GPU ASSISTED VOLUME RENDERING WITH OptiX

Student Name: Sambhav Satija
Roll Number: 2013085

BTP report submitted in partial fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Engineering
on 17th November, 2016

BTP Track: Research

BTP Advisor
Dr. Ojaswa Sharma (Thesis Advisor)
Dr. Viswnath Gunturi (Internal Examiner)
Dr. Ganga Bahubalindruni(Internal Examiner)

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

I hereby declare that the work presented in the report entitled **"GPU assisted volume rendering with OptiX"** submitted by me for the partial fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Engineering* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of my work carried out under guidance of **Dr. Ojaswa Sharma**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

.... **New Delhi, 17th November 2016**

Sambhav Satija

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

.... **New Delhi, ... November 2016**

Dr. Ojaswa Sharma

Abstract

Algorithms for rendering volumes (which evaluate the Volume Rendering Integral to different extents), can be be transformed into their parallel versions with varying improvements in performance. GPUs (Graphics Processing Units) have improved a lot in the past few years, however rendering systems still need to be designed in a way so as to utilise GPUs efficiently. In this project, we explore the possibility of using Nvidia® OptiX™ , a ray tracing framework by Nvidia to architect a fast volume rendering system.

Keywords: Computer Graphics, Volume rendering, GPU, Nvidia OptiX

Acknowledgments

I would like to thank Dr. Ojaswa Sharma for being a tremendously patient advisor. This project, until now, has been an incredible learning experience, which was possible only because of his guidance. I am grateful to my family and friends for all their love and support.

Work Distribution

This is an independent project.

Contents

Chapter 1

Introduction

1.1 Problem description

There are a lot of sources of volumetric data (3D scalar fields), typically medical imaging diagnostic techniques such as CT/MRI scans or abstract mathematical data such as 3D probability distributions. There is a growing need to render these volumes beautifully and in realtime, which to a more enriching experience for the viewer while trying to comprehend the information. GPUs are extremely good at solving parallel algorithms, with limitations on how the algorithms can be designed for improving performance. We'll be using the raytracing middleware Nvidia® OptiX™ to build a volume rendering system on top of it.

1.2 Current solution

Since GPU designers have opened platforms like CUDA® for using GPUs as general purpose computing devices (termed GPGPUs), it has been extremely easy to port naive parallel algorithms. Nvidia® OptiX™ provides the bare minimal framework for building raytracing solutions, which we'll be selectively using to push the computation for volume rendering to the GPU.

1.3 Organisation of this report

This report includes the work done in the first semester of this project. The goal was to use Nvidia® OptiX™ to build a functional, dynamic and real time volume rendering system. The motiviation behind choosing Nvidia® OptiX™ as the framework is covered in chapter ,an analysis on Nvidia® OptiX™ is done in and the system built is described in chapters

Chapter 2

Literature Prerequisites

2.1 Basic Terminology

Volumetric data is usually in the form of 3D scalar fields, which can be extracted from medical imaging devices, 3D distribution functions and computational fluid dymanics. A volume data can be extracted from any signal f where

$$f(\boldsymbol{x}) \in \mathbb{R} \; and \; \boldsymbol{x} \in \mathbb{R}^3$$

Readily available volume data is usually in the form of a dump of intensity value at each voxel in the volume. A voxel is the basic single unit of the representation of 3D space, analogous to a pixel in 2D space. Each intensity value can be thought of as belonging to different isosurfaces in the volume.

Direct Volume Rendering (DVR) methods, instead of extracting isosurface information, render the data by evaluating an optical model based on how the raw volume emits, scatters and absorbs light. In DVR, each intensity value is thus mapped to different colour and alpha values. This mapping function is typically termed as a **transfer function**, which can be a one-to-one mapping, a linear interpolation of colours and alpha values at control points, or any function in general.

$$transfer(\boldsymbol{i}) \in \mathbb{R}^4 \; and \; \boldsymbol{i} \in isovalues \; in \; dataset$$

Image order algorithms render the data by iterating over the pixels in the image, instead of the voxels or isosurfaces in the volume data.

2.2 Raymarching

One of the widely used volume rendering techniques is the Raymarching method. Raymarching is an image-order algorithm which renders the image one pixel at a time and employs DVR. Raymarching emits rays for every pixel on the screen from the camera.

Each ray is denoted by $\boldsymbol{r_{xy}}$ for every $\boldsymbol{x}$, $\boldsymbol{y}$ in the screen. The ray $\boldsymbol{r_{xy}}$ is parameterised by $\boldsymbol{t}$, where $\boldsymbol{t}$ is the distance of the point from the camera. A point $\boldsymbol{x}$ on $\boldsymbol{r_{xy}}$ is given by

$$\boldsymbol{x = camera + t \cdot dir(r_{xy})}$$

The ray $\boldsymbol{r_{xy}}$ traverses through the volume data, updating its colour as it passes through different intensities and finally sets the pixel $\boldsymbol{x}$, $\boldsymbol{y}$ to the composited colour. The ray $\boldsymbol{r_{xy}}$ traverses the volume by stepping through the it with a fixed **stepping distance**, $\boldsymbol{\Delta t}$. A lookup on the volume data returns the intensity at that point $\boldsymbol{x(t)}$. A transfer function maps this intensity value to a colour and alpha value.

This method of accumulation of colour using the alpha values is termed as **alpha blending** and be solved using 2 approaches:

- **Front to back:** In this technique, the ray is traversed in the front to back manner; with $\boldsymbol{i} : (n-1) \to 0$. $\boldsymbol{C'_0}$ is the final colour of the pixel.

$$C'_i = C_i + (1 - A_i)C'_{i+1} \quad where\ C'_n = 0$$

- **Back to front:** In this technique, the ray is traversed in the back to front direction; with $\boldsymbol{i} : 1 \to n$. $\boldsymbol{C'_n}$ is the final colour of the pixel.

$$C'_i = C'_{i-1} + (1 - A'_{i-1}) \cdot C_i \quad where\ C'_0 = 0$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1}) \cdot A_i \quad where\ A'_0 = 0$$

In both the above methods, $\boldsymbol{n = \frac{exit(r_{xy}) - enter(r_{xy})}{\Delta t}}$, $\boldsymbol{enter(r_{xy})}$ returns the point $\boldsymbol{x}$, the point where $\boldsymbol{r_{xy}}$ enters the volume and $\boldsymbol{exit(r_{xy})}$ returns the point $\boldsymbol{x}$, the point where $\boldsymbol{r_{xy}}$ exits the volume.

$\boldsymbol{C_i}$ and $\boldsymbol{A_i}$ are the colour and alpha values returned after two texture lookups. The first lookup returns the intensity value at point $\boldsymbol{i}$ on $\boldsymbol{r_{xy}}$. The second lookup gives the colour and alpha for this intensity value.

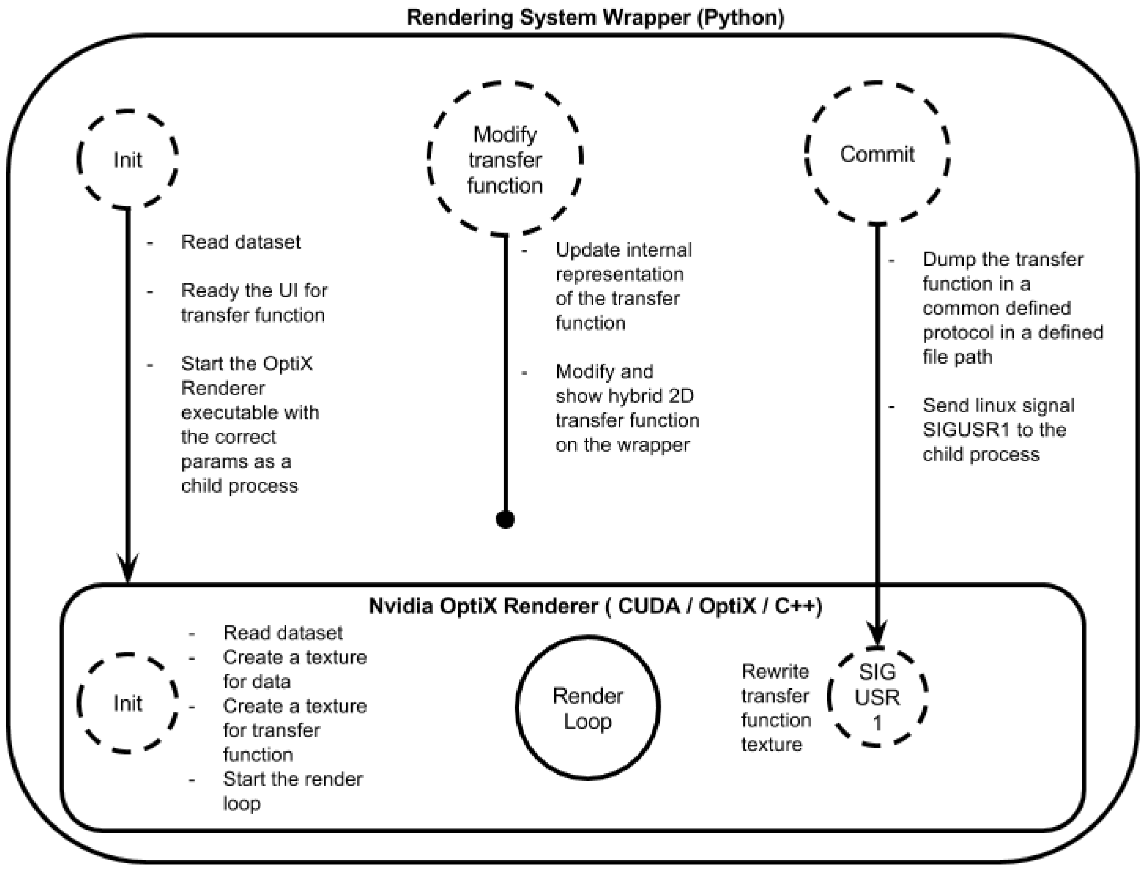
Front to back compositing has a major optimisation that it can terminate early, as soon as the alpha crosses 1. However, this usually leads to decreased performance in GPUs due to processors in a warp running in lockstep. So back to front compositing is usually implemented.

Chapter 3

System Overview

This chapter is split into 3 section. The first section gives a broad overview of the architecture. The second section gives a walkthrough of the system. The third section gives an overview on how the project evolved.

3.1 Architecture overview



Rendering System Wrapper (Python)
Init
- Read dataset
- Ready the UI for transfer function
- Start the OptiX Renderer executable with the correct params as a child process
Modify transfer function
- Update internal representation of the transfer function
- Modify and show hybrid 2D transfer function on the wrapper
Commit
- Dump the transfer function in a common defined protocol in a defined file path
- Send linux signal SIGUSR1 to the child process
Nvidia OptiX Renderer (CUDA / OptiX / C++)
Init
- Read dataset
- Create a texture for data
- Create a texture for transfer function
- Start the render loop
Render Loop
Rewrite transfer function texture
SIG USR 1

Figure 3.1: System Architecture.

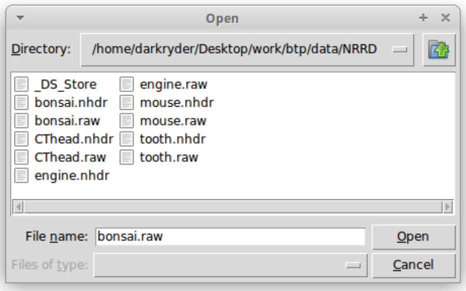
Figure 3.1 shows the architecture of the system designed. The volume rendering computation is done in an binary executable written in C++ on top of the Nvidia® OptiX™ platform. A Python wrapper is written for convenience which allows the user to dynamically create and update the transfer functions.

This transfer function can be pushed to the executable dynamically. The rendering executable maps the transfer function texture to the device at runtime again, showing immediate results. The rendering binary interops between OpenGL to display the rendered buffer on-screen.

3.2 Working

This section describes a sample workflow of the program.

1. Starting the python script starts the wrapper program, which asks for the volume data file 3.2. The current project can understand NRHD data file formats.



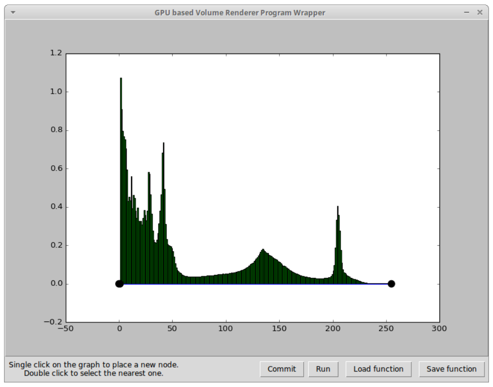
Directory: /home/darkryder/Desktop/work/btp/data/NRRD

File name: bonsai.raw

Figure 3.2: Prompt for datafile.

2. The wrapper shows a transfer function editing screen 3.3 that allows the user to modify the transfer function.

   A histogram of the isovalues in the dataset is the background of the transfer function editor. This can act as a help to the user for selecting good control points.



Single click on the graph to place a new node.
Double click to select the nearest one.

Figure 3.3: Transfer function editor.

3. Clicking anywhere on the graph editor adds a control point to that isovalue/alpha graph. The color for that control node can be selected from the prompt 3.4.

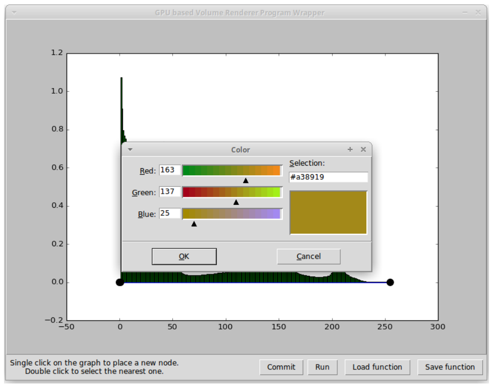

GPU based Volume Renderer Program Wrapper
Color
Red: 163
Green: 137
Blue: 25
Selection:
#a38919
OK
Cancel
Single click on the graph to place a new node.
Double click to select the nearest one.
Commit
Run
Load function
Save function

Figure 3.4: Selecting a color for control node.

4. This allows creating simple transfer functions extremely fast. The transfer function does linear interpolation for the alpha value and color for any isovalue for which the values are not explicitly defined. 3.5.

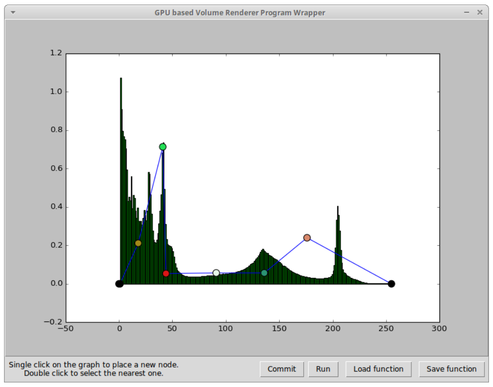

GPU based Volume Renderer Program Wrapper
Single click on the graph to place a new node.
Double click to select the nearest one.
Commit
Run
Load function
Save function

Figure 3.5: Fast creation of control points in nodes.

5. Clicking on **Run** starts the rendering executable as a child process of the wrapper with the corresponding required arguments. 3.6.

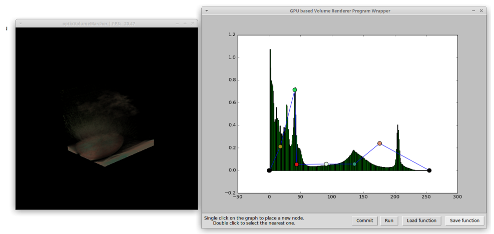
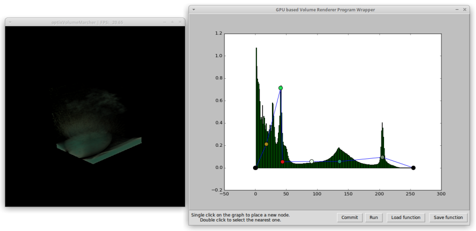


Figure 3.6: Booting up the rendering system.

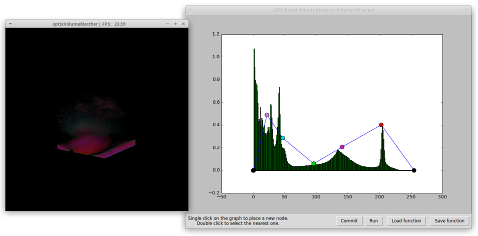
6. Already added nodes can be intuitively modified/deleted by double clicking and moving them around in the editor. Simply clicking the **Commit** button will send this modified transfer function to the rendering process.
   The rendering executable dynamically reloads the transfer function texture to the device and the new results are instantly visible. 3.7



Single click on the graph to place a new node.
Double click to select the nearest one.

Figure 3.7: Instant update of transfer function.

7. Convenience features like loading and saving transfer functions are also present 3.8.



Single click on the graph to place a new node.
Double click to select the nearest one.

Figure 3.8: A transfer function loaded from a previous session and auto reloaded into the executable.

3.3 Progress Report

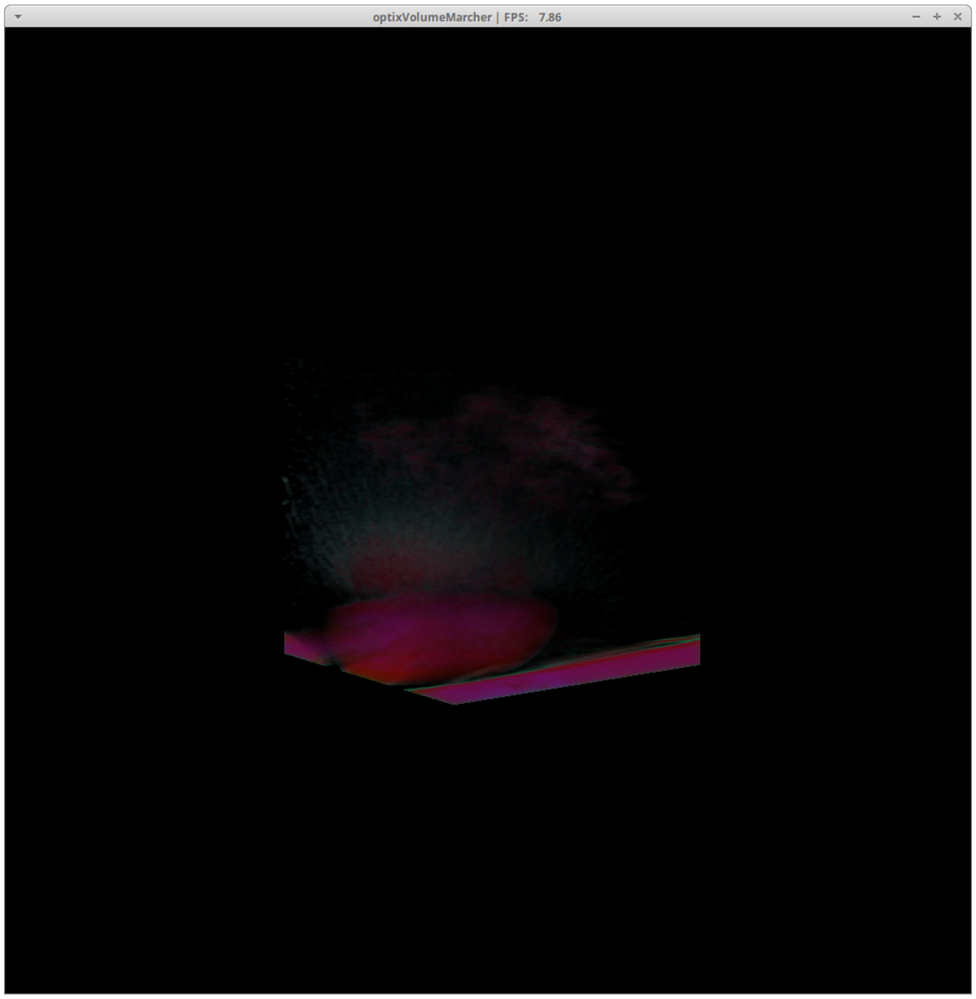


Figure 3.9: Scene from rendering executable.

Chapter 4

Discussion

Chapter 5

Appendix: Evaluating Nvidia® OptiX™