University of Alberta

Interactive Simulation and Visualization of Complex Physics Problems Using the GPU

by

Cailu Zhao

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring, 2011

Edmonton, Alberta

Examining Committee

Pierre Boulanger, Computing Science

Herb Yang, Computing Science

Bruce Cockburn, Electrical & Computer Engineering

Abstract

Physical simulations are in general very computationally intensive and required large and costly computing resources. Most of those simulations are rarely interactive as the link between visualization, interaction, and simulation is too slow. The recent development of parallel Graphic Processing Unit (GPU) on graphic cards has enabled us to develop real-time interactive simulators of complex physical phenomenon. In this thesis, two GPU-based implementations of interactive physical simulations are presented: (1) visualization of the electron probability distribution of a hydrogen atom, (2) visualization and simulation of particle based fluid dynamic model using smoothed particle hydrodynamics. These simulations were developed in the context of the Microscopic and Subatomic Visualization (MASAV) project as a demonstration of the capabilities of the GPU to create realistic interactive physical simulators for scientific education.

Table of Contents

List of Tables

List of Figures

CHAPTER 1 -

Introduction

The main goal of the Microscopic and Subatomic Visualization (MASAV) project [Pinfold, 2008] is to visualize the fundamental quantum nature of matter at the birth of the universe during the Big Bang, similar conditions will be recreated and studied with the highest energy accelerator in the world - the Large Hadrons Collider (LHC) or "Big Bang Machine" - at the European Centre for particle physics research (CERN) situated near Geneva, Switzerland (see Figure 1.1).

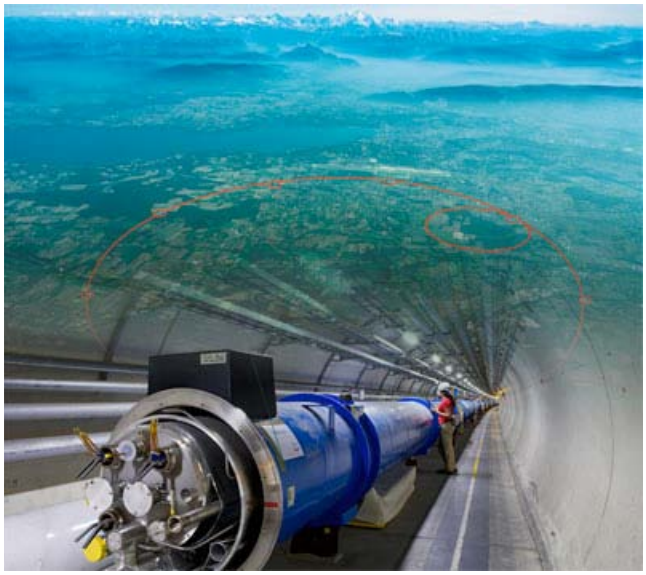


Figure 1.1: The LHC: with circumference radius of 27km buried 100m underneath the French-Swiss countryside.

This thesis is a pilot project and is one of the first contributions to the MASAV project. The MASAV project is very large but this pilot project focuses on creating a real-time interface to help illustrate the basic aims of the MASAV project, i.e. to reveal a

glimpse of the microcosm as envisaged by the latest theoretical and experimental efforts at CERN. In order to do so a basic toolbox was developed to render and simulate in real-time the atomic structures of the hydrogen and the interaction of molecules in a fluid.

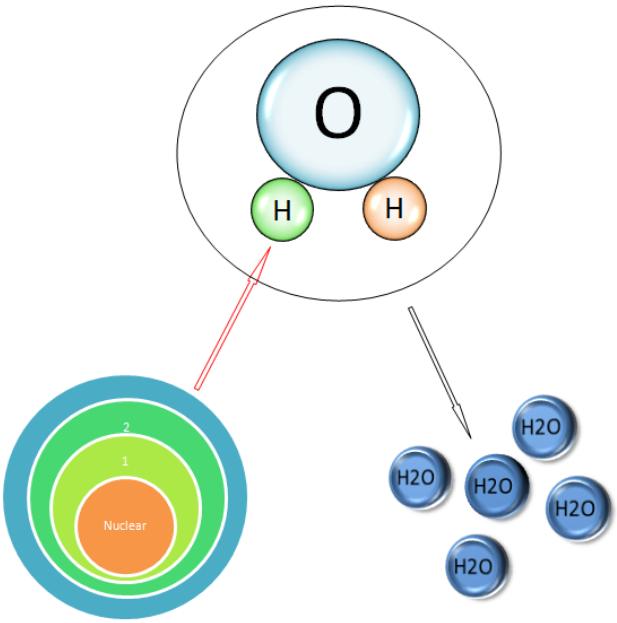


Figure 1.2: Project Structure: Electron Cloud, Hydrogen Molecule and Fluid Particles

This thesis consists of two major parts: atomic electron probability distribution visualization and molecular fluid flow simulation, as shown at Figure 1.2. The first part starts with the advanced real-time visualization of the electron probability distribution of the hydrogen atom in various energy states. In order to do so, Chapter 2 and Chapter 3 introduce two new rendering algorithms for electron distribution visualization: a point-based Radiosity algorithm using a Monte Carlo diffusion model where the electron distribution is lit as if it was part of a nebula where the nebula center or star is the atomic nucleus. The second project uses the lighting model produced by the Nebula Visualization (NV) to pre-render the electron probability distribution using a GPU

implementation of a volume rendering algorithm based on ray casting. Using this combination of volume rendering and pre-calculated inner radiosity, we will demonstrate that it is possible to observe in more detail the structure of the electron probability distribution. In Chapter 4, we journey out of the atomic level to consider the fluid flow of a group of small molecules, with a GPU implementation of a Smoothed Particle Hydrodynamics (SPH) simulation. SPH simulates the attraction and repulsion forces between fluid molecules and their motion.

Instead of reviewing this pertinent literature in one large literature review chapter, we present this information in each individual chapter.

1.1 Real-time Implementation Using CUDA

This thesis describes the implementations both for CPU (Central Processing Unit) and GPU (Graphics Processing Unit) of the various elements of the MASAV toolbox. Real-time implementation of these algorithms is now made possible with the use of GPUs' computing capability with thousands of thread computing elements and very fast memory that surpass by orders of magnitude the capability of the standard CPU. In order to program the GPU, we decided to choose CUDA (Computer Unified Device Architecture), one of the most popular and newest programming models for GPU programming. As most parallel algorithm implementations are specific to the hardware architecture, let's review a general programming framework for CUDA and let's review the basic building block of the GPU hardware.

1.1.1 Basic GPU Architecture

With the increasing requirement for large dataset computation and visualization, the modern GPU has recently evolved, providing multiple Teraflops at very low cost. Two dominant computing architectures exist in parallel computing today: Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD). Most MIMD architectures have to handle different instruction sequences and procedures, and is at the base of the modern CPU. On the other hand, SIMD architecture has been used for data intensive applications such as graphical rendering, image pixels computation, point-cloud processing, and etc. As an example of a highly efficient SIMD parallel model, CUDA was introduced by NVIDIA Corporation to abstract the real GPU hardware into a SIMD like architecture that can compute more than 60,000 data element in parallel on a high-end GPU [NVIDIA, 2007]. During the computation, each GPU program executes the same instruction sequence simultaneously on different index of data element. This mode of computation is ideal for graphics and scientific computing and for the MASAV project. For example, in the fluid simulation in Chapter 4, all the particles exert forces on other neighbouring particles at the same time, which is the natural fundamental phenomenon in real physical world and is ideal for GPU implementation. In the following sections let's review the building blocks of CUDA.

1.1.2 Grid of Thread Blocks

In CUDA, each kernel function calling the GPUs creates a grid of thread blocks, with a limited number of threads in a block. A thread is a small piece of code running on a single core. In fact a GPU is not really SIMD architecture but, more realistically, a Single Thread Multiple Data (STMD) architecture where a basic processing sequence called a

thread, is processing in parallel multiple data. This model allows a larger number of threads to execute the same program (called kernel) with one invocation. The blocks are identifiable via block ID (one- or two-dimensional arrays), which leads to a reduction in thread cooperation.

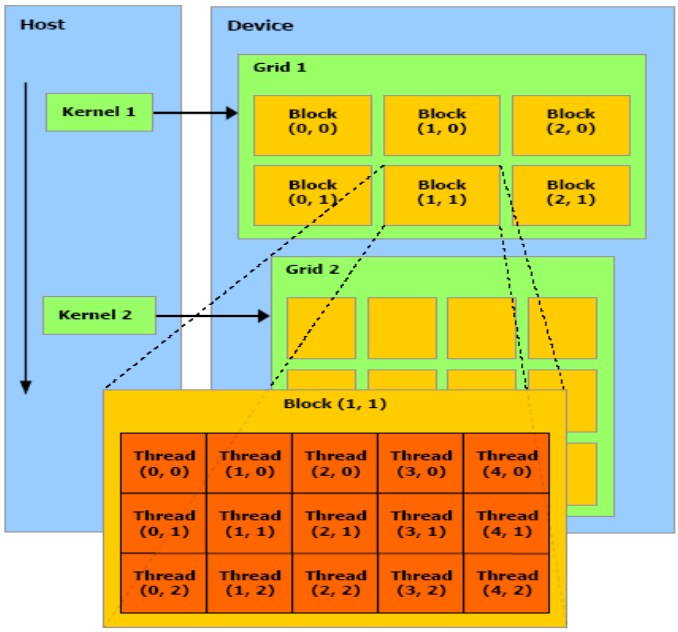


Figure 1.3: Tread Processing in CUDA [NVIDIA, 2007]

Figure 1.3 shows a CUDA program containing two kernel functions running on a group of blocks called a Grid. In Kernel-1, Grid-1 is designed as two-dimensional array in both grid size and block size. So every block in the grid has a unique two-dimensional index and it is the same for every thread in a specific block.

1.1.3 Memory in CUDA

There are three different levels of memory in a GPU, as shown at Figure 1.4:

- *Global-level Memories* consists of Global Memory, Constant Memory and Texture Memory. All global-level memories could be read from and written to by

the Host (CPU Program), and kept constant during kernel launches by the same application.

- *Block-level Memory or Shared Memory* is designed to be shared by all the threads in the same block during an application. As on-chip memory, shared memory is much faster than global-level memories, and in some extreme cases when all threads in only one block, the speed of shared memory fetching could even match that of accessing registers.
- *Thread-level Memories* like registers and local memory are on-chip memories and have the best read and write speed for threads, on which the kernel functions are really running.

The global memory space is not cached but its addresses could be simultaneously accessed by every thread of a kernel function. Compared to other memories in the device, global memory has the lowest read/write speed. Constant memory space is cached; therefore, a read from constant memory costs one memory read only on a cache miss. For all the threads of a kernel, reading from the constant cache is as fast as reading from a register as long as all the threads read the same address. The cost increases linearly with the rise of the number of different addresses read by all threads. The texture memory space is also cached; hence a texture fetch costs one memory read from device memory only on a cache miss. The texture cache is optimized for 2D spatial locality, so threads of the same kernel that read texture addresses that are close together will achieve the best performance. By contrast, as in the on-chip memory, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads within one SIMD group or "warp", accessing the shared memory is as fast as accessing a register, as long

as there are no bank conflicts between the threads -- shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously and yields *n* times higher bandwidth where *n* indicates the number of distinct memory addresses in a shared memory [NVIDIA, 2007].

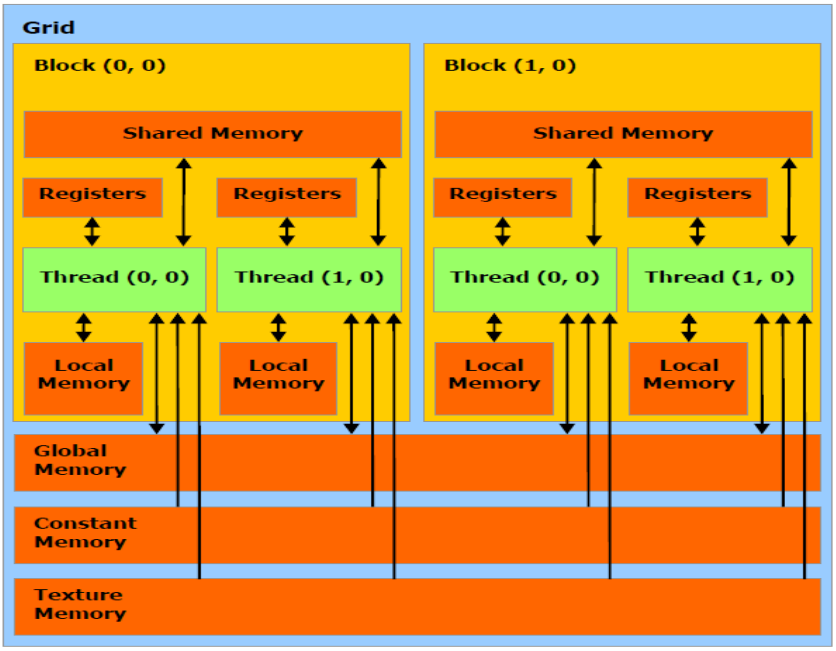


Figure 1.4: Memory Model in CUDA Framework [NVIDIA, 2007]

1.1.4 Grid of Thread Blocks

In CUDA, each kernel function calling the GPUs creates a grid of thread blocks, with limited number of threads in a block. This model allows larger number of threads to execute the same kernel with one invocation. The blocks are identifiable via block ID (one- or two-dimensional arrays), which leads to a reduction in thread cooperation.

1.1.5 Basic Development Environment

A typical development environment for CUDA programming on Windows consists of Visual Studio and CUDA Software Development Kit (SDK). The CUDA programming has always been a challenging and time-consuming task, not only because of the settings of environmental parameters with multiple compilers from both C++ and CUDA, but also because of the lack of debugging tools until August, 2010 when NSIGHT (also called Nexus) was officially released by NVIDIA. In general, CUDA programmers prefer to write and debug the code in C++ first and then adapt into CUDA line by line. Visual Studio serves only in the host part, as the programmers have no access to variable values on GPU device.

Recent development at NVIDIA for the CUDA community is the release of the industry's first development environment NSIGHT for GPU programming and debugging. This powerful plug-in allows programmers to develop for both GPUs and CPUs within Microsoft Visual Studio environment. With the help of NSIGHT, one can debug computing kernels directly on the GPU hardware, examine thousands of threads executing in parallel using the familiar Locals, Watch, Memory and Breakpoints functions in Visual Studio. One can view GPU memory directly using the standard memory windows in Visual Studio. Conditional breakpoints can be used to quickly identify and correct errors in massively parallel code and identify memory access violations using the CUDA C/C++ Memory Checker. For more information please see the pertinent literature at: http://www.nvidia.com .

All the CUDA programs for this thesis have been developed with manual debugging before NSIGHT.

1.2 Thesis Contributions

The first part of the thesis focuses on the atomic structure of the hydrogen atom. In Chapter 2, using a physical analogy between a nebula system and the electron cloud of a hydrogen atom, we developed a new lighting algorithm by introducing a Monte Carlo diffusion model used in astrophysics research for an electron cloud model. The algorithm was implemented in both CPU and GPU and a detailed comparison and analysis in terms of speed performance was performed. Speed-up of two orders of magnitude was achieved.

In order to present more detail information of an electron clouds structure, especially near low probability areas, a GPU ray-casting-based volume rendering algorithm was added to the result of the point-based radiosity calculation to allow real-time visualization of the electron cloud distribution. By acting on the control bar of the interface, the user can compare the basic distribution (derived from the results of nebula radiosity calculation) and the interaction of light with the probability density create by the volume rendering in order to get a deep understanding of the inner structure inside electron cloud at different energy states. Analysis on the experimental results and speed performance is given at the end of that chapter. In this case as well real-time performance was achieved.

The thesis then journeyed at the molecular interaction level where we implemented a Smoothed Particle Hydrodynamics (SPH) algorithm. SPH simulates the motion of molecules controlled by attractive and repulsive forces between all fluid molecules and the external forces applied to them. We have improved the precision of the SPH implementation by a classical 4^{th} order Runge Kutta integration method without any

loss of performance for the GPU. We also found the proper adjustment of different forces' parameters which enables SPH to simulate not only the motion of water-like fluid, but as well the explosion of thin fluid such as gas, and even the motion of thick heavy fluid such as wet mud. Speed-up in the order of 20 to 30 times was achieved.

CHAPTER 2 -

Visualization of the Hydrogen Atom Electron Distribution Using a Nebula Metaphor

In this chapter, we represent a method for rendering the electron probability distribution of a hydrogen atom using a Nebula visualization model [Magnor, 2005] based on a Monte-Carlo algorithm. The basic idea lies in the similarity between the hydrogen atom electron probability distribution and its astronomical counterpart, a nebula where the interstellar dust is similar from a graphic viewpoint to the hydrogen atom electron density probability. We will demonstrate that such a visualization technique will allow us to better understand how the hydrogen atom electron probability distribution evolves with its energy state. As the first element of our contribution to the MASAV toolbox, point-based technique is used as it is much easier to narrow down the huge gap between macrophysics of astronomy and our atomic model. The following sections introduce the background knowledge for nebula cloud simulation: Section 2.1 describes the reason for choosing a reflection nebula visualization algorithm rather than any other methods. Section 2.2 introduces the necessary background knowledge both in atomic physics and astronomy. Section 2.3 describes in detail the reflection nebula visualization algorithm. Section 2.4 describes the GPU implementation of the proposed rendering algorithm. Section 2.5 presents the experimental results and compares the GPU implementation with its CPU version.

2.1 Nebula vs. Electron Probability Distribution

A good visualization model is necessary in order to convey detailed information to the users of the atomic structure and its properties for various quantum energy levels. As we

decided to render an electron density distribution like a cloud, there are many modeling techniques and rendering methods to choose from. Some widely used mature lighting techniques include global illumination with photons [Jensen, 1996] [Christensen, 1997], various volume rendering algorithms such as: ray–casting [Amanatides and Fournier, 1984], texture mapping [Weinhaus and Devarjan, 1997], or splatting [Zwicker *et al.*, 2001], and finally radiosity [Durand *et al*, 1999] algorithms from which our proposed algorithm is borrowed. Widely used for flat planes rendering such as polygon, global illumination algorithms are not really able to deal with large point models as they tend to gives poor visualization result and miss use of the rendering pipeline as each point must be converted into spheres raising the polygon count by a factor of at least 8. It is really difficult to determine the size and normal of every particle in the system in order to perform a Phong shading model. Volume rendering algorithms, although perfect for observing volume data from the outside, are not directly applicable to our problem as we would like to have the lighting source to come from the inside, as if the electron probability density were lit by the atomic nucleus revealing the electron probability distribution from the inside-out. In many ways volume rendering techniques can be applied to our problem if one can compute a lighting model from the inside-out using the proposed radiosity technique.

In Section 2.2, we will demonstrate the similarity between the two models, and explain the reason why one would prefer to use reflection nebula visualization model for electron probability distribution rendering. Each reflection nebula system contains at least one star and the surrounding interstellar dust. This interstellar dust does not produce light, but reflects, refracts and absorbs light energy while photons traverse the region. All

photons come from one single source, the star in a one-star reflection nebula system. The intensity of the light fades away from the center to the boundary after so many reflections, refractions, and absorptions revealing the internal distribution of the cloud. Similarly, when we try to find a meaningful method for rendering an electron probability density, one can represent thousands of copies of a single electron traversing space according to the probability distribution and its energy, one can view those models as if the electrons would create a smoke or dust around the nucleus, called for now on an “electron cloud”. A well-known lighting model that has been successfully implemented for dust or smoke can be equally useful to visualize an electron probability density distribution. Although an electron cloud is not actually a cloud which contains millions of electrons, but only the probability of the electron to be presented at that position, where the local electron probability density represent one (or many) electrons’ possible appearance.

Another consideration is the advantage of a reflection nebula model in the placement of the light source. In such a nebula system, the light is from the single star (or the sum of multiple outcomes of many of them, if there is more than one star). For a single–star system, the sun is always placed in the middle, which gives enough information to present a better display of the inner-structure to an observer. A nucleus-electrons system could be considered as single-star system. This solves the problem of normal rendering paradigm which assumes the placement of a light source outside the cloud and would hide the true distribution of the probability density. In this respect, the lighting method of a nebula system matches perfectly our needs.

2.1.1 Physics & Astronomy Background

2.1.1.1 Atom Nucleus & Electron Probability Density

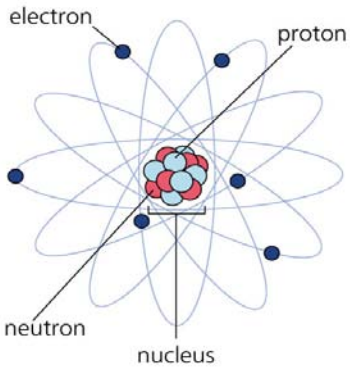


Figure 2.1: Planetary Model (Bohr Model) of the Atomic Structure [Milstid, 2010].

An atom consists of a tiny dense core in the center and some electrons outside the core moving around it. Figure 2.1 shows an inexact image of an atomic structure, called the Bohr model [Sapirstein, 1995]. At the center, the small core, called nucleus, corresponds to most of the mass of the whole atom as electrons are very light. There are two kinds of particles in the nucleus: proton and neutrons, where the proton is charged and its mass is slightly larger than the neutron that is not charged. In different atoms, the number of protons and neutrons can change but, in general, the same element will always have the same number of proton but not necessarily the same number of neutron. In this case those elements are called *isotope*. The proton attracts the electrons by Coulombian law and the proton and neutron are attracted by strong forces primarily and by weak forces when they are very close to each other. Further subdivision of the proton and neutron bring up to the quarks which states define a neutron from proton in family call leptons. In this case the quarks are define by their internal state (up/down, charm/strange, top/bottom) are they are attracted to each other by a Quantum Chromo-dynamic Field called Gluons which is responsible for the strong force.

Ultimately the MASAV project would be able to visualize the atom all the way to the quarks but for this pilot project, we will limit ourselves to a quantum probability analysis of the electron.

2.1.1.2 Electron Wave Function

A wave function, in quantum mechanics, is a function from space to complex numbers describing the motion of sub-atomic particles in an electromagnetic field. The modulus of the wave function is equal to the probability that the particle is at a specific position in space. The hydrogen atom wave function is the wave function in the case of one single electron been attracted by a single proton in three-dimensional space and is expressed in a spherical coordinate system by:

$$\psi_{n,l,m}(r,\theta,\varphi) \tag{2.1}$$

where r represents the distance of the electron from the nucleus, and θ and φ are two angles in polar coordinates system. A probability density function representing the probability of an electron to be located at a distance r and at an angle θ and φ is given and at a given quantum state *(n, l, m)* is: $\left|\psi_{n,l,m}\right|^2$. Because the probability density of each possible state should integrate over all space to 1, the norm of the wave function must be 1:

$$\int_{\mathrm{v}} \psi_{n,l,m}|r,\theta,\varphi|^2 \,\mathrm{d}v = 1 \tag{2.2}$$

- **State**

  The term “state” is used to describe the total energy of the electron. The shape of the probability density of the electron differs drastically depending on its quantized

energy state. Generally speaking, a lower energy always leads to a more compact probability distribution that looks like orbit in a planetary system. The number of possible energy states also differs from one atom to another. Smaller and lighter-weighted atoms have fewer possible quantized states than large heavy atoms.

- **n, l, m, s**

For the same atom, each state of electron has its unique quantum numbers *n, l, m* and *s*. These four parameters represent the principal, azimuthal, magnetic and spin quantum numbers of an atomic orbital.

The principal quantum number *n* represents the total energy of each orbital and increases as the electron is more distant from the nucleus increases. The sets of orbitals with the same *n*-value are often referred to as electron shells or energy levels.

The azimuthal (or angular momentum quantum number) *l* determines the angular momentum energy of different sub-shells or sub-orbits on the same shell. In each orbital determined by the principal quantum number *n*, the possible number of sub-orbitals *l* ranges from 0 to *n*-1.

The magnetic quantum number *m* refers to the direction of the angular momentum vector as the electron rotates around the nucleus. The magnetic quantum number *m* does not affect the electron's energy, but it does affect its probability distribution in space. The value of *m* could be chosen as any integer from between –*l* to *l*.

As the fourth quantum number, the spin quantum number *s* determines the angular momentum as it rorates around on itself. Since spin does not affect the position of the electron, this parameter is not useful in our point-based model.

At Table 2.1 one can see examples of possible values of the principal, angular and magnetic quantum numbers at different energy states, from state 1 to state 10.

Table2.1: Quantum numbers for ten energy states.

State	1	2	3	4	5	6	7	8	9	10
N	1	2	2	2	3	3	3	3	3	3
L	0	0	1	1	0	1	1	2	2	2
M	0	0	0	1	0	0	1	0	1	2

- **Bohr Radius**

As mentioned previously, the simplest atom, hydrogen, has a single electron orbiting the nucleus and its smallest possible orbit, with lowest energy, is called the Bohr radius and is the most likely position of the electron when the atom is at its lowest energy state. As the radius of the basic state, the Bohr radius of hydrogen has a value of $5.2917720859 \times 10^{-11}$ m.

- **Polar & Cartesian Coordinates**

In Polar Coordinates, positions in 3D space are determined by their own specific r, θ and φ values. Here r represents the distance from the center, and θ and φ are two angles in polar coordinates system. Accordingly, in the Cartesian Coordinates system which use x, y and z coordinates to represent the position of a particle, a conversion between its x, y and z value and their counterparts in polar system are defined by:

$$\begin{cases} x = r\,sin\theta cos\varphi, \\ y = r\,sin\theta\,sin\varphi, \quad 0 \le \theta \le \pi, \ 0 \le \varphi \le 2\pi \\ z = r\,cos\varphi, \end{cases} \tag{2.3}$$

One can see at Table 2.2 various hydrogen electron wave functions for every energy state from *n,l,m* (1, 0, 0) to *n,l,m* (3, 2, 2). For these wave functions, the Bohr radius of the fundamental state is denoted as a_0, and any other state *n* is equal to: $a_n = a_0 n^2$.

Table 2.2: Electron wave functions for some basic energy states

n	l	m	$\psi_{nlm}(r,\theta,\varphi)$
1	0	0	$\frac{1}{\sqrt{\pi}}\alpha_1^{3/2}e^{-\alpha_1 r}$
2	0	0	$\frac{1}{\sqrt{\pi}}\alpha_2^{3/2}(1-\alpha_2 r)e^{-\alpha_2 r}$
2	1	0	$\frac{1}{\sqrt{\pi}}\alpha_2^{5/2}re^{-\alpha_2 r}\cos\theta$
2	1	1	$\frac{1}{\sqrt{2\pi}}\alpha_2^{5/2}re^{-\alpha_2 r}\sin\theta e^{i\varphi}$
3	0	0	$\frac{1}{3\sqrt{\pi}}\alpha_3^{3/2}(3-6\alpha_3 r+2\alpha_3^2 r^2)e^{-\alpha_3 r}$
3	1	0	$\frac{2}{\sqrt{3\pi}}\alpha_3^{5/2}(2-\alpha_3 r)re^{-\alpha_3 r}\cos\theta$
3	1	1	$\frac{1}{\sqrt{3\pi}}\alpha_3^{5/2}(2-\alpha_3 r)re^{-\alpha_3 r}\sin\theta e^{i\varphi}$
3	2	0	$\frac{1}{3\sqrt{2\pi}}\alpha_3^{7/2}r^2e^{-\alpha_3 r}(3\cos^2\theta-1)$
3	2	1	$\frac{1}{\sqrt{3\pi}}\alpha_3^{7/2}r^2e^{-\alpha_3 r}\sin\theta\cos\theta e^{i\varphi}$
3	2	2	$\frac{1}{2\sqrt{3\pi}}\alpha_3^{7/2}r^2e^{-\alpha_3 r}\sin^2\theta e^{i2\varphi}$

2.1.1.3 Reflection Nebula & Interstellar Dust

In astronomical research, a reflection nebula is a cloud of gas or dust in the region near one or multiple central stars. The cloud of gas cannot be seen until it scatters and absorbs photon energy from the central star(s). The amount of light strength reaching the viewer depends on local gas density all the way from the star to the viewer. In some region within the boundary of reflection nebula, because of the high density of dust, all light is absorbed, and so caused one type of reflection nebula called a *dark nebula*. In other cases,

a portion of light, after scattering and absorbing, could reach the viewers' position through the gas.

The term *interstellar dust* refers to the low density chemical smoke-like materials between the stars. Compared to the density of air in the Earth's atmosphere, the density of interstellar dust is extraordinary low, experiment shows that if the density of interstellar dust in the space if condensed as thick as Earth's air, all light emitted by stars other than the sun will be totally blocked from reaching us [J. Binney and M. Merrifield, 1998].

2.1.2 Related Work of Reflection Nebula Visualization

Henyey and Greenstein [1938] provided the first fundamental formula for the computation for brightness and color of reflection nebula. Later astronomy researchers used this model into their own work such as A. Witt and G. Walker [1982] who used this formulation into their own phase function to simulate interstellar grains, K. Sellgren and M. Werner [1992] derived their model to compute infrared radiation produced by dust, S. Gibson and K. Nordsieck [2003] and K. Gordon [2004] cited this work to explain and analyze the dust property and scattering geometry of reflection nebula.

Nadeau *et al*. [2001] was able to animate and visualize a nebula system from a large amount of observational data. With volume rendering, Magnor [2004] rebuild a planetary nebula with constrained amount of input data of emissive volume to simulate ionized gas, which consider the local volume to emit light. Magnor [2005] published a new method to consider accurately that a reflection nebula changes the color of light due to wavelength-dependent scattering and extinction properties of interstellar dust. In addition, Magnor introduced a Monte-Carlo simulation into the visualization model to

simulation multiple scattering within the local cubic subspace, which was very efficient in terms of computational power.

In the following section, we will describe a visualization algorithm based on the simulation of a reflection nebula system (but for an electron cloud instead of interstellar dust) where the central star is the nucleus emitting a monochromatic light in all directions.

2.2 Algorithm Overview

The following are the three major parts of the proposed visualization algorithm:

- Particle cloud creation using probability density distribution
- Pre-calculation based on Monte-Carlo simulation
- Visualization of the electron cloud using a GPU

As shown in Figure 2.2, after choosing an energy state, a point cloud is first generated into a quantized space around the hydrogen nucleus, according to the local probability density function of the specific state. Then the initial parameters of a modified version of a Monte-Carlo Simulation (MCS) representing the multiple scattering of light in the subspace-level local synthetic cloud are performed. These pre-computed results are then stored into a 2D array that is then sent to the GPU memory, and a GPU kernel function is called to run in parallel the interactive visualization program using the MCS results for multiple scattering at every local subspace. In the following sections, we will describe the algorithm in more details.

2.2.1 Electron Cloud Creation

As described in the previous section, the electron wave function of the hydrogen atom is defined by $\psi_{n,l,m}(r,\theta,\varphi)$. One can use this function to calculate the spatial distribution

of a synthetic electron cloud. Using this scheme, the probability density function $|\psi|^2$ is used to give a specific probability value at a specific particle position. So the main idea behind particle placement becomes how to map the local probability value to the local particle properties.

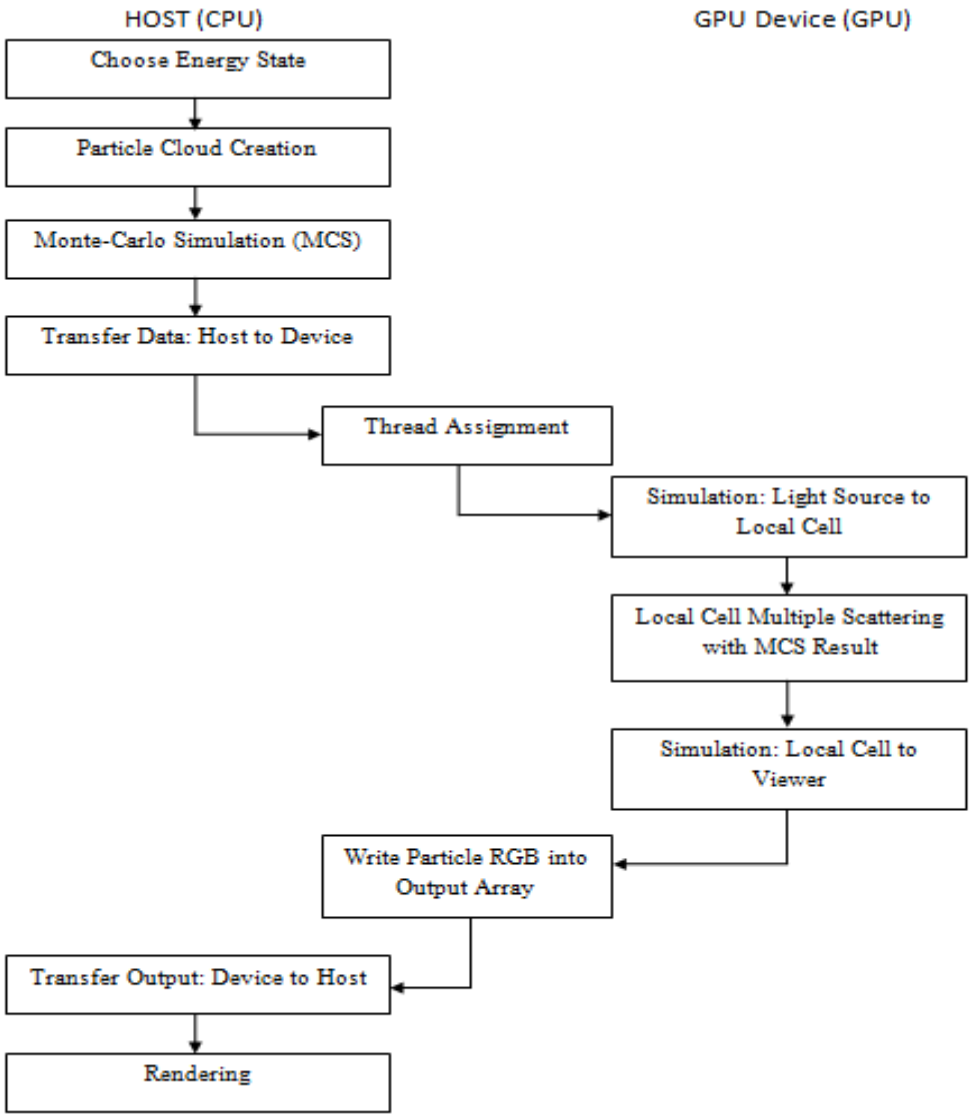


Figure 2.2: Flow chart of the proposed atomic visualization algorithm.

Possible methods include: 1) to equally place particles in the 3D space and assign different RGB & transparency values to a specific particle according to the local density

of the electron probability density at the particle's position; 2) to use the local density value of particle numbers to represent the local density value of probability. In our project, we chose the second method because the first method gives the illusion of a uniform distribution of particle placement around the nucleus leading to a misunderstanding of the physics reality. On the other hand the second method, apart from being more realistic physically, the local density of particle at each location could be easily used for volume rendering (see Chapter 3 for details).

So the main idea is for each loop, increase the radius and then produce a sphere with the present radius; for each particle (r, θ, φ) on the surface of the sphere (Figure 2.3 shows a point on the surface of sphere determined by a polar coordinate), consider it as the "local center" and calculate its local density probability value from the wave function. Then place a corresponding number of particles arbitrarily around the local center with a small offset on its position. The algorithm terminates when the radius exceeds the max possible radius outside the nuclear. Pseudo-code of the detailed implementation of the particle placement algorithm is provided in Algorithm 2.1.

Algorithm 2.1: Pseudo-code of the particle placement

```
r = r0;  // r0 is the radius of the central nuclear of in atom
while  r < r_max
    for θ ← 0 up to 2π
        for θ ← 0 up to π
            f ← ψ_{n,l,m}(r, θ, φ)
            f ← f ^2;
            Place particles according to probability density value f;
    r ← r + r_inc;  // Increase the radius by r_inc in the end of every loop
```

As shown previously, inside the loop, when we place a particle according to its probability density value, an arbitrary function is chosen to place a specific number of particle near the "local center", which means, a small offset vector (dx, dy, dz) is added to the local center (x, y, z) equally converted from the present polar coordinate (r, θ, φ).

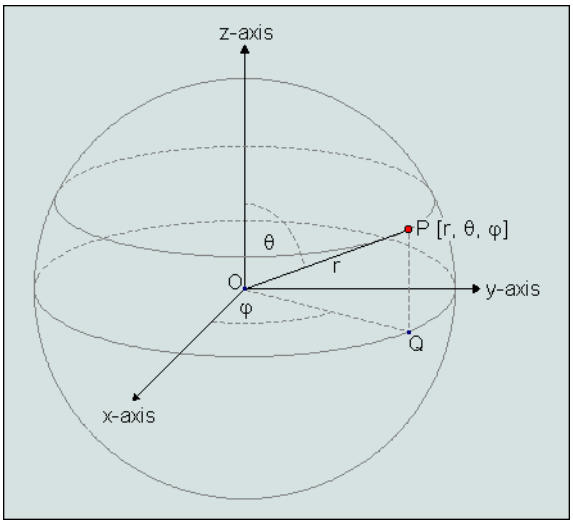


Figure 2.3: Point on the surface of a sphere [Compendium, 2007].

2.2.2 Visualization Model

In the visualization model a 3D grid is used as a technique of neighbor searching and is used to divide the whole space into subspaces, called *cubes* or *cells*. As shown in Figure 2.4 (a), each point has its own coordinate which lies in a corresponding cube. We develop a hash table where each element in the table represents one cube in the 3D grid. When one needs to search the local particle neighbors (as used in Chapter 4 SPH fluid simulation method), a 3D grid is an easy and efficient structure to use. We then compute a particle's hash value according to its position, and then go to the relative element in the hash table and find out the information of other particles in the same cube, or check other

cubes in other elements. Figure 2.4 (b) shows cubic data structure used. Although the size of the cube is equal to each other, they are not necessarily identical to the coordinate system. The detailed implementation of a 3D grid will be provided in Section 4.2.

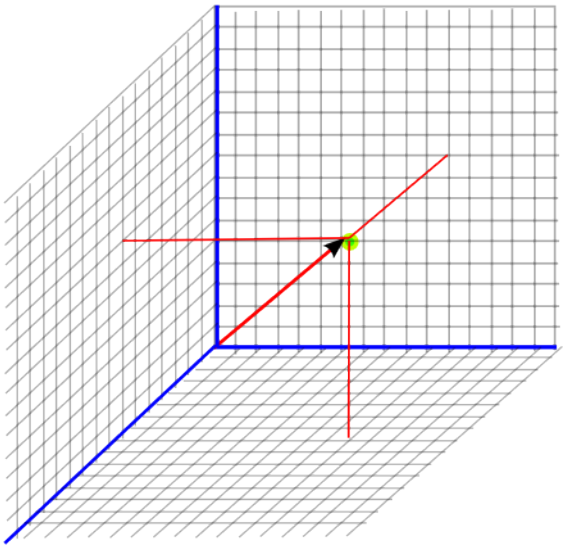


Figure 2.4 (a): Point in 3D Grid
[Euclideanspace, 1998]

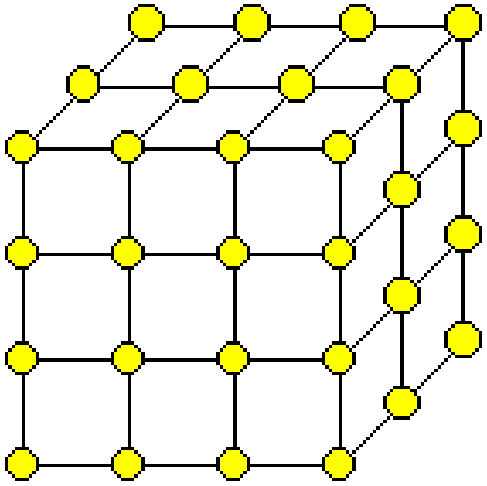


Figure 2.4 (b): 3D Grid
[APEmille, 2006]

Figure 2.4: 3D grid and the cubic data structure used.

The creation of the hash table takes place after the computation of all particle positions, and the particles within the same cube are mapped into the same hash table element to calculate the number of particles as the local density value. As shown in Figure 2.5, particles p4 and p6 have different positions but have the same hash value, so they are considered in the same cube in 3D grid, and then mapped into hash table element No.2 and the local density number of cube No.2 is “2”. So after going through every particle in particle LIST, the number in every element in N_LIST shows the local density value in every cube in the space.

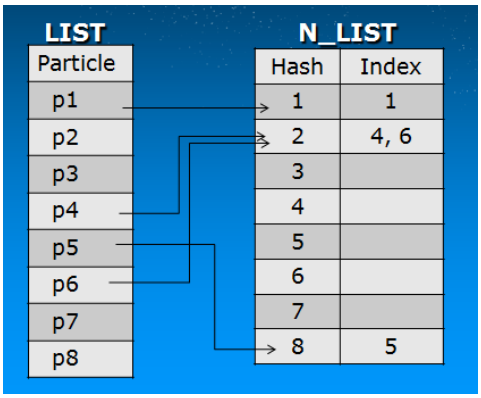


Figure 2.5: Particle list and hash table.

The visualization model (Figure 2.6) and three relative equations (Equations 2.4, 2.5 and 2.6) are from [Magnor 2005], which deals with the color (RGB) values of the particles.

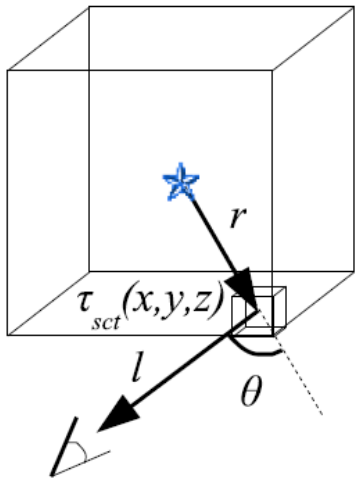


Figure 2.6: Visualization model used in [Magnor 2005]

In Figure 2.6, one can see that light from the star experiences three type source intensities (the central star shown in the figure) to the viewer's eye.

First, if one considers that the star radiant energy is Φ_{star}, then the energy L_{ill} (Equation 2.4) represents the portion of the star Φ_{star} radiant energy received by a cube, at position (x, y, z) located at a distance r from the star. The accumulated optical depth is

equal to $\tau_{opt} = \tau_{sct} / a$ and represent the ratio of the average percentage of radiation incident to a dust particle that is being scattered. The constant *a* is the albedo and is set equal to 0.6. The scattering depth is equal to $\tau_{sct} = \sigma_{sct} * l$, where *l* is the size of cube and σ_{sct} , is the local scattering coefficient. τ_{sct} is directly proportional to the local particle density:

$$L_{ill} = \frac{\Phi_{star}}{4\pi r^2} e^{-\frac{\int_0^r \tau_{opt}(r')}{r} dr'} \quad . \tag{2.4}$$

After being scattered by all particles in local cube in which a particle located at (x, y, z), the amount of light emitted from the cube is calculated by Equation 2.5. Here, the value of $P(\tau_{sct}, \theta)$ had already been established in Monte-Carlo simulation. The parameter θ is the intersection angle between the original direction ***r*** from the energy source to the particle, and the new direction ***l*** from the particle to the viewer's eye, and τ_{sct} indicates the local cube's scattering depth, which is also, proportional to local particle density value:

$$L_{\text{sct}} = L_{\text{ill}} \cdot P(\tau_{\text{sct}}, \theta) \quad . \tag{2.5}$$

Similar to Equation 2.5, the accumulated optical depth τ_{opt} is calculated using Equation 2.6 to set the portion of energy which could eventually reach the observer's eye. The final energy value L_{sct} is considered directionally proportional to the green value G. The red and blue values are set equal to the green value:

$$L_{\text{sct}} = L_{\text{ill}} \cdot e^{-\frac{\int_0^l \tau_{opt}(l')}{l} dl'} \quad . \tag{2.6}$$

2.2.3 Monte-Carlo Simulation

The visualization approach is based on a full Monte-Carlo simulation, which is pre-computed before the process of coloring the particles at set intensity values. The benefit of Monte-Carlo simulation is that it represent a multiple-scattering probability distribution which could be used in any arbitrary volume medium (a cube of particles), even for a single-particle phase function [Magnor 2005].

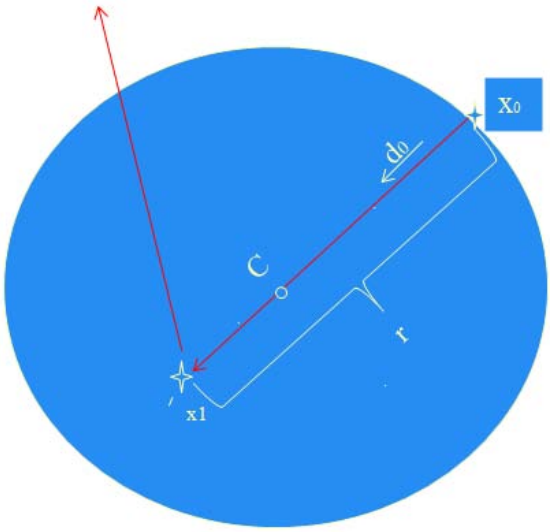


Figure 2.7: Monte-Carlo Simulation

However, unlike Magnor's work, our Electron Cloud Visualization project only considers inside-cell multiple scattering probabilities. Other multiple scattering phenomena, such as the scattering of photons between different cubes of particles, are not taken into consideration in this implementation.

The main Monte-Carlo procedure (as shown in Figure 2.7) is the following: emit N photons - each with a weight $w_0 = 1/N$ with a starting point $\mathbf{X}_0$ on the surface of one sphere with a center C and a radius that is the same value as the size of cube (marked as l) in 3D grid. All photons have the same original direction $\mathbf{d}_0$. Scattering each photon

happens after it travels for a distance r from the previous position $\mathbf{X}i$ and direction $\mathbf{d}_i$, and reaches the new position $\mathbf{X}_{i+1}$. After scattering, the new direction $\mathbf{d}_{i+1}$ is computed and the photon starts to travel for a distance r from the new position $\boldsymbol{X}_{i+1}$ again. In this scheme the weight of the photon always decreases with a fixed factor: $w_{i+1} = w_i * a$, where the constant $a = 0.6$ is the albedo.

The simulation stops when, for every photon, one of the following two conditions happens: if the new position is out of the boundary of the sphere, in which case the photon's current weight w_i is added into a special approximate directional bin $B[cos\theta]$; or if after n scattering the photon weight drop down below a fixed threshold. As long as there is any photon in the system with its weight higher than the threshold or inside the sphere, the simulation continues [Magnor 2005].

The following is the detailed description on how a particle travelling a distance r from an original position $\mathbf{X}_i$ and direction $\boldsymbol{d}_i$ to its new position $\mathbf{X}_{i+1}$, and new direction $\mathbf{d}_{i+1}$ is computed:

- Parameters:

  u, v, w – Three random variables ranging from [0,1]

  a – Dust albedo; a= 0.6 in our project

  g – Dust anisotropic factor, g=0.6 in our project

  l – Cube size, or length of sphere radius

  $\mathbf{X_0}$ – Initial position

  $\mathbf{d_0}$ – Initial direction

  C – Center of the sphere

  τ_{sct} – the scattering depth, and $\tau_{sct} = \sigma_{sct} * l$

σ_{sct} – local scattering coefficient, directly proportional to the local particle density

- The travelling length [Magnor 2005]:

$$r = -\frac{\ln(1-u)}{\sigma_{sct}}.$$

- Get New position $\boldsymbol{X}_{i+1}$ from the present position $\boldsymbol{X}_i$ [Magnor 2005]:

$$\mathbf{X}_{i+1} = \mathbf{X}_i + r \cdot \mathbf{d}_i$$

- Get New direction $\mathbf{d}_{i+1}$ (d'_x, d'_y, d'_z) from the present direction $\mathbf{d}_i$ (d_x, d_y, d_z)

1. Scattering anisotropy is considered by the use of Henyey-Greenstein's function [Magnor 2005]:

$$\cos\theta = \frac{1}{2g} \cdot \left(1 + g^2 - \left(\frac{1-g^2}{1-g+2gv}\right)^2\right)$$

$\phi = 2\pi w.$

Both random variables v, w are uniformly distributed.

2. If the present direction $\boldsymbol{d}_i$ (d_x, d_y, d_z) is almost parallel to the z-axis, e.g., $||d_z|| > 0.9999$, then:

$$d'_x = \sin\theta\cos\phi$$

$$d'_y = \sin\theta\sin\phi$$

$$d'_z = \frac{d_z}{\|d_z\|} \cdot \cos\theta\ .$$

3. Otherwise, in the normal case:

$$d'_x = \frac{\sin\theta}{\zeta} \cdot \left(d_x\, d_z \cos\phi - d_y \sin\phi\right) + d_x \cos\theta$$

$$d'_y = \frac{\sin\theta}{\zeta} \cdot \left(d_y\, d_z \cos\phi - d_x \sin\phi\right) + d_y \cos\theta$$

$$d'_z = -\zeta \cdot \sin\theta\cos\phi + d_z\cos\theta$$

$$\zeta = \sqrt{1 - d_z^2} \ .$$

The output of the full Monte-Carlo simulation is a two-dimensional [1000 * 72] array $P\ (\tau_{sct}, \theta)$, in which τ_{sct} varies from 0 to 10 by 0.01, and θ varies from -1 to +1 by 1/36. With a specific accumulated scattering depth τ_{sct}, the simulation storage bin $B[cos\theta]$ establishes one row of the 2D array.

2.3 Experimental Results and Analysis

The algorithm was implemented on a CPU and GPU using a computer with 3.0 GB of memory with an Intel Core2 Duo CPU P8400 (2.26 GHz and 2.27 GHz) and an NVIDIA Quadro FX 5800 Graphics Card.

The CPU version computes the color of every particle serially according to the visualization model. For a specific particle, the program traces a ray from the energy source to the particle, and after a scattering event, the residual lighting energy is traced from the particle to the viewer's eyes. The amount of energy left when the ray reaches the viewer eyes will determine the final color of the particle, in the perspective of the viewer. Particle_List is traversed from top to bottom to finish this procedure, and then a "Rendering Phase" after all particle colors being computed.

In contrast, the GPU version requires transferring data from host to device for the use in parallel computation. Before calling of kernel functions, constant values including Source_Power, Source_Position and Viewer_Postion, are transferred into the GPU constant memory; and large datasets, including Particle_List, Density_List and a 2D Mont_Carlo_Result are transferred into GPU texture memory. By a linear assignment of threads, the procedure of the CPU version is executed on GPU in parallel, and results in a

3D texture containing RGB of each particle as the output of device. In this implementation, we use the same linear assignment of threads and texture memory storage method as we will introduced in Chapter 4.

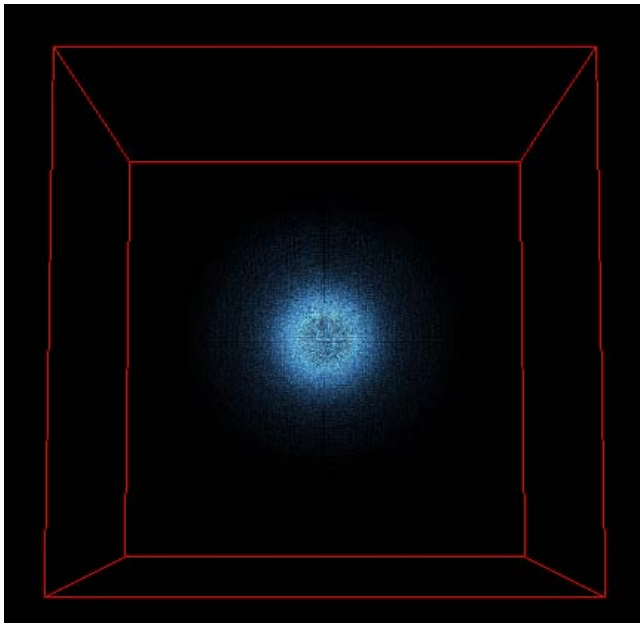


Figure 2.8: Details of state1 (N = 1, L = 0, M = 0)

Figure 2.8 shows an image of the electron cloud with a lower energy state using a middle-level intensity energy source. The result is convincing and physically correct because the particles which are near the center (the energy source) should have more luminous energy then the one in the outer regions where the energy fades away gradually as photons go through all the inner particles.

As discussed previously, interactivity was important for the MASAV project. By using the GPU implementation, one can change interactively the viewpoint and the power of the energy source revealing different aspects of the electron cloud. A control bar on the user interface provides over 10 steps of different source strength to choose from. In each

case, the time required to compute the change to power sources change is not noticeable for the GPU version.

Figure 2.9 shows three different atomic energy states (State1, State3 and State8) for different lighting conditions: lowest (50K Lumens), middle (250K Lumens), and high (500K Lumens). State1 forms a sphere, State2 has two face-to-face portions of hemi-spheres and State8 has a ring round an empty middle space between two hemi-spheres. One can notice that no matter how strong the light source is, one could always see the lighting power fading away from the center to the outer layers of the electron cloud. One can also notice that the light intensity is proportional the local density of the electron, visually revealing the true nature of the distribution.

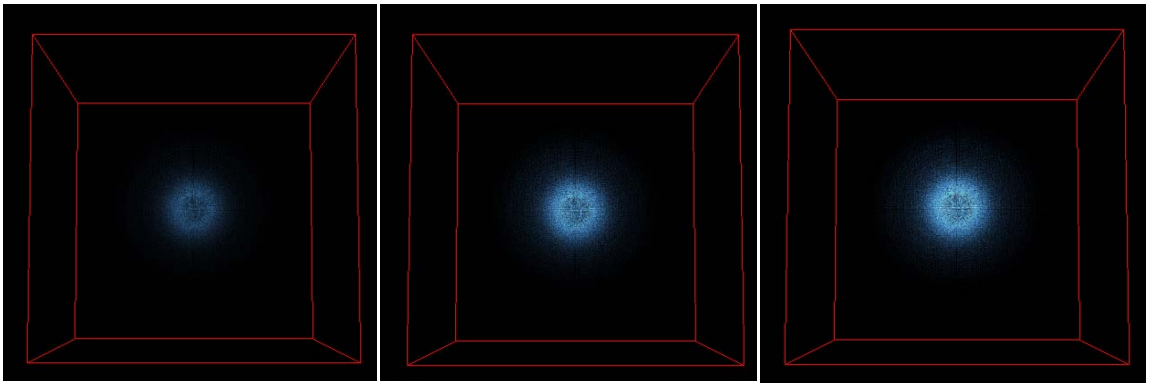


Figure 2.9 (a): Energy State1 (N = 1, L = 0, M = 0)

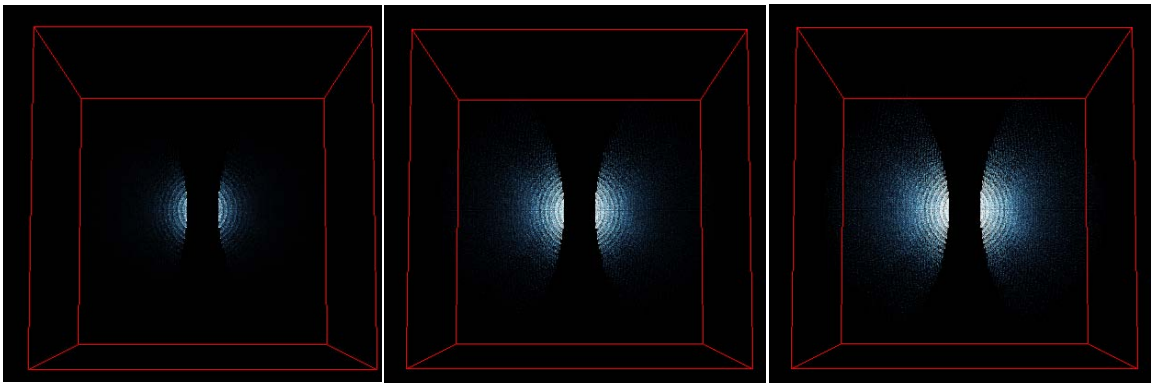


Figure 2.9 (b): Energy State3 (N = 2, L = 1, M = 0)

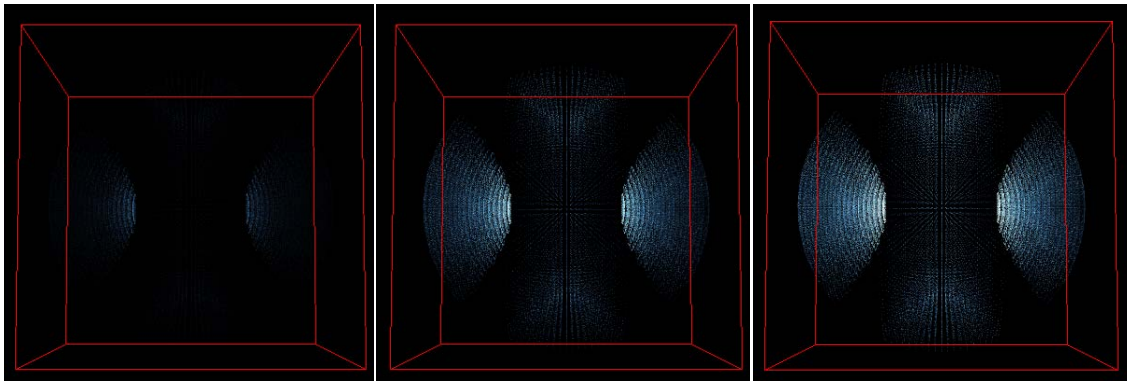


Figure 2.9 (c): Energy State8 (N = 3, L = 2, M = 0)

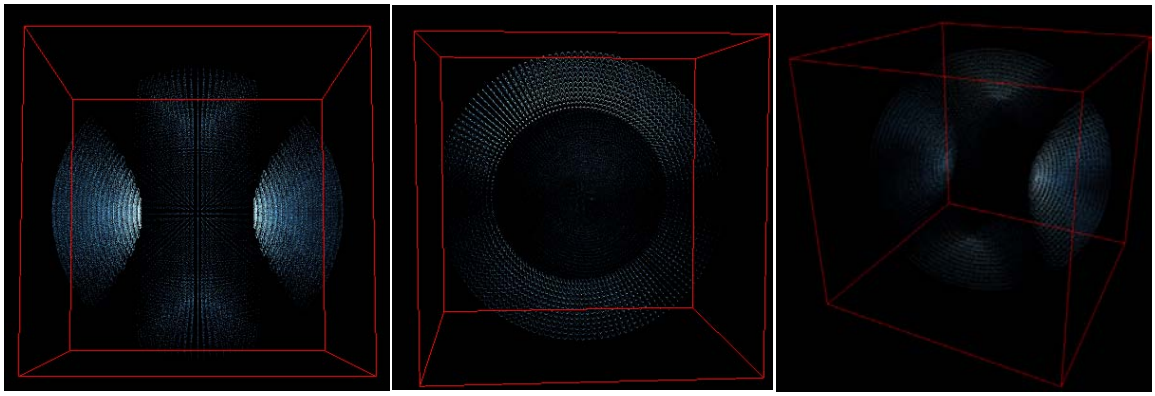
Figure 2.9: Energy State1, State3 and State8 with lighting strength: 50K (left image), 250K (middle image), and 500 K (right image) Lumens.

Our program provides viewer's position at three different positions, from top, left and front in the same energy strength. As shown in Figure 2.10, one can see in Figure 2.10(b) that most particles in the middle region have very limited intensity compare with the same particles in Figure 2.10(a) and Figure 2.10(c), revealing the true density distribution of the electron.

Detailed information on the performance of the two implementations of the ten atomic energy states is provided in Table 2.3. The first row indicates the number of particles used to render each electron cloud state and their relative time to create the electron cloud in seconds. Roughly speaking, the time and number of particles is approximately linear.

"ColorCPU" and "ColorGPU" rows provide the time to color the cloud and the intensity computation for the CPU and the GPU version. Experimental results show the CPU has relatively poor performance (tens of seconds). One can see that the average

speed-up of the GPU version is at least 150 times faster than the CPU version, opening the door to real-time interaction.



(a)Top (b) Left (c) Front

Figure 2.10: Result of energy State8 (N = 3, L = 2, M = 0) from tree viewing directions.

2.7 Remarks

The idea of comparing the atomic electron distribution to a gas in a nebula seems to work. One can easily see that the proposed rendering technique conveys more information on the inner structure of the electron cloud distribution but does not reveal the outer structure very well. In the next chapter, we will show that by combining the radiosity algorithm with a real-time ray tracing rendering, we will be able to fulfil our requirement of displaying in a realistic way the electron distribution of the hydrogen atom in real-time, allowing for true interactive visualization and exploration.

Table 2.3: CPU vs. GPU Speed for Various Atomic Energy States

State	1	2	3	4	5
N	1	2	2	2	3
L	0	0	1	1	0
M	0	0	0	1	0
No. Particle (unit)	630720	483552	316512	1124640	452016
Electron Cloud Creation (sec)	4.1	3.6	3.2	9.8	3.8
Color CPU (sec)	44.7	35.5	26.4	67.4	34.3
Color GPU (sec)	0.288	0.227	0.164	0.516	0.211
GPU/CPU (multiple)	155	156	160	131	163
State	6	7	8	9	10
N	3	3	3	3	3
L	1	1	2	2	2
M	0	1	0	1	2
No. Particle (unit)	318528	1123632	165312	132192	1000001
Electron cloud Creation (sec)	3.2	10.3	1.9	1.5	8.7
Color CPU (sec)	24.8	67.1	29.3	27.5	64.1
Color GPU (sec)	0.161	0.499	0.126	0.105	0.197
GPU/CPU (multiple)	154	134	233	260	325

CHAPTER 3 -

Volume Rendering

Facing the reality that our proposed algorithm method produces satisfactory result which conveys more information on the interior of the electron cloud, but lays little emphasis on outer-structure description, we decided to establish a bridge between our present radiosity based algorithm and some perspective rendering method that would better highlight the outer structures. As a sophisticated technique developed for many years to conveniently 3D density information, Volume Rendering (VR) was chosen.

3.1 Introduction to Volume Rendering

Volume Rendering is a widely-used rendering technique for visualizing volumetric data. In practice, the input of volume rendering is often produced by medical imaging data such as Magnetic Resonance (MR) [Lauterbur, 1973] and Computed Tomography (CT) [Herman, 2009] to display organ details inside the human body. Other usage of volume rendering are to render Confocal Microscopy (CM) data, an optical imaging technique that is used to increase the optical resolution and contrast of a micrograph by using point illumination and a spatial pinhole to eliminate out-of-focus light in specimens that are thicker than the focal plane [Tsien *et al*., 1995]. Figure 3.1 shows two examples of Volume Rendering result of a tooth and skull.

Volume rendering tries to simulate the transmission of light through a semi-transparent material which is proportional to the local material property represented by block or voxels if the data is organized as a 3D grid [Subramanian and Fussell, 1990] [Zuiderveld *et al*., 1992]. The light sent out from the ray source can be absorbed, diffracted, scattered, and emitted by the local voxel as the light traverse the data set.

Since each local volume has different optical properties, an integration process is needed. Figure 3.2 shows the standard ray tracing model as an outer light source emits light into a 3D grid of semi-transparent material represented as voxels. Similar to the nebula case, the light is absorbed, scattered, reflected depending of the local material properties. Each time the light path is changed its power is reduced until it reaches a specific pixel of the simulated sensor.

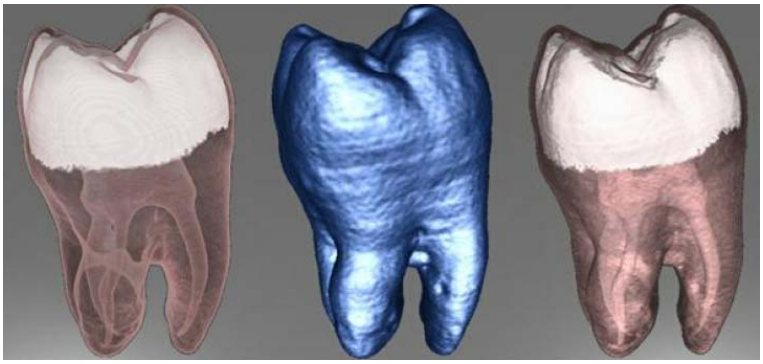


Figure 3.1 (a): Tooth

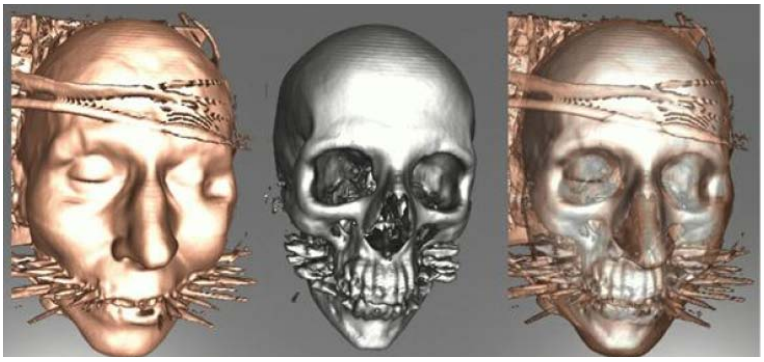


Figure 3.1 (b): Skull

Figure 3.1: Volume rendering examples [Kruger and Westermann, 2003].

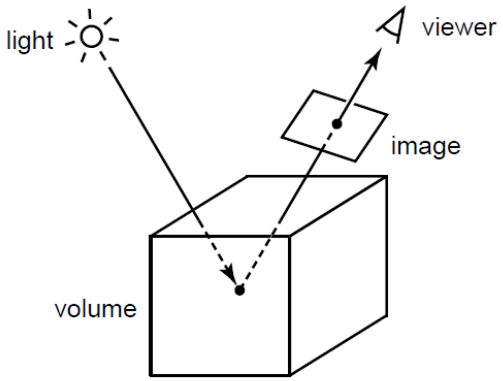


Figure 3.2: Basic principle of ray tracing [Lacroute, 1995].

Ray tracing, although accurate, is very complex for the general case as one has to represent complex optical interactions. One can develop a simplified rendering algorithm that is based on the following assumptions (as shown in Figure 3.3):

- *Single Scattering*: all the photons travelling to the output image are scattered only once after being sent out from the light source.
- *Half Absorption*: absorption between the light source and the scattering event are ignored; we only consider scattering events in the direction of the image plane.
- *Isotropic Condition*: the portion of light after scattering is uniformly absorbed and re-emitted in all possible directions.

With the assumptions of single scattering and half absorption, one can avoid the integration computation process from light source to volume [Lacroute, 1995], which is approximately half the computation compared to the more accurate Monte Carlos rendering model. The only disadvantage is that this model cannot be used to produce shadows, as there is no absorption between the light source and the local volume. In this new model, each voxel gets the energy from the light sources according to its relative

position with the source, and then re-emits the energy to the image plane. The local voxel's scalar value is mapped into a RGBA vector according to a pre-calculated lookup table and integrated in the direction of a specific pixel in the image plane.

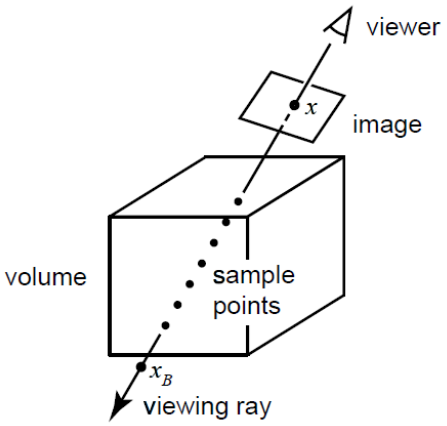


Figure 3.3: A simplified visualization model of VR [Lacroute, 1995].

3.2 GPU-Based Ray Casting

The use of VR for interactive applications has been greatly constrained by its high requirement for computational power. A typical implementations of volume rendering uses a group of dataset (generally images) in sequence according there spatial position in space as the input. In the early implementation, brute-force techniques were used and required hundreds of seconds to render a common medical dataset on a computer. In order to reduce the large amount of consumption time, optimized algorithms have been developed by different researchers to further accelerate the processing: like early ray termination and empty-space skipping [Levoy 1990; Danskin and Hanrahan 1992; Sobierajski 1994; Yagel and Shi 1993; Freund and Sloan 1997] approaches. Although those effects have high value in computing science and medical imaging research, and the results were encouraging – the running time in some algorithms has been reduced down

to the range of tens of seconds, which, however, is still far from real-time rendering. The limitation of CPU computational power still seems to be an insurmountable obstacle for real-time man-machine interaction of volume rendering.

The feasibility of GPU-based Ray Casting depends on the possibility of parallel processing of the output image pixels generation: each pixel sends out a ray which goes independently through a semi-transparent material. Each ray computation is the same, fetching information from the same dataset (but different fragment of it), using the same rendering algorithm and writing the result to the same data in the same image plane. A graphics card using SIMD architecture could apply the same computation to each pixel in parallel.

Purcell and his colleague proposed a stream model for volume rendering with ray-tracing on GPU computation in [Purcell *et al*., 2002], which was considered the most relevant work for our rendering problem. They developed a high efficiency structure to use the graphics card's power by making use of parallel fragment units and high bandwidth to texture memory. It is considered as a SIMD algorithm because each fragment unit fetch its own information from the input stream of homogeneous data created by a rasterization stage.

While implementing VR in ray-casting model, we used a skeleton code from CUDA SDK and made use of the result of Monte Carlo point-based model to add a radiosity component to the ray-casting results.

3.3 Algorithm Overview

The key to volume ray-casting on GPU is to establish an effective parallel model that give a one-to-one mapping between 2D output texture pixels and GPU chips which

execute the ray-casting algorithm in parallel. In previous sections, after the introduction of CUDA hardware structures, we consider NVIDIA graphics card as blocks of threads, so ray casting implementation based on dividing the whole Output image (2D) into sub-areas, each have the same size as a threads block and inside the sub-areas the operation on each pixel, is assigned on one single thread in that block.

3.3.1 Overall Algorithm of GPU Ray-Casting Volume Rendering

As shown in Figure 3.4, the whole algorithm of GPU Ray Casting consists of the following steps:

1. Transfer the density probability and the radiosity results (computed by the Monte Carlo simulation) into GPU texture memory;
2. Perform thread-pixel assignment;
3. Perform ray-box intersection check;
4. Re-sample the input scalar value into RGBA pixel value ;
5. Interpolate the re-sampled RGBA values for each pixels of the imaging plane;
6. Output 2D image from device to host for rendering.

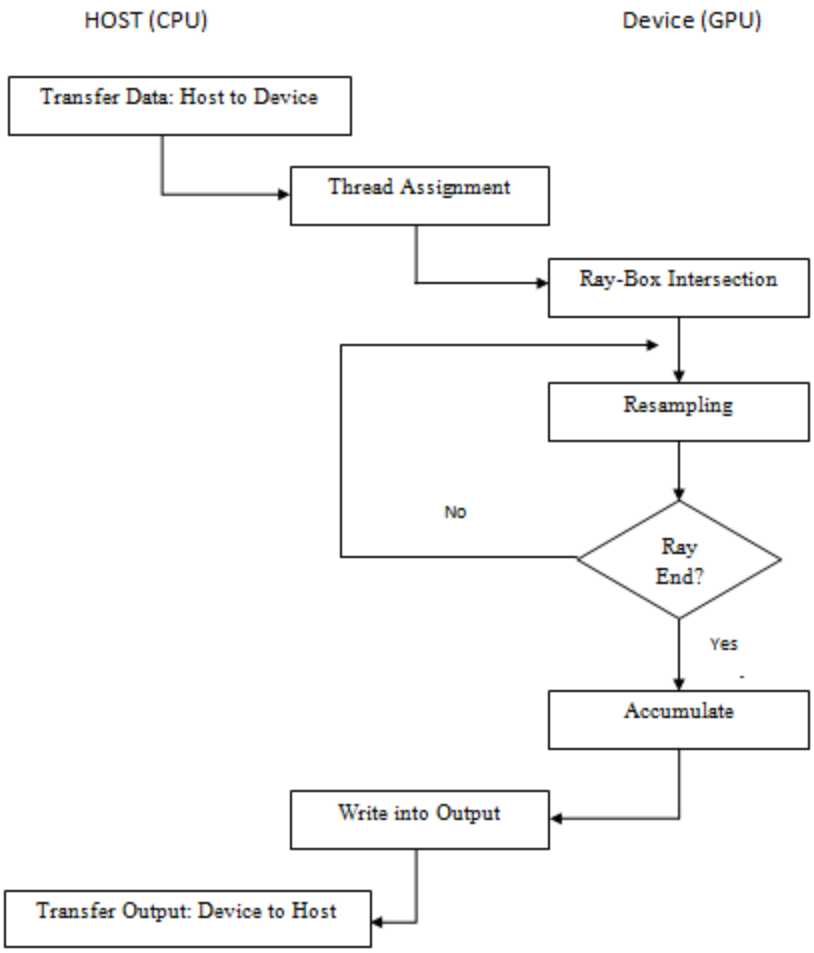


Figure 3.4: Flow chart of the proposed ray casting volume rendering.

To initiate the algorithm, the program inputs is the 3D texture which contains a probability density value and a RGB result of the Monte Carlo simulation for each sub-cell in the whole 3D grid space. Then we assign each thread on the GPU its unique pixel position (x, y, z) according to a two-dimensional mapping from the thread index and the pixel position:

```
x = blockIdx.x * blockDim.x + threadIdx.x
y = blockIdx.y * blockDim.y + threadIdx.y
z = C0
```

Here, z has a constant value c_0. Each block has a two-dimensional index number *blockIdx(x, y),* and another 2D vector *blockDim(x, y)* indicating the maximum possible value of *blockIdx.x* and *blockIdx.y*. Perform the same for a thread index *threadIdx(x, y)*. The block area size (*blockDim.x* * *blockDim.y*) has been fixed before running the kernel function. Using this technique it is possible to divide the 2D image into sub-sections which has exactly the same size of a block, where each thread has a corresponding pixel.

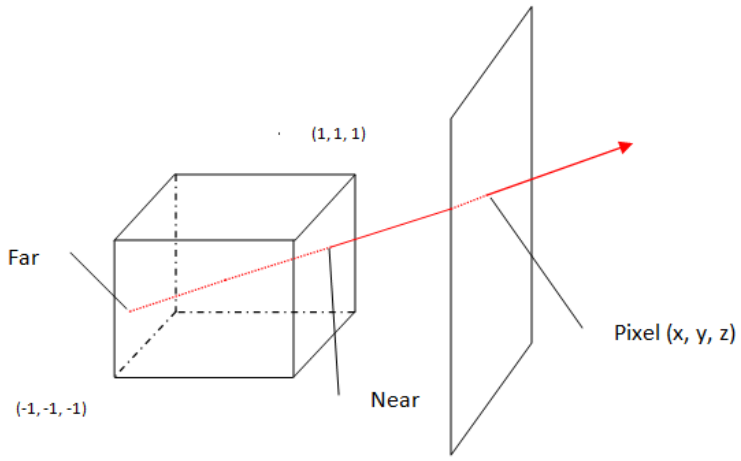


Figure 3.5: GPU-based visualization model.

As the pixel position is assigned to a specific thread, a ray is sent from each pixel of the image to the origin of the coordinate system to determine the ray-grid intersection points, see Figure 3.5. A Ray – Box intersection algorithm provided by CUDA SDK has solved the problem of efficiently finding two intersection points FarPoint and NearPoint. This algorithm will be described in Section 3.3.3 in detail. The program then traces back from FarPoint, via the origin and NearPoint, goes to the thread's assigned pixel. In the ray direction, we gradually add increment to its length and get the new position (p_x, p_y, p_z). The input texture provides the probability density and RGB result from the Monte Carlo simulation of the cell which contains that position (p_x, p_y, p_z). In Monte Carlo simulation, the resulting Red and Blue values are the same to the final Green value, so in practice our

input of RGB only contains one single floating point value – Green [Magnor 2005]. One simple linear interpolation converts the Green value and the probability density value into one single scalar value between [0.0, 1.0], and the transformation function transfers the scalar result into a 4-dimensional floating-point vector (float4) RGBA (color and alpha transparency value). Because we consider the space empty between NearPoint and pixel, only the in-box part between far and near points is taken into consideration. The loop continues to resample the new scalar values and interpolate the result into summary value with a *lerp* function, until the ray reaches the NearPoint and the integrated value is given to the pixel as its RGBA. This thread-level loop is executed in parallel for each pixel on its assigned GPU tread. At the end, the 2D image output is transferred from GPU memory to host memory.

3.3.2 Fusion of the Probability Density Rendering & RGB Results of the Monte Carlo Simulation

A pre-calculation procedure is required to bridge the Nebula Metaphor simulation to volume rendering. The problem is that the Nebula Metaphor simulation uses a point-based method, in which each particle has its unique position, color and probability. So it is necessary to use some simple tricks to convert this multiple data into two scalar values (radiosity and density) for each cell. The most obvious is to compute the radiosity inside the cell by simply computing the average particle radiosity values of the particle inside the cell. The cell density is computed by the ratio of the number of particle in the cell relative to the total amount of particle used, i.e.,

$$p(i) = \frac{NumParticles\ CellNo.i}{Total\ Particle\ Num},$$

$$\sum_{i=0}^{No.of\ Cells} p(i) = 1 \quad .$$

Here, p (i) indicates the probability value of this cell, *NumParticleInCellNo.i* is the local particle number of the i^{th} cell, and *ParticleNum* is the number of all particles in the whole 3D grid. For the green values, g (i) indicates the average value for this cell, *particle* (j).*green* indicates the green value of a specific particle j that is inside cell i as defined by:

$$g(i) = \sum_{j} \frac{particle\ (j).green}{NumParticles\ CellNo.i}\ , \quad particle\ (j) \in cell(i).$$

One simple linear interpolation between the Green value and probability density value is then performed:

$$sample = s \cdot g(\mathrm{i}) + \ t \cdot p(i)\ .$$

Here, s, t are both constant between [0.0, 1.0] and $s > t$. In *sample*, the g (i) makes a larger contribution because we need a lighting model with volume rendering method rather than a classical density field volume rendering method. The value of g (i) from the Nebula Metaphor result is the basic part in the final result; and probability density p (i) describes not lighting but distributions of volume in the grid space, and provides an indispensible continuous change of volume colors without any threshold. For example, in point-based method, if the threshold of probability density value for placing one single particle is 0.1, the space with a smaller value than 0.1 is ignored and consider empty. However, volume rendering method still renders a relative color in this space and provides more interior and exterior information in this case. By the interpolation between the two inputs, we combine the advantage of the two methods as a volume rendering lighting model.

3.3.3 Ray – Box Intersection

The Ray – Box Intersection algorithms is performed using Kay and Kayjia's "Slabs" method [Kay and Kayjia, 1996]. The term "slab" represents the space between a pair of parallel planes. And this method use slabs both in ray – box intersection computation and ray – volume intersection examination. Here, a ray is defined as **Ray** (T) = **O** + T * **d**, where **O**(x, y, z) indicates the ray origin and **d**(x, y, z) represents the ray direction. Algorithm 3.1 gives the details of the algorithm in pseudo code.

Algorithm 3.1: Slabs Ray – Box intersection method.

T_{near} = - infinity, T_{far} = infinity // output T_{near} and T_{far}

For the pair of X planes X_l and X_h, do:

if direction $\mathbf{d}_x = 0$ then the ray is parallel to the X planes

if origin $\mathbf{O}_x$ is not between the slabs ($O_x < X_l$ or $O_x > X_h$)

then return false //this example using x planes X_l and X_h

else if the ray is not parallel to the plane then

begin:

$T_1 = (X_l - \mathbf{O}_x) / \mathbf{d}_x$

$T_2 = (X_h - \mathbf{O}_x) / \mathbf{d}_x$ // compute the intersection distance T_1, T_2 of the planes

If $T_1 > T_2$, swap (T_1, T_2) // since T_1 is the intersection with near plane

If $T_1 > T_{near}$, $T_{near} = T_1$ // want largest T_{near}

If $T_2 < T_{far}$, $T_{far} = T_2$ // want smallest T_{far}

If $T_{near} > T_{far}$, return false // box is missed so return false

If $T_{far} < 0$, return false // box is behind ray return false

end;

For the pair of Y and Z planes Y_l, Y_h and Z_l, Z_h do the same tests.

If Box survived all above tests, return true with intersection point T_{near} and exit point T_{far}.

As shown in Algorithm 3.1, T_{far} is chosen as the smallest from all the T_{far} in X, Y and Z slabs, and T_{near} is selected as the largest. The result of the algorithm provided two values (T_1 and T_2) to determine two endpoints on the ray.

3.3.4 Transform Function for Texture Re-sampling

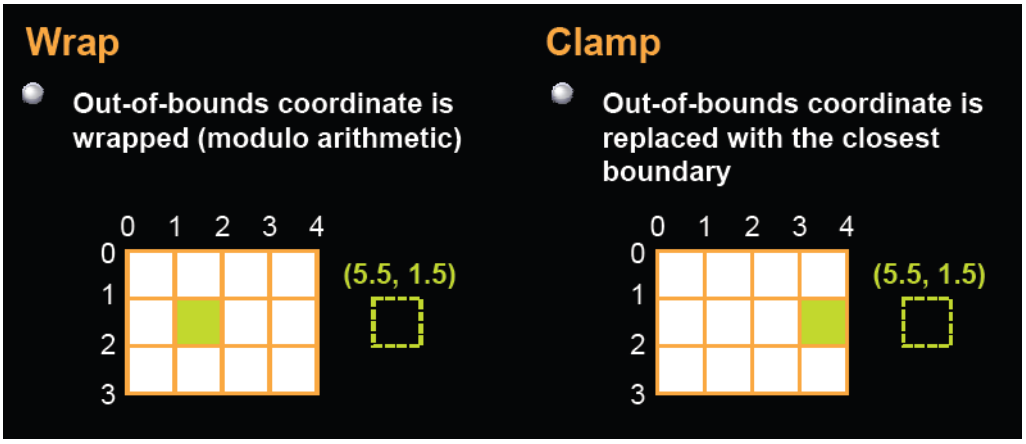


Figure 3.6: Difference between Wrap and Clamp [NVIDIA, 2008].

A 2D transfer matrix is stored in host memory when the kernel functions are activated. The matrix, in which each element is a float4 vector, is then copied into the card global constant memory and reformed into a 1D texture called *TransferTex*. This texture, however long and value-distributed, will later be normalized (mapping between [0, width] texture into [0, 1] coordinate) and then wrapped (to deal with out-boundary cases). Figure 3.6 from NVIDIA CUDA programming guide explained in detail the difference between wrapped texture and clamped texture operation in dealing with the margin.

So when a GPU thread fetches its own local probability density value and average radiosity value from the input texture, the scalar value in [0, 1] then determines a relative position in the *TransferTex*, which provides a corresponding RGBA float4 vector.

3.3.5 Linear Interpolation for Integrated Pixel Value

A linear interpolation *"lerp"* function is performed between the present summary value and the new transferred float4 value. A lerp function "*lerp (a, b, amt)*" calculates a number between two numbers *a*, *b* at a specific increment *amt*. The *amt* parameter is the amount to interpolate between the two values, where 0.0 equal to the first point, 0.1 is very near the first point, 0.5 is half-way in between, etc. The *lerp* function is convenient for creating motion along a straight path and for drawing dotted lines.

3.4 Experimental Results and Analysis

The machine running the program is an NVIDIA Quadro FX5800 Graphics card and an Intel Core2 Duo P8400 (2.26 GHz and 2.27 GHz) CPU. Our volume size is 60x60x60. The average processing time is 151 milliseconds including transferring the data from CPU to GPU in real time, while different electron cloud energy states have very limited value of variation (<5 milliseconds) from this value. Besides the data transferring time, the VR computation time is from 32 to 37 milliseconds, with their comparative Frame per Second (fps) between 27 fps to 31 fps. From the users' experience viewpoint, while the user dragging the controlling bar to adjust the source power, the final image is given simultaneously at the end of the mouse event with no noticeable delay. In our design, the local electron cloud particles change colors from blue, via green to red, which indicates the change of energy from low radiance to high level of radiance. See Figure 3.7 as an example where the amount of energy received from the source decreases.

Figure 3.7 (a) 50K Lumens

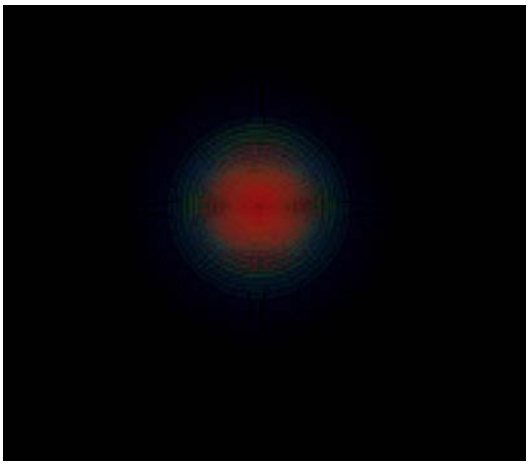


Figure 3.7 (b) 150K Lumens

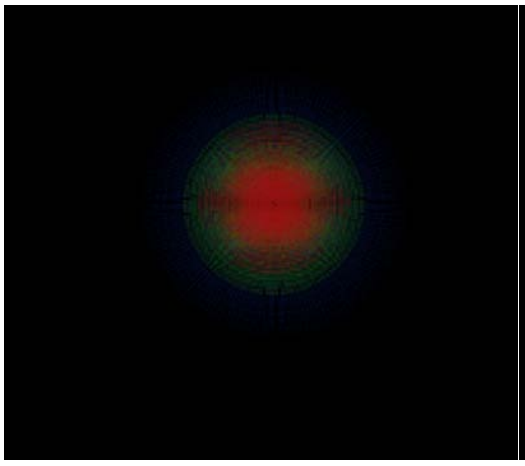


Figure 3.7(c) 200K Lumens

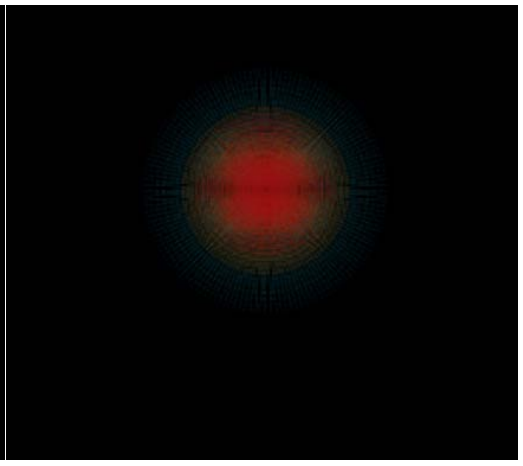


Figure 3.7 (d) 250K Lumens

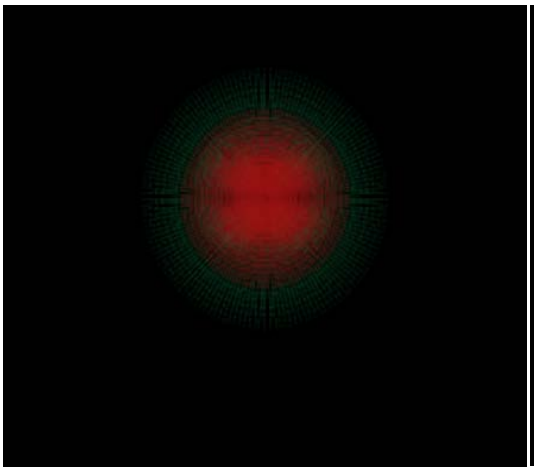


Figure 3.7 (e) 300K Lumens

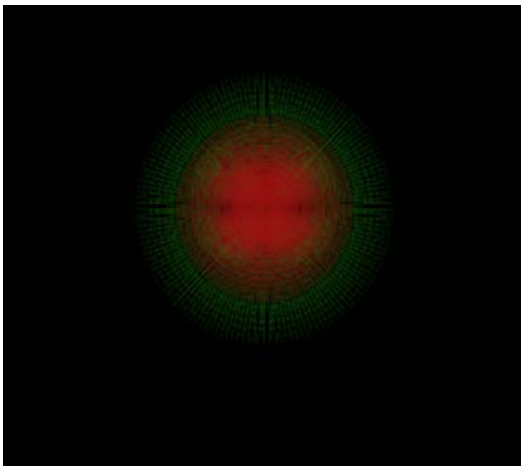


Figure 3.7 (f) 350K Lumens

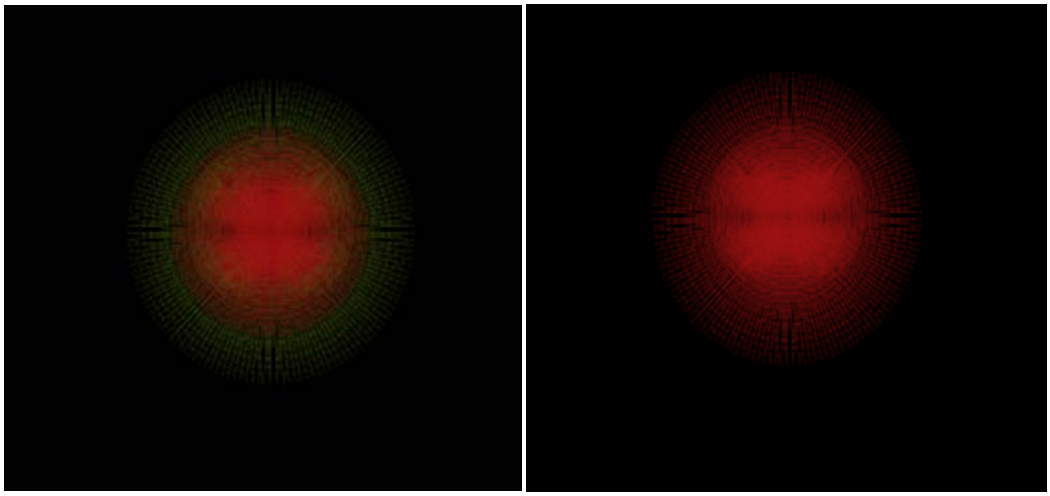


Figure 3.7 (g) 400K Lumens Figure 3.7 (h) 500K Lumens

Figure 3.7: Energy State1 (N = 1, L = 0, M = 0) from 50K to 500K Lumens

As shown in Figure 3.7 (a), at the beginning of the program, when the power source is in the middle in the default lowest value, most of the particles' colors are dark and their transparency has the highest score. In the middle-level of the atomic energy states, the region of red particles increases gradually with the increase of energy, which means that stronger energy radiates to outer area from the center. At the end, as shown in Figure 3.7 (h), the source power gets strong enough to heat the whole model into red. Figure 3.8 shows a similar energy increase result for State5 (a sphere-like state) of the electron cloud.

The reason that different states have little deviation in speed performance from the average value lies in the input of our program: the input is a 3D texture containing 60x60x60 volume input values of every small cube in the whole space, and all ten electron cloud states differ only in the those value, not the size of the input.

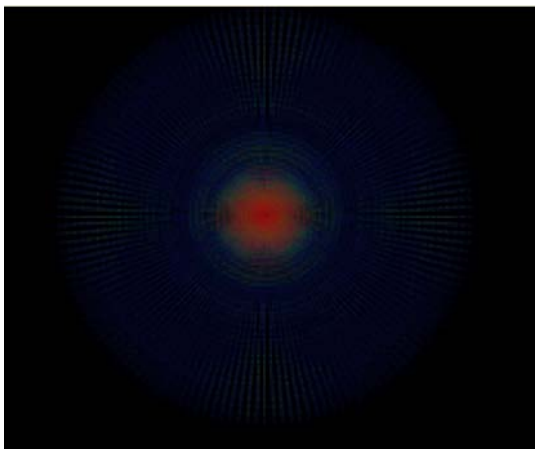


Figure 3.8 (a) 50K Lumens

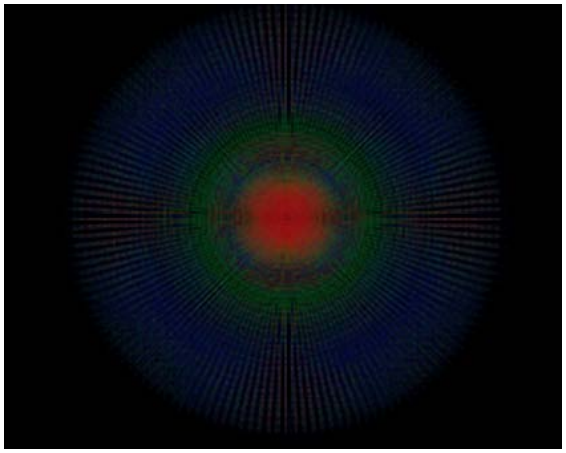


Figure 3.8 (b) 150K Lumens

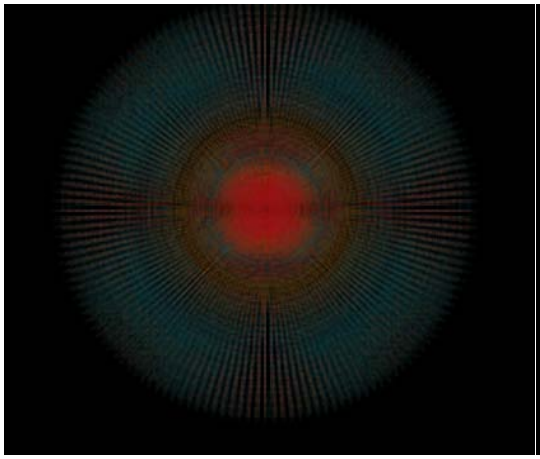


Figure 3.8 (c) 200K Lumens

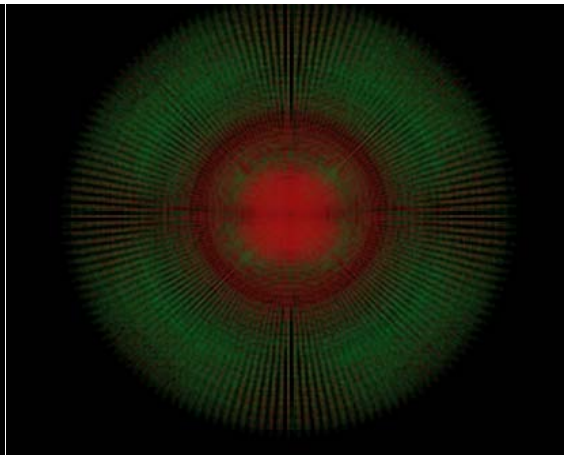


Figure 3.8 (d) 250K Lumens

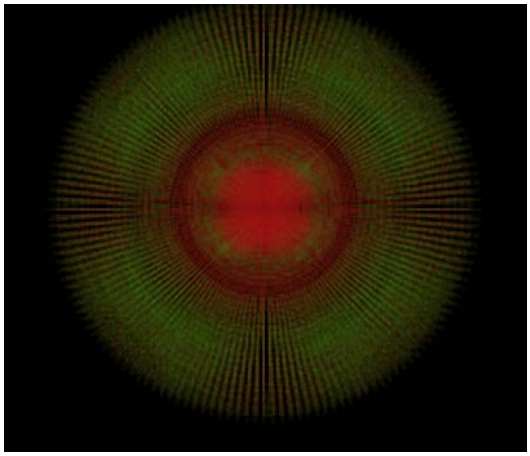


Figure 3.8 (e) 300K Lumens

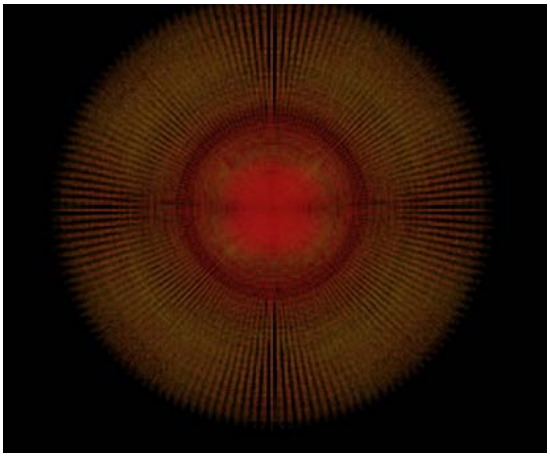


Figure 3.8 (f) 350K Lumens

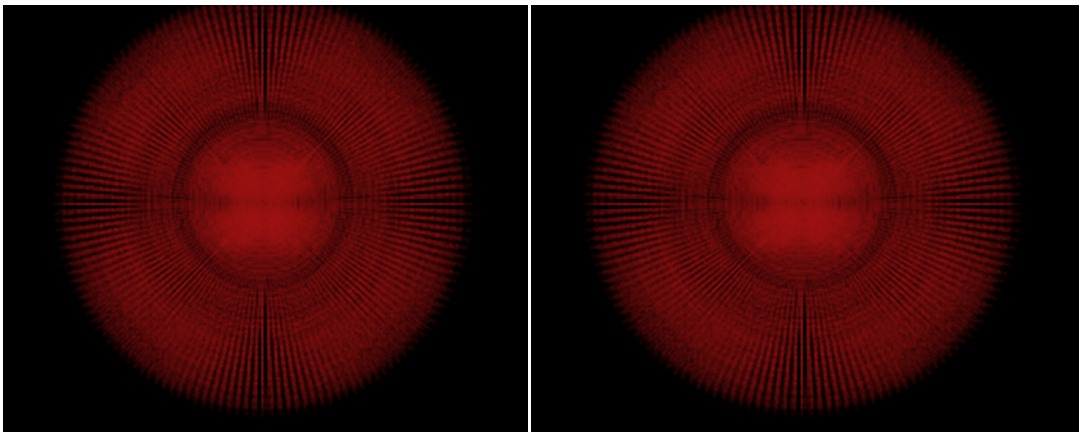


Figure 3.8 (g) 400K Lumens Figure 3.8(h) 500K Lumens

Figure 3.8: Energy State5 (N = 3, L = 0, M = 0) from 50K to 500K Lumens

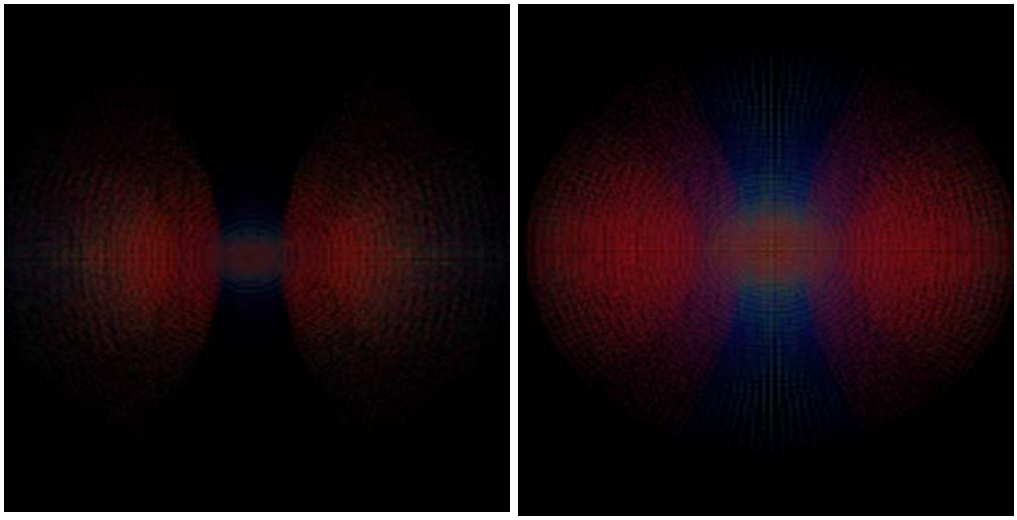


Figure 3.9 (a) 150K Lumens Figure 3.9 (b) 400K Lumens

Figure 3.9: Energy State3 (N = 2, L = 1, M = 0) with lighting conditions: 150K and 400K Lumens

In other render energy states such as State 3, State 8 and State 10 shown in Figure 3.9, Figure 3.10 and Figure 3.11, one can see the importance of the rendering produced by the Monte Carlo simulation to be the basic framework of the structure. The increment

added to low-density blocks leads to a big change in their color so that when the user controls the program interactively, they could gain a better understanding of the electron.

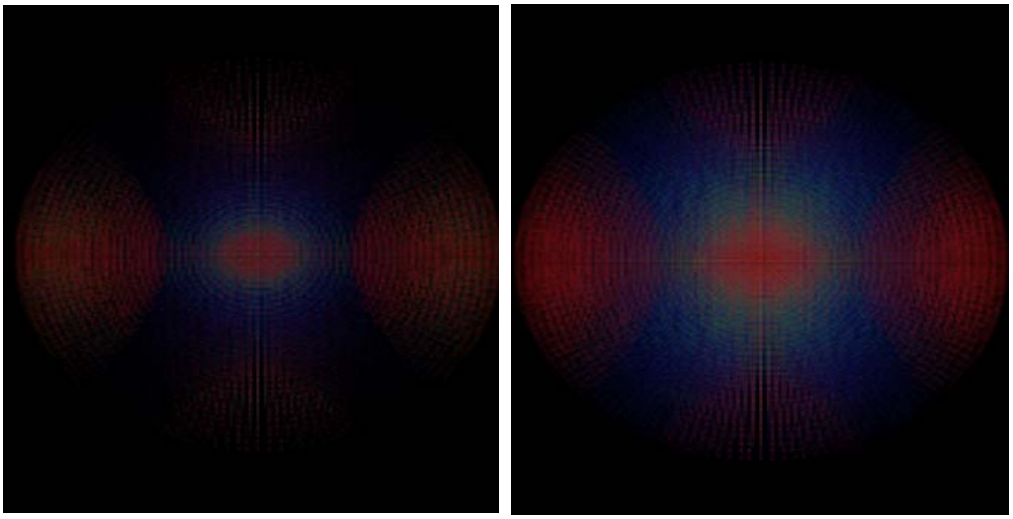


Figure 3.10 (a) 150K Lumens Figure 3.10 (b) 400K Lumens

Figure 3.10: Energy State8 (N = 3, L = 2, M = 0) with lighting conditions: 150K and 400K Lumens

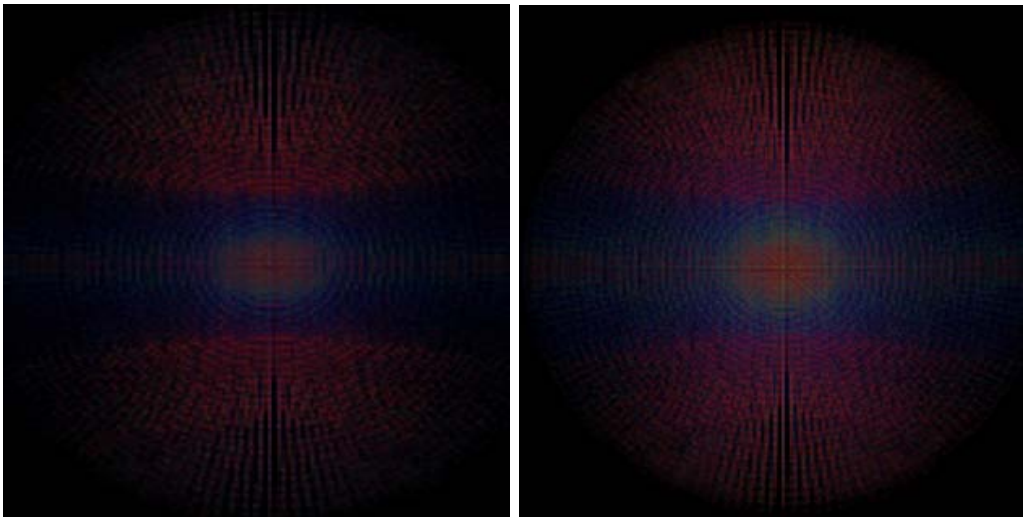


Figure 3.11 (a) 150K Lumens Figure 3.11 (b) 400K Lumens

Figure 3.11: Energy State 10 (N = 3, L = 2, M = 2) with lighting conditions: 150K and 400K Lumens

CHAPTER 4 -

Particle-Based Fluid Motion Simulation

This Chapter introduced an implementation of a point-based fluid simulation technique using Smoothed Particle Hydrodynamics (SPH). We first developed the code on CPU to verify the correctness of the SPH implementation. Then, faced with the constraint of limited computational power of the CPU and its inability to run and render the simulation in real-time, we transformed the code to a CUDA implementation. In this chapter, we will first describe in detail the basic theory behind SPH, then its GPU and CPU implementation. Towards the end of the chapter we will demonstrate some experimental results and then provided a detailed performance analysis of the difference between the CPU and the GPU implementation.

4.1 Related Work

In fluid dynamics simulation there are three main approaches to solve numerically the problem: continuous, Lagrangian, and Eulerian frameworks. The famous Navier-Stoke equation [Foster *et al.* 1997, 2001] falls in the first category and has been used to study the behaviour of fluids in various conditions for decades. The Navier-Stoke equations arise from applying Newton's second law to fluid motion, together with the assumption that the fluid stress is the sum of a diffusing viscous term (proportional to the gradient of velocity), plus a pressure term. For the discrete version of the problem, the assumption here is that the fluid traverses elementary volume elements preserving the basic assumptions of the Navier-Stoke equations. The solutions of the Navier–Stokes equations result in a flow field, which is a description of the velocity of the fluid at a given point in

space and time. This is different from what one normally sees in classical mechanics, where solutions are typically trajectories of position of a particle. This is a problem for real-time simulation as there is a disconnection between rendering and simulation. To solve this problem Stam et al. [Stam *et al.* 1999] introduce a semi-Lagrangian framework where particle motions are visualized using an advection equation. Stam's work later inspired Enright *et al* [2002], who developed an atomic-level algorithm dealing with Lagrangian particles. Carlson *et al.*[2004] contributed a method for the interplay between rigid body and fluid, and a method for animating a viscoelastic fluid was presented by Goktekin *et al.* [2004]. Two and three-dimensional techniques, such as vortex particle and surface tension, were then introduced by Irving *et al.* [2006].

For pure Lagrangian methods, the main contributions came from two major particle-based fluid dynamic simulation methods: the Moving Particle Semi-implicit (MPS) method and Smoothed Particle Hydrodynamics (SPH). MPS was originally used by Koshizuka *et al.* [1996] in Nuclear Science research to study the incompressibility of molecule by solving the Poisson equation, and then was introduced to computer graphics by Premoze *et al.* [2003].

Monaghan *et al.* [1992] originally developed SPH in the context of astronomy simulation of galaxies, and later by Muller who experimented and demonstrated that SPH could be successfully applied into computer graphics. After years of rapid development of GPU computational power, Amada *et al.* [2004] was the first to use the GPU for the acceleration of SPH. The main problem is that their method for neighbouring particle search was based on CPU computations (which take almost half of the computational power) not including the very inefficient transfer of data between CPU and GPU. Both

Kolb *et al.* [2005] and Harada *et al.* [2007] were the first researchers to entirely implement SPH on the GPU. The major problem with Kolb's implementation lies in the interpolation error, because in their method they first compute physical values of the grid and then use those values to interpolate particles' values.

Our implementation is mainly based on Harada's work, which contained a new neighbour-searching method for inter-particle force computation and all the values for each particle was calculated individually without any need for grid-level computation. However, Harada used explicit Euler integration which is known to be imprecise for long integration periods. To improve Harada's SPH implementation, we use a 4^{th} order Runge-Kutta method for integration.

4.2 Smoothed Particle Hydrodynamics (SPH)

4.2.1 Data Structure

To use particles to represent the movement of fluid, we need a data structure containing position, color, normal (optional, for light effects), radius and the velocity of a particle. The code below shows the data structure used in our program, where bold characters represent vectors:

```
Particle
{
   position (x, y, z);
   velocity (vx, vy, vz);
   color (r, b, g);
   normal (nx, ny, nz);
 }
```

4.2.2 Neighbor Search Using a 3D Grid

A 3D *grid* is used to represent the whole space in which the fluid motion happens. See detailed information in Harris's work [2003]. In our algorithm, we use a hash function to assign a *neighbour bucket* to a specific visible point (surfel) according to its position:

- **Grid**

  We consider the whole cubic space as one single *grid*. The term '*grid*' is also used in this thesis in CUDA graphics card memory structure description, to express the whole patch of different blocks of parallel threads in a running kernel function.

- **Cell / Cube**

  In the large grid, space is divided into equal-size subspaces with the shape of cube, called '*cells*' or '*cubes*'. Each cell has its own unified index for all particles in the same cell. The total number of cubes in the grid is computed by using the length of grid and the length of cell:

$$L = x / CellLength$$
$$M = y / CellLength$$
$$N = z / CellLength \quad (4.1)$$

  So, in total, there are "$L \cdot M \cdot N$" small cells in the full grid.

- **Hash Value**

  The hash number is computed according to the hash Equation 4.2. The main idea is to put the present particle, of which we search for the neighbours, in the center of an invisible sphere in 3D space, search all possible cells which may contain particles inside the sphere, calculate relative values for the inside particles while ignoring the outside-sphere particles in these cells. The equation is:

$$HashValue = x \cdot M \cdot N + y \cdot N + z \ . \tag{4.2}$$

In practice, the search radius equals the cubic cell length, and for every particle, one has to search $3 \times 3 \times 3$ cells to make sure that we successfully take every neighbouring particle into consideration. As shown in Figure 4.1, the search for neighbours is similar to picking up a Braille Rubik's Cube and the searching sphere is exactly the internally tangent sphere of the Braille Rubik's Cube. The central cell in a Braille Rubik's Cube represents the cell that contains the current particles and the central cell and its 26 neighbouring cells must be taken into consideration. No matter how many particles there are inside each cell, if the particle we are considering is near the boundary of the central cell, we still need to search all the neighbour cells.

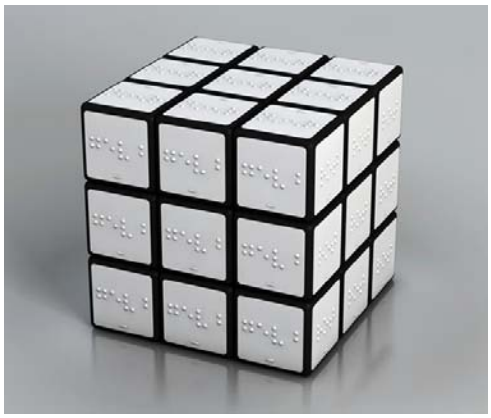


Figure 4.1: Braille Rubik's cube to represent the neighbourhood search [Spice, 2010].

4.2.3 Algorithm Overview

The proposed particle system was designed to run entirely on the GPU, and to minimize the transfer time between the host computer and the GPU card, hence fully exploiting the real GPU computational potentials. A similar project, called *Particle System*, is provided in CUDA SDK for programmers, which contains the same framework as we requires but

in different physical models. In this thesis, we implement SPH on that framework and added all necessary modifications. The basic simulation loop is composed of the following stages:

1. Bucket Generation;
2. Density Computation;
3. Velocity Update;
4. Position Update.

A flow chart is provided in Figure 4.8. Once a simulation loop is complete with all particle positions updated, the rendering phase takes place and the particles are sent through the graphics pipeline to create an image frame.

4.2.3.1 Bucket Generation

The first stage of our simulation loop is responsible for advancing the particles in time, integrating the governing equation. Then for each particle, we build a bucket to store all neighbours for future computation, using the method for fetching neighbouring particles described in Section 4.2.2.

4.2.3.2 Density Computation

In SPH, a physical value at position **x** is calculated as a weighted sum of physical values ϕ_j from every one of the neighbouring particles. The function $\phi\ (x)$ is expressed by:

$$\phi(\mathbf{x}) = \sum_j \ m_j \cdot \frac{\phi_j}{\rho_j} \ \boldsymbol{W}\left(\mathbf{x} - \mathbf{x}_j\right) \quad . \tag{4.3}$$

So in order to get the density formula $\rho(\mathbf{x})$ for a specific particle at **x**:

$$\rho(\mathbf{x}) = \sum_j \, m_j \cdot \boldsymbol{W}\left(\mathbf{x} - \mathbf{x}_j\right) \quad . \tag{4.4}$$

The force of viscosity $\mathbf{F}^{vis}$ and the pressure force $\mathbf{F}^{press}$ for a specific particle p_i are given by Equations 4.5 and 4.6, and the parameters in these equations are shown in Equation 4.7 [Harada *et al.* 2007]. In theory, the viscosity force comes from the relative velocity between two neighbouring particles attracting each other, and the pressure force using the incompressibility condition and the Coulombian forces is in Equation 4.6.

$$\mathrm{F}_i^{vis} = v \sum_j \mathrm{m}_j \cdot \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \cdot \nabla W_{vis}\left(\mathbf{r}_{ij}\right) \quad , \tag{4.5}$$

$$\mathrm{F}_i^{press} = -k_p \cdot \sum_j \, m_j \cdot \frac{\rho_j - \rho_i}{2\rho_j} \cdot \nabla W_{press}\left(\mathbf{r}_{ij}\right) \quad , \tag{4.6}$$

$$\nabla W_{press}\,(\mathbf{r}) = \frac{45}{\pi r_e^6}\left(r_e - |\mathbf{r}|\right)^3 \frac{\mathbf{r}}{|\mathbf{r}|} \,,$$

$$\nabla W_{vis}\,(\mathbf{r}) = \frac{45}{\pi r_e^6}\left(r_e - |\mathbf{r}|\right),$$

$$\boldsymbol{W}(\mathbf{r}) = \frac{315}{64\pi\, r_e^9}(r_e^2 - |\mathbf{r}|^2)^3 \quad . \tag{4.7}$$

In Equations 4.5, 4.6 and 4.7, r_e is the effective distance between neighbouring particles, which are considered to contribute to the local density and forces, and **r** stands for a radius vector with its direction from a specific neighbour particle to the central particle. The parameter *v* in Equation 4.5 is called the Dynamic Viscosity Constant (DVC), and k_p, called the Pressure Constant (PC) in Equation 4.6, is a constant value that used to adjust the pressure force.

4.2.3.3 Velocity & Position Update

After the computation of density and two forces, the program calculates the acceleration of each particle from its total force (the combination of viscosity and pressure force).

The original numerical solution of ordinary differential equation to update velocity **v** and position **p** in the work of Harada *et al.* [2007] was based on the Euler method: $\mathbf{v}_1 = \mathbf{v}_0 + \mathbf{a}\ \mathrm{d}t$ and $\mathbf{P}_1 = \mathbf{P}_0 + \mathbf{v}\ \mathrm{d}t$, where d*t* is the time offset between two sequential iterations.

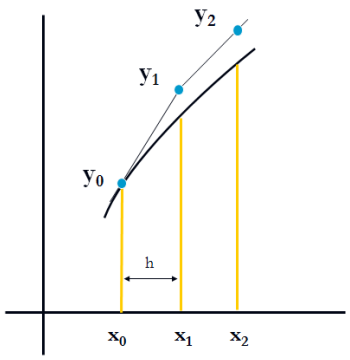


Figure 4.2: Euler method [Thacker, R., 2007].

A standard Euler method denotes the time offset as *h* and is widely used in engineering or scientific computation. As shown in Figure 4.2, in order to compute a new result y_1 of function $y(x) = f(x)$, we let $y_1 = y_0 + hf_0$, where $f_1 = f(x_1, y_1)$ predict forward from y_1; and $y_2 = y_1 + hf_1$, where $f_2 = f(x_2, y_2)$; and so on. The problem with the Euler method, as we can see clearly contains both a large local error per iteration, and an accumulative global error which could be unacceptable after a few iterations. Roughly speaking, if the local error is O (h^n) then the global error will be O (h^{n-1}). This happens because $n = (x_n - x_0)/h$ and you sum over n intervals [Thacker, R., 2007].

The Euler method is flawed because we use the beginning of the interval to predict the end. There are other higher-order methods with more sub-steps of evaluation using midpoint values as well to further reduce interpolation error. Runge-Kutta, provided by R. England [1968], is one of the most widely used interpolation methods as a solution for ordinary differential equations. The Runge-Kutta methods series considers the Euler's as its own first order solution and extends the interpolation into a higher-order scheme. The classical Runge-Kutta (the 4th order version) is a 4-step iteration described in Equations 4.8 to 4.12. See Figure 4.3 for details: the 1st step is to initialize the original position of the curve (the left dot point in the figure), which is fixed by Equation 4.8. Step 2 and 3 calculate the slope and position at midpoint from Equation 4.9 and 4.10, and finally the slope of target position is given by Equation 4.11. The resulting value from the previous step is immediately inserted into the next iteration for consideration and at the end of the four steps, the final interpolation value of next position y_{n+1} (shown as the right dot node at position y_{n+1} in Figure 4.3) is provided with all values in each of the 4 steps (see Equation 4.12 for detail).

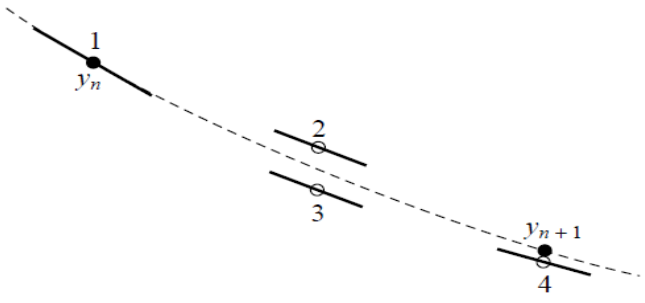


Figure 4.3: Classical (4th Order) Runge-Kutta method [Press, W., *et al.*, 1992].

The Runge-Kutta equations are:

$$f_0 = f(x_0, y_0) \quad , \tag{4.8}$$

$$\tilde{f}_{\frac{1}{2}} = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2} f_0\right) \quad , \tag{4.9}$$

$$f_{\frac{1}{2}} = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2} \tilde{f}_{\frac{1}{2}}\right) \quad , \tag{4.10}$$

$$\tilde{f}_1 = f\left(x_0 + h, y_0 + h f_{\frac{1}{2}}\right) \, , \tag{4.11}$$

$$y_1 = y_0 + \frac{h}{6}\left(f_0 + 2\tilde{f}_{\frac{1}{2}} + 2f_{\frac{1}{2}} + \tilde{f}_1\right) \, . \tag{4.12}$$

The interpretation of the sum over interior points gives an average slope. The local error is proportional to h^5, so halving *h* leads to a 1/32 reduction in the local discrete error [Thacker, R., 2007]. Thacker provided a detailed comparison of global error for different *h*-values using the Runge-Kutta method, Euler method and improved Euler method in Figure 4.4, where the improved Euler method could be considered as an intermediate method from the original Euler to Runge-Kutta. This is the reason we chose use Runge Kutta to solve our SPH integration problem. The experimental results in Figure 4.4 demonstrate that by comparison with Euler method with the same *h*, the Runge-Kutta method greatly reduces the global error between two sequential integrated positions, and on average, the error in Runge-Kutta is approximately at powers three times smaller than that the Euler method, while the time offset h varies from 0.01 to 0.1.

Above all, with the same time step, Runge Kutta provides much more accuracy than the Euler method but may require more computational power which affects its speed performance. In the later sections, we will compare the result of both Euler and Runge Kutta version of SPH in terms of speed.

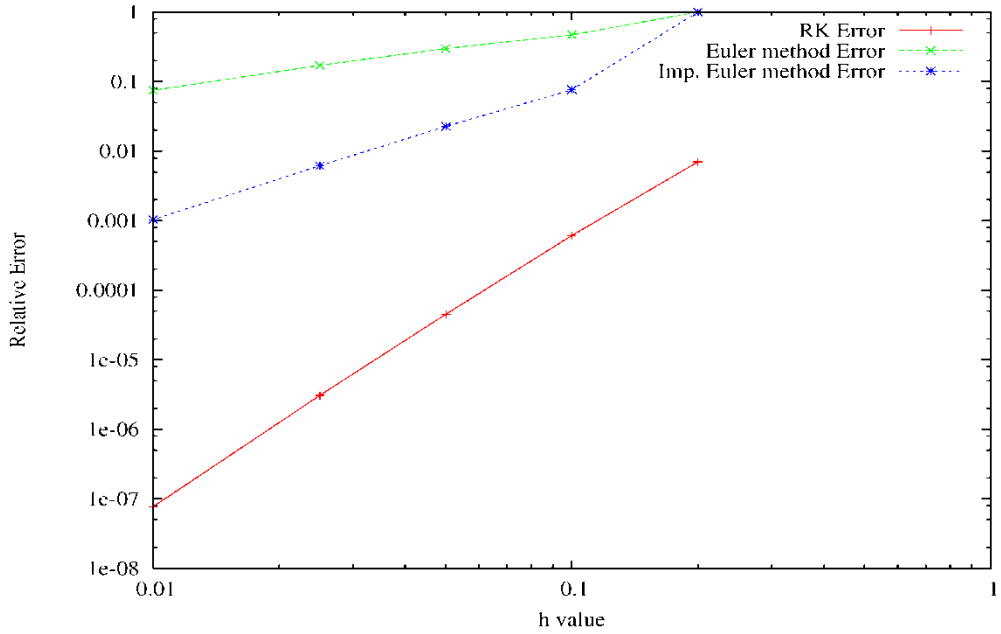


Figure 4.4: Comparison of classical Runge Kutta and Euler method errors. [Thacker, 2007]

4.3 CPU Implementation

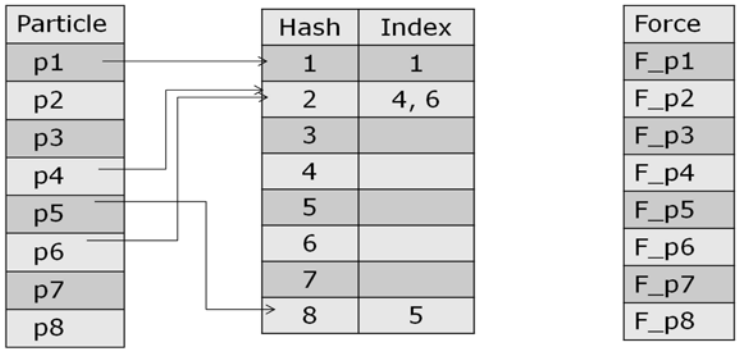


Figure 4.5: CPU implementation data structures.

In the CPU version, we use a "vector" in standard C++ as the structures of *Particle List*, *Neighbour List* and *Force List*, as shown in Figure 4.5 in left, middle and right positions. For each element of Particle List, its position coordinates *x*, *y* and *z* are used to compute the hash value as a mapping between *Particle List* and *Neighbour List.*

In the Neighbour List, column *Index* store indices of all particles with the same local hash value. And the Force List is used to store the total force exerted on a specific particle for the future position & velocity update process. For every iteration, we first updates the Particle List according to the previous Force List, and then, use the new position of every element in *Particle List* to construct a new *Neighbour List*, and store the newly calculated force values in *Force List* one by one. The element from *Particle List* and *Force List* has a 1-to-1 mapping, that is, *p1* in Particle List has it force *F_p1* stored in *Force List* with the same index.

The programming pipelines of CPU and GPU are the same, and we will be introduced in the following section.

4.4 GPU Implementation

This section introduces the implementation detail of the GPU version of SPH.

4.4.1 Thread Assignment

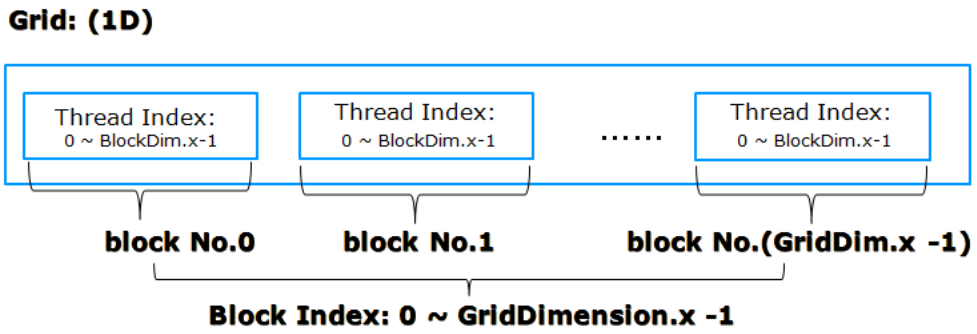


Figure 4.6: Linear thread assignments.

We use a linear assignment of threads in the GPU computation, in which, only the x coordinate of the block index and block dimension are used. The function is:

$$Index = blockIdx.x \cdot blockDim.x + threadIdx.x \quad .$$

As shown in Figure 4.6, each block in the kernel grid has an unique index, which only uses the *x* value: *blockIdx.x*; and within each block, the number of threads equals to block dimension's *x* value, so the thread index value varies from 0 to *blockDim.x*-1.

4.4.2 Hardware Memory & Data Structure

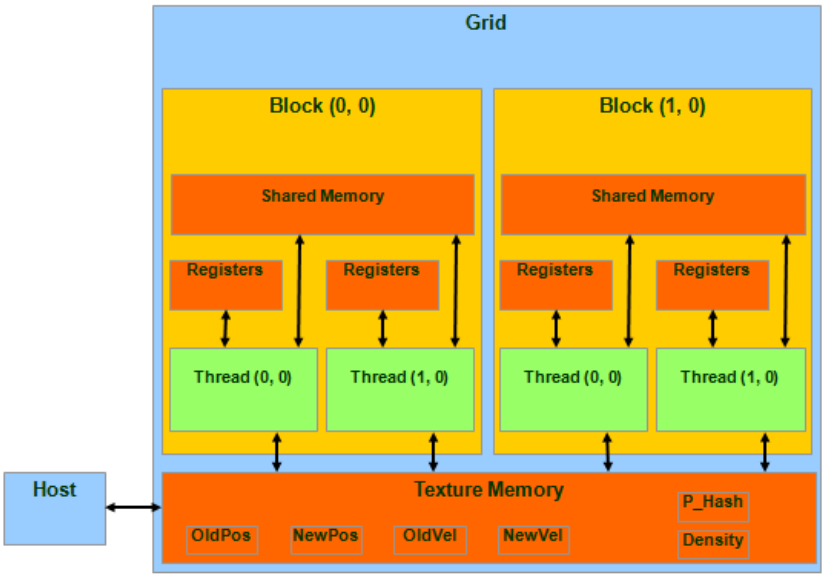


Figure 4.7: Six key dataset in GPU global memory in the form of texture.

As the particles can move to any position in the whole 3D grid, the memory which stores the position and velocity values of the neighbouring particles could be rather randomly distributed. If we use global memory arrays to obtain this data, there will always be un-coalesced. For this reason, we bind the global memory arrays to textures and use texture lookups (*tex1Dfetch*) instead (see Figure 4.7), which improves performance by 45% since texture reads are cached [Green, S., 2008]. As an additional optimization when using the sorting method, we actually re-order the position and velocity arrays into sorted order to

improve the coherence of the texture lookups during the collision processing stage. With the benefit from its design in CUDA SDK, the texture is fast to fetch. And considering the feasibility, a texture could be used because the number of fluid particles in the system stays unchanged during the simulation.

We chose not to use shared memory in the system for SPH because the physical reality of SPH method allows large numbers of fluid particles have the possibility to gather together in some specific time in one dense area, and the size of shared memory may be exceeded by the memory space required by the specific cell's particles; and more over, our demand of flexibility of the project allows the user to adjust the size of each cell in the 3D grid. Large cell length leads to larger number of particles in one cube, which again makes the whole system unreliable. And finally, unlike the global memory, texture memory is cached, so a texture fetch costs one memory read from device memory only on a cache miss. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. With only the texture cache but shared memory, the program's performance is satisfying.

4.4.3 Program Pipeline &Major Kernel Functions

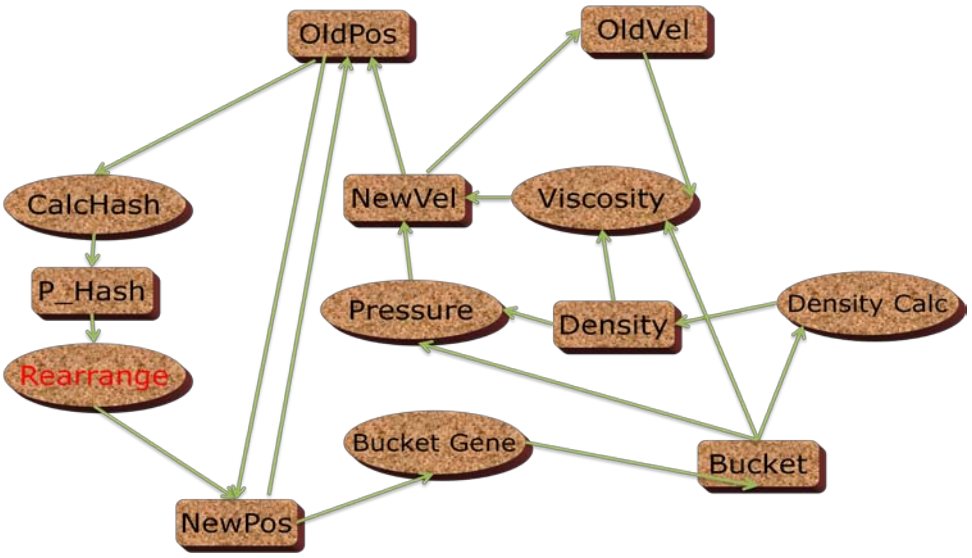


Figure 4.8: SPH programming pipeline.

In the beginning of each loop, as shown in Figure 4.8, kernel function ‘CalcHash’ first fetch the information of the present particle and produces a new position hash table P_Hash. Then a ‘Rearrange’ function is called to arrange the particle-ordered OldPos into cell-ordered NewPos to store the position for all particles. Then on each thread, a unique index of its corresponding particle is assigned and a bucket for this specific particle is built by “Bucket Generator”. On the same thread, the particle’s density value and according pressure force and viscosity force is then calculated. A texture called OldVel storing the previous velocity value for each particle is always updated for the use of computation for viscosity force in next iteration. With the particles’ present velocity (stored in NewVel) and old position (in NewPos), its newest position will be integrated with Runge Kutta 4^{th} order method and inserted into OldPos again. The program keeps running until some pre-fixed time or a QUIT button pressed by users.

4.5 Experimental Results and Analysis

4.5.1 Correctness

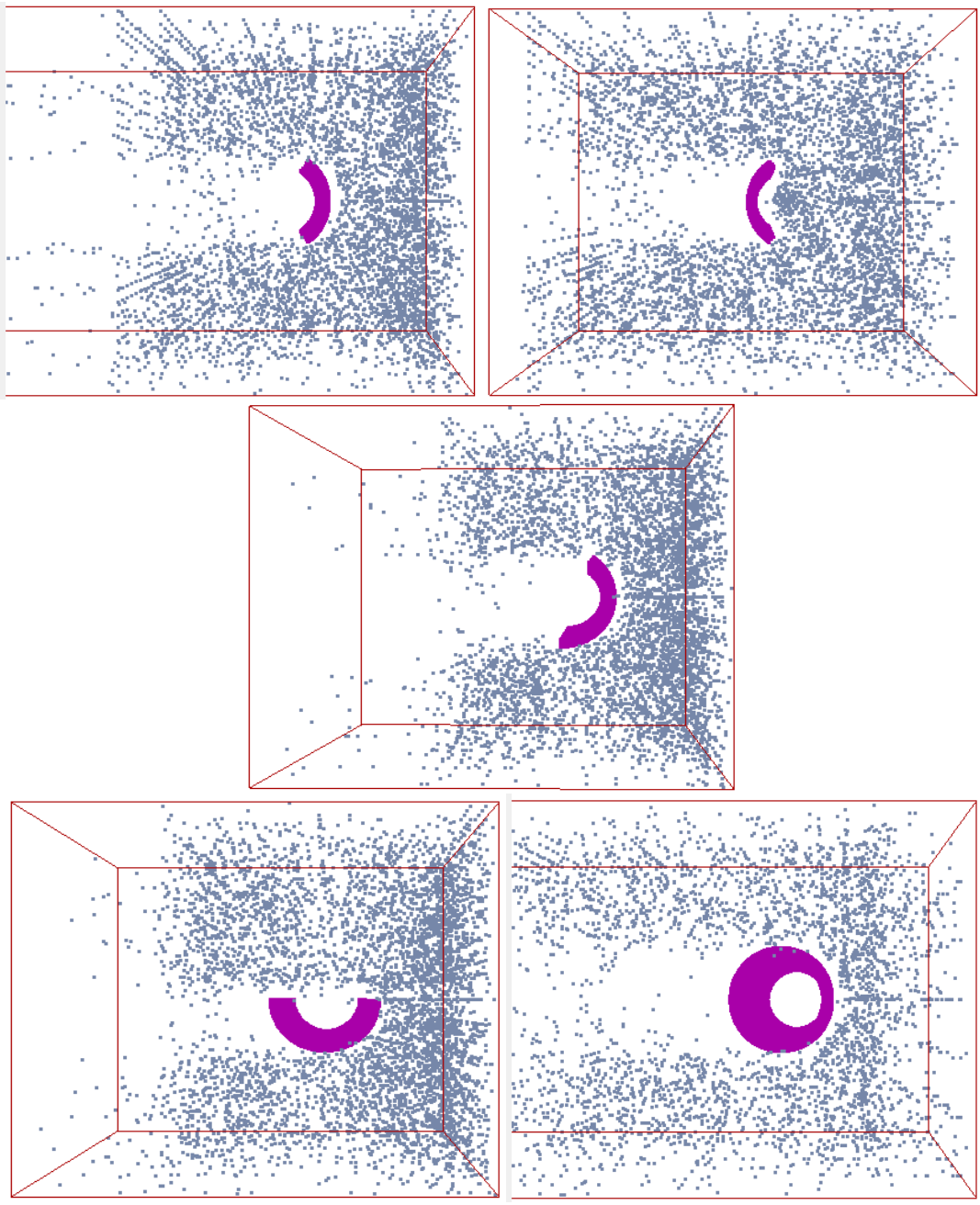


Figure 4.9: Simple CPU implementation for validation.

We firstly produced a simple CPU version with about 2K fluid particles to validate the correctness of the simulation. We lay much stress on the result when fluid particles impinge a solid HEAVY obstacle which is also made by hundreds of solid particles that share all properties with fluid particle but have a huge mass.

As shown in Figure 4.9, we lay the fluid particles, which all have an original velocity to the left, at the right side of the box, and place the obstacle containing only solid particles in the middle of the box. When the difference between fluid particles and obstacle particles gradually reduces to the effective radius of SPH, pressure force and viscosity force is computed and exert according effect on the fluid particles' velocity.

4.5.2 Results

In the official CPU and GPU version of SPH, we experimented both with the same number of particles in three different physical models including the gas explosion, fluid motion within a tank and some wet mud. The only difference lies in the adjustment of comparative portions of attraction force (viscosity) and repulsion force (pressure) in the total force exerted on a specific particle from all neighbouring particles. This could be realized only to change the attraction force, when they overcome the repulsive pressure force, will give all the fluid particles an inner-directed exert which prevent any particle to escape from the group, which looks from outside, more like some viscid mud. On the other hand, when the pressure force becomes the dominating force, particles could hardly get close to each other, and so that all pushed out to some low-density regions.

Images in Figure 4.10 from (a) to (e) shows in time sequence of a group of cube-shaped placed particles to explode in the tank. All boundaries in of cube-shaped tank are

rigid and could rebound any particle rushing onto it with damping of the particle's velocity.

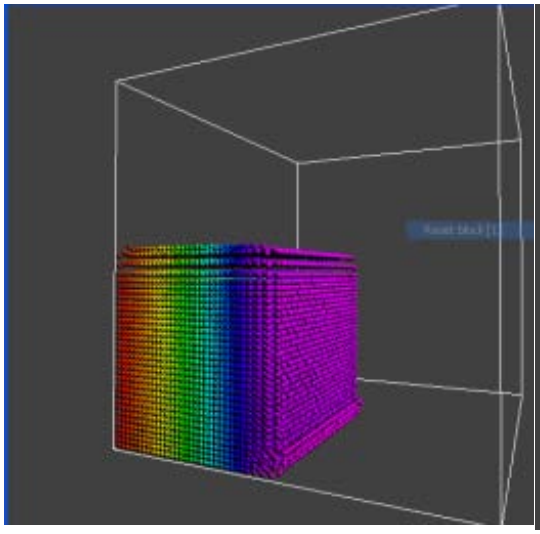


Figure 4.10 (a) Start

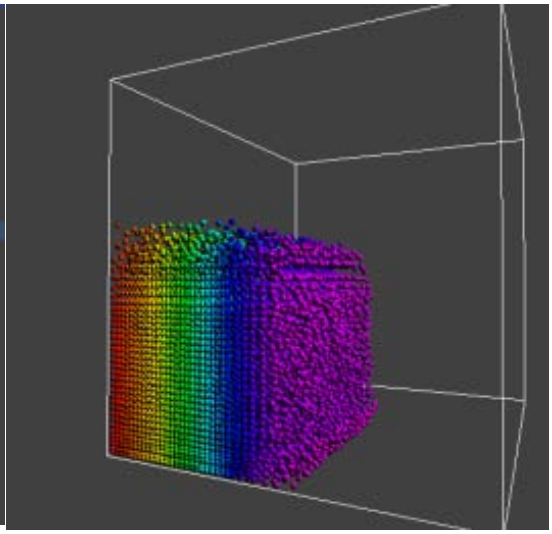


Figure 4.10 (b) 0.8 sec. later

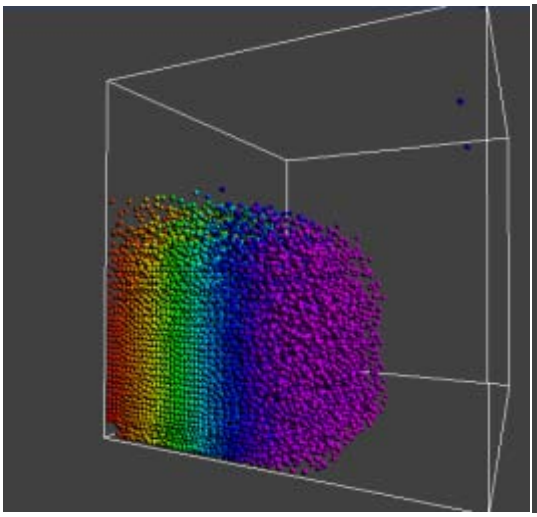


Figure 4.10 (c) 1.5 sec. later

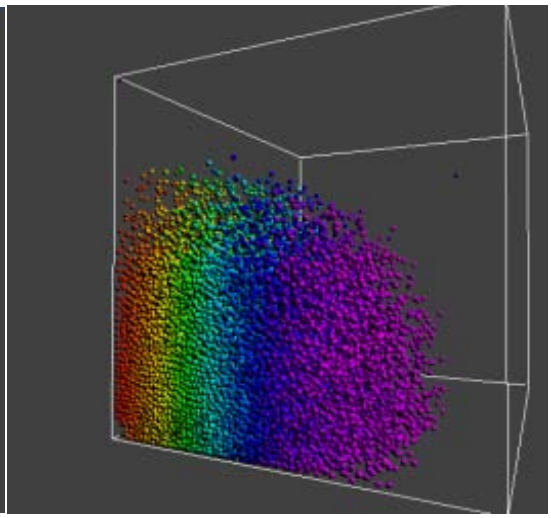


Figure 4.10 (d) 3.0 sec. later

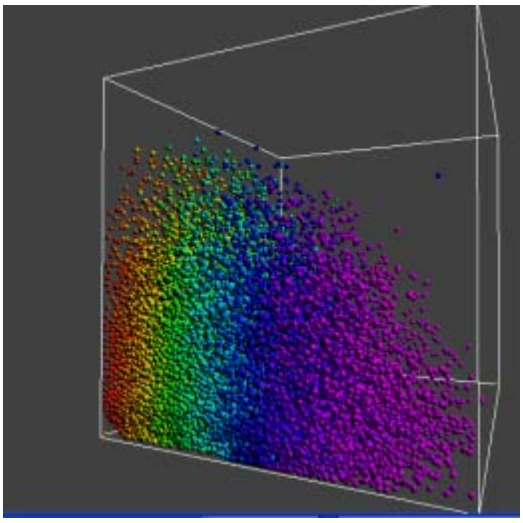


Figure 4.10 (e) 5.0 sec. later

Figure 4.10: Experimental result of an explosion (Dynamic Viscosity Constant $v = 5.37 \times 10^{-9}$, Pressure Constant $k_p = 1.22 \times 10^{-7}$).

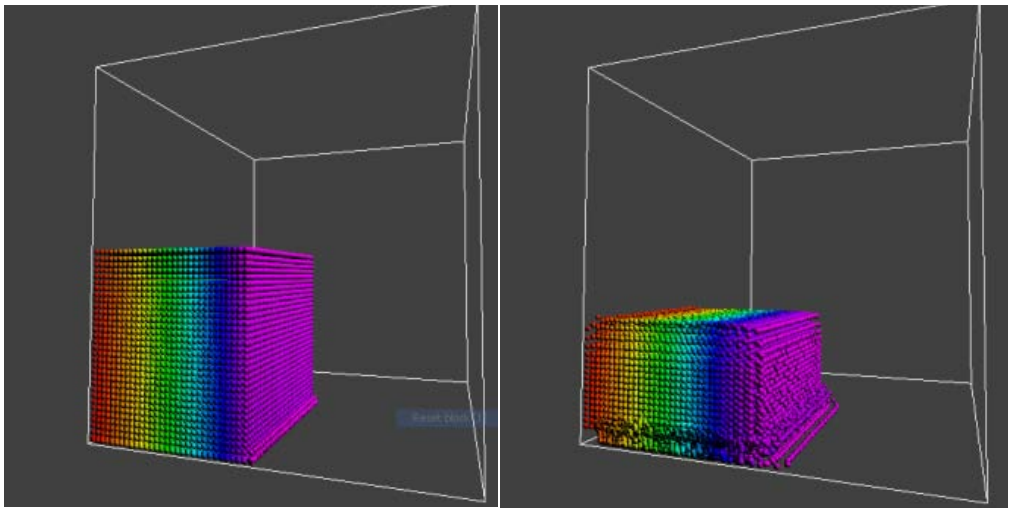


Figure 4.11(a) Start

Figure 4.11 (b) 1.5 sec. later

Similarly, images in Figure 4.11 and 4.12 show that, from two different viewing positions, some fluid motions in a tank after placing a small cube inside the tank. This simulation could be considered as the case that suddenly takes off all six transparent planes of the inside glass cube, and the fluid particles drops to the ground of the tank first,

some may bounce up then interact with other particles which are still falling down. When local density values of the lower layer begin to increase, particles will press some out-layered particles horizontally (see in Figure 4.11(b) for detail).

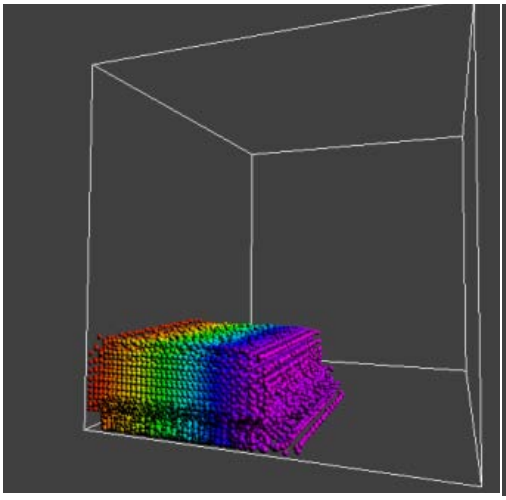


Figure 4.11 (c) 2.5 sec. later

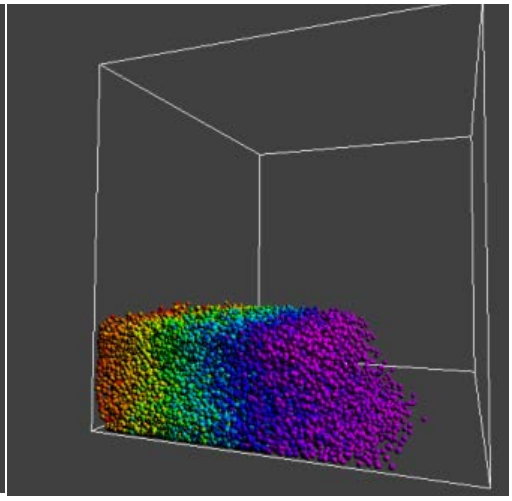


Figure 4.11 (d) 4.0 sec. later

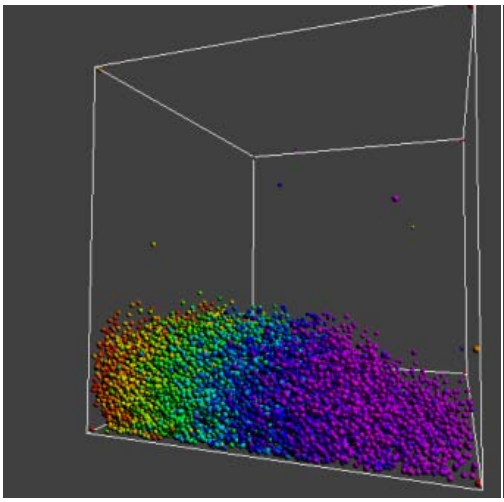


Figure 4.11 (e) 5.5 sec. later

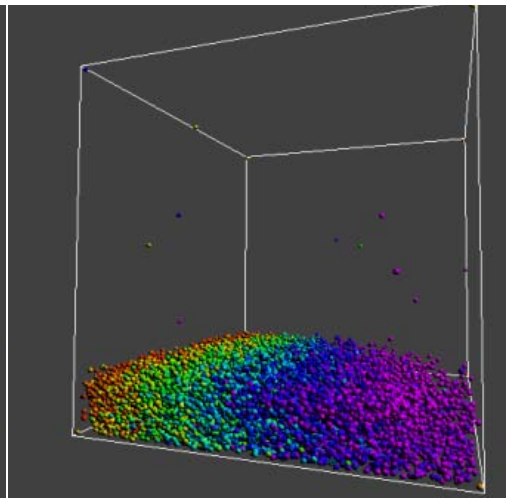


Figure 4.11 (f) 8.0 sec. later

Figure 4.11: Experimental result of fluid motion (Dynamic Viscosity Constant $v = 5.35 \times 10^{-5}$, Pressure Constant $k_p = 1.41 \times 10^{-5}$)

After the rebound damping, continuous exert of the gravity and viscosity force from neighbouring particles which reduces the intensity of relative motion between the

particles, none of the particles could reach their original height again. While the pressure force continues to give horizontal push, the fluid spreads to wider places. As in Figure 4.11 (e), some particles near the boundary corners could be push to the up by a suddenly dense particle cluster, and the rebound from the tank wall.

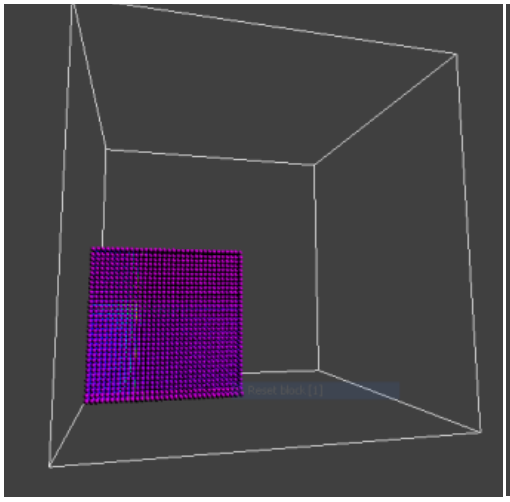


Figure 4.12 (a) Start

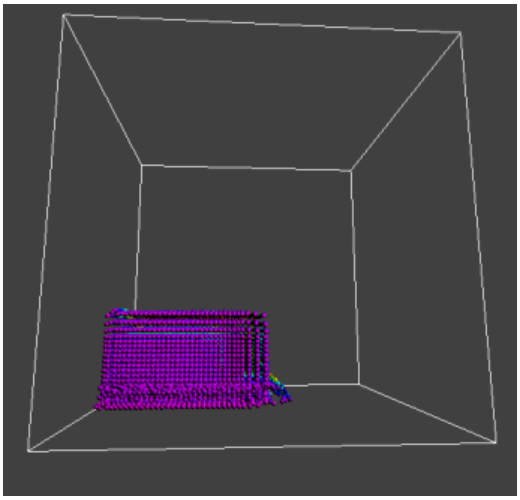


Figure 4.12 (b) 1.5 sec. later

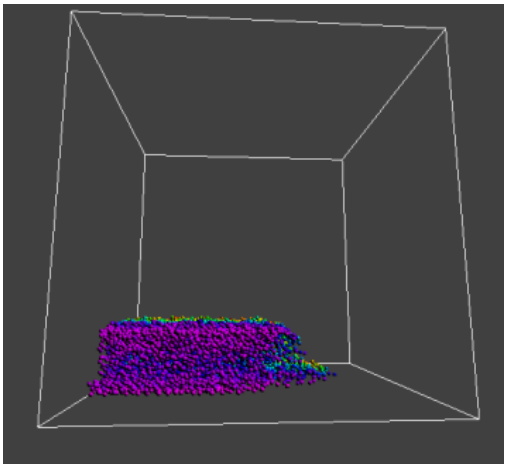


Figure 4.12 (c) 3.0 sec. later

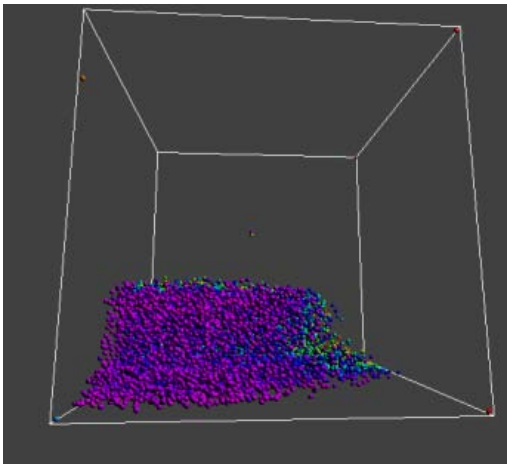


Figure 4.12 (d) 4.0 sec. later

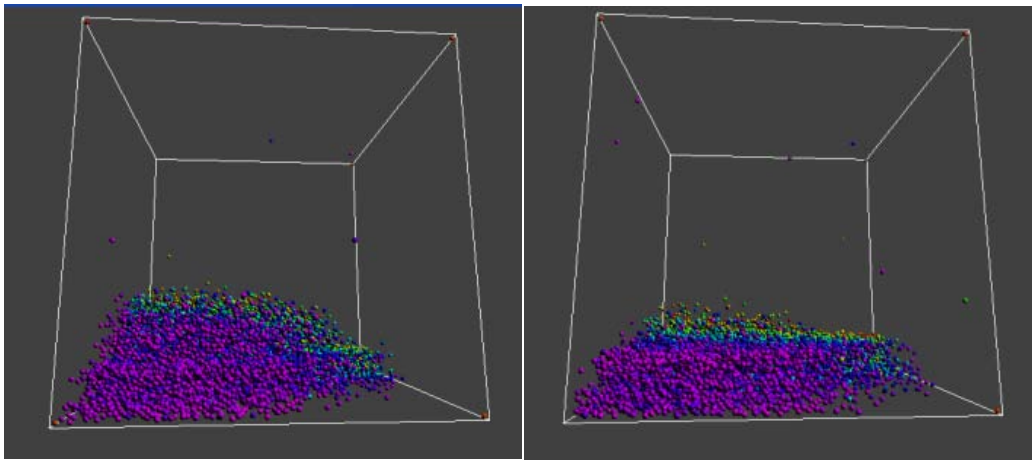


Figure 4.12 (e) 5.5 sec later

Figure 4.12 (f) 7.0 sec. later

Figure 4.12: Experimental Result of Fluid Motion at other Viewing Position (Dynamic Viscosity Constant $v = 5.35 \times 10^{-5}$, Pressure Constant $k_p = 1.41 \times 10^{-5}$)

Figure 4.12 images shows the same simulation but at another viewing direction. From this direction, the result is more apparent and gives the impression that a big dam experiencing a sudden burst of a large water pressure burst. In terms of the comparative portion between attraction and repulsion forces, the fluid motion is the in-between case of explosion and viscous wet mud, in which the viscosity value and pressure influence have a better balance with each other.

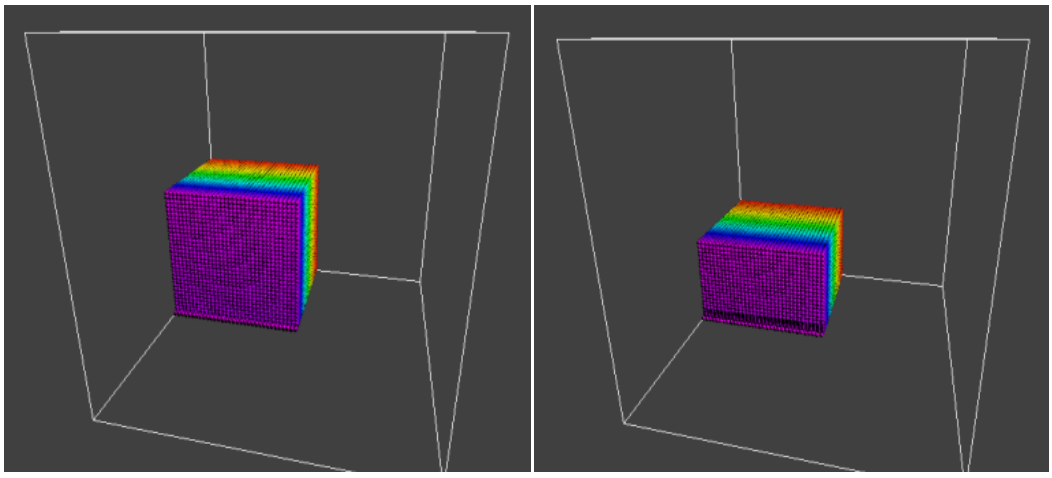


Figure 4.13 (a) Start

Figure 4.13 (b) 1.5 sec. later

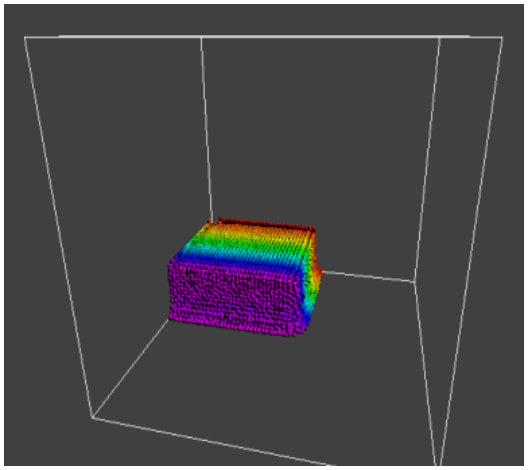


Figure 4.13 (c) 2.5 sec. later

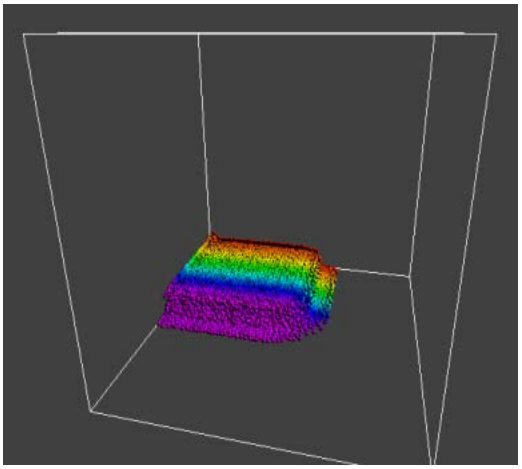


Figure 4.13 (d) 4.0 sec. later

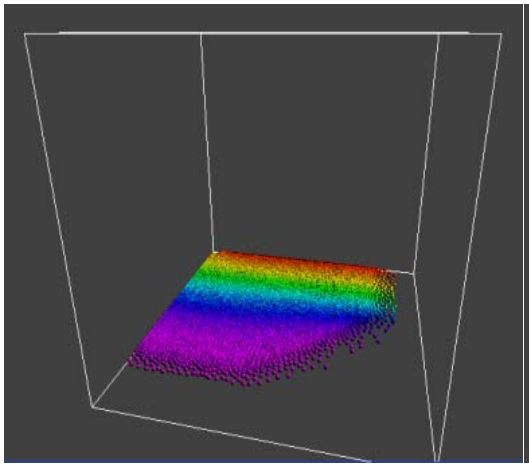


Figure 4.13 (e) 8.0 sec. later

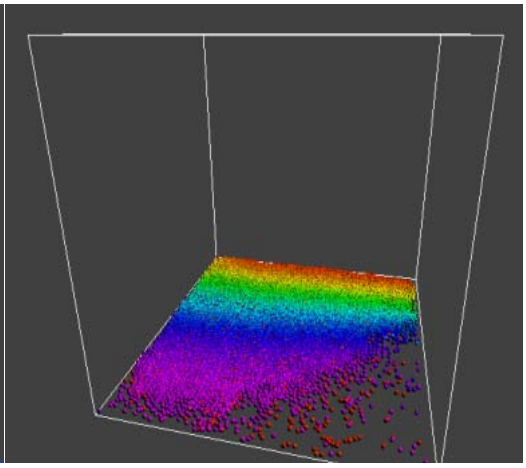


Figure 4.13 (f) 10.0 sec. later

Figure 4.13: Experimental Result of Wet Mud (Dynamic Viscosity Constant $v = 3.73 \times 10^{-4}$, Pressure Constant $k_p = 1.50 \times 10^{-5}$)

Unlike fluid motion and explosion, the rebound of particles in the wet mud model seems unnoticeable, as shown in Figure 4.13 (d), and the reason is quite simple: as a resisting force, the viscosity force, which is comparatively much larger in this model than others, reduced the damped velocity of the rebounding force immediately, and then, with

more particles falling down to the lower layer, local density and viscosity force continues to increase and further contains the upper rebounded particles.

The relation among the particles begins to change when most the particles lay on the ground and has very limited vertical velocity. By pressure force, periphery particles start to move to low-density areas, as shown in Figure 4.13 (e). After a while, with a continuous repulsive force from inside and its relative acceleration onto the outer particles, their velocity increases over than the threshold for attractive force to prevent them from escaping. So as shown in Figure 4.13 (f), some particles ran out of the group region.

4.5.3 Performance Analysis

The code included in the GPU version simulates 32,728 fluid particles on an NVIDIA Quadro FX 5800 at 8.8 frames per second (fps) in the steady state for the fluid motion model, 10.5 fps for the wet mud and 7.2 fps for the explosion model. It uses the sort-based algorithm provided in CUDA SDK. This algorithm improves the coherence of memory access so it is considered currently the fastest method [Green, S., 2008]. The ability of CUDA to perform scattered memory writes makes it possible to build dynamic data structures on the GPU. Sorting can be used to improve memory coherence when accessing these kinds of data structures. This combined with the computational power of the GPU makes it possible to simulate large systems of interacting particles at interactive rates. And the code included in the CPU version simulates the same number of fluid particles in the same environment (with gravity and wall-particle collision) on an Intel Core2 Duo P8400 CPU (2.26 GHz and 2.27 GHz) with Memory (RAM) 3.0 GB.

Table 4.1 shows the experimental results in speed performance: in the first part "CPU Performance", we provide the CPU version of the SPH code in different model and different integration methods (Euler or Runge Kutta), and compare the speed decrease from Euler to Runge Kutta. We offered the similar information of GPU version of the SPH in the "GPU Performance" part in Table 4.1 as well. Finally in the last part, we compared GPU vs. CUP speed performance and calculate the acceleration ratio.

Table 4.1: Performance Comparison

CPU Performance			
	Fluid Motion	Wet Mud	Explosion
CPU + Euler (fps)	0.41	0.31	0.48
CPU + RK (fps)	0.25	0.21	0.35
CPU RK / Euler	61%	67%	73%
GPU Performance			
GPU + Euler (fps)	8.8	10.5	7.2
GPU + RK (fps)	8.6	10.4	7.0
GPU Euler / RK	97%	99%	97%
GPU vs. CPU Speed Comparison			
Euler GPU / CPU	21	34	15
RK GPU / CPU	34	50	20

By comparing the Euler and Runge Kutta integration methods, one could conclude that with the same time step *h*, Runge Kutta method requires more computation

than Euler so that has slower speed in both CPU and GPU version. But the difference affects the speed on CPU version much more than on the GPU version: RK has about 70% of the speed of Euler in CPU, but only has less 3% speed drop in GPU version. So the Runge Kutta method seems more suitable to be used in GPU version of SPH to reduce the integration error.

A comparison between the different fluid models shows that both CPU and GPU versions vary enormously from equally-placed particles searching (such as the Explosion) to unevenly-placed particles searching (such as wet mud in which all the particles are in the same ground floors of cells in steady state). The decrease in fps in the CPU version is due to the fact that unevenly placed particles model has a larger number of neighbours so that the bucket of each particle may contain more neighbouring particles to search; in another word, in the sequential CPU computation, the Wet Mud model provided larger neighbouring density value for every particle, so that the total number of neighbouring particles of all the particles in the system increased considerably, which greatly reduce the performance. In the GPU models the performance of the Explosion model is even worse than Wet Mud. This is because of the "*Rearrange*" step in GPU, which move and rearrange the different memory spaces of *NewPos* according to the newly calculated position values. In Wet Mud, all the particles remains in the bottom floors of the cells in the space, so that very few position values in the *NewPos* texture have to be moved to new places, while the task much heavier in evenly placed particles models. Although for each particle in Wet Mud, the threads in the GPU computation has to deal with more neighbours than Explosion, the difference in this parallel computation is relatively slight

compared to the difference of rearrange position data in texture memory, and so that Wet Mud, in total, exceeded Explosion in GPU result .

The comparison between CPU and GPU is tabulated at Table 4.1, we found that the GPU version is 15 – 34 times faster than the CPU version in different models with Euler method; and with Runge Kutta, the GPU version has even a better performance: 20 – 50 times faster than the CPU version.

Conclusion

In this thesis, we presented real-time techniques to visualize the electron statistical distribution of the hydrogen atom at various energy states and the real-time dynamic fluid simulation or molecules/particles using SPH.

By using a nebula visualization lighting metaphor based on Monte Carlo diffusion, we are able to visualize a point-based electron cloud model on both CPU and GPU. The result successfully shows that the lighting energy distribution in different spatial area does improve the perception of the subtle changes in the electron density distribution. We were able to demonstrate that the GPU version exceeds its CPU counterpart by over 100 times in speed. This is good news for the MASAV project as it is now possible to visualize and explore the electron distribution of the hydrogen atom in real-time.

Faced with the reality that the point-based method fails to describe low density areas because of the threshold of particle placement, and notify its disadvantage in enhancing only interior information, in Chapter 3, we implemented a Ray-Casting Volume Rendering technique that complements this algorithm. By changing lighting strength, the user could easily gain a deep understanding on electron probability distribution in different areas. Because of the same size input, the results of different models have similar speed performance: their processing time varies from 32 to 37 milliseconds, which converts into update speeds between 27 to 31 fps.

Finally, in Chapter 4, we simulated the fluid dynamics of large sets of particles using SPH. By changing parameters of the pressure force and viscosity force, we are able to create three fluid models, including explosive gas, water and wet mud. Experiment

results also show that the Runge Kutta integration method does affect significantly the performance of the CPU version compared to its Euler counterpart. This is bad news as the Runge Kutta method is 100 times more precise than the Euler method. But on the contrary, the GPU version is not affected much by the use of the Runge-Kutta integration method resulting in a high-speed performance with no loss of precision. The acceleration of GPU over CPU varies for different model and different integration methods, but in general, GPU speed performance is 20-50 times faster for SPH than its CPU version.

These experimentations with GPU implementation demonstrate without any doubt the power of this new processor to solve physics problems. It is indeed now possible to create new interface to teach complex physics as if you were performing an experiment in real-time. This is good news for the MASAV project as it is now possible to foresee the development of more complex physical simulator that could illustrate in an intuitive way new development in physics such as the physics of sub-atomic particles like quarks and leptons, and the collision of those particles in the LHC.

The new development of the GPU is truly promising and impressive. This year introduction by NVIDIA of the FERMI processor is a sign of thing to come. With over 3 billion transistors, 512 CUDA cores, error correcting memory and double precision floating arithmetic it is truly a revolution in computing but also a scientific revolution in simulation as contrary to the previous paradigm one can now render and simulate on the same hardware. In addition to the new hardware, this year`s introduction of the NSIGHT development environment makes parallel code development much easier to debug. This is critical for the future as parallel code development is not a trivial thing to do especially with massive datasets like those in physics and engineering.

Reference

1. Amanatides, J., and Fournier, A. (1984). Ray Casting using Divide and Conquer in Screen Space. Proc. Intl. Conf. of Engineering and Computer Graphics, Beijing, China.
2. Amada, T.; Imura, M.; Yasumoto, Y.; Yamabe, Y.; Chihara, K. (2004). Particle-Based Fluid Simulation on GPU. ACM Workshop on General-Purpose Computing on Graphics Processors.
3. APEmille. (2006). 3D Grid. [Online]. Available: URL: [http://apegate.roma1.infn.it/mediawiki/index.php/APEmille_project]
4. Bargteil, A.W.; Goktekin, T.G.; O'Brien, J.F.; Strain, J.A. (2006). A semi-lagrangian contouring method for fluid simulation. ACM Transactions on Graphics 25(1), pp. 19–38.
5. Binney, J.; Merrifield, M. (1998). Galactic Astronomy. Princeton University Press.
6. Carlson, M.; Mucha, P.; Turk, G. (2004). Rigid fluid: Animating the interplay between rigid bodies and fluid. ACM Transactions on Graphics 23(3), pp 377–384.
7. Christensen, P. H. (1997). Global illumination for professional 3D animation, visualization, and special effects. Rendering Techniques '97 (Proceedings of the 8th Eurographics Workshop on Rendering), pages 321–326.
8. Compendium G. (2007) A point in the spherical coordinate system [Online]. Available: URL: [http://www.vias.org/comp_geometry/math_coord_sphere.htm]
9. Danskin, J., and Hanrahan, P. (1992). Fast Algorithms for Volume Ray Tracing. In ACM Workshop on Volume Visualization '92, 91–98.
10. Durand, F.; Drettakis, G.; Puech, C. (1999). Fast and accurate hierarchical radiosity using global visibility. ACM Transaction on Graphics, 18(2) pp. 128 – 170.
11. England, R. (1968) Error estimates for Runge-Kutta type solutions to systems of ordinary differential equations, Research and Development Department, Pressed Steel Fisher Ltd., Cowley, Oxford, UK.
12. Enright, D.; Marschner, S.; Fedkiw, R. (2002). Animation and rendering of complex water surfaces. ACM Transactions on Graphics 21, pp. 721–728.
13. Euclideanspace. (1998). Left and right hand coordinate systems. [Online]. Available: [http://www.euclideanspace.com/maths/geometry/space/coordinates/index.htm]

14. Foster, N.; Metaxas, D. (1997). Controlling fluid animation. Proc. of the 1997 Conference on Computer Graphics International, pp. 178–188.
15. Foster, N.; Fedkiw, R.(2001). Realistic animation of liquids. Proc. of ACM SIGGRAPH, pp. 471–478.
16. Freund, J., and Sloan, K. (1997). Accelerated Volume Rendering using Homogeneous Region Encoding. In Processing IEEE Visualization, ’97, 191–197.
17. Gibson, S.; Nordsieck, K. (2003). The Pleiades Reflection Nebula. II: simple model constraints on dust properties and scattering geometry. Astro- physical Journal, 589: pp. 362.377.
18. Goktekin, T.; Bargteil, A.; O’Brien, J. (2004). A method for animating viscoelastic fluids. ACM Transactions on Graphics 23, pp. 464–467.
19. Gordon, K. (2004). Interstellar dust scattering properties. In A.Witt, G. Clayton, and B. Draine, editors, Astrophysics of dust, ASP conference series.
20. Green, S. (2008). CUDA Particles. CUDA SDK Samples White Paper, 9.
21. HARADA, T.; KOSHIZUKA, S.; KAWAGUCHI, Y. (2007). Smoothed particle hydrodynamics on GPUs. In Computer Graphics International, pp. 63–70.
22. Harris, M.; Baxter, W.; Scheuermann, T.; Lastra, A. (2003). Simulation of cloud dynamics on graphics hardware. Proc. of the SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 92–101.
23. Henyey, L.; Greenstein, J. (1938). The theory of the colors of reflection nebulae. Astrophysical Journal, 88:580.604.
24. Herman, G. T. (2009). Fundamentals of computerized tomography: Image reconstruction from projection, 2nd edition.
25. Irving, G.; Guendelman, E.; Losasso, F.; Fedkiw, R. (2006). Efficient simulation of large bodies of water by coupling two and three dimensional techniques. ACM Transactions on Graphics 25, pp. 812–819.
26. Jensen, H. W. (1996). Global illumination using photon maps. Rendering Techniques '96 (Proceedings of the 7th Eurographics Workshop on Rendering), pages 21–30.
27. Kay, T. L. and Kayjia, J. T. (1996). Ray tracing complex scenes. In Computer Graphics (SIGGRAPH ’86 Proceedings), vol. 20.

28. Kruger, J. and Westermann, R. (2003) Acceleration techniques for GPU-based volume rendering, Proceedings of the 14th IEEE Visualization 2003 (VIS'03), pp. 38.
29. Koshizuka, S. and Oka, Y. (1996). Moving-particle semi-implicit method for fragmentation of incompressible flow. Nucl. Sci. Eng. 123, pp. 421–434.
30. Lacroute, P. (1995). Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, In Proceedings of the SIGGRAPH 94 Conference, pp. 451 – 457
31. Lauterbur, P. C. (1973). Image formation by induced local interactions: examples of employing nuclear magnetic resonance. Nature, 242: 190–191.
32. Levoy, M. (1990). Efficient Ray Tracing of Volume Data. ACM transactions on Graphics 9, 3 (July), 245–261.
33. Le Grand, S. (2007). Broad-phase collision detection with CUDA. GPU Gems 3, Addison Wesley.
34. Magnor, M.; Hildebrand, K.; Lintu, A.; Hanson, A., (2005). Reflection nebula visualization. Proc. IEEE Visualization 2005, Minneapolis, USA (Oct.), pp. 255-262.
35. Monaghan. (1992). J. Smoothed particle hydrodynamics. Annu. Rev. Astrophys. 30, pp. 543–574.
36. Muller, M.; Charypar, D.; Gross, M. (2003). Particle-based fluid simulation for interactive applications. Proc. of Siggraph Symposium on Computer Animation, pp. 154–159.
37. Milstid. (2010). Science with Mr. Milstid. [Online]. Available: URL [http://www.sciencewithmrmilstid.com/category/physics/matter-properties-of-matter/atomic-structure/]
38. NVIDIA. (2007). NVIDIA CUDA programming guide. [Online]. Available: URL [http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf]
39. NVIDIA. (2008). CUDA Programming model overview. [Online]. Available: URL[http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf]
40. Pinfold, James L. (2008). Project proposal: a pilot project to MASAV – visualizing the beginning. [Online]. Available: URL:

[http://web.me.com/jamespinfold/Site/MASAV_project_files/MASAV%20Pilot%20Project.pdf]

41. Premoze, S.; Tasdizen, T.; Bigler, J.; Lefohn, A.; Whitaker, R. (2003) Particle-based simulation of fluids. Computer Graphics Forum 22(3), pp. 401–410.
42. Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. (1992). Numerical recipes in C. The art of scientific computing (Cambridge: University Press 2nd ed.), pp. 704-708.
43. Sapirstein, J. (1995). Advances in the theory of atomic structure. A/P Conference Proceedings, 323: pp. 45–62.
44. Sellgren, K. and Werner, M. (1992). Scattering of infrared radiation by dust in NGC 7023 and NGC 2023. Astrophysical Journal, 400:238.247.
45. Sobierajski, L. (1994). Global Illumination Models for Volume Rendering. PhD thesis, The State University of New York at Stony Brook. Disertation.
46. Subramanian, K. R.; Fussell, D. S. (1990). Applying space subdivision techniques to volume rendering, Proceedings of Visualization ’90, San Francisco, California, pp. 150–159.
47. Spice A. (2010). Color Rubik for the blind [Online]. Available URL: [http://www.yankodesign.com/2010/03/17/color-rubik-cube-for-the-blind/]
48. Thacker, R. (2007). PHYS 3437 – Computational Methods in Physics, lecture 12 [Online]. Available: URL [http://www.ap.smu.ca/~thacker/teaching/3437/lectures.html]
49. Tsien, R.Y.; Waggoner, A. (1995). Fluorophores for Confocal Microscopy. Handbook of Biological Confocal Microscopy, 2nd Ed.; Pawley, J.B., Ed.; Plenum Press: pp. 267–280.
50. Weinhaus, M. and Devarjan, V. (1997). Texture mapping 3D models of real-world scenes, ACM Computing Survey, 29(4), pp. 325 – 365.
51. Witt, A.; Walker, G.; Bohlin, R.; and Stecher, T. (1982). The scattering phase function of interstellar grains: The case of the reflection nebula NGC 7023. Astrophysical Journal, pp. 261.
52. Yagel, R., and Shi, Z. (1993). Accelerated Volume Animation by Space-Leaping. In Processing IEEE Visualization ’93, 62–69.

53. Zuiderveld, K. J.; Koning, A. H.; Viergever, M. A. (1992). Acceleration of ray-casting using 3D distance transforms, Proceedings of Visualization in Biomedical Computing 1992, Chapel Hill, North Carolina, pp. 324 – 335.
54. Zwicker, M.; Pfister, H.; van Baar, J.; Gross, M.H. (2001). Surface splatting. In: Proc. SIGGRAPH, pp. 371–378.