D. Silver, "Volume animation using the skeleton tree," in *Proceedings of the IEEE Symposium on Volume Visualization*, 1998, pp. 47-53.

[21] Y. Chen, Q.-h. Zhu, and A. Kaufman, "Physically based animation of volumetric objects," Technical report (TR-CVC-980209), 1998.

[22] A. Nealen, M. Mueller, R. Keiser, E. Boxerman, and M. Carlson, "Physically based deformable models in computer graphics," *STAR report Eurographics 2005,* vol. 25(4), pp. 809-836, 2005.

[23] S. P. Raya and J. K. Udupa, "Shape-based interpolation of multidimensional objects," *IEEE Transactions on Medical Imaging,* vol. 9(1), pp. 32-42, 1990.

[24] K. W. Chun, M. S. Lee, and J. B. Ra, "Directional 3-D interpolation technique for volume rendering," in *Proceedings of the Nuclear Science Symposium and Medical Imaging Conference*, 1991, pp. 2145 - 2148.

[25] V. Interrante, "Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*, 1997.

[26] J. Hladuvka and M. E. Groller, "Direction-driven shape-based interpolation of volume data," Technical Report (TR-186-2-01-12), Institute of computer graphics and algorithms, Vienna University of Technology, 2001.

[27] B. Csebfalvi, L. Neumann, A. Kanitsar, and E. Groller, "Smooth shape-based interpolation using the conjugate gradient method," in *Proceedings of Vision, Modeling, and Visualization (VMV '02)*, 2002, pp. 123-130.

[28] R. R. Bertram, J. M. Daida, J. F. Vesecky, G. A. Meadows, and C. Wolf, "Reconstructing incomplete signals using nonlinear interpolation and genetic algorithms," in *Proceedings of the Third Annual Conference on Genetic Programming'98*, 1998, pp. 19-27.

[29] Z. Mihajlovic, L. Budin, and J. Radej, "Gradient of B-splines in Volume Rendering," in *Proceedings of the IEEE MELECON'04*, 2004, pp. 239-242.

[30] S. Fu, Q. Ruan, and W. Wang, "Feature Preserving Image Interpolation and Enhancement Using Adaptive Bidirectional Flow," *World Academy of Science, Engineering and Technology* vol. 2(1), pp. 144-147, 2005.

[31] S. Fu, Q. Ruan, and W. Wang, "Feature Preserving Nonlinear Image Interpolation," in *Proceedings of the 8th International Conference on Signal Processing*, 2006.

[32] S. Fang, Y. Dai, F. Myers, M. Tuceryan, and K. Dunn, "Three-dimensional microscopy data exploration by interactive volume visualization," *Journal of Scanning Microscopies,* vol. 22(4), pp. 218-226, 2000.

[33] J. L. Clendenon, C. L. Phillips, R. M. Sandoval, S. Fang, and K. W. Dunn, "Voxx:a pc-based, near real-time volume rendering system for biological microscopy," *Am J Physiol Cell Physiol,* vol. 282(1), pp. C213-C218, 2002.

[34] J. Chen, D.-H. Hong, and X.-X. Zheng, "An interactive volume rendering algorithm for laser scanning confocal microscope data," *Space Medicine and Medical Engineering (Space Med Med Eng (Beijing)),* vol. 17(3), pp. 229-231, 2004.

[35] H.-J. Choi, I.-H. Choi, T.-Y. Kim, N.-H. Cho, and H.-K. Choi, "Three-dimensional visualization and quantitative analysis of cervical cell nuclei with confocal laser scanning microscopy," *Analytical and Quantitative Cytology and*

Histology / The International Academy of Cytology and American Society of Cytology (Anal Quant Cytol Histol), vol. 27(3), pp. 174-180, 2005.

[36] Q. Wang, Y. Sun, and J. P. Robinson, "GPU-based visualization techniques for 3D microscopic imaging data," in *Proceedings of the SPIE '07*, 2007, p. 64981H.

[37] K. Mosaliganti, L. Cooper, R. Sharp, R. Machiraju, G. Leone, K. Huang, and J. Saltz, "Reconstruction of cellular biological structures from optical microscopy data," *IEEE Transaction on Visualization and Computer Graphics,* vol. 14(4), pp. 863-876, 2008.

[38] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics Application (IEEE CGA),* vol. 8(3), pp. 29-37, 1988.

[39] J. Marks, B. Andalman, P. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, P. Hanspeter, W. Ruml, K. Ryall, J. Seims, and S. Shieber, "Design Galleries: A general approach for setting parameters for computer graphics and animation," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH' 97)*, Los Angeles, CA, 1997, pp. 389-400.

[40] T. He, L. Hong, A. Kaufman, and H. Pfister, "Generation of transfer functions with stochastic search techniques," in *Proceedings of the IEEE Visualization'96 (VIS '96)*, 1996.

[41] S. Fang, T. Biddlecome, and M. Tuceryan, "Image-based transfer function design for data exploration in volume visualization," in *Proceedings of the IEEE Visualization 1998 (VIS '98)*, 1998.

[42] Y. Sato, C.-F. Westin, and A. Bhalerao, "Tissue classification based on 3D local intensity structures for volume rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 6(2), pp. 160-180, 2000.

[43] G. Kindlmann and J. W. Durkin, "Semi-automatic generation of transfer functions for direct volume rendering," in *Proceedings of the 1998 IEEE Symposium on Volume Visualization (VIS '98)*, Research Triangle Park, North Carolina, United States, 1998.

[44] C. Rezk-Salama, P. Hastreiter, J. Scherer, and G. Greiner, "Automatic adjustment of transfer functions for 3D volume visualization," in *Proceedings of the Vision, Modeling and Visualization (VMV'2000)*, 2000.

[45] J. Hladuvka, A. Konig, and E. Groller, "Curvature-based transfer functions for direct volume rendering," in *Proceedings of the Spring Conference on Computer Graphics 2000*, 2000, pp. 58-65.

[46] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller, "Curvature-based transfer functions for direct volume rendering: methods and applications," in *Proceedings of the 14th IEEE Visualization (VIS'03)*, 2003.

[47] D. S. Ebert, C. J. Morris, P. Rheingans, and T. S. Yoo, "Designing effective transfer functions for volume rendering from photographic volumes," *IEEE Transactions on Visualization and Computer Graphics (IEEE TVCG),* vol. 8(2), pp. 183-197, 2002.

[48] S. Potts and T. Moller, "Transfer functions on a logarithmic scale for volume rendering," in *Proceedings of Graphics Interface 2004 (GI '04)*, London, Ontario, Canada, 2004.

[49] J. Kniss, G. Kindlmann, and C. Hansen, "Multidimensional transfer functions for interactive volume rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 8(3), pp. 270-285, 2002.

[50] S. Roettger, M. Bauer, and M. Stamminger, "Spatialized transfer functions," in *Proceedings of the IEEE/Eurographics Symposium on Volume Visualization (EuroVis'05)*, 2005, pp. 271-278.

[51] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma, "A novel interface for higher-dimensional classification of volume data," in *Proceedings of the 14th IEEE Visualization (VIS'03)*, 2003.

[52] A. Tappenbeck, B. Preim, and V. Dicken, "Distance-based transfer function design: Specification methods and applications," in *Proceedings of Simulation and Visualisierung*, 2006, pp. 259-274.

[53] C. Rezk-Salama, M. Keller, and P. Kohlmann, "High-level user interfaces for transfer function design with semantics," *IEEE Transactions on Visualization and Computer Graphics,* vol. 12(5), pp. 1021-1028, 2006.

[54] S. Bruckner and E. Groller, "Style transfer functions for illustrative volume rendering," *Computer Graphics Forum,* vol. 26(3), pp. 715-724, 2007.

[55] T. Ropinski, J.-S. Praßni, F. Steinicke, and K. H. Hinrichs, "Stroke-based transfer function design," in *Proceedings of the IEEE/EG International Symposium on Volume and Point-based Graphics*, 2008, pp. 41-48.

[56] J. T. Kajiya and B. P. V. Herzen, "Ray tracing volume densities," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, 1984, pp. 165-174.

[57] M. P. Garrity, "Raytracing irregular volume data," in *Proceedings of the 1990 Workshop on Volume Visualization (VVS '90)*, San Diego, California, United States, 1990.

[58] P. Bunyk, A. Kaufman, and C. T. Silva, "Simple, fast, and robust ray casting of irregular grids," in *Proceedings of the Conference on Scientific Visualization*, 1997.

[59] M. Brady, K. Jung, H. T. Nguyen, and T. Nguyen, "Two-phase perspective ray casting for interactive volume navigation," in *Proceedings of the 8th Conference on Visualization '97 (VIS '97)*, Phoenix, Arizona, United States, 1997.

[60] K. Koji, "Fast ray-casting for irregular volumes," in *Proceedings of the Third International Symposium on High Performance Computing*, 2000.

[61] R. Westermann and B. Sevenich, "Accelerated volume ray-casting using texture mapping," in *Proceedings of the IEEE Visualization '01 (VIS'01)*, San Diego, California, 2001.

[62] B. Mora, J.-P. Jessel, and R. Caubet, "A new object-order ray-casting algorithm," in *Proceedings of the Conference on Visualization (VIS '02)*, Boston, Massachusetts, 2002.

[63] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3D texture hardware," Technical Report, University of North Carolina at Chapel Hill,1994.

[64] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proceedings of the Symposium on Volume Visualization'94*, Tysons Corner, Virginia, United States, 1994.

[65] A. Van Gelder and K. Kim, "Direct volume rendering with shading via three-dimensional textures," in *Proceedings of the 1996 Symposium on Volume Visualization (VIS'96)*, San Francisco, California, United States, 1996.

[66] R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*, 1998.

[67] M. Meissner, U. Hoffmann, and W. Strasser, "Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions," in *Proceedings of the Conference on Visualization '99 (VIS'99)*, San Francisco, California, United States, 1999.

[68] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Interlaken, Switzerland, 2000.

[69] H. Younesy, T. Möller, and H. Carr, "Improving the quality of multi-resolution volume rendering," in *Proceedings of the Eurographics/IEEE-VGTC Symposium on Visualization (2006)*, Lisbon, Portugal, 2006, pp. 251-258.

[70] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen, "Interactive texture-based volume rendering for large data sets," *IEEE Computer Graphics and Applications (IEEE CGA),* vol. 21(4), pp. 52-61, 2001.

[71] C. Rezk-Salama and A. Kolb, "A vertex program for efficient box-plane intersection," in *Proceeding of Vision, Modeling and Visualization (VMV '05)*, 2005, pp. 115-122.

[72] M. Hadwiger, T. Theu, H. Hauser, and E. Groller, "Hardware-accelerated high-quality reconstruction on pc hardware," in *Proceedings of the Vision, Modeling and Visualization Conference 2001 (VMV '01)*, 2001.

[73] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser, "Fast and flexible high-quality texture filtering with tiled high-resolution filters," in *Proceedings of Vision, Modeling, and Visualization*, 2002, pp. 155-162.

[74] S. Roettger, M. Kraus, and T. Ertl, "Hardware-accelerated volume and isosurface rendering based on cell-projection," in *Proceedings of the Conference on Visualization 2000 (VIS '00)*, Salt Lake City, Utah, United States, 2000.

[75] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and T. Ertl, "High-quality unstructured volume rendering on the pc platform," in *Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop*, 2002, pp. 119-125.

[76] P. Ivo, "Advanced volume ray casting on GPU," Diploma, Department of Software and Computer Science Education, Charles University, Prague, 2009.

[77] T. Vassilev and B. Spanlang, "A mass-spring model for real-time deformable solids," in *Proceedings of East-West Vision*, Graz, Austria, 2002, pp. 149-154.

[78] J. Mosegaard, P. Herborg, and T. S. Sorensen, "A GPU accelerated spring mass system for surgical simulation," in *Proceedings of the Health Technology and Informatics 111*, 2004, pp. 342-348.

[79] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on GPUs," in *Proceedings of the Simulation and Visualization*, 2005.

[80] J. Georgii and R. Westermann, "Mass-spring systems on the GPU," *Simulation Practice and Theory,* vol. 13, pp. 693-702, 2005.

[81] Y. Shen, Z. Xiangmin, Z. Nan, and T. K. S. Robert, "Realistic soft tissue deformation strategies for real-time surgery simulation," Research Report (UMSI 2009/13), University of Minnesota supercomputing institute,2009.

[82] C. A. Diaz Leon, S. Eliuk, and H. T. Gomez, "Simulating soft tissues using a GPU approach of the mass-spring model," in *Proceedings of the Virtual Reality Conference*, 2010, pp. 261-262.

[83] E. Tejada and T. Ertl, "Large steps in GPU-based Deformable Bodies Simulation," *Simulation Modeling Practice and Theory,* vol. 13(8), pp. 703-715, 2005.

[84] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models," *Computer Graphics (ACM SIGGRAPH 87 Conference Proceedings),* vol. 21(4), pp. 205-214, 1987.

[85] M. Bro-Nielsen and S. Cotin, "Real-time volumetric deformable models for surgery simulation using finite elements and condensation," *Computer Graphics Forum,* vol. 15(3), pp. C57-66, 1996.

[86] D. James and D. K. Pai, "ArtDefo: Accurate realtime deformable objects," *Computer Graphics (ACM SIGGRAPH 99 conference proceedings),* vol. 5(1), pp. 65-72, 1999.

[87] C. A. Felippa, "A systematic approach to the element independent corotational dynamics of finite elements," Technical Report (CUCAS-00-03), Center for Aerospace Structures, 2000.

[88] M. Muller and M. Gross, "Interactive virtual materials," in *Proceedings of Graphics Interface (GI 2004)*, 2004, pp. 239-246.

[89] M. Hauth and W. Strasser, "Corotational simulation of deformable solids," in *Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'04)*, 2004, pp. 137-145.

[90] J. Georgii and R. Westermann, "Corotated finite elements made fast and stable," in *Proceedings of the 5th workshop on Virtual Reality, Interaction and Physical Simulation*, 2008, pp. 11-19.

[91] Y. Zhuang and J. Canny, "Real-time simulation of physically realistic global deformation," in *Proceedings of the IEEE Visualization 1999 (VIS '99)*, 1999, pp. 270-273.

[92] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr, "Dynamic real-time deformations using space and time adaptive sampling," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*, 2001, pp. 31-36.

[93] C. Mendoza and C. Laugier, "Simulating soft tissue cutting using finite element models," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2003, pp. 1109-1114.

[94] G. Picinbono, H. Delingette, and N. Ayache, "Non-linear anisotropic elasticity for real-time surgery simulation," *Graphical Models,* vol. 6(5), pp. 305-321, 2003.

[95] H. Zhong, M. P. Wachowiak, and T. M. Peters, "A real time finite element based tissue simulation method incorporating nonlinear elastic behavior," *Computer Methods Biomechan. Biomed. Eng.,* vol. 6(5), pp. 177-189, 2005.

[96] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation,* vol. 31(138), pp. 333-390, 1977.

[97] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*: SIAM, 2000.

[98] J. Georgii and R. Westermann, "A multi-grid framework for real-time simulation of deformable volumes," in *Proceedings of the Workshop on Virtual Reality Interactions and Physical Simulations (VRIPHYS '05)*, 2005.

[99] C. Dick, J. Georgii, and R. Westermann, "A real-time multigrid finite hexahedra method for elasticity simulation using CUDA," *Simulation Modelling Practice and Theory,* vol. 19(2), pp. 801-816, 2010.

[100] K. Miller, G. Joldes, D. Lance, and A. Wittek, "Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation," *Internal Journal for Numerical Methods in Biomedical Engineering,* vol. 23(2), pp. 801-816, 2007.

[101] Z. Taylor, M. Cheng, and S. Ourselin, "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units," *IEEE Transactions on Medical Imaging,* vol. 27(5), pp. 650-663, 2008.

[102] O. Comas, Z. A. Taylor, J. Allard, S. Ourselin, S. Cotin, and J. Passenger, "Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework sofa," in *Proceedings of the International Symposium on Computational Models for Biomedical Simulation*, 2008, pp. 28-39.

[103] J. Allard, H. Courtecuisse, and F. Faure, "Implicit FEM and fluid coupling on GPU for interactive multiphysics simulation," in *Proceedings of the ACM SIGGRAPH (SIGGRAPH'11)*, 2011.

[104] M. Kraus, M. Eissele, and M. Strengert, "GPU-based edge-directed image interpolation," in *Proceedings of the 15th Scandinavian Conference on Image Analysis (SCIA'07)*, 2007.

[105] R. Cheng, E. Yang, T. Liu, and L. Wu, "CUDA-based directional image/video interpolation," in *Proceedings of the International Conference on Audio Language and Image Processing (ICALIP'10)*, 2010, pp. 125-129.

[106] N. Robidoux, M. Gong, J. Cupitt, A. Turcotte, and K. Martinez, "CPU, SMP and GPU implementations of Nohalo Level 1, a fast co-convex antialiasing image resampler," in *Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*, 2009, pp. 185-195.

[107] C. Sigg and M. Hadwiger, "Fast Third-Order Texture Filtering," *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation,* pp. 313–329, 2005.

[108] D. Ruijters, B. M. t. H. Romeny, and P. Suetens, "Efficient gpu-based texture interpolation using uniform b-splines," *Journal of Graphics, GPU and Game Tools* vol. 13(4), pp. 61-69, 2008.

[109] P. Thévenaz, T. Blu, and M. Unser, "Interpolation Revisited," *IEEE Transactions on Medical Imaging,* vol. 19(7), pp. 739-758, 2000.

[110] D. Ruijters and P. Thévenaz, "GPU prefilter for accurate cubic B-spline interpolation," *The Computer Journal,* vol. 2(1), 2010.

[111] D. Ruijters, B. M. t. H. Romeny, and P. Suetens, "Accuracy of GPU-based B-Spline Evaluation," in *Proc. Tenth IASTED International Conference on Computer Graphics and Imaging (CGIM'08)*, Innsbruck, Austria, 2008, pp. 117-122.

[112] F. Champagnat and Y. L. Sant, "Efficient cubic B-spline image interpolation on a GPU " *Journal of Graphics, GPU, and Game Tools,* 2012.

[113] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," *Computer Graphics (Proceedings of the ACM SIGGRAPH '90),* vol. 24(5), pp. 63-70, 1990.

[114] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno, "Tetrahedral projection using vertex shaders," in *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, Boston, Massachusetts, 2002.

[115] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based view-independent cell projection," *IEEE Transactions on Visualization and Computer Graphics,* vol. 9(2), pp. 163-175, 2003.

[116] M. Kraus, W. Qiao, and D. S. Ebert, "Projecting tetrahedra without rendering artifacts," in *Proceedings of the Conference on Visualization (VIS '04)*, 2004.

[117] K. Moreland and E. Angel, "A fast high accuracy volume renderer for unstructured data," in *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, 2004.

[118] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Proceedings of the 14th IEEE Conference on Visualization (VIS '03)*, 2003, pp. 333-340.

[119] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 11(3), pp. 285-295, 2005.

[120] J. Georgii and R. Westermann, "A generic and scalable pipeline for GPU tetrahedral grid rendering," *IEEE Transaction on Visualization and Computer Graphics,* vol. 12(5), pp. 1345-1352, 2006.

[121] R. Marroquim, A. Maximo, R. Farias, and C. Esperança, "Volume and isosurface rendering with GPU-accelerated cell projection " *Computer Graphics Forum,* vol. 27(1), pp. 24-35, March 2008 2008.

[122] A. Maximo, R. Marroquim, and R. Farias, "Hardware-assisted projected tetrahedra," *Computer Graphics Forum,* vol. 29(3), pp. 903-912, 2010.

[123] J. Kruger and R. Westermann, "Acceleration techniques for GPU-based volume rendering," in *Proceedings of the 14th IEEE visualization (VIS'03)*, 2003.

[124] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting," in *Proceedings of the Fourth International Workshop on Volume Graphics* 2005, pp. 187 - 241.

[125] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*, 1998.

[126] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," Technical Report, 1994.

[127] S. Bhattacharjee, S. Patidar, and P. J. Narayanan, "Real-time Rendering and Manipulation of Large Terrains," in *Sixth Indian Conference on Computer Vision, Graphics & Image Processing (ICVGIP 2008)*, Bhubaneswar, India, 2008, pp. 551-559.

[128] EXT_transform_feedback. *The OpenGL Specification Registry, Available online: http://www.opengl.org/registry/specs/EXT/transform_feedback.txt, accessed in 2012*.

[129] ARB_transform_feedback2. *The OpenGL Specification Registry, Available online: http://www.opengl.org/registry/specs/ARB/transform_feedback2.txt accessed in 2012*.

[130] ARB_transform_feedback3. *The OpenGL Specification Registry, Available online: http://www.opengl.org/registry/specs/ARB/transform_feedback3.txt accessed in 2012*.

[131] G. Xiahu and Q. Hong, "Realtime Meshless Deformation," *Computer Animation and Virtual Worlds,* vol. 16, pp. 189-200, 2005.

[132] B. Adams, M. Ovsjanikov, M. Wand, H.-P. Seidel, and L. J. Guibas, "Meshless modelling of deformable shapes and their motion," in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2008, pp. 77-86.

[133] F. Faure, B. Giles, G. Bousquet, and D. K. Pai, "Sparse meshless models of complex deformable solids," *ACM Transactions on Graphics* vol. 30, 2011.

[134] M. Muller, D. Charypar, and M. Gross, "Particle based fluid simulation for interactive applications," in *Proc. ACM SIGGRAPH Symp. Comput. Anim.*, 2003, pp. 154-159.
[135] I. Buck, "Taking the plunge into GPU computing," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds., ed: Addison-Wesley, 2005, pp. 509-519.
[136] R. D. Skeel, "Variable Step Size Detabilizes the Stömer/Leapfrog/Verlet Method," *BIT Numerical Mathematics,* vol. 33, pp. 172-175, 1993.
[137] C. M. Stein, B. G. Becker, and N. Max, "Sorting and hardware assisted rendering for volume visualization," in *Proceedings of the 1994 Symposium on Vlume Visualization*, Tysons Corner, Virginia, United States, 1994.
[138] P. Williams, N. Max, and C. Stein, "A high accuracy volume renderer for unstructured data," *IEEE Transactions on Visualization and Computer Graphics,* vol. 4(1), pp. 37-54, 1998.
[139] D. M. Shotton, "Confocal scanning optical microscopy and its applications for biological specimens," *Journal Cell Science,* vol. 94, pp. 175-206, 1989.
[140] D. Astle, P. Baker, D. Christopoulos, J. Citron, and A. Dorbie, "More OpenGL Game Programming," 2005, p. 600.
[141] K. Engel, M. Kraus, and T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Los Angeles, California, United States, 2001.
[142] M. Chris, "The Official WebGL Specifications," *Available online: http://www.khronos.org/registry/webgl/specs/latest/ accessed in 2012*.
[143] C. John, K. Luis, and M. Aitor, "Interactive visualization of volumetric data with WebGL in real-time," in *The 2011 Web3D ACM Conference*, 2011.
[144] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 1(2), pp. 99-108, 1995.
[145] S. Bruckner and M. E. Groller, "Instant Volume Visualization using Maximum Intensity Difference Accumulation," *Computer Graphics Forum,* vol. 28(3), pp. 775-782, 2009.
[146] B. R. Haxel, M. Goetz, R. Kiesslich, and J. Gosepath, "Confocal endomicroscopy: a novel application for imaging of oral and oropharyngeal mucosa in human," *Eur Arch Otorhinolaryngol,* vol. 267(7), pp. 443-448, 2010.