A Fast Fluid Simulator

Using Smoothed-Particle Hydrodynamics

by

Gustavo Figueroa

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved October 2012 by the
Graduate Supervisory Committee:

Gerald Farin, Chair
Ross Maciejewski
Yalin Wang

ARIZONA STATE UNIVERSITY

December 2012

ABSTRACT

This document presents a new implementation of the Smoothed Particles Hydrodynamics algorithm using DirectX 11 and DirectCompute. The main goal of this document is to present to the reader an alternative solution to the largely studied and researched problem of fluid simulation. Most other solutions have been implemented using the NVIDIA CUDA framework; however, the proposed solution in this document uses the Microsoft general-purpose computing on graphics processing units API. The implementation allows for the simulation of a large number of particles in a real-time scenario. The solution presented here uses the Smoothed Particles Hydrodynamics algorithm to calculate the forces within the fluid; this algorithm provides a Lagrangian approach for discretizes the Navier-Stockes equations into a set of particles. Our solution uses the DirectCompute compute shaders to evaluate each particle using the multithreading and multi-core capabilities of the GPU increasing the overall performance. The solution then describes a method for extracting the fluid surface using the Marching Cubes method and the programmable interfaces exposed by the DirectX pipeline. Particularly, this document presents a method for using the Geometry Shader Stage to generate the triangle mesh as defined by the Marching Cubes method. The implementation results show the ability to simulate over 64K particles at a rate of 900 and 400 frames per second, not including the surface reconstruction steps and including the Marching Cubes steps respectively.

DEDICATION

I dedicate this thesis to my wife, Veronica, who never lost faith in me and who gave me the necessary strength to complete my work. Without her patience and sacrifice my research would have been impossible.

ACKNOWLEDGMENTS

This thesis document would not have been possible without the assistance and guidance of several individuals whose valuable contribution and knowledge helped in the preparation and completion of my research.

First, I wish to thank my advisor Dr. Gerald Farin, Professor of Computer Science and Engineering at ASU and co-director of PRISM, for his guidance and encouragement. For the past four years, regardless of his busy schedule, he has always made the time to answer questions and share his extensive and deep knowledge in computer graphics.

Dr. Gregory Nielson, Emeritus Professor from ASU, whose knowledge and dedication inspired me to complete my research. I am grateful to my committee members, Dr. Ross Maciejewski and Dr. Yalin Wang, Assistant Professors at ASU, for reviewing my thesis and their support during this time.

Finally, I would like to thank my friends and colleagues for their relentless and unconditional support and my parents for their encouragement and faith in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In this first chapter, we introduce the reader to the main motivation behind this research and its contribution to the computer graphics community.

1.1 Motivation

Fluid simulation has been a subject of study in computer graphics for several years. Researchers and developers have used a wide range of algorithms and techniques in order to generate high performance fluid simulators. Overall, there exists a wide variety of simulators that range in complexity; from high-end, accurate and time-consuming, to mathematically-simplified and real-time applications. The latter is the focus of this research. There exist several publications and white papers written describing different algorithms for solving the mathematical equations involved with fluid simulators. Most of the solutions proposed by researchers and developers were designed with a specific technology in mind, for example the NVIDIA CUDA framework. The lack of solutions using alternative technologies is the main motivation behind this research. In the following pages, we will discuss one of these alternatives: Direct3D and DirectCompute.

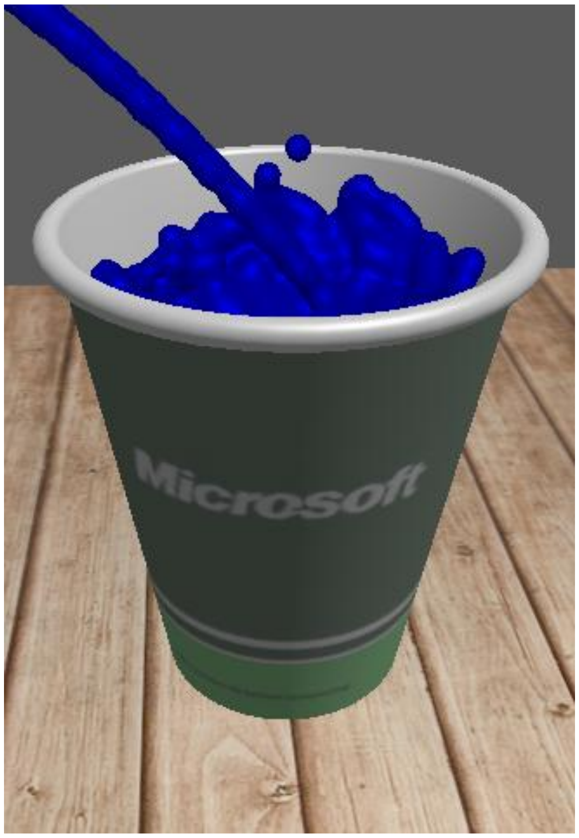

Figure 1. Fluid Simulation using SPH and DirectX

1.2 Objective

The main goal of this research is to provide an alternative solution to the existing fast fluid simulators developed using the NVIDIA CUDA framework. In this document we will present an algorithm and implementation designed using Microsoft Direct3D and the well-known Smoothed Particle Hydrodynamics and Marching Cubes methods. We then show the results obtained using different parameters and settings. These results validate that it is practical and efficient to build real-time fluid simulators using Direct3D as an alternative solution to CUDA.

1.3 Document Outline

The document is organized as follows. Chapter 2 describes other research and implementations used in order to develop fluid simulators and how these relate to this research. Chapter 3 provides a high level description of Direct3D programmable pipeline and DirectCompute. Chapter 4 reviews the Smoothed Particle Hydrodynamics algorithm, its physical properties and mathematical formulations. Chapter 5 provides a high level description of the Marching Cubes method and its implementation details. Chapter 6 introduces the reader to the algorithms necessary to develop a fast fluid simulator. Chapter 7 provides the details of the implementation and results. And Chapter 8 discusses alternative algorithms and methods in order to provide possible improvements to the algorithm presented in this document.

Chapter 2

RELATED WORK

For the last couple of decades, hundreds – if not thousands – of fluid simulators have been developed in computer graphics applications. These applications vary from video games, to water flood simulations, to flight simulation, to bloodstream simulations. Some of the earliest approaches for solving the fluid simulation problem used Eulerian grid-based solutions [1]. For several years this was the most used and well known approach. A few years later, additional algorithms were introduced using semi-Lagrangian algorithms and adaptations of the Eulerian solution proposed by Foster and Metaxas. It was in 2003 when Mathias Müller [2] introduced the Smoothed Particles Hydrodynamics (SPH) algorithm into the fluid simulation computer graphics world. It has since then been used in a countless number of research papers and applications. The use of Lagrangian solutions enabled scientists and researchers to implement a wide range of new solutions.

In the last few years, the introduction of General-Purpose Computation on Graphics Processing Units (GPGPU, or also referred to as simply GPU) has increased the research of fluid simulation. The ability to transform almost any PC into a powerful machine capable of performing millions of float-point operations per second (flops) has created a lot of attention in the research community.

In 2007 NVIDIA introduce CUDA to the world of computer graphics, and with it, a new playground for researchers. With the ability to generate general-purpose shaders, it is now possible to speedup solutions in an unprecedented way (see Figure 2).

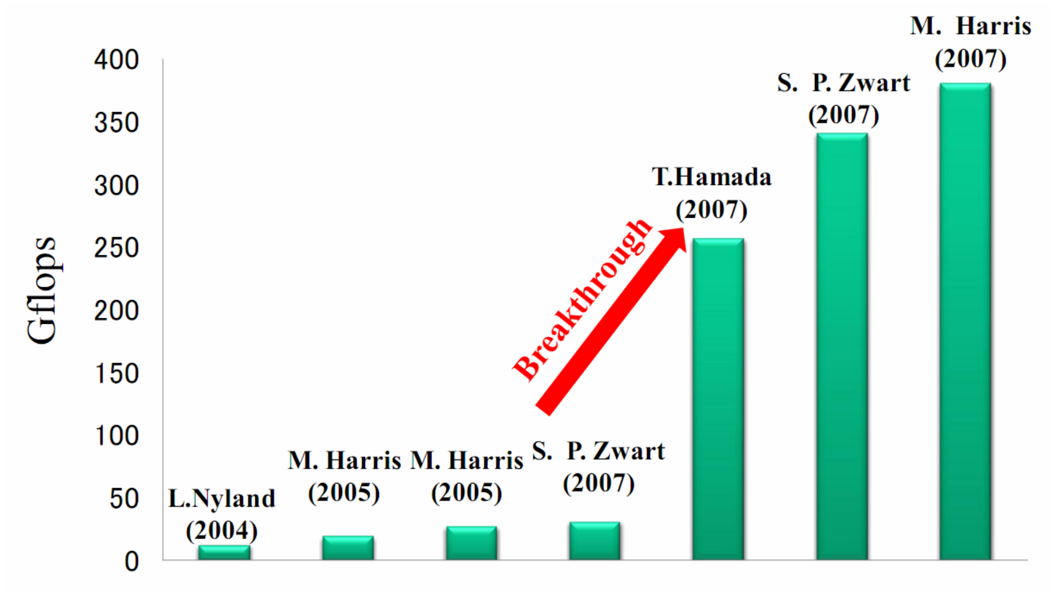

Figure 2. GPU N-body speedup timeline (Berczik, 2008)

Several new implementations and algorithms have been developed using CUDA and SPH, including research documents like “Hybrid Smoothed Particle Hydrodynamics,” which exposes a new algorithm using “a Poisson solve on a coarse grid to enforce a divergence free velocity field, followed by a local density correction of the particles” [3]. The solution presented a set of results ranging from rendering 480,000 particles in 326 minutes to 200,000 in 146 minutes. Another document includes “Interactive SPH Simulation and Rendering on the GPU,” which exposes a new method for the neighbor particle search using the "Z-indexing and parallel sorting which eliminates GPU memory overhead due to grid or hierarchical data structures" [4]. In this case, the results range from rendering 16,128 particles at 28 fps (frames per second) to 255,600 particles at 3 fps. Finally, the research document "A SPH-based method for interactive fluids simulation on the multi-GPU" presents an approach for using multiple GPUs in parallel to resolve the SPH equations [5]. The solution presented in this document shows speedup results over single-GPU solutions of up to almost 3.5 times.

More recently, Microsoft introduced DirectCompute with the latest release of Direct3D. Although CUDA continues to be the most popular framework for fluid simulation and research, this document focuses on the technology developed by Microsoft as an alternative solution. As described in the following sections, the algorithm proposed here uses the Compute, Vertex, Geometry and Pixel Shaders exposed by this API.

Due to the early adoption of CUDA as the framework to resolve computer graphics problems such as fluid simulation, there is almost no research documents discussing the implementation of Direct3D and SPH and/or Marching Cubes. However, in 2011 Kristoffer Lindstrom authored a white paper discussing the implementation of the Marching Cubes algorithm using Direct3X's DirectCompute [6]. In the document he outlines the benefits (and drawbacks) of using DirectCompute and the results of his research.

Chapter 3

DIRECT3D 11

Direct3D is a Microsoft DirectX API subsystem component available on Windows operating systems (Windows 95 and above). Its most important feature is the ability to allow applications to render three dimensional graphics in a high-performance environment. There have been several releases and updates, each one of them adding functionality that allows programmers to take advantage of the latest improvements on hardware, particularly on GPUs. Direct3D 11 was released in 2008 and included several major improvements and additional functionality over its previous versions. A full list of features can be found online at “Programming Guide for Direct3D 11” [7].

The algorithm presented on this document was designed with Direct3D11 in mind only. It specifically uses the new functionality available through DirectCompute and the Vertex, Geometry and Pixel Shader Stages (see Figure 3). The latter were originally introduced within Direct3D 10; however, there are some elements that were improved and added for version 11, such as allowing a higher number of threads and performance improvements, which are essential for a successful implementation of this algorithm.

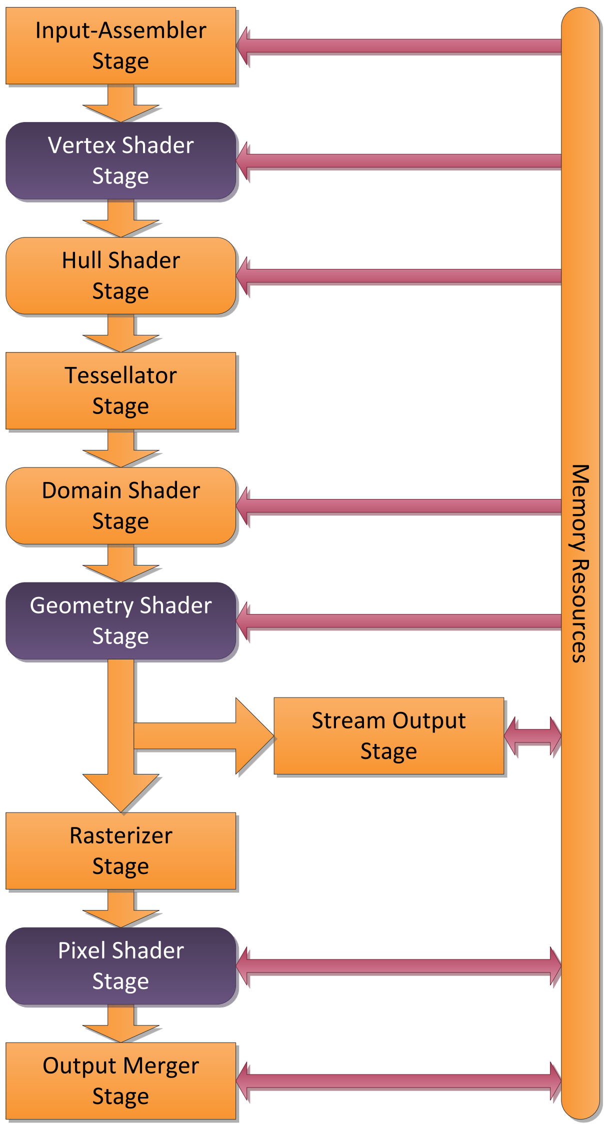

Figure 3. Direct3D 11 Programmable Pipeline

3.1 Direct Compute

One of the critical pieces this algorithm uses is the Direct3D 11 DirectCompute technology, sometimes also referred to as ‘compute shader technology’. It allows for general-purpose computing on graphics processing units [8]. That is, instead of using the GPU to handle computer graphics, DirectCompute performs computational operations on GPUs that otherwise would have been traditionally performed on CPUs. Although GPUs are limited to a range of operations and data structures due to their nature, DirectCompute has become a powerful and important tool when developing high performance solutions. Programmers can now migrate some of the highly intensive mathematical operations from the CPU to the GPU, so that such operations can be handled in a multithreaded manner and more efficient hardware architecture for such operations.

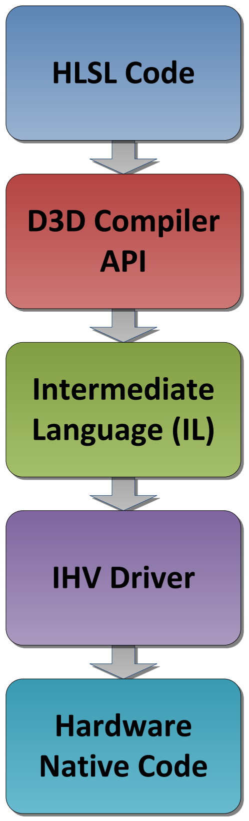

Figure 4. DirectCompute Compilation Steps (image by MSDN, 2010)

DirectCompute is not the only technology that provides this functionality, it actually “shares a range of computational interfaces with its competitors – the Khronos Group’s Open Computing Language (OpenCL) and NVIDIA’s Compute Unified Device Architecture (CUDA)” [8]. However, the algorithm being presented on this document focuses only on Direct3D and its components.

3.2 Vertex Shader Stage

The Vertex Shader (VS) stage is the first programmable interface in the Direct3D pipeline. The main feature of this stage is that it processes a single input vertex and produces a single output vertex. Also, “this stage must always be active for the pipeline to execute. If no vertex modification or transformation is required, a pass-through vertex shader must be created and set to the pipeline” [9]. This stage allows for multiple operations, including but not limited to per-vertex transformations and lighting; these two being the most common ones.

3.3 Geometry Shader Stage

The Geometry Shader (GS) stage is executed immediately before the rasterization stage (originally introduced in Direct3D 10). Its most important feature is the ability to work with multiple vertices as input (three for triangles, two for lines, and one for points), and generate additional vertices. This shader “is capable of outputting multiple vertices forming a single selected topology (GS stage output topologies available are tristrip, linestrip, and pointlist)” [9]. The number of primitives generated within the shader can vary according to the implementation; however, the maximum number of vertices must be specified during implementation and cannot be updated dynamically (there exists no limit for the maximum number of vertices) [9]. Another important feature is the ability to choose different topology types for the input and output parameters; for example, the shader can receive a list of points as input, and produce a list of triangles as output.

3.4 Pixel Shader Stage

The Pixel Shader (PS) stage is the last programmable interface. It allows for “rich techniques such as per-pixel lighting and post-processing” [9]. After rasterization, this shader is executed once per pixel covered by a primitive. As opposed to the VS, this shader can be set to ‘NULL’ (no shader) in order to avoid running this stage. The input to this stage can vary depending on the requirements; however, the output of this shader can only be a ‘color’ output (or an additional depth-test value). This data is then evaluated at the ‘Output Merger Stage’ to determine if the color is written to the depth buffer or it is discarded [9].

Chapter 4

SMOOTHED PARTICLE HYDRODYNAMICS

Smoothed Particle Hydrodynamics (SPH) is a computational method originally developed by Gingold and Monaghan (1977) and Lucy (1977) to simulate and resolve astrophysical problems. In 2003 Matthias Müller [2] used this method to render fluids in interactive applications. He used the SPH algorithm to develop an alternative solution for the fluid simulations systems as opposed to the grid-based solutions that existed at the time. Since then it has been adopted as one of the most common approaches for creating fluid simulators.

SPH is a mesh-free Lagrangian method which works by dividing the system into discrete set of atomic elements normally referred to as particles. Each one of these particles has physical properties such as mass, position, velocity and acceleration. There are several advantages to using this method compared to the alternative solutions that use grid-based numerical methods for solving the problem. As described by Liu & Liu, the most significant advantage is the adaptive nature of the SPH method. “This adaptability of SPH is achieved at the very stage of the field variable approximation that is performed at each time step based on a current local set of arbitrarily distributed particles. Because of the adaptive nature of the SPH approximation, the formulation of SPH is not affected by the arbitrariness of the particle distribution” [10].

The SPH method discretizes the fluid into a set of particles, each one having a set of physical quantities thus defining the fluid. The algorithm computes these quantities' values by evaluating the properties of the neighboring particles. The contribution of each neighbor particle is weighted based on its distance from a given particle. This distance is known as the smoothing length (or core kernel radius), represented by h in the equations (1) – (3) and (10) – (17).

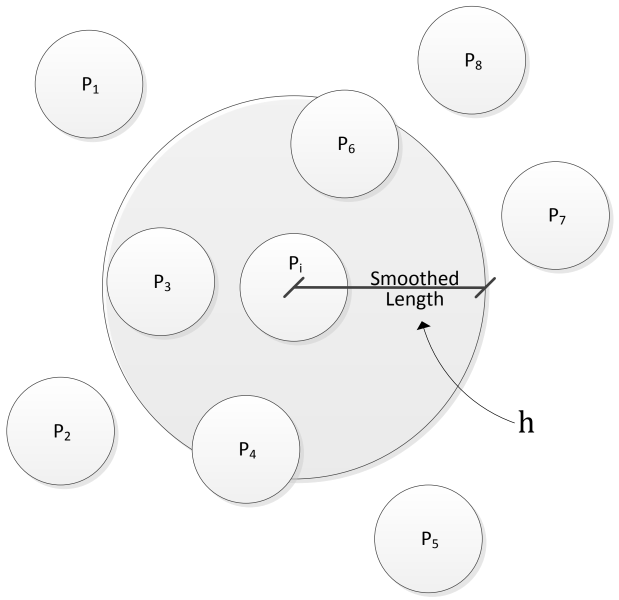

Figure 5. Fluid Particles and the Smoothed Length

The Smoothed Particle Hydrodynamics method allows evaluating only those particles within a specific area, regardless of the particle position. Basically, it "distributes quantities in a local neighborhood of each particle using radial symmetrical smoothing kernels" [10]. So in order to calculate each particles' physical scalar quantities at location r, SPH defines the following formula:

$$A_s(r) = \sum_i m_i \frac{A_i}{\rho_i} W(r - r_i, h) \tag{1}$$

where m_i is the mass of particle i, ρ_i is its density, h is the smoothing length, A_i is the field quantity at r_i, and the function $W(r - r_i, h)$ is the smoothing kernel with smoothed length h. Also, the kernel function must be even and normalized in order to accomplish a second order accuracy; that is:

$$\int W(r)\, dr = 1 \tag{2}$$

$$W(r, h) = W(-r, h) \tag{3}$$

As it will be described next, it is critical to select the appropriate kernel function for each scalar quantity. For the purpose of the algorithm outlined in this document, we implemented the kernel functions Müller suggested in his 2003 research document [2].

4.1 Forces

It is common to use computational techniques using four types of forces: attractive, repulsive, damping and external. In SPH, it is assumed that the first three forces act within the fluid. As described in Müller's document [2], we can start analyzing these internal forces using the equations used in the Eulerian solutions:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0 \tag{4}$$

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla p + \mu \nabla^2 v + \rho g \tag{5}$$

where v is the fluid velocity, ρ is the density field, p is the pressure field, $-\nabla p$ is the fluid force, $\mu \nabla^2 v$ the fluid viscosity, and ρg the external forces. Here, Eqn. (4) assumes conservation of mass and Eqn. (5) (a simplified version of the Navier-Stokes equations for incompressible fluids) assumes conservation of momentum.

Due to the nature of SPH, these equations can be greatly simplified. First, conservation of mass is guaranteed since the number of particles and their masses are constant. Then, since the particles compose the fluid, they move with the fluid, and so the convective term $v \cdot \nabla v$ is not needed in Eqn. (5). Therefore, we get:

$$\rho\left(\frac{\partial v}{\partial t}\right) = -\nabla p + \mu \nabla^2 v + \rho g \tag{6}$$

The left hand side of Eqn. (6) represents the force density fields of the SPH method. We can rewrite this equation by first defining the acceleration as:

$$a = \frac{\partial v}{\partial t} \tag{7}$$

and the summation of the force fields as:

$$f = -\nabla \mathrm{p} + \mu \nabla^2 v + \rho g \tag{8}$$

Thus we get the acceleration of particle i as:

$$a_i = \frac{f_i}{\rho_i} \tag{9}$$

The next section describes the steps necessary in order to calculate the different force density fields of f_i.

4.2 Density Force Fields

As defined in Eqn. (8), there are three density force fields values that we must calculate in order to compute the acceleration of each particle in the fluid. The following sections describe in detail each one of them and their mathematical formulation.

For the purpose of the algorithm and implementation described later on this document, we will be using the kernel functions described by Müller [2]. These kernels have been adopted by the graphics community due to their properties and stability as the distance between particles varies.

4.2.1 Pressure

The first density force field is pressure; within the fluid system, this term belongs to the internal repulsive forces. In the SPH Method, it is necessary to calculate each particle pressure and density at the particle location prior to the fluid pressure being calculated. This is achieved using the following formulas:

$$p_i = k\,(\rho_i - \rho_0) \tag{10}$$

$$\rho_i = \sum_j m_i W(r_i - r_j, h) \tag{11}$$

$$W_p(r,h) = \left(\frac{315}{64\pi h^9}\right)\begin{cases}(h^2 - r^2)^3, & 0 \le r \le h \\ 0, & otherwise\end{cases} \tag{12}$$

where p_i and ρ_i are the particle pressure and density at particle location, r_i is the particle position, k and ρ_0 are the gas constant and rest density from the ideal gas state equation respectively, and h is the kernel core radius. Eqn. (12) is the pressure's smoothing kernel.

Once the particle pressure at location r is calculated, it is necessary to calculate the fluid's pressure density force field. This is achieved using the following formulas:

$$-\nabla p_i = -\frac{1}{2}\sum\nolimits_j m_j\left(\frac{p_i + p_j}{\rho_j}\right)W\left(r_i - r_j, h\right) \tag{13}$$

$$W_s(r,h) = \left(\frac{15}{\pi h^6}\right)\begin{cases}(h-r)^3, & 0 \le r \le h\\ 0, & otherwise\end{cases} \tag{14}$$

where Eqn. (14) is know as the Desbrun spiky kernel [2]. It allows the fluid to generate the necessary repulsion force to avoid building particle clusters when pressure is high.

4.2.2 Viscosity

The second density force field is viscosity; it belongs to the damping forces of the fluid. This field is "caused by friction and, thus, decreases the fluid's kinetic energy by converting it into heat" [10]. In the SPH method, this value can be calculated as follows:

$$\mu\nabla^2 v = \mu\sum\nolimits_j m_j\left(\frac{v_j - v_i}{\rho_j}\right)W^2\left(r_i - r_j, h\right) \tag{15}$$

$$W_v(r,h) = \left(\frac{15}{2\pi h^3}\right)\begin{cases}-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1, & 0 \le r \le h\\ 0, & otherwise\end{cases} \tag{16}$$

with the property:

$$W_v{}^2(r,h) = \left(\frac{45}{\pi h^6}\right)(h - r) \tag{17}$$

where μ is the fluid viscosity, m_j the particle mass, v_j is the particle velocity, v_i is the velocity of the particle being analyzed, ρ_j the particle velocity, r_i is the particle position, and W is the kernel function. Here Eqn. (16) and Eqn. (17) represent the viscosity's kernel functions.

4.2.3 External Force

The last force in the formula represents all external forces applied to the fluid; these are simply accumulated and added to the total particle force f_i. These forces typically represent gravity, objects moving through the fluid, collisions, and other effects.

4.3 Integrator Method

As described above, we will use SPH to calculate three different physical scalar quantities per particle: density, pressure, and viscosity. Each one of these fields will be used to calculate the particles' acceleration, and subsequently, their velocities and positions. In order to accomplish this, we will use the Leapfrog integration method [14], a second-order accurate numerical solution. The Leapfrog integrator offers two main benefits, its time-reversibility (that is, the ability to "integrate n steps and then reverse the direction of the integration and integrate backward n steps to arrive at the same position" [14]) and its strength. The latter implies that the method conserves the energy of the dynamical system (the biggest benefit over other higher-ordered integrators) [14].

The Leapfrog integrator “is a simple method for numerically integrating differential equations of the form:”

$$\ddot{x} = F(x) \tag{18}$$

In the case of velocities and positions, this function takes the form of:

$$x_i = x_{i-1} + v_{i-1/2}\Delta t \tag{19}$$

$$a_i = F(x_i) \tag{20}$$

$$v_{i+1/2} = v_{i-1/2} + a_i\Delta t \tag{21}$$

where x_i is the position at step i, $v_{1+1/2}$ is the velocity, or first derivative of x at step $i + 1/2$, $a_i = F(x_i)$ is the acceleration, or second derivative of x at step i, and Δt is the size of each time-step. It is critical for Δt to be constant on each time-step in order to maintain the stability of the system [14].

Chapter 5

MARCHING CUBES

Marching Cubes (MC) is one of the most popular and used algorithms in computer graphics for surface reconstruction. It was developed by William E. Lorensen and Harvey E. Cline and published in the 1987 SIGGRAPH conferences.

The purpose of MC is to “extract a polygonal mesh of an isosurface from a three-dimensional scalar field” [11]. The basic principle “is to subdivide space into a series of small cubes. The algorithm then instructs us to ‘march’ though each on the cubes testing the corner points and replacing the cube with an appropriate set of polygons” [12]. The set of polygons generated through this method is what we know as the isosurface, which is an approximation of what the three-dimensional scalar field describes.

It is important to note that several new approaches have been developed in order to resolve some of the issues and drawbacks associated with the original Marching Cubes algorithm; these other algorithms can be used within our solution, and we can measure their impact on image quality and performance. Some of these alternative solutions are addressed later on this document.

Marching Cubes was selected as the method to generate the isosurface from the particles of the fluid due to its nature. We can easily translate the fluid information into a grid based data field and manipulate its topology during the Geometry Shader Stage of the Direct3D pipeline. The MC scalar field (data grid) is created by using the density values of each particles and their “influence” on the vertices of the grid cubes. These values are then used to calculate the isosurface based on the iso-value specified by the user. More implementation details are explained in the following sections.

Mark Blackburn gave a presentation based on the work by Lorenson and Cline’s [11] work in which he described the Marching Cubes algorithm in three basic steps:

1. Locate the surface corresponding to a user-specific value (iso-value)
2. Create triangles
3. Calculate normal to the surface at each vertex

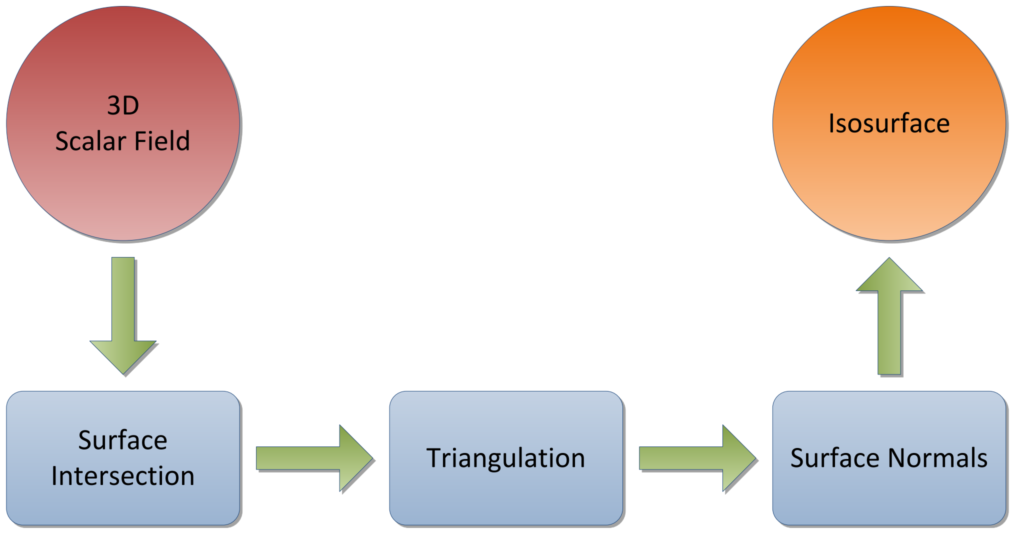

Figure 6. Marching Cubes Basic Steps

5.1 Step 1: Surface Intersection

The first step in the Marching Cubes algorithm is accomplished by interpreting the scalar field as a collection of points, each one represented by a field value. The algorithm first transverses the entire grid using each vertex and its seven neighboring values in order to generate a virtual cube (see Figure 7). Then the algorithm analyzes each cube and its vertex values in reference to an arbitrary value normally referenced to as iso-value.

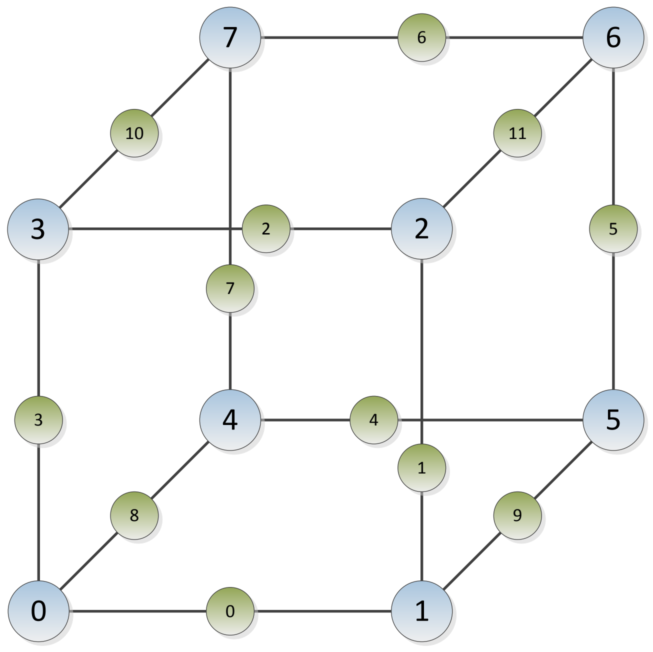

Figure 7. MC Virtual Cube: Vertices (blue) & Edges (green) (image by Mark Blackburn, 2005)

Then, in order to find the isosurface within the cube, the algorithm assigns a ‘one’ if the value of the vertex is greater or equal to the iso-value (inside); otherwise it assigns a ‘zero’ to the given vertex (outside). If an edge contains one vertex inside and one outside the surface, then it is said that the edge intersects the Isosurface.

5.2 Step 2: Triangulation

The Marching Cubes' second step uses as input the edge information computed in the first step. Now, since each cube has eight vertices, and each can be inside or outside, then we know that there are 2^8=256 possible combinations. However, due to its symmetry, these combinations can be reduced to only 15 different combinations (greatly simplifying the lookups). In this step, a set of triangles are generated within each cube (see Figure 8) using the interpolated values between the vertex "inside" and the one "outside."

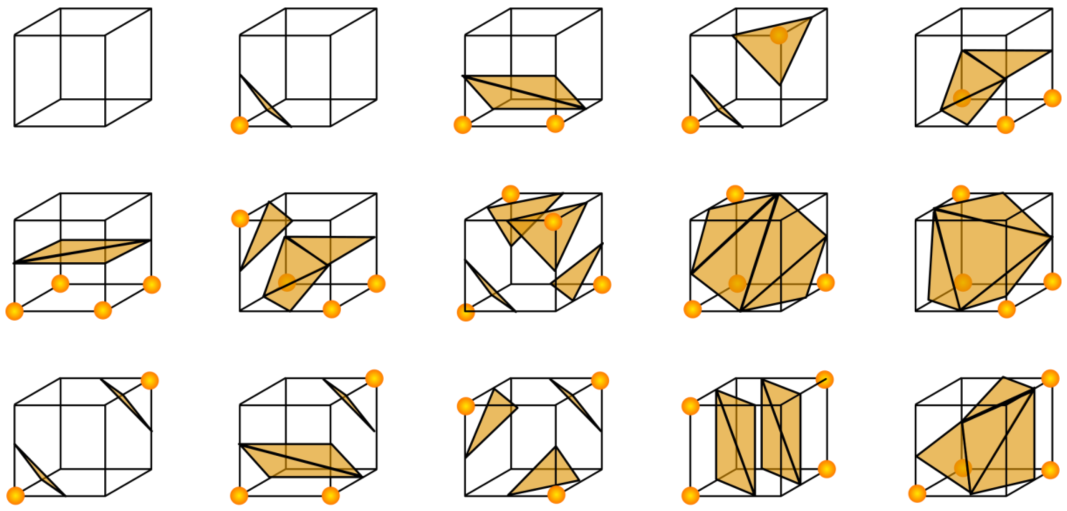

Figure 8. The original published Marching Cubes configuration (image by Wikipedia: Marching Cubes, 2012)

The process of calculating the intersections of the surface and the grid vertices by computing an interpolated value is referred to as ‘Adaptive Marching Cubes.’ This approach provides a better approximation of the original scalar field surface. In this case, the interpolated values will be “closer” to the actual intersection between the surface and the grid vertices than selecting simply the midpoint or any other arbitrary value along the edge. It is also important to point out that, although the algorithm does not generate any edge connectivity information, the connectivity between triangles is preserved due to the cubes being selected as shown in Figure 8; thus also guaranteeing the isosurface mesh connectivity 0 .

5.2 Step 3: Surface Normals

The last step of the Marching Cubes algorithm is to calculate the normals of the triangles. This step is required for rendering purposes; in order to be able to apply lighting and other effects. According to the original Marching Cubes algorithm presented by Lorenson and Cline, this step can be achieved by calculating first the gradient vector $\vec{g}$ as a derivative of the density function. As described in their paper, “to estimate the gradient vector at the surface of interest, we first estimate the gradient vectors at the vertices and interpolate the gradient at the intersection” [11]. Where the gradients are calculated as follows:

$$G_x(i,j,k) = \frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x} \qquad (22)$$

$$G_y(i,j,k) = \frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y} \tag{23}$$

$$G_z(i,j,k) = \frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z} \tag{24}$$

where $D(i,j,k)$ is the scalar field value at location (i,j,k), and Δx, Δy, Δz are the lengths of the cube edges. And the normal then is obtained by normalizing the gradient. Finally the algorithm interpolates the normal at the cube vertices to the intersection point found in the previous step.

Chapter 6

ALGORITHM

The algorithm described in this document can be broken into multiple blocks and grouped based on their functionality as follows: Initialization, Fluid Setup, SPH Calculations, and Surface Reconstruction (see Figure 9). With the exception of the Initialization block, all the blocks are executed in the GPU on a per-frame basis. The Initialization block is executed on the CPU only once during the entire execution of the application.

This algorithm was developed with only Direct3D 11 and DirectCompute in mind. As such, the blocks described next are not meant to be implemented using any other technologies; although, it is possible to translate them based on other technologies functionality and requirements.

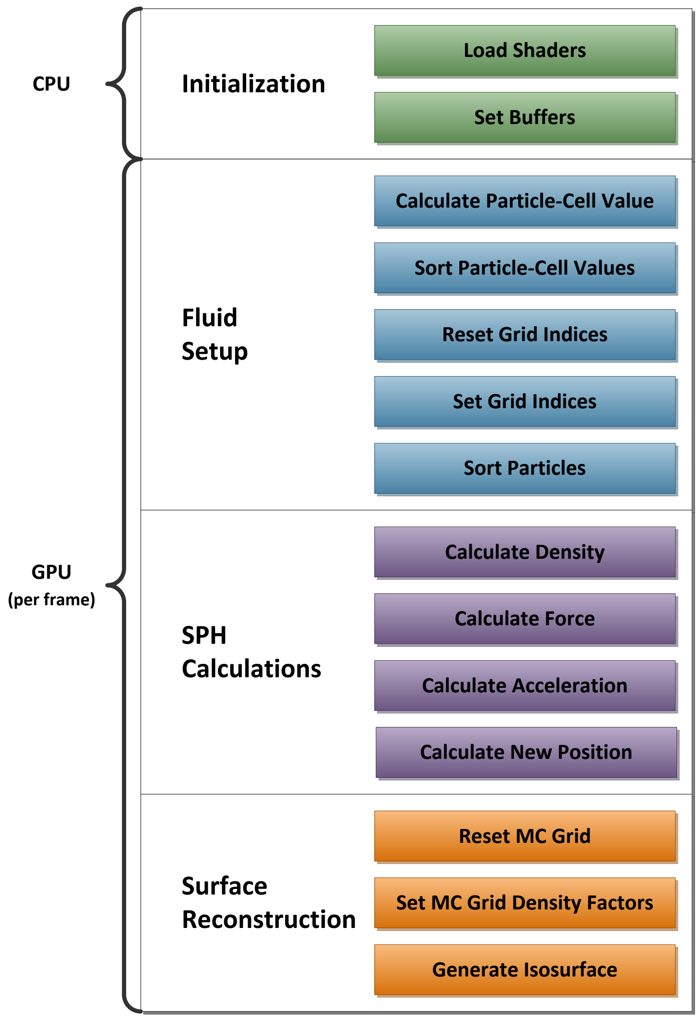

Figure 9. Algorithm Structure

6.1 Initialization

This block is composed of only two steps: “Load Shaders” and “Set Buffers.” Each one of the steps on the following blocks represents a DirectCompute shader. These shaders can be loaded into memory by either using precompiled shaders or by compiling them during the application execution. After the loading step, then it is necessary to set all the buffers required by each one of these shaders. This block can also be used to initialize any predefined setting such as minimum/maximum number of particles, create objects for the fluid to interact with, and set all the SPH global settings.

6.2 Fluid Setup

This block is composed of a series of steps required to group and sort the particles in a manner in which we can apply the SPH calculations in an efficient manner. The nature of SPH, as it can be observed in its formulas, is to look at each particle and calculate the influence that other particles have over it. This can trivially be translated into a series of loops; however, due to the nature of the DirectCompute shaders, organizing the particles in such a format is not a trivial task.

To accomplish this task, each one of the steps in this block uses a grid set. This grid contains all the particles in the fluid and arranges them in individual units called cells. The biggest benefit of placing all the particles within these cells is to limit the area the algorithm needs to look into when analyzing the influence of the particles over each other.

In order to minimize the number of cells the algorithm needs to look at, we set the virtual size of each cell to $2h$. This size guarantees that all the particles which can influence the forces affecting a given particle are contained within the adjacent cells. Furthermore, since each cell is of size 2h, then we can determine the quadrant in which a given particle is within its cell and only search for other particles on the adjacent cells to that quadrant; allowing us to narrow the search from 27 cells to only 8 cells.

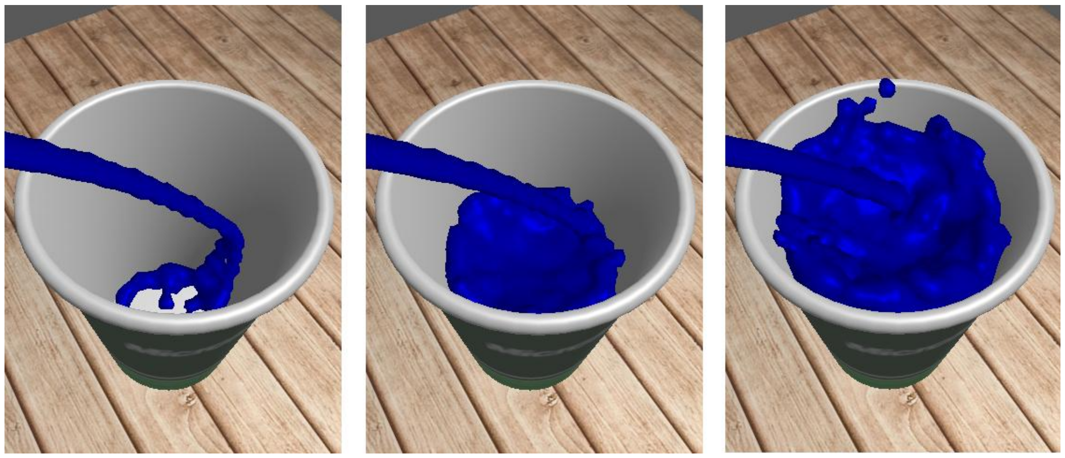
Figure 10. Fluid Simulation: Liquid interacting with a mesh object

6.2.1 Data Structures

The steps described next use multiple different data structures based on the DirectCompute and HLSL guidelines. Next, we present the structured buffers required for the steps described on the Fluid Setup and SPH Calculations blocks (actual buffers implementation details are described later on the document):

$StructuredBuffer[Particle]Particles: register$

$StructuredBuffer[uint]Grid: register$

$StructuredBuffer[uint2]GridIndices: register$

$StructuredBuffer[float]Density: register$

$StructuredBuffer[float]Pressure: register$

$StructuredBuffer[float]Force: register$

$StructuredBuffer[float]Acceleration: register$

6.2.2 Calculate Particle-Cell Value

This step loops through all the particles and calculate the cell to which each particle belongs too. These cell values vary by the specific location of each particle, so they must be recalculated on each frame.

```
01 $For\ each\ particle$
02   $cell = CellValue(particle.Position)$
03   $Grid[particle.ID] = (cell, particle.ID)$
```

6.2.3 Sort Particle-Cell Values

This step takes the grid values derived on the previous step and sorts them all in ascending order. There are multiple different algorithms that can be used to achieve this using the power of the GPU. The implementation described later in this document uses the Bitonic Sort algorithm (more details on this algorithm are described on the following section).

6.2.4 Reset Grid Indices

This is trivial step that simply resets all the values of the $GridIndices$ buffer to zero. This is required in order to regroup all the particles by cells in the previous step. Since a particle can move to any cell from one step to another, the algorithm needs to be reset on every frame.

6.2.5 Set Grid Indices

This step loops through all the grid cell values and groups the particle IDs based on cell they belong to. At the end of this step, the GridIndices buffer contains a reference to the particles on each cell.

```
$For\ each\ cell$
  $Set\ currentCellID =\ cell.ID$
  $Set\ previousCellID =\ currentCellID - 1$
    $if\ previousCellID < 0$
      $prviousCelID = numberOfParticles - 1$
  $Set\ nextCellID = currentCellID + 1$
    $if\ nextCellID = numberOfParticles$
      $nextCelID = 0$
  $if\ currentCellID \neq previousCellID$
    $GridIndices[currentCellID] = cell.ID$
  $if\ currentCellID \neq nextCellID$
    $GridIndices[currentCellID] = cell.ID + 1$
```

6.2.6 Sort Particles

This step simply rearranges every particles based on the cell value they belong. This is necessary to assign the particles to the values assigned in the previous step.

```
$For\ each\ particle$
  $ID = IDValue(Grid[particle.ID])$
  $Particles[particle\ ID] = Particles(ID)$
```

6.3 SPH Calculations

After all the particles have been arranged and the $GridIndices$ buffer is setup with the correct indices, the algorithm is now ready to calculate the particles' next positions in order to simulate how they move along the fluid. To accomplish this, the algorithm applies the SPH force formulas described previously following the next steps.

6.3.1 Calculate Density

This step loops through every particle and calculates their density and pressure values based on the particles on the adjacent cells.

```
01 $For\ each\ particle$
02   $ID = particle.ID$
03   $Cell = GetCell(Grid[ID])$
04   $Quadrant = GetQuadrant(particle.Position)$
05   $NCells = GetNeighbors(Quadrant)$
06   $Set\ DenF = 0.0$
07   $For\ each\ neighbor\ cell$
08     $DenF += DFact(particle, GridIndices[cell])$
09   $Density[ID] = DenF * DenCoef$
10   $Pressure[ID] = (Density[ID] - RDen) * IGC$
```

where the function $DFact$ is defined as follows:

```
01 $Set\ Total = 0.0$
02 $For\ each\ particle\ in\ cell$
03   $if\ particle \neq cell.Particle$
```

```
    $Dist = [distance\ between\ particles]$
    $DistSq = DistanceSquare(dist)$
    $if\ (SmoothLenSq > distSq)$
      $Total \mathrel{+}= (SmoothLenSq - distSq)^3$
```

6.3.2 Calculate Force

After the pressure and the density of each individual particle have been calculated, this step computes the SPH force for all particles.

```
$For\ each\ particle$
  $ID = particle.ID$
  $Cell = GetCell(Grid[ID])$
  $Quadrant = GetQuadrant(particle.Position)$
  $NCells = GetNeighbors(Quadrant)$
  $Set\ Force = 0.0$
  $For\ each\ neighbor\ cell$
    $Force \mathrel{+}= FFact(particle, GridIndices[cell])$
  $Force[ID] = Force$
```

where the function $FFact$ is defined as follows:

```
$Set\ Total = (0.0, 0.0, 0.0)$
$For\ each\ particle\ in\ cell$
  $if\ particle \neq cell.Particle$
    $dist = [distance\ between\ particles]$
    $distSq = DistanceSquare(dist)$
    $if\ (SmoothLenSq > distSq)$
```

```
07      $PressureF = CalculatePressure()$
08      $ViscosityF = CalculateViscosity()$
09      $Total \mathrel{+}= PressureF + ViscosityF$
```

6.3.3 Calculate Acceleration

Once the forces have been calculated this step derives the particles' acceleration. This is accomplished by looping through each particle and computing first their acceleration, next the algorithm applies the collision forces – if a collision is detected, and finally the algorithm applies all external forces.

```
01 $For\ each\ particle$
02   $ID = particle.ID$
03   $Acceleration = Force[ID] * particle.Mass$
04   $Velocity = Acceleration^2$
05   $if\ Velocity > MaxVelocity^2$
06     $Acceleration = Acceleration\left(\frac{MaxVelocity}{\sqrt{Velocity}}\right)$
07   $For\ each\ triangle$
08     $if\ collision\ detected$
09       $Acceleration \mathrel{+}= CollisionForce()$
10   $Acceleration \mathrel{+}= [External\ Forces]$
11   $Acceleration[ID] = Acceleration$
```

where the $CollisionForce$ method is a specific implementation function. The document describes in detailed an algorithm for implementing this function on the following section. In this particular implementation, the final force applied to each particle is affected by the distance between the given particle and the object.

6.3.4 Calculate New Position

This step used the acceleration of each particle and computes the particles' new positions using a time-step approach.

```
01 $For\ each\ particle$
02   $ID = particle.ID$
03   $V0 = particle.VelT0$
04   $v = V0 + (Acceleration[ID] * TimeStep)$
05   $particle.VelT1 = \frac{1}{2}(particle.VelT0 + v)$
06   $particle.VelT0 = v$
07   $particle.Position += v * TimeStep$
```

6.4 Surface Reconstruction

In order to derive the surface out of the particles, this solution presented next uses the Marching Cubes algorithm. The first two steps set up the data set with the corresponding iso-values, and then the set is used to create a triangle-mesh representing the fluid.

6.4.1 Data Structures

The following are the structured buffers required for the steps required within the Surface Reconstruction block (actual buffers implementation details are described later on the document):

$StructuredBuffer[Particle]Particles: register$

$StructuredBuffer[float]Grid: register$

$StructuredBuffer[uint2]GridIndices: register$

6.4.2 Reset MC Grid

This step resets the grid indices buffer to zero before calculating the densities at each cell vertex, as described on the following step. This step must be executed on every frame.

6.4.3 Set MC Grid Density Factors

The algorithm loops through each vertex in the grid and calculates the density factor (iso-value) according to the "influence" of the particles on the neighbor cells. A particle's density "influences" the density factor of a vertex if the particle is within the kernel core radius, h; the "influence" is proportional to the distance between the particle and the vertex, similar to how particles influence each other on the SPH algorithm.

```
01 $For\ each\ vertex\ in\ grid$
02   $Index = MapToFluidGrid(vertex)$
03   $Set\ DenF = 0.0$
04   $For\ each\ neighbor\ cell$
05     $DenF += DFact(vertex, GridIndices[Index])$
06   $MCGrid[ID] = DenF * DenCoef$
```

6.4.4 Generate Isosurface

This step uses the Vertex, Geometry and Pixel Shaders from the Direct3D pipeline. These three shaders are part of the improved Direct3D 11 and they play an important role on this algorithm.

6.4.4.1 Vertex Shader

For this algorithm, the Vertex Shader (VS) is simply used as a pass-through stage. The vertex ID is simply passed to the next stage on the pipeline. This ID will be used to identify the cell (or cube) in which the marching cube algorithm will operate on. That is, the vertex shader is called for each cell on the data grid.

6.4.3.2 Geometry Shader

This shader is responsible for generating the isosurface using the Marching Cubes algorithm and data grid generated during the ‘Set MC Grid Density Factors’ step.

This step loops through each cell on the grid, calculates the vertices “inside” and “outside” the isosurface. It then determines the cube configuration (as described in the Marching Cubes algorithm) and generates the triangles accordingly.

```
$For\ each\ cell\ in\ grid$
  $CubeIndex = 0$
  $For\ each\ cellVertex\ in\ cell$
    $Index = CalculateIndex()$
    $cellVertex.IsoValue = MCGrid[Index]$
    $cellVertex.Index = Index$
    $if\ cellVertex.IsoValue < IsoValueThreshold$
      $CubeIndex = CubeIndex \mid 2^{cellVertex.ID}$
  $EdgeFlag = EdgeTable[CubeIndex]$
```

```
$Base = Calulate(cellVertex[0].Index)$
$For\ each\ edge\ on\ cube$
  $if\ edge\ belongs\ to\ EdgeFlag$
    $cellVertex[ID].Position = Position(Base)$
    $cellVertex[ID].Normal = Normal(Base)$
$VertexList[12] = 0$
$NormalList[12] = 0$
$For\ i = 0;\ i < 12$
  $if\ (EdgeFlag\ \&\ 2^i) \neq 0$
    $VertexList[i] = Lerp(Edge\ Positions)$
    $NormalList[i] = Lerp(Edge\ Normals)$
$While\ TriangleTable[i] \neq\ -1$
  $v0.Position = VertexList[i]$
  $v0.Normal = NormalList[i]$
  $SpriteStream.Append(v0)$
  $v1.Position = VertexList[i+1]$
  $v1.Normal = NormalList[i+1]$
  $SpriteStream.Append(v1)$
  $v2.Position = VertexList[i+2]$
  $v2.Normal = NormalList[i+2]$
  $SpriteStream.Append(v2)$
  $i = i + 3$
```

```
32      SpriteStream.RestartStrip()
```

6.4.3.3 Pixel Shader

There is no specific use for the Pixel Shader on this algorithm. Depending on the implementation, this stage can be used to create per-pixel lighting calculations or simply to pass through the color calculated at the Geometry Shader. Even though there is no specific PS algorithm, this shader plays an important role and it cannot be overlooked. Anyone implementing this algorithm will have to set a Pixel Shader in order to properly render the isosurface extracted by the Geometry Shader.

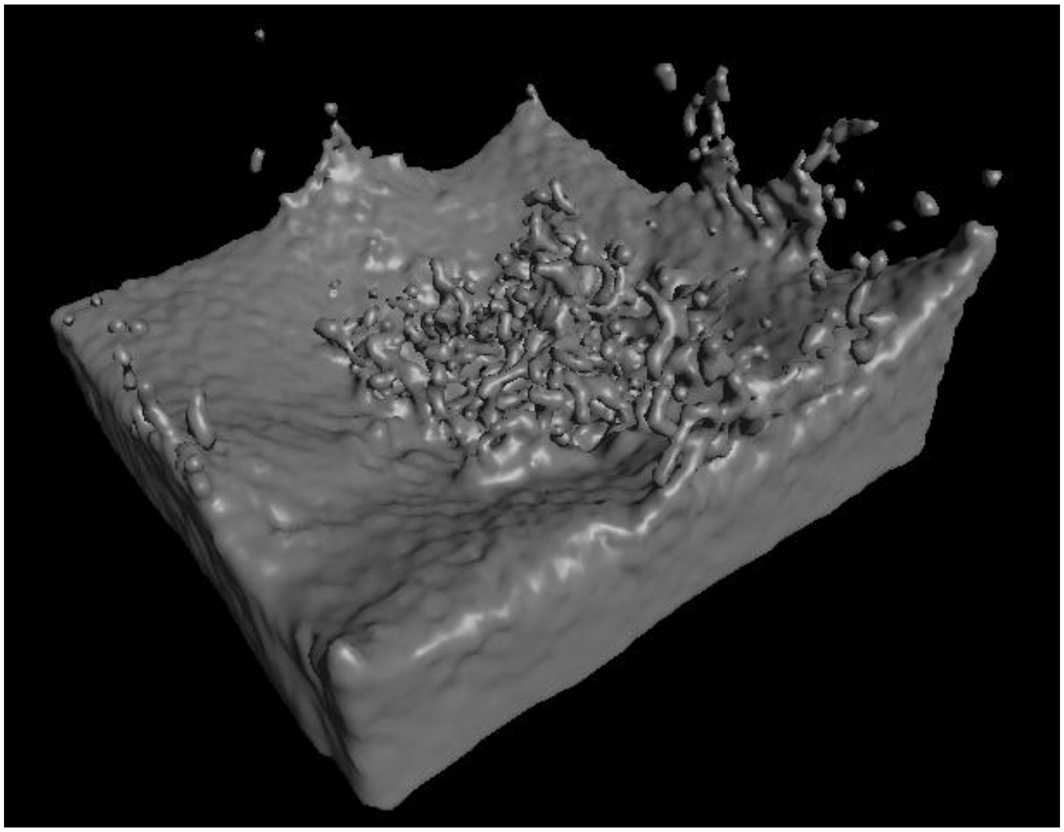

Figure 11. Pixel Shader Fluid Rendering

Chapter 7

IMPLEMENTATION

The document now focuses on describing the details of the application developed using the algorithm described on the previous section. The implementation was created to simulate a liquid fluid, particularly in this case: water.

7.1 Assumptions

As in all implementations, there exist multiple assumptions that were made during the development. Here is the list of the assumptions made on this implementation:

1. Particles' masses and sizes are constant and equal. This can be considered accurate in real-live scenarios where all the particles are water molecules. It also allows for a greatly simplifications of the SPH calculations.
2. All particles are contained within a predefined grid; that is, no particles can exist outside the grid. This implies a limitation on the simulator, which in most cases, it would not have a significant impact, but it does have to be observed. In addition, the grid exists at a fixed location and it is of a fixed size; guaranteeing that the neighboring lookups are constant.
3. All objects in the scene are triangle-meshes with the same physical properties, such as elasticity and stiffness. Also, the objects are static within the time-step specified in the SPH Calculations block; that is, the

object is not moving while the particles’ forces, acceleration and new position values are being calculated.

7.2 Fluid Shaders

In order to guarantee that all compute shaders are executed for each fluid particle, the implementation uses the ID3D11DeviceContext.Dispatch method and the NUMTHREADS DirectCompute attribute. Following is an example of the setting required to accomplish this:

In Code:

```
Dispatch(NumberParticles / NUMBER_THREADS, 1, 1);
```

In HLSL:

```
[numthreads(NUMBER_THREADS, 1, 1)]
void Step
      (
        uint3 Gid  : SV_GroupID,
        uint3 DTid : SV_DispatchThreadID,
        uint3 GTid : SV_GroupThreadID,
        uint  GI   : SV_GroupIndex
       )
{
   // Code
}
```

where NUMBER_THREADS is set to 2014 (the maximum number of threads allowed by Direct3D 11).

7.3 Sorting Shaders

For this implementation, we used the Bitonic algorithm in order to sort the grid indices values generated during the “Calculate Particle Cell-Value” step. The Bitonic algorithm was originally developed by Ken

Batcher as a network sorting algorithm [26]. The algorithm provides a $O(n\,log^2(n))$ worst case complexity and a best case performance of $O(\,log^2(n))$.

Appendix A shows a full implementation of the Bitonic algorithm using C++ and HLSL.

7.4 Marching Cubes Shaders

The first two shaders of the Surface Reconstruction block use the same approach as the fluid shaders, where the variable ‘NumberParticles’ is replaced with the variable ‘NumberVertices.’ Then, as the algorithm describes, the following steps use the Vertex, Geometry and Pixel shaders. In this implementation we use a per-pixel lighting technique.

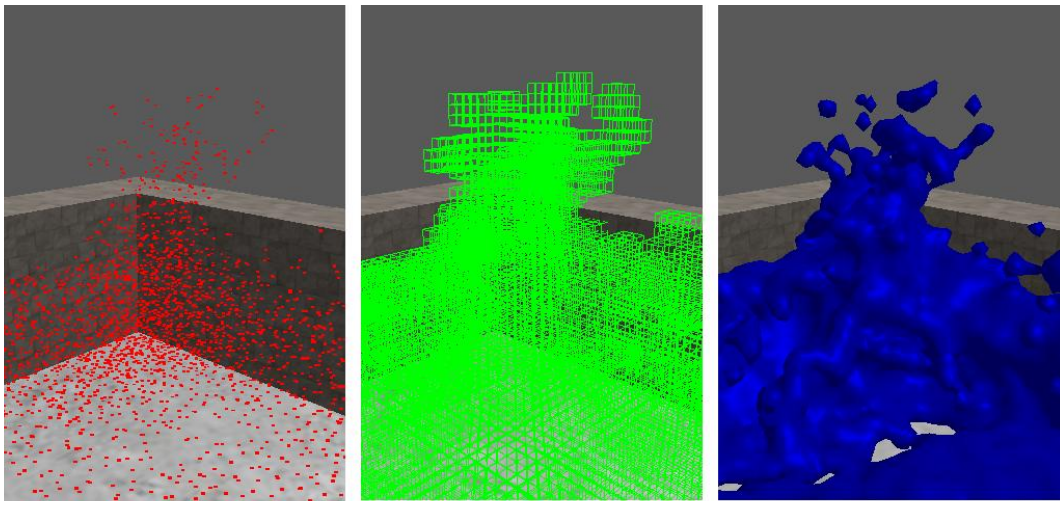

Figure 12. Marching Cubes Surface Reconstruction Steps Illustration. The left image shows the particles rendered as textured squares. The middle image shows the cubes with at least one vertex “inside” the isosurface. The right image shows the mesh fluid rendered using the MC algorithm.

7.5 Results

In this section we describe the results of the implementation we just described. As a point of reference, the application was tested using the following system:

System 1 Characteristics
▪ Intel Core i7-2700 CPU @3.50GHz (8 CPUs) ▪ 16 GB RAM ▪ Windows 7 Enterprise 64-bit (6.1, Build 7601) ▪ NVIDIA GeForce GTX 580 (2736 MB) ▪ DirectX 11 ▪ Windows Experience Index: 7.6

Table 1: System 1 Characteristics

System 2 Characteristics
▪ Intel Core i7 CPU 950 @3.07GHz (8 CPUs) ▪ 12 GB RAM ▪ Windows 7 Enterprise 64-bit ▪ NVIDIA GeForce GTX 430 (1024 MB) ▪ DirectX 11 ▪ Windows Experience Index: 5.9

Table 2: System 1 Characteristics

The first tests were designed in order to validate the performance of the algorithm and the implementation. Table 3 & Table 4 describe the results obtained when testing the implementation using the both System 1 and System 2. The application was tested using a different set of configurations and number of particles. As it can be seen on the following tables, one of the most relevant results is the application's ability to handle in real-time over 65,000 particles using a 128*128*128 Marching Cubes grid. However, the results also show that the bottleneck in the application is the Marching Cubes algorithm. Alternative solutions are analyzed in the following section.

System 1			
Configuration	Number of Particles		
	16,384	32,768	65,536
SPH-Only	920 fps	915 fps	910 fps
SPH + Particles (texture based)	903 fps	895 fps	860 fps
SPH + MC (grid size 32)	608 fps	524 fps	440 fps
SPH + MC (grid size 64)	314 fps	241 fps	184 fps
SPH + MC (grid size 128)	86 fps	65 fps	49 fps
SPH + MC (grid size 256)	12 fps	8 fps	7 fps

Table 3: System 1 Simulation Results

System 2			
Configuration	Number of Particles		
	16,384	32,768	65,536
SPH-Only	835 fps	805 fps	798 fps
SPH + Particles (texture based)	800 fps	790 fps	755 fps
SPH + MC (grid size 32)	537 fps	463 fps	389 fps
SPH + MC (grid size 64)	277 fps	212 fps	180 fps
SPH + MC (grid size 128)	78 fps	60 fps	44 fps
SPH + MC (grid size 256)	10 fps	7 fps	4 fps

Table 4: System 2 Simulation Results

Additional results are shown in the following images:

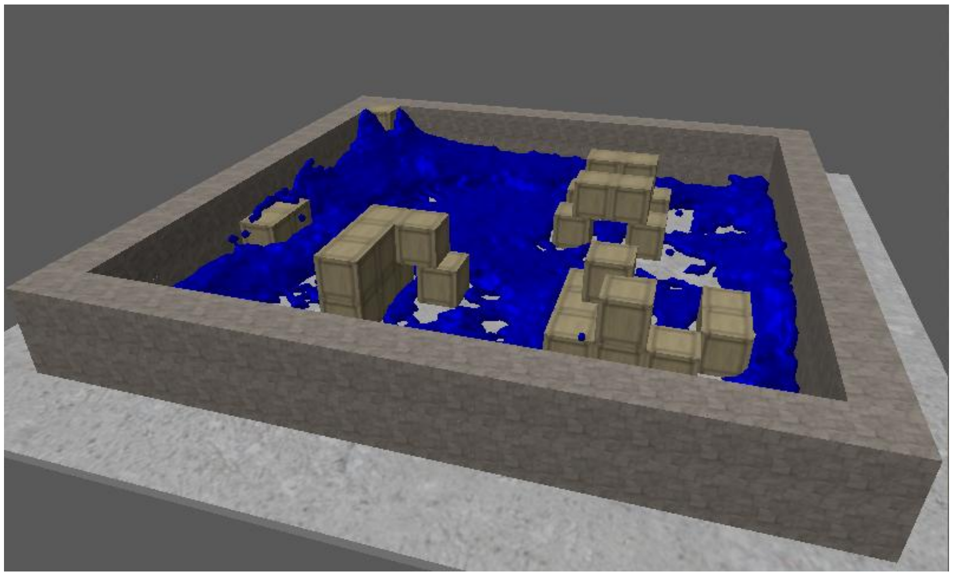

Figure 13. Fluid interacting with multiples objects (a). Number of Particles: 32,768; Marching Cubes Grid: 128x128x128.

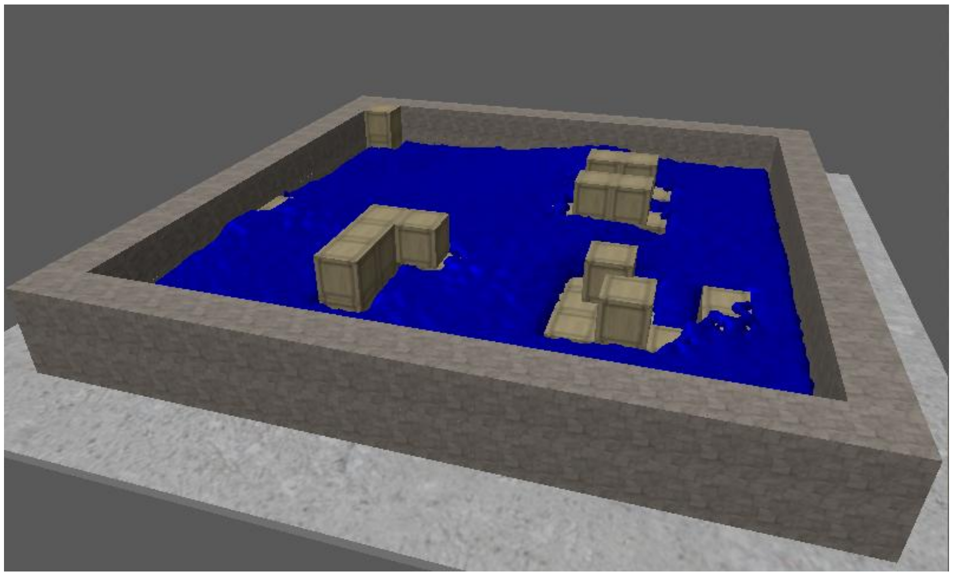

Figure 14. Fluid interacting with multiples objects (b). Number of Particles: 65,636; Marching Cubes Grid: 128x128x128. The image shows the fluid rendered using Marching Cubes.

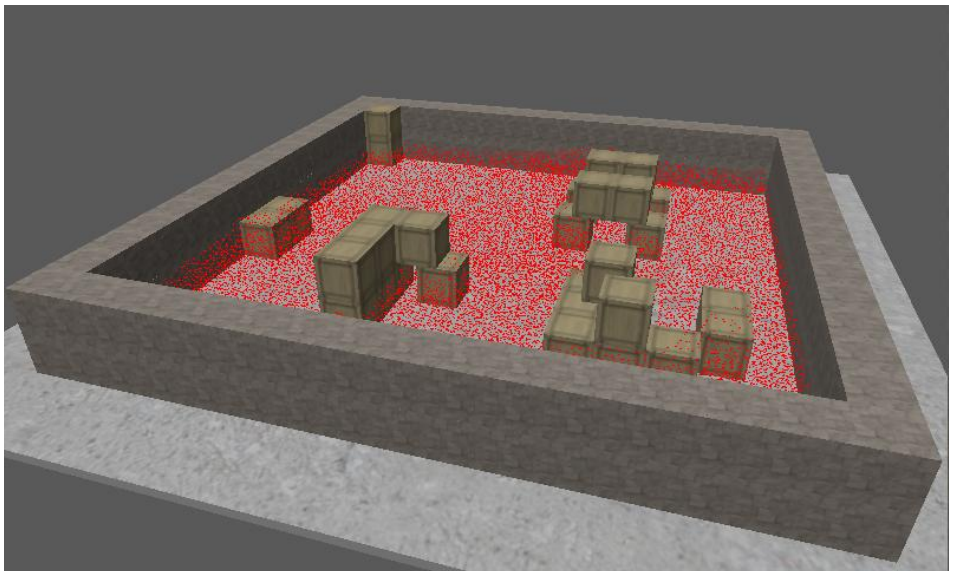

Figure 15. Fluid interacting with multiples objects (c). Number of Particles: 65,636; Marching Cubes Grid: 128x128x128. This image shows the individual particles rendered as textured squares.

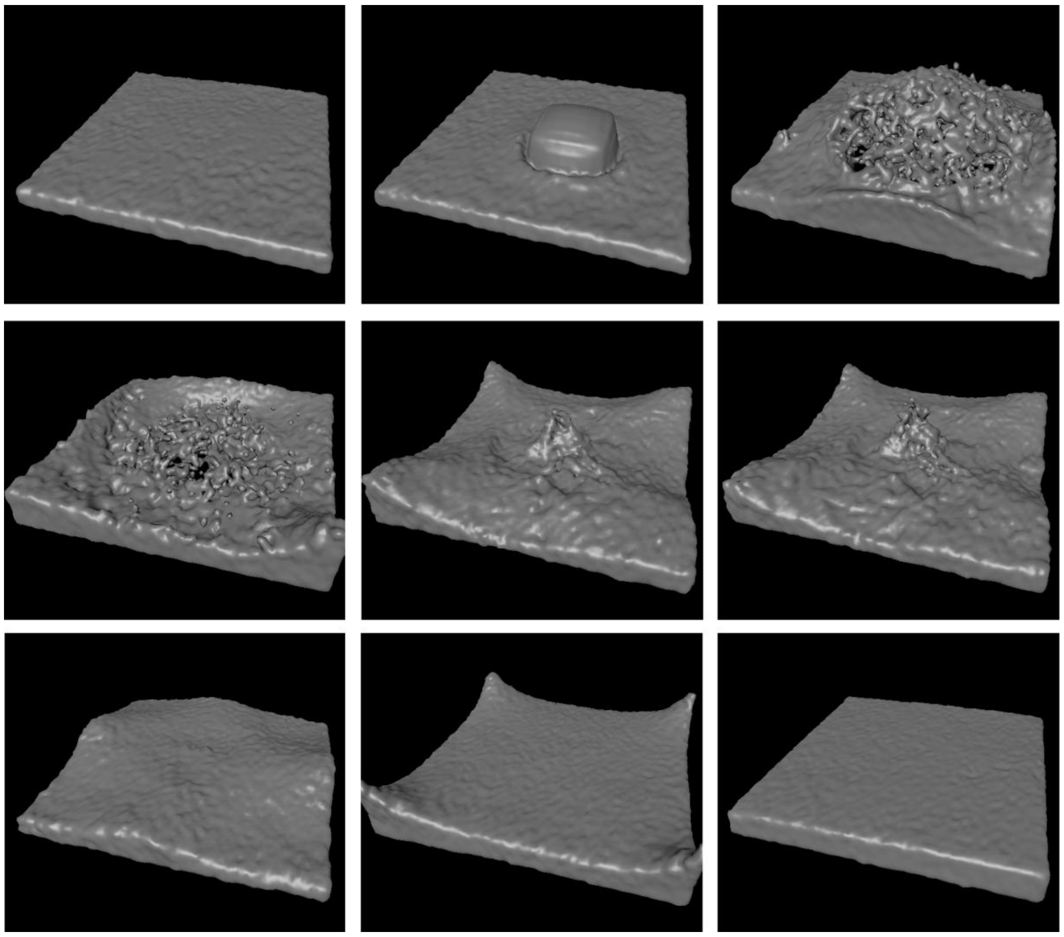

Figure 16. Test Fluid Sequence. The top image represents the initial state of the fluid. All other images represent the sequence after “dropping” a “block of liquid”. Total Number of Particles: 32,768; MC Grid: 128x128x128

Chapter 8

FUTURE WORK

The solution presented in this document shows a promising approach for creating fast fluid simulators using Direct3D. However, the solution is not ideal; there are areas that can be improved to generate faster and more photorealistic simulators. Following we present a list of possible areas and solutions which can improve the overall outcome of the algorithm just presented. These possible areas of improvements are grouped by the block or step they belong to.

- ‘Sort Particle’ step. Our solution uses the Bitonic Sort algorithm in order to sort, in each frame, all the fluid particles. However, it was observed that most particles do not have to be rearranged on every frame; that is, most particles do not move from one cell/cube on every frame. So, a new algorithm which tackles this issue, and only sorts those particles that have moved to another cell/cube could improve the overall the performance greatly.
- ‘Surface Reconstruction’ block. There exist several algorithms that can be used to reconstruct the surface based on the fluid particles data. Each one of these algorithm would need to be tested and compared against the results presented here to determine if they actually represent an improvement. One of these alternatives is the “Dual Marching Cubes” algorithm developed by Dr. Gregory M. Nielson [16]. This algorithm resolves some of the “artifacts” generated by the

Marching Cubes algorithm; thus generating a visually better isosurface. Other alternatives include smooth isosurface extraction [17], anisotropic kernels [18] and adaptive sampling [19]. All of these present benefits and drawbacks. Further research is required to determine their best implementation and to analyze their results.

- ‘Collisions’ (within the ‘Calculate Acceleration’ step). Our algorithm uses a basic point-triangle collision detection method. This method is efficient, but it can be improved by using other approaches such as the one proposed by da Silva, Clua, Pagliosa and Montenegro: “Fluid Simulation with Rigid Body Triangle Accuracy Collision using a Heterogeneous GPU/CPU Hardware System.”

REFERENCES

[1] Nick Foster and Demitri Metaxas. “Realistic Animation of Liquids”. In Wayne A. Davis and Richard Bartels, editors, Graphics Interface ’96, pages 204–212. Canadian information Processing Society, Canadian Human-Computer Communications Society, May 1996.

[2] Matthias Müller, David Charypar, and Markus Gross. 2003. “Particle-Based Fluid Simulation for Interactive Applications”. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '03). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 154-159.

[3] Karthik Raveendran, Chris Wojtan, and Greg Turk. 2011. “Hybrid Smoothed Particle Hydrodynamics”. In Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '11), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 33-42.

[4] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. 2010. “Interactive SPH Simulation and Rendering on the GPU”. In Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '10). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 55-64.

[5] Fengquan Zhang, Lei Hu, Jiawen Wu, and Xukun Shen. 2011. “A SPH-Based Method for Interactive Fluids Simulation on the Multi-GPU”. In Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry (VRCAI '11). ACM, New York, NY, USA, 423-426.

[6] Kristoffer Lindstrom. 2011. “Performance of Marching Cubes using DirectX Compute Shaders Compared to using HistoPyramids”. Blekinge Institute of Technology. Karlskrona, Sweden, 2-4.

[7] Programming Guide for Direct3D 11. MSDN. Microsoft. 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476345(v=vs.85).aspx/

[8] DirectCompute. Wikipedia. Wikimedia Foundation, Inc. May 2, 2012. http://en.wikipedia.org/wiki/DirectCompute/

[9] Shader Stages. MSDN. Microsoft. 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/bb205146(v=vs.85).aspx#Vertex_Shader_Stage/

[10] G.R. Liu, M. B. Liu. “Smoothed Particle Hydrodynamics: A Meshfree Particle Method”. World Scientific Publishing Co. Pte. Ltd. 2003. USA.

[11] William E. Lorensen and Harvey E. Cline. 1987. “Marching cubes: A high resolution 3D Surface Construction Algorithm”. In Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH '87), Maureen C. Stone (Ed.). ACM, New York, NY, USA, 163-169.

[12] Marching Cubes. Wikipedia. Wikimedia Foundation, Inc. September 4, 2012. http://en.wikipedia.org/wiki/Marching_cubes

[13] James Sharman. “The Marching Cubes Algorithm”. http://www.exaflop.org/docs/marchcubes/

[14] Leapfrog Integration. Wikipedia. Wikimedia Foundation, Inc. August, 12, 2012. http://en.wikipedia.org/wiki/Leapfrog_method/

[15] Tomas Müller and Ben Trumbore. 2005. “Fast, Minimum Storage Ray/Triangle Intersection”. In ACM SIGGRAPH 2005 Courses (SIGGRAPH '05), John Fujii (Ed.). ACM, NY, USA, Article 7.

[16] Gregory M. Nielson. 2004. “Dual Marching Cubes”. In Proceedings of the conference on Visualization '04 (VIS '04). IEEE Computer Society, Washington, DC, USA, 489-496.

[17] Lars Linsen, Vladimir Molchanov, Petar Dobrev, Stephan Rosswog, Paul Rosenthal, Tran Van Long. “SmoothViz: Visualization of Smoothed Particle Hydrodynamics Data”. In Hydrodynamics – Optimizing Methods and Tool. INTECH. October 26, 2011, pages 3-28.

[18] Jihun Yu and Greg Turk. 2010. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (SCA '10). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 217-225.

[19] Zhang, Y; Solenthaler, B; Pajarola, R (2008). “Adaptive Sampling and Rendering of Fluids on the GPU”. In: Proceedings Symposium on Point-Based Graphics, Los Angeles, 10 August 2008 - 11 August 2008, 137-146.

[20] Jose Ricardo da Silva, Jr, Esteban W. Cluay, Paulo A. Pagliosaz, and Anselmo Montenegrox. 2010. “Fluid Simulation with Rigid Body Triangle Accuracy Collision using a Heterogeneous GPU/CPU Hardware System”. In Proceedings of the 2009 symposium on Interactive 3D graphics and games (I3D '09). ACM, New York, NY, USA, Article 20, 1 pages.

[21] Rama Hoetzlein. R.C. Hoetzlein Web Site. http://www.rchoetzlein.com/eng/

[22] Peter Berczik. SPH Astrophysics - State-of-the-art. ERCOFTAC SIG SPHERIC III International Workshop Keynotes. 2008.

[23] Michal Chladek, Roman Durikovic. Smoothed Particle Hydrodynamics in Flood Simulations. Proceedings of the Spring Conference on Computer Graphics - SCCG2010, ISSN: 1335-5694, Budmerice, Slovakia, pages 34 – 37, 2010.

[24] Robert Bridson, Matthias Müller-Fischer. Fluid Simulation SIGGRAPH 2007 Course Notes, SIGGRAPH Symposium 2007, pages 65 – 67. April 30, 2007.

[25] Marcus Vesterlund. 2004. “Simulation and Rendering of a Viscous Fluid using Smoothed Particle Hydrodynamics”. Master Thesis. December 3, 2004.

[26] Bitonic Sorter. Wikipedia. Wikimedia Foundation, Inc. September 8, 2012. http://en.wikipedia.org/wiki/Bitonic_sorter.

APPENDIX A

BIOTONIC SORT ALGORITHM

BiotonicSort.cpp

```
#define BIT_BSIZE 1024
#define TRANS_BSIZE 32
#define MATRIX_WIDTH = BIT_BSIZE;
#define MATRIX_HEIGHT = NUM_ELEMENTS/ BIT_BSIZE;

// Constant Buffers
_DECLSPEC_ALIGN_16_ struct SortCB
{
    UINT Level;
    UINT LevelMask;
    UINT Width;
    UINT Height;
};

// Structured Buffers
ID3D11ComputeShader*             pSortBitonic = NULL;
ID3D11ComputeShader*             pSortTranspose = NULL;
ID3D11Buffer*                    pSortCB = NULL;
ID3D11Buffer*                    pTemp = NULL;
ID3D11ShaderResourceView*        pTempSRV = NULL;
ID3D11UnorderedAccessView*       pTempUAV = NULL;


//-----------------------------------------------------------------------
// GPU Bitonic Sort
// For more information, please see the DirectX samples by Microsoft
//-----------------------------------------------------------------------
void BitonicSort
        (
          ID3D11DeviceContext* pd3dCxt,
          ID3D11UnorderedAccessView* inUAV,
          ID3D11ShaderResourceView* inSRV
        )
{
    pd3dCxt ->CSSetConstantBuffers(0, 1, &pSortCB);

    // Sort the data
    // First sort the rows for the levels <= to the block size
    for(UINT level=2; level<=BIT_BSIZE; level<<=1)
    {
        SortCB cons = {level, level, MATRIX_HEIGHT, MATRIX_WIDTH};
        pd3dCxt->UpdateSubresource(pSortCB, 0, NULL, &cons, 0, 0);

        // Sort the row data
        pd3dCxt->CSSetUnorderedAccessViews(0, 1, &inUAV, NULL);
        pd3dCxt->CSSetShader(pSortBitonic, NULL, 0);
        pd3dCxt->Dispatch(NUM_ELEMENTS/BIT_BSIZE, 1, 1);
    }

    // Sort the rows and columns for the levels > than the block size
    // Transpose. Sort the Columns. Transpose. Sort the Rows.
    for(UINT level=(BIT_BSIZE<<1); level<=NUM_ELEMENTS; level<<=1)
    {
        SortCB cons1 =
        {
           (level/BIT_BSIZE),
           (level&~NUM_ELEMENTS)/BIT_BSIZE, MATRIX_WIDTH, MATRIX_HEIGHT
        };

        // Set constant buffer
        pd3dCxt->UpdateSubresource(pSortCB, 0, NULL, &cons1, 0, 0);

        // Transpose the data from buffer 1 into buffer 2
        ID3D11ShaderResourceView* pViewNULL = NULL;
        pd3dCxt->CSSetShaderResources(0, 1, &pViewNULL);
        pd3dCxt->CSSetUnorderedAccessViews(0, 1, &pTempUAV, NULL);
        pd3dCxt->CSSetShaderResources(0, 1, &inSRV );
        pd3dCxt->CSSetShader(pSortTranspose, NULL, 0);
        pd3dCxt->Dispatch(WIDTH/TRANS_BSIZE, HEIGHT/TRANS_BSIZE, 1);

        // Sort the transposed column data
        pd3dCxt->CSSetShader(pSortBitonic, NULL, 0);
        pd3dCxt->Dispatch(NUM_ELEMENTS/BIT_BLOCK_SIZE, 1, 1);

        SortCB cons2 = {BIT_BSIZE, level, MATRIX_HEIGHT, MATRIX_WIDTH};
        pd3dCxt->UpdateSubresource(pSortCB, 0, NULL, &cons2, 0, 0);

        // Transpose the data from buffer 2 back into buffer 1
        pd3dCxt->CSSetShaderResources(0, 1, &pViewNULL);
        pd3dCxt->CSSetUnorderedAccessViews(0, 1, &inUAV, NULL);
        pd3dCxt->CSSetShaderResources(0, 1, &pTempSRV);
        pd3dCxt->CSSetShader(pSortTranspose, NULL, 0);
        pd3dCxt->Dispatch(HEIGHT/TRANS_BSIZE, WIDTH/TRANS_BSIZE, 1);

        // Sort the row data
        pd3dCxt->CSSetShader(pSortBitonic, NULL, 0);
        pd3dCxt->Dispatch(NUM_ELEMENTS/BIT_BSIZE, 1, 1);
    }
}
```

BiotonicSort.hlsl

```
#define BIT_BSIZE 1024
#define TRANS_BSIZE 32

// Constant Buffers
cbuffer CB : register( b0 )
{
    uint Level;
    uint LevelMask;
    uint Width;
    uint Height;
};

// Structured Buffers
StructuredBuffer<uint> Input : register( t0 );
RWStructuredBuffer<uint> Data : register( u0 );

//-----------------------------------------------------------------------
// For more information, please see the DirectX samples by Microsoft
//-----------------------------------------------------------------------

// Bitonic Sort Compute Shader
groupshared uint shared[BIT_BSIZE];

[numthreads(BIT_BSIZE, 1, 1)]
void BitonicSort
        (
          uint3 Gid  : SV_GroupID,
          uint3 DTid : SV_DispatchThreadID,
          uint3 GTid : SV_GroupThreadID,
          uint  GI   : SV_GroupIndex
        )
{
    // Load shared data
    shared[GI] = Data[DTid.x];
    GroupMemoryBarrierWithGroupSync();

    // Sort the shared data
    for (uint j=Level>>1 ; j>0 ; j>>=1)
    {
        // ‘Copy’ data
        uint result =
        ((shared[GI&~j].x<=shared[GI|j].x)==(bool)(LevelMask& DTid.x))?
           shared[GI^j] : shared[GI];
        GroupMemoryBarrierWithGroupSync();

        // ‘Paste’ data
        shared[GI] = result;
        GroupMemoryBarrierWithGroupSync();
    }

    // Store shared data
    Data[DTid.x] = shared[GI];
}
```

```
// Matrix Transpose Compute Shader
groupshared uint shared [TRANS_BSIZE*TRANS_BSIZE];

[numthreads(TRANS_BSIZE, TRANS_BSIZE, 1)]
void MatrixTranspose
        (
          uint3 Gid  : SV_GroupID,
          uint3 DTid : SV_DispatchThreadID,
          uint3 GTid : SV_GroupThreadID,
          uint  GI   : SV_GroupIndex
        )
{
    // Load shared data
    shared[GI] = Input[DTid.y*Width+DTid.x];
    GroupMemoryBarrierWithGroupSync();

    // Find destination
    uint2 P = DTid.yx-GTid.yx+GTid.xy;

    // Store shared data
    Data[P.y*Height+P.x] = shared[GTid.x*TRANS_BSIZE+GTid.y];
}
```