A GPU ACCELERATED SMOOTHED PARTICLE HYDRODYNAMICS CAPABILITY FOR HOUDINI

A Thesis

by

MATHEW ALLEN SANFORD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Visualization

A GPU ACCELERATED SMOOTHED PARTICLE HYDRODYNAMICS CAPABILITY FOR HOUDINI

A Thesis

by

MATHEW ALLEN SANFORD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Frederic Parke
Committee Members,	John Keyser
	Ann McNamara
Head of Department,	Tim McLaughlin

August 2012

Major Subject: Visualization

ABSTRACT

A GPU Accelerated Smoothed Particle
Hydrodynamics Capability for Houdini. (August 2012)
Mathew Allen Sanford, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Frederic Parke

Fluid simulations are computationally intensive and therefore time consuming and expensive. In the field of visual effects, it is imperative that artists be able to efficiently move through iterations of the simulation to quickly converge on the desired result. One common fluid simulation technique is the Smoothed Particle Hydrodynamics (SPH) method. This method is highly parellelizable. I have implemented a method to integrate a Graphics Processor Unit (GPU) accelerated SPH capability into the 3D software package Houdini. This helps increase the speed with which artists are able to move through these iterations. This approach is extendable to allow future accelerations of the algorithm with new SPH techniques. Emphasis is placed on the infrastructure design so it can also serve as a guideline for both GPU programming and integrating custom code with Houdini.

ACKNOWLEDGMENTS

I would first like to acknowledge my wife Stephanie and daughter Madison. Without you putting up with my late nights and procrastinations, this wouldn't have been possible.

I would also like to thank my committee for their guidance along the way.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

The goal of this work is to create a Graphics Processor Unit (GPU) accelerated Smoothed Particle Hydrodynamics (SPH) capability for the Houdini software package. Motivation for doing so is explained below.

I.1. Motivation

In the three-dimensional (3D) computer animation industry there are multiple parts of the production process that are each integral to the success of a project. Each of these segments requires artists with different sets of talents and abilities. Of these segments, the creation of visual effects requires the most reliance on computer algorithms to generate the end result.

Simply stated, effects are the creation of a virtual representation of natural or supernatural phenomena through computer algorithms. These representations are generally too time consuming or difficult to create in a more direct manner by hand. Examples of this include fire, explosions, complex object fracturing, and fluid simulations. See Fig. 1 for examples in modern film. These realistic simulations are generally computationally intensive. Therefore, using current commercially available simulation methods it takes a substantial amount of time for the effects artist to iterate through results to converge on the desired final look of an effect.

The Houdini software package, developed by Side Effects Software [15], is a commercially available tool commonly used in the entertainment industry for the

---

The journal model is *IEEE Transactions on Visualization and Computer Graphics.*

(a) (b)

Fig. 1. Examples of visual effects in film: (a) Fire in *How To Train Your Dragon* [2]. (b) Water and object fracturing in *Inception* [18].

creation of effects. It is very versatile and allows users to easily develop their own tools using several languages, including Python and C/C++, and integrate them with the software package.

Recent advancements in graphics card technology have made large amounts of computation power available at a very affordable cost. These graphics cards generally consist of a GPU chip containing multiple stream processors and shared memory. It is possible to write parallel non-graphics related code to take advantage of these GPUs, allowing performance gains over traditional single threaded processing algorithms. Several companies have developed proprietary tools that speed up simulation algorithms using GPUs, but there are very few non-proprietary implementations available. In this thesis I describe the creation of an extended capability for the Houdini software package that makes use of the GPU to accelerate the SPH algorithm, which is commonly used to simulate liquids and other substances with fluid-like behavior.

I.2. Introduction

Creation of this GPU accelerated SPH capability for Houdini has been driven by the following goals.

- *Create a usable Houdini asset.* The Houdini asset should allow for GPU accelerated SPH simulation on a particle system. Within Houdini, this asset should be easy to work with and not a burden for the user.
- *Determine speed increase, if any, of integrating GPU acceleration in Houdini.* Analysis is made to determine how much the GPU implementation helped increase the speed of calculation.
- *Provide a useful template for future Houdini Development Kit (HDK) projects.* The resulting HDK template should be easy for another developer or student to use as a base for creating their own GPU accelerated Houdini asset.

This thesis is organized as follows: Chapter II summarizes related work. Chapter III discusses the process used for development and the details of the implementation. Chapter IV presents and discusses the results. Chapter V concludes and discusses future work.

CHAPTER II

RELATED WORK

This chapter discusses previous work that applies to this thesis. Section II.1 discusses Smoothed Particle Hydrodynamics (SPH) from conception to more recent advancements. Section II.2 discusses spatial subdivision schemes and the scheme that is implemented in this thesis. Section II.3 discusses Houdini and the Houdini Development Kit (HDK).

II.1. Smoothed Particle Hydrodynamics

Gingold and Monaghan [3] and Lucy [6] first introduced SPH as a means to simulate problems in astrophysics. The authors focused solely on the application of the algorithm to astrophysics.

Monaghan later discussed the application of SPH to a broader selection of problems in 1992 [7]. The basic idea behind SPH is that fluid is represented by a set of discrete elements, or particles. These particles each have attributes, such as mass, position, and velocity. The SPH algorithm uses these attributes along with the distances between particles to approximate fluid properties, such as density, viscosity, and pressure. These properties are calculated by averaging the attributes across the particles based on the distance between the particles. This is accomplished using a set of smoothing kernels, one for each of the properties. Examples of commonly used smoothing kernels are the Gaussian function and cubic splines. The following equation represents the generalized form of this averaging function.

$$A(r) = \sum_j m_j \frac{A_j}{\rho_j} W\left(|r - r_j|, h\right) \tag{2.1}$$

The variables in this function are defined as follows:

- $A(r)$ represents the property to be calculated
- m_j represents the mass of particle j
- A_j represents the value of property A at particle j
- ρ_j represents the density of particle j
- W is the smoothing kernel function
- $|r - r_j|$ is the distance between the particle of calculation and particle j
- h is the influence distance of the smoothing kernel, the smoothing length

In this generalized case, the summation over j implies a summation over all particles in the simulation. In practical implementations, this is not feasible and a spatial partitioning is used so that the summation is only performed on particles that are within the smoothing distance.

To calculate the differential form of the generalized SPH averaging function, we need only use a smoothing kernel that is differentiable. Because of this, we can use normal differentiation and do not have to use finite differences or a grid. For example, to calculate the gradient ∇A, we can use the following equation:

$$\nabla A(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W\left(|r - r_j|, h\right) \tag{2.2}$$

The following subsections detail how SPH has been used to simulate fluid flow. Section II.1.1 discusses the first application of SPH to fluids. Section II.1.2 discusses some of the improvements made to SPH implementations for fluids. Section II.1.3 compares the SPH fluid simulation method with more common Eulerian voxel based simulation methods.

II.1.1. Modeling Fluids With Smoothed Particle Hydrodynamics

SPH was first introduced as a means to simulate fluid flow in 3D applications in 2003 [8]. The algorithm was then modified to create a realistic simulation in a real-time application. These modifications apply the standard Navier-Stokes equation to the algorithm. New smoothing kernels were derived that allowed real-time interactivity.

II.1.1.1. Applying Simplified Physics to Fluids

In Eulerian, grid-based fluid simulation, fluids are described by a velocity field v, a density field ρ, and a pressure field p. There are two general equations that describe the fluid with respect to these changing quantities.

$$\frac{\partial p}{\partial t} + \nabla \cdot (\rho v) = 0 \tag{2.3}$$

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \nabla v \right) = -\nabla p + \rho g + \mu \nabla^2 v \tag{2.4}$$

To create a physically plausible and realistic simulation, both conservation of mass and conservation of momentum must be observed. For Eulerian simulations, equation 2.3 guarantees conservation of mass and equation 2.4 guarantees conservation of momentum. In Equation 2.4, g is an external gravity force and μ is the viscosity of the fluid. This is a form of the Navier-Stokes equation, which describes the motion of fluid substances [13]. There are many forms of the Navier-Stokes equation, this form represents a simplified version used for incompressible fluids.

The use of particles in SPH allows several simplifications of these equations when compared to grid based Eulerian simulations. First, since the number of particles in an SPH simulation is constant, and each particle has a constant mass, conservation of mass is inherently guaranteed. This makes equation 2.3 unnecessary in SPH calcu-

lations. Second, in equation 2.4, first replace the $\partial v/\partial t + v \cdot \nabla v$ term on the left hand side with the substantial derivative Dv/Dt. The substantial derivative of a property in fluid dynamics defines the rate of change of the property subjected to a velocity field. Since particles move with the fluid, the substantial derivative of the velocity field is simply the time derivative of the velocity of the individual particles. This allows us to eliminate the convective term $v \cdot \nabla v$, which leaves the following equation.

$$\rho \left(\frac{Dv}{Dt} \right) = -\nabla p + \rho g + \mu \nabla^2 v \tag{2.5}$$

In equation 2.5 there are three terms on the right hand side of the equation used to model pressure ($f^{pressure} = -\nabla p$), external forces ($f^{external} = \rho g$), and viscosity ($f^{viscosity} = \mu \nabla^2 v$). The sum of these three terms is the fluid force f that determines the change of momentum of the particles.

$$f = f^{pressure} + f^{external} + f^{viscosity} \tag{2.6}$$

Using this, we can calculate the acceleration of particle i with the following equation:

$$a_i = \frac{dv_i}{dt} = \frac{f_i}{\rho_i} \tag{2.7}$$

In equation 2.7, v_i is the velocity of particle i, f_i is the fluid force on particle i, and ρ_i is the density field evaluated at particle i.

II.1.1.2. Calculating Density

Notice that both the particle mass and density appear in equation 2.1. The mass of a given particle is a constant throughout the simulation, but density may change. To find the density at particle i, ρ_i, just insert the desired values into equation 2.1. This

yields the following.

$$\rho_i(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W\left(|r - r_j|, h\right) = \sum_j m_j W\left(|r - r_j|, h\right) \tag{2.8}$$

II.1.1.3. Modeling Pressure

To model the pressure term of equation 2.5, $-\nabla p$, first apply the SPH equation 2.1. This gives the following equation for the pressure on particle i.

$$f_i^{pressure} = -\nabla p\left(r_i\right) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W\left(|r_i - r_j|, h\right) \tag{2.9}$$

A problem with equation 2.9 is that it does not conserve energy as the forces on two interacting particles are not inherently symmetric. When particle i interacts with particle j, it uses only the pressure of particle j to compute the contributing force, and particle j uses only particle i to compute the corresponding opposite force. Since the pressure forces on each particle are not guaranteed to be equal, the pressure forces will not be symmetric in most cases. Muller [8] proposes a simple method of computing the pressure force between two particles by taking the arithmetic mean of the pressures of the two particles. This ensures that the pressure force between two interacting particles is symmetric. This results in the following equation.

$$f_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W\left(|r_i - r_j|, h\right) \tag{2.10}$$

To calculate the pressure at a given particle i, a modified version of the ideal gas state equation is used.

$$p_i = k\left(\rho_i - \rho_0\right) \tag{2.11}$$

In equation 2.11, the pressure of particle i, p_i, is calculated using a pre-determined

pressure constant k, the rest density of the fluid ρ_0, and the current density of the particle ρ_i as calculated by equation 2.8. Using this computed pressure in equation 2.10 will yield the desired fluid force on particle i due to pressure, $f_i^{pressure}$.

II.1.1.4. Modeling Viscosity

Similar to the derivation for pressure, to model the viscosity term of equation 2.5, $f^{viscosity} = \mu\nabla^2 v$, first apply the SPH equation 2.1. This gives the following equation for the viscosity force on a particle i.

$$f_i^{viscosity} = \mu\nabla^2 v\left(r_i\right) = \mu\sum_j m_j \frac{v_j}{\rho_j}\nabla^2 W\left(\left|r_i - r_j\right|, h\right) \tag{2.12}$$

Similar to the derivation for pressure, there is a problem in equation 2.12. The force on two interacting particles are not inherently symmetric because the velocity field varies from one particle to the next. The viscosity term is a way for a particle i to look at neighboring particles and accelerate in the direction of the average velocity of it's environment, in relation to it's own velocity. Therefore, the forces can be made symmetrical by defining the viscosity force as being dependent on the velocity difference between two particles instead of the absolute velocities. This yields the following equation, which is the viscosity force on particle i.

$$f_i^{viscosity} = \mu\sum_j m_j \frac{v_j - v_i}{\rho_j}\nabla^2 W\left(\left|r_i - r_j\right|, h\right) \tag{2.13}$$

II.1.1.5. Selecting Smoothing Kernels

When choosing smoothing kernels to use, the key considerations are stability, computational speed, and accuracy. For stability, it is beneficial to choose kernels whose value and derivatives approach zero as the particle seperation, r, increases. For speed,

it is beneficial to be able to precompute as much as possible of it and minimize the amount of per-particle calculations that must be performed. Muller proposed the use of the following smoothing kernels. $W_{pressure}$ is used in the pressure calculations, $W_{viscosity}$ is used in the viscosity calculations, and W_{other} is used in all other calculations. These kernels were developed specifically for the SPH algorithm through testing by Muller.

$$W_{other}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

$$W_{pressure}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

II.1.1.6. Modeling External Forces

SPH supports the inclusion of external forces on the simulation by applying the forces directly on the individual particles independent of all SPH calculations. Some external force examples could be gravity, collisions, wind, or user interaction forces.

For example, assuming a gravitational force, the following equation would be used:

$$f_i^{external} = m_i g \tag{2.17}$$

Where m_i is the mass of particle i, and g is the acceleration due to gravity.

II.1.2. SPH Improvements

Since the initial introduction of SPH in 3D applications much research has been put into improving both the efficiency and the realism of these simulations. Some of the major improvements are discussed in this section.

One source of unnecessary calculation in SPH simulations is simulating the particles that are beneath the surface of the fluid. Since these are generally fairly stationary particles that are packed at the rest density, simulating all of these particles is an unnecessary process. There has been research on using an adaptive method that modifies the particle size based on the level of detail needed in the particle's area. For example, the particles at the surface of the fluid where detail is needed will be smaller than the particles underneath the surface. This is exhibited in the research of Adams et. al [1], Yan et. al [19] and Hong, House and Keyser [4]. Another take on adaptive sampling was done by Solenthaler and Gross [16]. They propose definining areas of the simulation where complex flow behavior occurs, and using a smaller particle size in these areas as compared to the rest of the simulation.

Another source of inefficiency in the SPH algorithm is that the simulations tend to produce fluids that are compressible if too large a timestep is used, which results in an undesirable 'springy' looking fluid. In physical simulations, a timestep is the amount of time that each simulation calculation uses for it's calculations. Solenthaler and Pajarola address this issue by implementing a predictive-corrective scheme [17]. In this scheme, particle density fluctuations are propogated through the fluid and the pressure values are updated each time step until a target density is met. This sometimes takes longer per timestep to compute, but allows the use of a much larger timestep per simulation step. Results showed that this method outperformed the weakly compressible SPH model by an order of magnitude if a truly incompressible

solution is desired.

II.1.3. Comparison With Voxel Fluid Simulations

SPH has several benefits over more traditional voxel based fluid simulation methods, but also some drawbacks. One major benefit is that SPH inherently guarantees conservation of mass, since the particles themselves represent the mass of the fluid. Another major benefit is that pressure at a given particle is computed from a weighted contribution of neighboring particles in SPH, whereas in a voxel based simulation a more complex linear system of equations is used.

One drawback of SPH is that the algorithm inherently creates a compressible fluid, meaning that particles can temporarily pack in tighter than the target density of the fluid until the particle pressures are re-smoothed to achieve the desired density. This smoothing process can sometimes take several timesteps to achieve. Specifying a larger fluid pressure constant can minimize this effect. However, this requires taking smaller timesteps to avoid an unstable simulation, causing simulation times to increase.

II.2. Spatial Subdivision

When working with a large number of 3D data points, it is beneficial to store the data points in a data structure that allows for efficient processing. For example, many algorithms, including SPH, require searching within a given radius of a point in 3D space for other data points that may affect it. Storing the data points in a smart data structure allows processing only data points that are within that radius, instead of processing every single data point. This results in an efficiency on the order of $O(n)$, depending on the data structure used, instead of $O(n^2)$ if a spatial subdivision

scheme is not used.

One common spatial subdivision scheme used in SPH is to partition the particles in a grid-based data structure [8]. Since the smoothing kernels in SPH have a finite support distance, h, using a grid structure where the grid cells have uniform dimension h can greatly increase simulation performance. This allows searching of a particle's own grid cell and its neighboring grid cells to find potential particles of influence, instead of the whole domain of available particles. A common method for implementing a grid sort as described above is to define a maximum number of particles that are allowed in each grid cell, as well as the maximum total number of cells, and then to preallocate an array of grid cell element structures large enough to accommodate this.

Other, more recent spatial subdivision schemes implemented on the GPU are illustrated by Liu et. al [5]. They describe a multi-level subdivision scheme designed for operation on GPUs that divides the simulation workspace up into uniform grid cells. Once the initial workspace subdivision is complete, they further divide the individual cells up into more discrete cells depending on how the given processing algorithm uses the space. By doing this, they are able to reduce the required final processing even futher by eliminating more false positives.

II.3. Houdini

Houdini is a 3D animation package developed by Side Effects Software [15]. It is used by visual effects and animation studios around the world. The chief distinction of Houdini, when compared to other prominent 3D animation packages, is that it was developed as a purely procedural environment from the ground up. This makes it an ideal platform for developing procedural effects solutions.

The following subsections describe the Houdini animation package, and go into depth on the Houdini Development Kit. Section II.3.1 discusses the procedural operators and network types of Houdini. Section II.3.2 discusses the Houdini Development Kit.

II.3.1. Houdini Operators and Networks

Houdini's procedural nature is based largely on the fact that all tools are built by connecting sequences of operator nodes. Data flows sequentially through and is manipulated by each operator in turn. This node based system allows non-linear workflows. Non-linear refers to the fact that any node in the processing chain can be updated at any time.

The following is a list of the available operator types in Houdini.

- OBJs - Top level object nodes that contain transform information. These are usually container nodes that contain other operator types within them. A 'container' node is simply a node that is used to package a node tree inside, simplifying the top level node graph.
- SOPs - Surface OPerators that contain surface generation nodes. These are generally used for procedural modeling.
- POPs - Particle OPerators that contain nodes used to drive particle simulations.
- CHOPs - CHannel OPerators that contains nodes used to procedurally drive channels for animation or audio manipulation. A channel refers to any input to a node that can be modified. Examples of this are translation, rotation, and scale inputs.
- COPs - Compositing OPerators used to drive a compositing network.
- DOPs - Dynamic OPerators used to drive dynamic simulations such as fluids, rigid bodies, and cloth.

- SHOPs - Shading OPerators used to represent shading types for different renderers.
- ROPs - Render OPerators used to build rendering networks consisting of different passes and dependencies.
- VOPs - Vector Expression (VEX) OPerators used to build nodes of any type. VOP operator networks are built using a series of Houdini provided processing nodes. This is similar to programming using pre-built operator types.

II.3.2. Houdini Development Kit

The Houdini Development (HDK) Toolkit is a set of freely distributed C++ libraries that allows the development of plugins for Houdini and Mantra, which is the rendering engine distributed with Houdini and developed by Side Effects [14]. This is the same set of libraries that Side Effects uses to develop the standard Houdini distribution.

HDK allows customization and the capability to add functionality to any part of Houdini. Here are a some examples of functionality that can be added.

- Adding custom expression functions.
- Adding custom commands.
- Developing custom operators, including but not limited to surface, particle, and dynamics operators.
- Adding the ability to output to a non-standard renderer.
- Adding custom effects to the standard Mantra Renderer.

II.4. Graphics Processor Units and CUDA

This section describes Nvidia GPUs and the CUDA programming language used to code them.

II.4.1. Graphics Processor Units

Graphics Processor Units (GPUs) are specialized integrated circuits found on graphics cards that typically are used to create and manipulate images output to a monitor or other display devices attached. The graphics cards generally contain one or more GPUs as well as a block of Random Access Memory (RAM). They connect to the computer in a Peripheral Component Interconnect (PCI) slot.

Recently, GPUs have been developed to also accomodate general purpose computing. Since the GPUs are designed as a series of stream processors with many processors capable of being run in parallel, they work well for algorithms where an identical processing sequence is executed on many different elements. [11] The processors on the GPU have access to the RAM block on the graphics card, known as their Global Memory, and also have access to a smaller amount of memory built into the GPU, called Shared Memory.

II.4.2. Nvidia CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA for graphics processing [9]. Software developers are able to access the CUDA computing engine by using 'C for CUDA'. This is a set of Nvidia extensions and libraries developed to be used with either C or C++. CUDA lets developers access the instruction set and memory of the GPUs.

CHAPTER III

METHODOLOGY AND IMPLEMENTATION

This chapter focuses on the design methodology used in creating the GPU SPH simulation, and also on the algorithmic implementation details. This thesis was done in part as a learning experience in GPU programming. Therefore, all algorithms were implemented first in a single threaded manner on the CPU, and then implemented in parallel on the GPU. The algorithms were first implemented and visualized using an OpenGL front-end, independent of Houdini. Once the results were verified, the Fluid Simulation code was compiled into a static library and used to create a Houdini asset using the HDK.

III.1. Development Environment

Since this is a GPU intensive task, the GPU and hardware used greatly impact the results. To put the results achieved in perspective, this section details the specifications of the development workstation used.

This work was developed on a Linux workstation running Scientific Linux 6.1. The workstation itself consists of an Intel i7-860 processor running at 2.93 GHz. The system has 12GB of RAM. The graphics card is an Nvidia GTX260 containing a GPU with 216 stream processors and 896 MB of local memory. The Houdini version used is Houdini 11.1.67.

The Eclipse development IDE was used for the coding, and the Subclipse subversion plugin was used for code version control.

III.2. Class Structure

Before jumping into the implementation of the SPH algorithm, a class structure with appropriate virtual functions was developed that allowed seamless transition between the GPU and CPU implementations of the fluid simulation. This scheme consists of an SPH_Fluid superclass with two subclasses, SPH_Fluid_CPU and SPH_Fluid_GPU. These inherit the proper functions and variables that allow seamless switching between the two implementations. This class structure allows code implementation reuse between both CPU and GPU implementations. Fig. 2 gives an overview of how this works.

```
fluid; {Create either a GPU or CPU fluid object named fluid.}
fluid.init(); {Initialize fluid.}
while simulation running do
   fluid.addParticle(); {Add any new particles for this frame.}
   fluid.stepForward(); {Step the simulation forward one frame.}
   fluid.getPartPos(); {Get the particle positions.}
   Move to next frame.
end while
```

Fig. 2. Generalized class implementation.

III.3. Implementing Basic SPH on the CPU

As a first step, a basic SPH implementation working solely on the CPU was created, following the class structure discussed above. This served as both a proof of concept of the algorithm, as well as a foundation to be used for improvements to be discussed

in following sections.

In this implementation, no spatial subdivision scheme was used to sort the particles. For each simulation step, every particle included every other particle in the simulation for pressure and viscosity contributions. This results in an algorithmic efficiency of $O(n^2)$. The outline for this implementation is shown in Fig. 3.

III.4. Spatial Subdivision on the CPU

After the basic SPH algorithm was implemented on the CPU, a spatial subdivision scheme was added to speed up calculations by not performing calculations on particles that are outside of each particle's smoothing radius.

In section II.2 grid-based storage structures were discussed. These were used as the basis for the spatial subdivision used. There were two derivations of this method developed as described below.

For the first derivation, a large block of memory was pre-allocated for the creation of the grid structure. The grid cells were cubic, with the dimensions of each cube being the smoothing radius of the simulation. Each grid cell was given enough memory space to hold a pre-defined number of pointers to particle structures. Fig. 4 gives an overview of this data structure.

```
fluidParameters; {Initialize all fluid parameters}
boundaryConditions; {Initialize implicit boundary conditions}
externalForces; {Initialize external forces}
initialize fluid;
while simulation running do
  Add new particles to simulate;
  s = 0;
  while s < subSteps do
    p = 0;
    while p < numParticles do
      Calculate density, pressure at p due to all particles in system;
    end while
    p = 0;
    while p < numParticles do
      Calculate pressure force at p due to all particles in system;
      Calculate viscosity force at p due to all particles in system;
      Calculate collisions on particle p;
      Calculate external forces on particle p;
      Integrate particle p;
    end while
  end while
end while
```

Fig. 3. Basic SPH algorithm process on CPU.

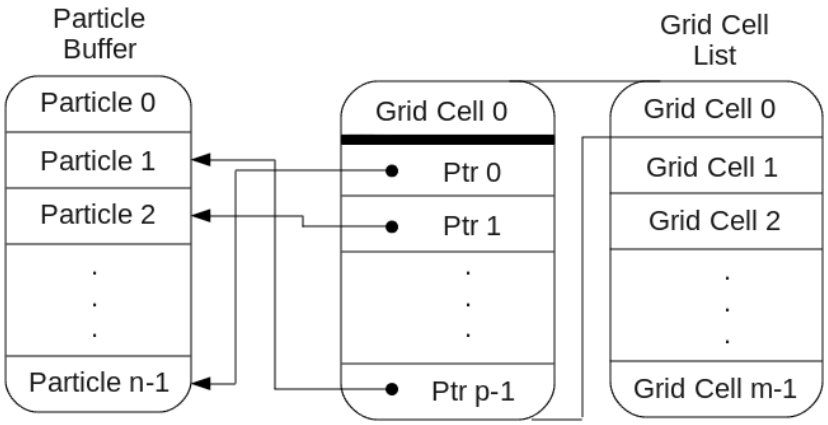


Fig. 4. Basic spatial subdivision structure.

For each simulation substep, the number of grid cells needed was determined by calculating the maximum and minimum position along the three axes. Then a grid structure was created in the pre-allocated memory block. All of the particles were then iterated through to determine in which grid cell they were currently located. A pointer to the particle structure was then placed in that grid cell's memory. When this step is completed, a particle needs to iterate through the particles in it's grid cell and it's neighbor grid cells to perform the fluid calculations. This results in an algorithmic efficiency of $O(mn)$, where m is the average number of particles in a given smoothing radius neighborhood.

Implementation of this derivation is straightforward, but it has some drawbacks. It requires a very large memory footprint to be pre-allocated. This especially comes into play when dealing with GPUs, as they generally have smaller amounts of available memory than the computer workstation. Secondly, since the grid cells themselves only store pointers to the particle data structures, it is not very efficient in terms of memory usage as the data will not be stored in contiguous memory blocks. It would be faster to allocate the grid structure to hold particle structures instead of pointers

to particle structures. However, this would create a memory footprint several times larger still.

To overcome these deficiencies, a modified version of this storage system was derived, as described below.

In the new derivation, memory is pre-allocated for two *int* data types per potential grid cell. These two integer values refer to the number of particles that reside in each grid cell, and the minimum index into the particle array of the particles in each grid cell. Fig. 6 shows this data structure.

All of the particles are then iterated through to determine the 3D coordinates of the grid cell they are located in. Each 3D coordinate is then transformed into a one dimensional index using Equation 3.1.

$$newIndex = indX + indY * dimX + indZ * dimX * dimY; \tag{3.1}$$

In this equation, $indX$, $indY$, and $indZ$ refer to the calculated three dimensional coordinates, while $dimX$ and $dimY$ refer to the x and y dimensions of the grid.

Once these one dimensional indices are calculated, a key-value radix sort is performed using the calculated indices as the keys, and the particle structures as the data. This results in both the particle array and two value array described above sorted by the grid cell that the particles reside in. Fig. 5 depicts the result of this sort.

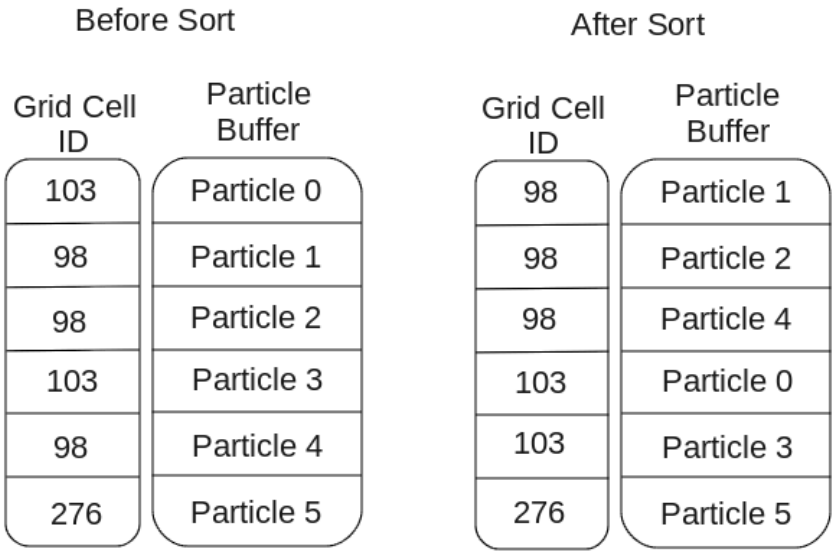


Fig. 5. Key-value radix sort of particles.

After this radix sort is complete, all of the particles are then iterated through once more. For each particle, the corresponding grid cell count is incremented, and the starting index of the particle array for that grid cell is updated. Fig. 6 depicts the result of this step of the algorithm.

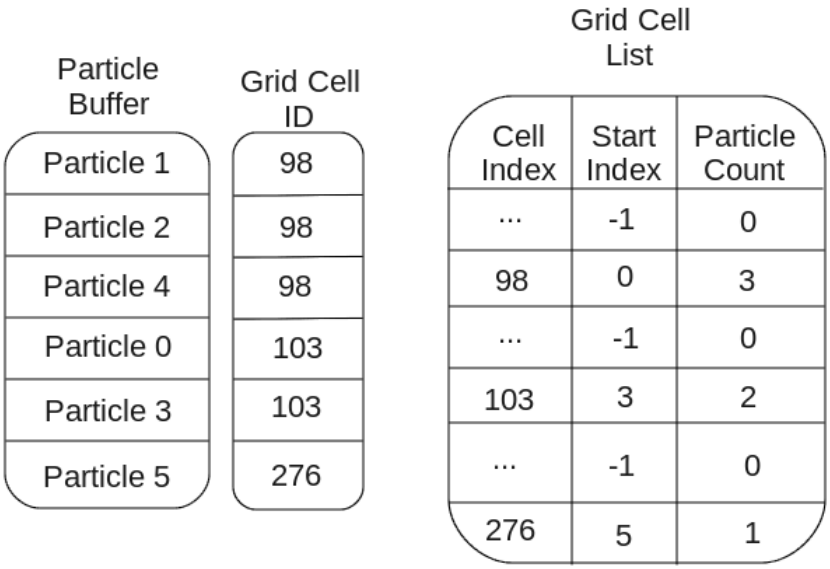


Fig. 6. Filling in the grid structure.

This method does have the drawback that it takes longer to create the grid structure. However, it requires much less memory overhead and is efficient in the SPH calculations as all of the particles will be in sequential memory locations.

Now that the grid cell structure is complete, all of the information is available to perform the SPH calculations. When searching through the particles for a given grid cell, we use the calculated start index into the particle array and the corresponding number of particles in that grid cell. From this point, the algorithm is carried out similar to the basic grid cell structure algorithm. Fig. 7 below depicts the algorithm used for the grid cell implementations.

```
fluidParameters; {Initialize all fluid parameters}
boundaryConditions; {Initialize implicit boundary conditions}
externalForces; {Initialize external forces}
initialize fluid;
while simulation running do
  Add new particles to simulate;
  s = 0;
  while s < subSteps do
    Calculate particle grid structure;
    p = 0;
    while p < numParticles do
      Calculate density, pressure at p due to all particles in neighborhood;
    end while
    p = 0;
    while p < numParticles do
      Calculate pressure force at p due to all particles in neighborhood;
      Calculate viscosity force at p due to all particles in neighborhood;
      Calculate collisions on particle p;
      Calculate external forces on particle p;
      Integrate particle p;
    end while
  end while
end while
```

Fig. 7. Grid accelerated SPH algorithm process on CPU.

III.5. Implementing Basic SPH on the GPU

Algorithmically, implementing a basic SPH algorithm on the GPU is identical to the process described in Section III.3. The only real difference is that data is processed in parallel on the GPU.

There are three additions to the code structure when using the GPU.

- Transfer data to the GPU before processing, and back from the GPU after processing.
- Creation of host GPU code to initiate the GPU processing.
- Creation of kernel GPU code to do the actual GPU processing.

Transferring data to the GPU memory is a straightforward process. To do this, a memory location must first be declared and allocated for the transfer. To declare the memory location on the GPU, a global variable declaration is defined in the CUDA kernel file.

```
__device__ float3* GPU_data;
```

Once the memory is declared on the GPU, it must be allocated. In the following, `dataSize` is the size to allocate for the data structure, in bytes.

```
cudaMalloc((void**)&GPU_data, dataSize));
```

Once memory is allocated in the GPU's memory, the particle data can be copied to the GPU. In the following, `CPU_data` is a character pointer to the data stored in host memory, `GPU_data` is the pointer to the memory allocated on the GPU, `size` is the size of the data transfer in bytes, and `cudaMemcpyHostToDevice` is a flag that specifies data is being transferred from the host CPU to the GPU.

```
cudaMemcpy((char*)GPU_data,CPU_data,size,cudaMemcpyHostToDevice));
```

When the particle data has been transferred to the GPU, it can then be processed. To process data on the GPU, CUDA code must be written for the host processor to

initiate the processing, and a CUDA kernel must be written to perform the processing on the GPU.

The host code consists of calculating the number of GPU blocks and threads to invoke, and calling the kernel code. Since the GPU processing infrastructure consists of a finite number of processor blocks, it must be determined beforehand how many of these processor blocks will be used, and how many processing threads to invoke on each block of processors. Each Nvidia GPU has a maximum number of threads per processor block.

In the case of SPH, this calculation is a simple process. For each frame of simulation, we have a constant number of particles to process. Therefore, to determine how many processor blocks are needed, we divide the total number of particles by the maximum threads per processor block. If this is an exact division with no remainder, the result is the number of blocks that are needed. If there is a remainder, the number of blocks needs to be incremented by 1. This calculation is shown below.

```
numThreads = maxThreadsForGPU;
if(particleNum % numThreads == 0)
    numBlocks = particleNum / numThreads;
else
    numBlocks = (particleNum / numThreads) + 1;
```

With these values calculated, the host code just needs to invoke the kernel code. This is done using the following template code. This code calls a kernel function, `SPH_GPU_Calc` on `numBlocks` GPU processor blocks, each processing `numThreads` threads. In this code, `GPU_data` is the pointer to the data in GPU global memory. `numPoints` is the number of particles.

```
SPH_GPU_Calc <<<numBlocks, numThreads>>> (GPU_data, numPoints);
```

The GPU kernel is a block of code that will be executed in parallel by each thread on each block. Since the same code is executed by all threads, it is necessary

for each thread to have a way of identifying which particle it is working on. CUDA provides a simple way to do this by providing `blockId` and `threadId` variables that are unique for each thread. CUDA provides functionality to access one, two, and three dimensional arrays using these variables. For this thesis a one dimensional array is used. The index of the current particle is calculated using the following code.

```
int ndx = blockIdx.x * blockDim.x + threadIdx.x;
```

Since all kernel invocations are called using one dimensional block and thread identifiers, the particle ID is determined simply by multiplying the current block ID by the dimension of each block, and adding the thread offset within that block.

Once the particle ID has been calculated, each particle's information can be accessed and calculations can proceed as described in Section III.3.

Once processing is completed on the GPU, data is transferred back to the host using the following. The parameter names mean the same as before, with `cudaMemcpyDeviceToHost` being the flag that specifies data is being transferred from the GPU to the host CPU.

```
cudaMemcpy(CPU_data,(char*)GPU_data,size,cudaMemcpyDeviceToHost));
```

Once the particle data has been transferred back from the GPU, it can be displayed using OpenGL or transferred to Houdini. Fig. 8 gives an overview of the simulation loop required for this algorithm.

III.6. Spatial Subdivision on the GPU

For the GPU spatial subdivision implementation, the algorithm used is identical to that in Section III.4, except the bulk of the calculations are performed on the GPU. First, space must be allocated on the GPU to hold the list structures.

```
fluidParameters; {Initialize all fluid parameters}
boundaryConditions; {Initialize implicit boundary conditions}
externalForces; {Initialize external forces}
initialize fluid;
while simulation running do
  Add new particles to simulate;
  Copy particle data to GPU;
  s = 0;
  while s < subSteps do
    Spawn p GPU threads to calculate density of particles;
    Spawn p GPU threads to calculate fluid forces on ;
    Spawn p GPU threads to calculate collisions;
    Spawn p GPU threads to calculate external forces;
    Spawn p GPU threads for integration;
  end while
  Copy partice data from GPU;
end while
```

Fig. 8. Basic SPH algorithm process on GPU.

```
__device__ unsigned int*  GPU_cellMarker;
__device__ List_GPU*  GPU_cellList;
cudaMalloc((void**)&GPU_cellMarker,maxCellSize*sizeof(int)));
cudaMalloc((void**)&GPU_cellList,maxCellSize*sizeof(List_GPU)));
```

When computing the grid structure, the min and max dimensions of the particle system are still determined on the CPU, as are the dimensions of the grid structure. This data is then passed to the GPU so that each particle can calculate it's grid cell location in a parallel fashion using the same algorithm described by Equation 3.1. This grid cell information is stored in the `GPU_cellMarker` memory that has already been allocated.

A key-value radix sort is then performed on the GPU using `GPU_cellMarker` as the keys and the particle data as the values. Nvidia provides a set of parallel sorting algorithms as part of their Thrust library [10]. The radix sort used for this implementation was provided by this library. Usage of this library consisted of typecasting both the `GPU_cellMarker` and the particle data as thrust vectors, and then performing the sort. The particle data and `GPU_cellMarker` structure are both sorted in place, meaning the result from the sort is stored in place of the unsorted version. The code for calling this library is

```
thrust::device_ptr<unsigned int> keyPtr =
    thrust::device_pointer_cast(GPU_cellMarker);
thrust::device_ptr<Fluid_Particle_GPU> valuePtr =
    thrust::device_pointer_cast(GPU_partBuf);
thrust::sort_by_key(keyPtr, keyPtr + numPoints, valuePtr);
```

Once the key-value pairs are sorted, the algorithm proceeds by generating the count of particles in each grid cell, and calculating the start index of each grid cell in the sorted particle list. Since this is done in parallel, memory locking methods must be used to ensure that multiple threads do not modify the same memory location simultaneously. This is achieved using the atomic operations built into CUDA, `atomicInc` and `atomicMin`.

Atomic operations are operations that appear to the rest of the system to occur instantaneously. This guarantees that multiple atomic accesses to memory locations will not conflict with each other. The atomic operation `atomicInc` increments the value in a specified memory location by a value of 1. The atomic operation `atomicMin` compares the value at a specified memory location with a provided value. If the provided value is less than the current value in the memory, it replaces the memory value with the provided value. Otherwise, it does nothing to memory. By creating a CUDA thread for each particle, performing an `atomicInc` on that particle's grid cell and an `atomicMin` between the current minimum grid cell index and the current index, the desired particle count and start index are generated. The following code depicts this operation.

```
__global__ void SPH_GPU_Grid_CreateIndexListStart(
                  unsigned int *cellMarker,
                  List_Element_GPU *cellList,
                  unsigned int particleNum)
{
    // Get Particle Index
    int ndx = blockIdx.x * blockDim.x + threadIdx.x;
    // Make sure we are not on an invalid mem location
    if(ndx >= particleNum) return;
    // Get Grid Index of Particle
    unsigned int currentIndex = cellMarker[ndx];
    // Run Atomic Min on Start Index
    atomicMin(&cellList[currentIndex].startIndex,ndx);
    // Run Atomic Inc on Particle Count
    atomicInc(&cellList[currentIndex].particleCount,particleNum+1);
}
```

Once this data is generated, all of the data needed to complete the algorithm as described in Section III.4 is present, using the principles of GPU coding described in Section III.5. Fig. 9 gives an overview of the simulation loop required for this algorithm.

```
fluidParameters; {Initialize all fluid parameters}
boundaryConditions; {Initialize implicit boundary conditions}
externalForces; {Initialize external forces}
initialize fluid;
while simulation running do
  Add new particles to simulate;
  s = 0;
  while s < subSteps do
    Calculate particle grid structure;
    Copy particle data to GPU;
    Spawn p GPU threads to calculate density of particles using grid;
    Spawn p GPU threads to calculate fluid forces using grid;
    Spawn p GPU threads to calculate collisions;
    Spawn p GPU threads to calculate external forces;
    Spawn p GPU threads for integration;
    Copy partice data from GPU;
  end while
end while
```

Fig. 9. Grid accelerated SPH algorithm process on GPU.

III.7. Implicit Collisions on the GPU

Particle collisions with implicit surfaces were included for both the CPU and GPU implementations. Implicit collisions are collisions with surfaces that can be defined by parametric equations. The implicit surfaces implemented for collision detection were axis aligned planes and vertically aligned cylinders. The following sections discuss the implementation of the implicit collisions.

III.7.1. Collision With Implicit Planes

Collisions with axis aligned implicit planes were implemented on both the GPU and the CPU. By axis aligned, it is meant that the plane is parallel to either the $X = 0$, $Y = 0$, or $Z = 0$ plane.

Using the $Y = 0$ plane as an example case, for each particle a test is done to see if the particle went from a position with a positive y value to a negative y value. If this happens, a collision has occured. To simplify calculations and decrease processing time, the particle is then moved to the $Y = 0$ plane, keeping it's X and Y position coordinates the same. To calculate the collision reflection velocity along the normal of the plane, the dot product of the incoming velocity and the plane normal is taken. For the $Y = 0$ plane example, this will result in the X and Z velocity components going to 0, and the Y velocity component remaining. This velocity along the normal is then subtracted from the incoming velocity, which yields the collision velocity tangent to the plane. Finally, the normal velocity is multiplied by -1 to reflect the particle off the surface.

The normal velocity is then multiplied by an 'elasticity' constant and the tangential velocity is multiplied by a 'friction' constant. This is a simple model of potential energy loss due to the collision. For the final collision response velocity, the tangential

and normal velocities are added together and applied to the particle.

III.7.2. Collision With Implicit Cylinders

Collisions with vertically aligned cylinders were implemented on both the GPU and the CPU. By vertically aligned, we mean that the major central axis of the cylinder is parallel to the Y-Axis.

Using the cylinder centered on the Y-axis with a radius of 1 as an example, each particle is tested to see how far it is from the Y-axis. If the position of a particle is within the radius of the cylinder, the particle is then moved to the approximate collision location on the cylinder by multiplying the X and Y components of the position by the ratio of the radius of the cylinder to the distance of the particle from the cylinder's major axis. This effectively moves the particle onto approximately the surface of the cylinder.

To calculate the response velocity, the normal at the calculated surface point is determined by finding the normalized vector from the cylinder axis to the surface point that is orthogonal to the cylinder axis. To calculate the response velocity along the normal vector, the dot product of the incoming velocity and the calculated normal is taken. This velocity along the normal is then subtracted from the incoming velocity, which yields the collision velocity tangent to the surface point. Finally, the normal velocity is multiplied by -1 to reflect the particle off of the surface.

The normal velocity is then multiplied by an 'elasticity' constant and the tangential velocity is multiplied by a 'friction' constant. This is a simple model of potential energy loss due to the collision. For the final collision response velocity, the tangential and normal velocities are added together and applied to the particle.

III.8. Integrating With Houdini

III.8.1. HDK Implementation

To integrate the fluid simulation code with Houdini, the Houdini Development Kit (HDK) is used to initiate the fluid simulation. The code must first be compiled into a static library that can be referenced by the HDK code. In the HDK code, several things are done.

First, the HDK wrapper file sets up all of the parameters that will be accessible when using the node in Houdini. This includes defining the parameters, setting up their default values, and organizing them in a tabular structure. To define the parameters, a list of `PRM_Name` structures are defined, as shown below. This shows only a subset of the parameters available in the final Houdini SPH solver.

```
static PRM_Name fluidParamNames[] =
{
    PRM_Name("sph_restDensity",          "Rest Density"),
    PRM_Name("sph_pressureConstant",     "Pressure Constant"),
    PRM_Name("sph_viscosity",            "Viscosity"),
    PRM_Name("sph_surfaceTension",       "Surface Tension"),
    PRM_Name("sph_defaultMass",          "Default Mass"),
    PRM_Name("sph_smoothRad",            "Smoothing Radius"),
    PRM_Name("sph_simScale",             "Simulation Scale"),
    PRM_Name("sph_simTimestep",          "Simulation Timestep"),
    PRM_Name("sph_simSubsteps",          "Simulation Substeps"),
    PRM_Name("sph_drag",                 "Drag"),
    PRM_Name("sph_gravity",              "Gravity"),
    PRM_Name(0)
};
```

To set the default values of the parameters, a list of `PRM_Default` structures are defined, as shown below.

```
static PRM_Default fluidParamDefs[] =
{
    PRM_Default(1000),
    PRM_Default(10),
    PRM_Default(0.15),
    PRM_Default(0),
```

```
    PRM_Default(0.00020543),
    PRM_Default(0.006),
    PRM_Default(1),
    PRM_Default(0.002),
    PRM_Default(10),
    PRM_Default(0.999),
    PRM_Default(-93)
};
```

The parameters and the default values must be bound together and have space allocated within HDK. To do this we create a parameter template list, shown below. In the template list, the first entry is the data type of the parameter, the second is the size of the parameter, the third is the memory location of the defined parameter name, and the third is the memory location of the default value.

```
PRM_Template
POP_SPH::myTemplateList[] =
{
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[0], &fluidParamDefs[0]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[1], &fluidParamDefs[1]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[2], &fluidParamDefs[2]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[3], &fluidParamDefs[3]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[4], &fluidParamDefs[4]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[5], &fluidParamDefs[5]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[6], &fluidParamDefs[6]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[7], &fluidParamDefs[7]),
    PRM_Template(PRM_INT,  1,&fluidParamNames[8], &fluidParamDefs[8]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[9], &fluidParamDefs[9]),
    PRM_Template(PRM_FLT_J,1,&fluidParamNames[10],&fluidParamDefs[10])
    PRM_Template())
};
```

Once the parameters are setup for evaluation, we use a couple of functions written in the HDK wrapper. One of them is used to initialize the fluid simulation when the HDK code is called. This creates a new fluid simulation structure with enough space to hold the current number of particles in the simulation, and accesses all of the parameters in Houdini and applies them to the fluid structure. Once the fluid structure has been initialized, a function is called that reads all of the particles from Houdini and writes them out to the fluid structure. The following code block

illustrates the general idea of how that is done.

```
GEO_PrimParticle*   part;
GEO_ParticleVertex* pvtx;
// Process each particle primitive fed into this POP and then
// each particle within that primitive.
for (part = myParticleList.iterateInit() ;
     part ; part = myParticleList.iterateNext())
{
    for (pvtx = part->iterateInit() ; pvtx ; pvtx = pvtx->next)
    {
        myCurrPt = pvtx->getPt();
        p = myCurrPt->getPos();
        v = myCurrPt->getValue<UT_Vector3>(data->getVelocityOffset());
        a = myCurrPt->getValue<UT_Vector3>(data->getAccelOffset());
        pos = Point3D(p.x(), p.y(), p.z());
        vel = Point3D(v.x(), v.y(), v.z());
        acc = Point3D(a.x(), a.y(), a.z());
        fluid->addParticle(pos, vel, acc, mass);
    }
}
```

In this code block, `GEO_PrimParticle` represents a particle system primitive. It is possible for multiple particle systems to be input into the SPH solver node within Houdini. This allows for effective processing of all present particle systems using the same simulation. `GEO_ParticleVertex` is used to hold the location of individual particles in the particle system.

Once the fluid structure is initialized and the particles have all been written into the fluid structure, the simulation can be run. This is accomplished by simply calling the `stepForward()` function present as a member of the fluid class.

When the simulation has completed for the current frame, all particles are read from the fluid simulation and written back into Houdini. The following code block illustrates the general idea of how this is done.

```
int i=0;
GEO_PrimParticle*   part;
GEO_ParticleVertex* pvtx;
// Process each particle primitive fed into this POP and then
// each particle within that primitive.
for (part = myParticleList.iterateInit() ;
```

```
     part ; part = myParticleList.iterateNext())
    {
    for (pvtx = part->iterateInit() ; pvtx ; pvtx = pvtx->next)
    {
        myCurrPt = pvtx->getPt();
        pos = fluid->getPosition(i);
        vel = fluid->getVelocity(i);
        acc = fluid->getAccel(i);
        p.assign(pos.x, pos.y, pos.z);
        v.assign(vel.x, vel.y, vel.z);
        a.assign(acc.x, acc.y, acc.z);
        myCurrPt->setPos(p);
        myCurrPt->setValue<UT_Vector3>(data->getVelocityOffset(), v);
        myCurrPt->setValue<UT_Vector3>(data->getAccelOffset(), a);
        i++;
    }
}
```

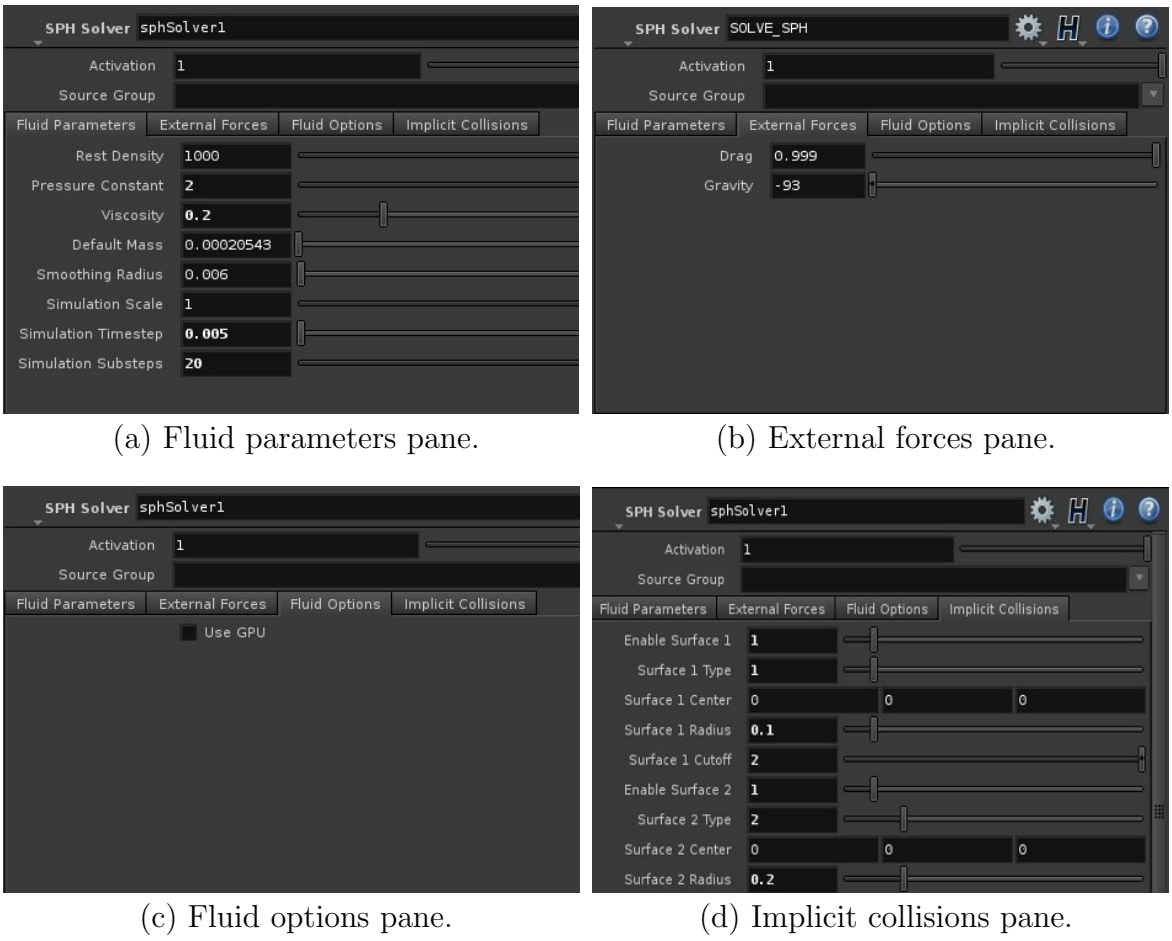
After this is done, the fluid structure memory is deallocated and HDK cleans up the rest of the variables that were used. The HDK code then exits back to Houdini, and will be called again from Houdini when the next frame is ready to be evaluated.

III.8.2. Houdini Implementation

Once the fluid simulation is compiled into a Houdini Particle Operator node using HDK, it can be imported into a Houdini Particle Operator network as a node. This node will have all of the parameters needed to control the fluid simulation. Since these parameters are inside of Houdini, they can be keyframed as desired. Fig. 10 shows the GUI parameter interface to the node.

The following list defines each of the parameters for the node.

- Activation - Whether or not the node is active.
- Source Group - If the input particles are grouped, which group or groups should be affected by this node.
- Rest Density - Rest density of the SPH fluid simulation.
- Pressure Constant - Pressure constant of the SPH fluid simulation.
- Viscosity - Viscosity of the SPH fluid simulation.



(a) Fluid parameters pane.

(b) External forces pane.

(c) Fluid options pane.

(d) Implicit collisions pane.

Fig. 10. Houdini SPH solver node attributes.

- Default Mass - Default particle mass to use if mass for the particle isn't already defined.
- Smoothing Radius - Smoothing radius of the SPH fluid simulation.
- Simulation Timestep - Timestep to use for each frame of the simulation.
- Simulation Substeps - How many substeps to take for each frame.
- Drag - Per timestep drag to apply to each particle.
- Gravity - Gravity to apply to each particle.
- Use GPU - Whether or not to use GPU acceleration.
- Enable Surface x - Enable use of implicit surface number x.
- Surface x Type - Which type of implicit surface to use. Options are: 0 - Constant

X Plane, 1 - Constant Y Plane, 2 - Constant Z Plane, 3 - Sphere Inside, 4 - Sphere Outside, 5 - Cylinder Inside, 6 - Cylinder Outside

- Surface x Center - Center of implicit surface x.
- Surface x Radius - Radius of implicit surface x.
- Surface x Cutoff - Max Y Axis cutoff to be used on cylinders.

The node's input requires a connection to some particle source. The node will process the particles provided to it's input on each timestep, and pass the particles along to the next node in the list. One complication that arose is that the Houdini POP network integrates the particle network at the end of each frame's processing. This is not desirable, as the fluid simulation sometimes requires multiple substeps per frame to complete it's calculation. To deal with this issue, a system of nodes were put in place to store the current velocities at the end of each frame and transfer them back to the particles at the start of the next frame. Each particle's velocity was then zeroed out at the end of a frame. By zeroing out the velocity, it effectively negates the built in integration by Houdini because there are no velocity values to update positions. Fig. 11 shows this basic node network. In this figure, the `SOURCE` node generates the particles. The `VEL_INIT` node creates a particle attribute that will be used to store the current velocity. The `VEL_SET` node sets the current velocity to the custom velocity particle attribute. The `SOLVE_SPH` node is the implemented SPH solver. The `VEL_GRAB` node then grabs the current velocity and stores it in the custom created velocity attribute. Finally, the `VEL_ZERO` node sets the current velocity to zero so the integration will not affect the current particles.

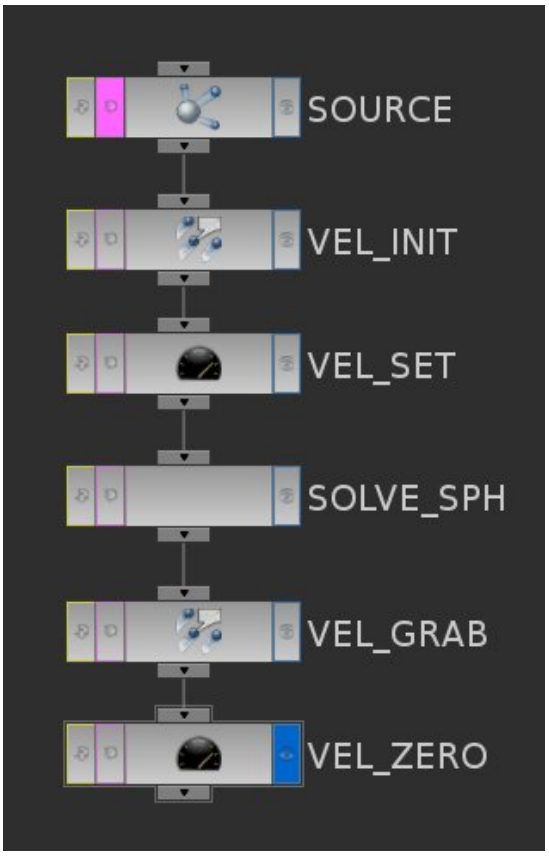


Fig. 11. Example Houdini particle operator network.

Now that the particle network is completed, a particle fluid surface was applied to the particles in a surface operator network. Fig. 12 shows this basic node network. In this figure, the `SOURCE_GEO` node is the source geometry used to generate the particles. The `TRANSFORM` node is used to transform the emitter in the scene as desired. The `POPNET` node is the particle operator network described by Fig. 11. The output of the `POPNET` node is a particle system, and therefore needs a particle fluid surface fitted to it for realistic rendering. The `SURFACE_GEO` node places the particle fluid surface around the particles.

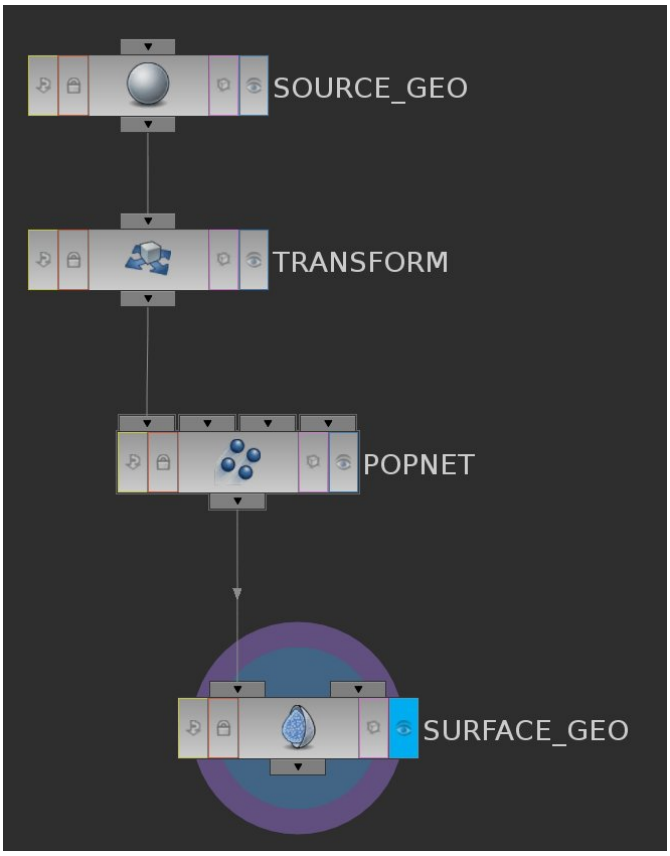


Fig. 12. Example Houdini surface operator network.

CHAPTER IV

RESULTS AND DISCUSSION

We set out to accomplish several goals in this thesis. First, we wanted to implement SPH fluid simulation on the GPU as a means of simulation acceleration. Second, we wanted to provide this GPU accelerated fluid simulation as an extended capability for Houdini. Third, we wanted to compare the results of the fluid simulation code on the GPU with the same algorithm code written for the CPU to determine how much, if any, the GPU accelerates the simulation. Fourth, we wanted to provide a framework for future users willing to develop their algorithms on the GPU, and also provide a framework for developing extended functionality for Houdini using the HDK.

IV.1. GPU Implementation

The first step was to get SPH simulations working in a Houdini-independent, OpenGL environment using the same algorithm on both the CPU and the GPU. Using the implementation process outlined in Chapter III, we were able to simulate fluid in multiple environments bounded by implicit surfaces.

The first environment simulated was fluid in a box with no top. The walls of the box were simulated as implicit planes. Fluid particles were dropped into the box from above and allowed to settle. Fig. 13 illustrates some of the results from this simulation. It is important to note that the fluid is represented only by particles in this work. Particle fluid surface extraction was not implemented. The Houdini Particle Fluid Surface was used for later renders.

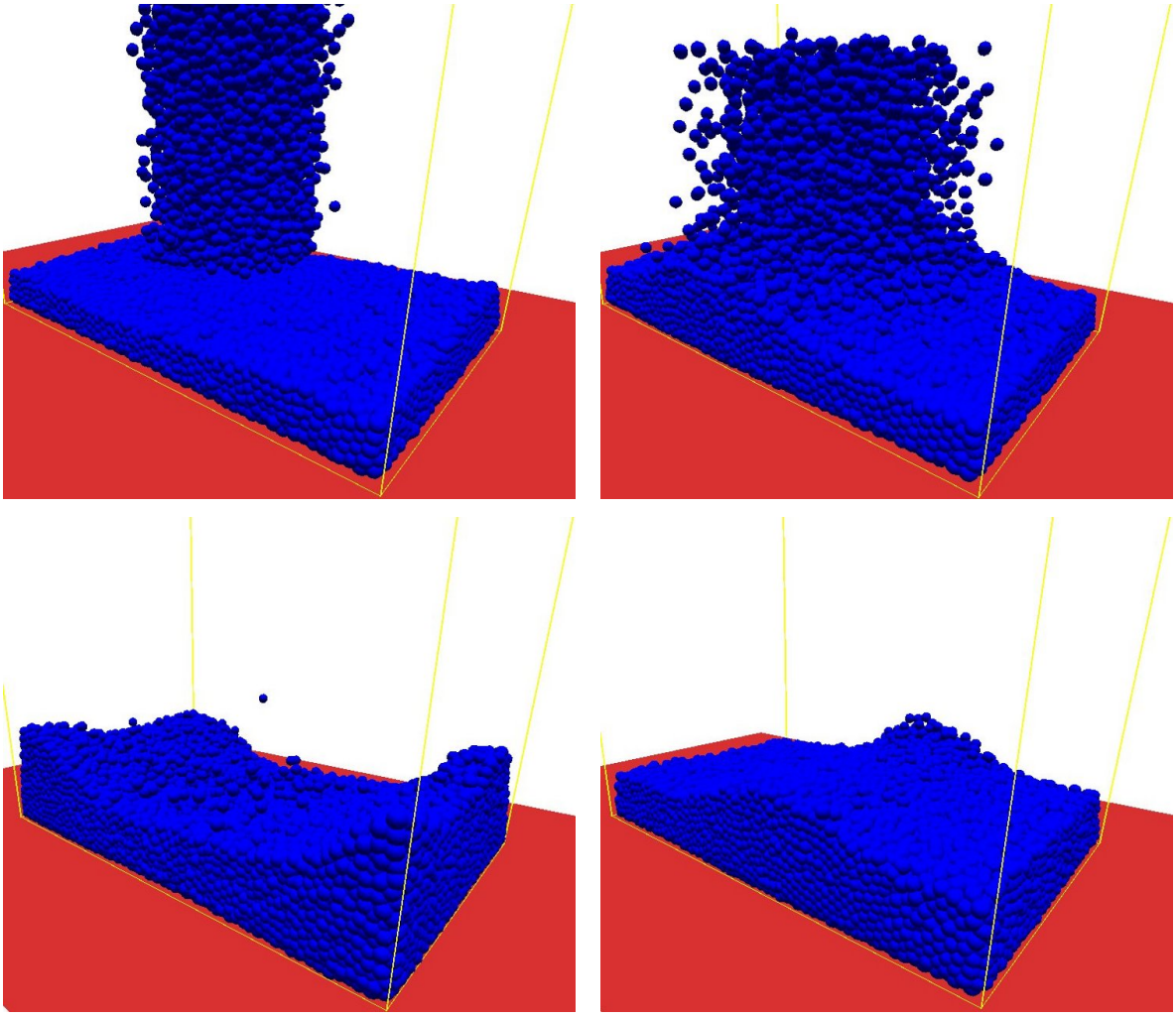


Fig. 13. OpenGL results from GPU simulation in a box environment.

The second environment simulated was fluid in a cylindrical "water fountain". Implicit cylinders were used as an interior and exterior wall of the fountain. A maximum heigth of intersection was introduced, allowing fluid to go over the walls of the fountain above a certain height.

IV.2. Adding Houdini Functionality

The CPU and GPU simulations were imported into a Houdini digital asset using HDK as described in Section III.8. When designing the Houdini interface, we decided that an important feature should be ease of use. To do this, a multi-tab GUI interface was designed with the simulation parameters grouped in tabs with other similar parameters. Fig. 10 shows this interface.

In Fig. 14 a series of rendered images from a scene using the result of this work are shown. In this scene, a rock falls into a fountain of water causing a splash.

IV.3. Performance Analysis

Performance data was collected for the four processing schemes.

- Basic CPU implementation as described in Section III.3 in both the OpenGL and Houdini environments.
- Basic GPU implementation as described in Section III.5 in both the OpenGL and Houdini environments.
- Grid accelerated CPU implementation as described in Section III.4 in both the OpenGL and Houdini environments.
- Grid accelerated GPU implementation as described in Section III.6 in both the OpenGL and Houdini environments.

Data was collected using several different particle counts in both simulation environments described in Section IV.1. Table 1 and Fig. 15 display the results from the OpenGL fountain implementation. Table 2 and Fig. 16 display the results from the OpenGL box implementation. Table 3 and Fig. 17 display the results from the Houdini fountain implementation. Table 4 and Fig. 18 display the results from the Houdini box implementation.

(a)

(b)

Fig. 14. Results from GPU simulation in a fountain environment in Houdini, rendered with Mantra.

(c)

(d)

Fig. 14. Continued.

TABLE 1
Tabular simulation results of fountain environment in OpenGL.
Frames per second limited to a maximum of 24.

Particle Count	CPU Simple	CPU Grid	GPU Simple	GPU Grid
100	24	24	24	24
1000	2.3	24	24	24
10000	<0.1	4.5	1.7	24
100000	<0.1	0.24	<0.1	5.7

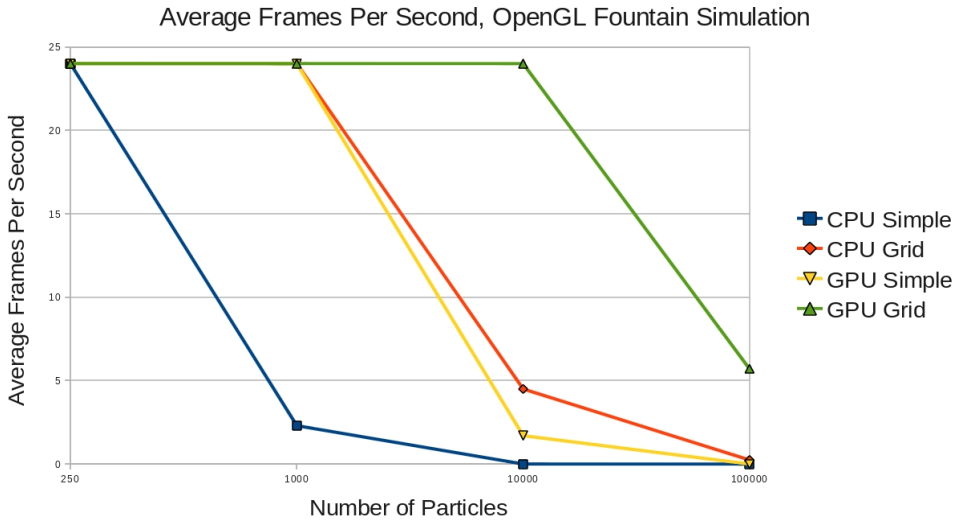


Fig. 15. Simulation results of fountain environment in OpenGL. Frames per second limited to a maximum of 24.

TABLE 2
Tabular simulation results of box environment in OpenGL.
Frames per second limited to a maximum of 24.

Particle Count	CPU Simple	CPU Grid	GPU Simple	GPU Grid
100	24	24	24	24
1000	2.2	24	24	24
10000	<0.1	2.7	1.67	24
100000	<0.1	0.17	<0.1	4.5

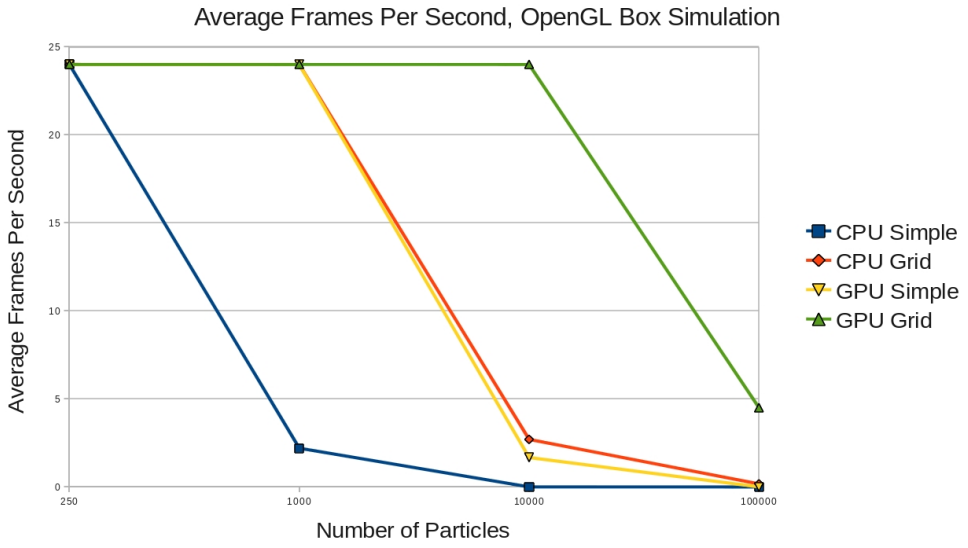


Fig. 16. Simulation results of box environment in OpenGL. Frames per second limited to a maximum of 24.

TABLE 3
Tabular simulation results of fountain environment in Houdini.
Results shown in seconds.

Particle Count	CPU Simple	CPU Grid	GPU Simple	GPU Grid
100	3.4	2.3	2.1	2.2
1000	111.3	4.3	5.3	3.2
10000	1200	58.7	150.8	10.9
100000	3000	700	1400	116.7

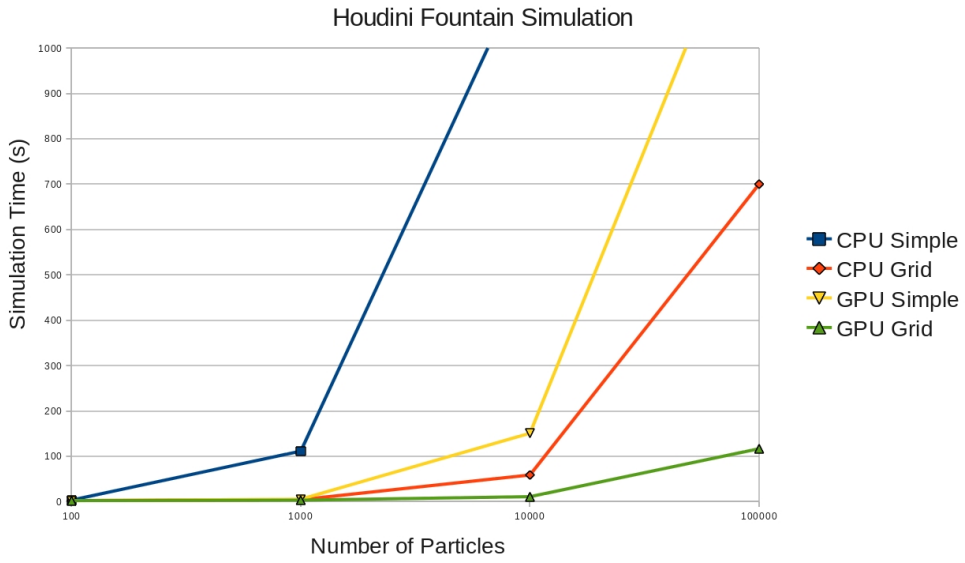


Fig. 17. Simulation results of fountain environment in Houdini.

TABLE 4
Tabular simulation results of box environment in Houdini.
Results shown in seconds.

Particle Count	CPU Simple	CPU Grid	GPU Simple	GPU Grid
100	3.3	2.3	2.3	2.3
1000	110.6	4.9	5.5	3.1
10000	1300	119.6	151.6	12.8
100000	3200	800	1500	161.6

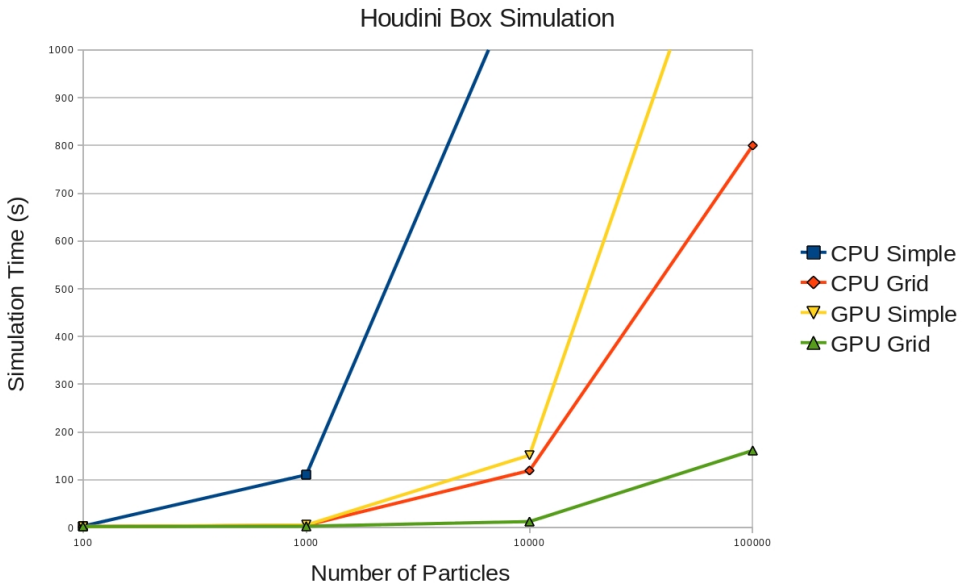


Fig. 18. Simulation results of box environment in Houdini.

The graphs clearly show that the GPU implementation is significantly faster than the same implementation on the CPU in both the OpenGL and Houdini environments. One interesting result was that the CPU grid implementation yielded similar results to the GPU simple implementation, particularly in the OpenGL environment. Another interesting note is that the OpenGL environment was significantly faster than simulation in the Houdini environment. This can be attributed to the overhead in reading and writing particle data to and from Houdini, and the overhead of processing the particle system inside Houdini itself.

Performance data was also gathered for two different graphics cards installed in the same workstation. Along with the Nvidia GTX260 used for development, the simulation was run on an Nvidia GT210. Table 5 shows the differences in these two cards.

Fig. 19 and Fig. 20 show the comparison results of these two cards in the OpenGL environments.

The graphs show that the GTX260 is faster in all cases when compared to the GT210. However, there is not a linear relationship between card processing power and the results because transferring data to and from the graphics card is the same in both cases and is a large portion of the overall processing time. Being able to decrease the number of simulation substeps by using a more advanced SPH method like the Predictive-Corrective Incompressible SPH method would help alleviate this data transfer overhead.

IV.4. Providing a Framework

We wanted to provide a framework for future users to develop their algorithms on the GPU. We also wanted to provide a framework to develop extended functionality for

TABLE 5
GT210 vs GTX260 hardware comparison.

GPU	Steam Processor Count	Stream Processor Clock	Memory
Nvidia GT210	16	520 MHz	512 MB
Nvidia GTX260	216	576 MHz	896 MB

Houdini using HDK. Both the GPU and HDK framework presented in this thesis could be adapted to many other algorithms. This will aid other students or professionals in implementing their own processing algorithms on the GPU and using Houdini.

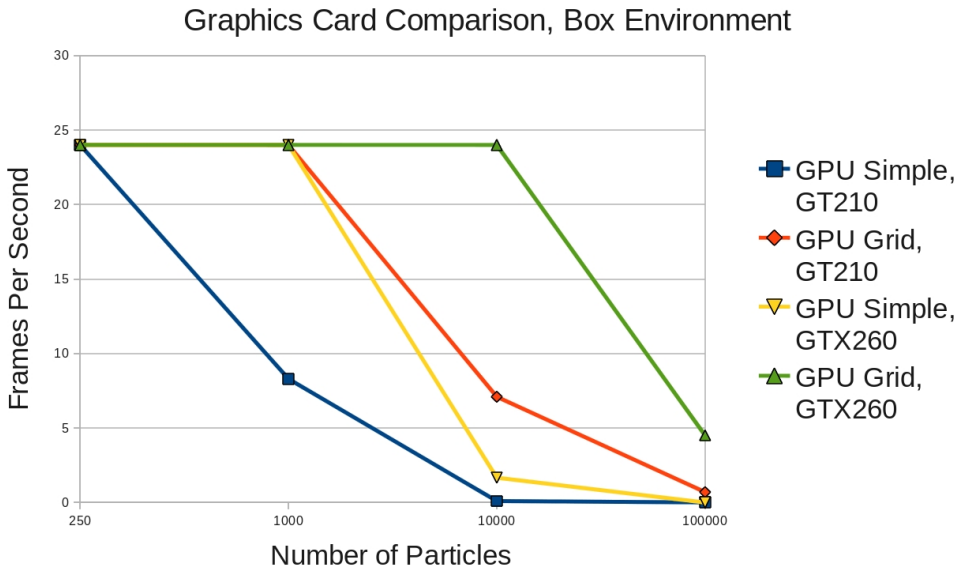


Fig. 19. GT210 vs GTX260 comparison in box environment. Frames per second limited to a maximum of 24.

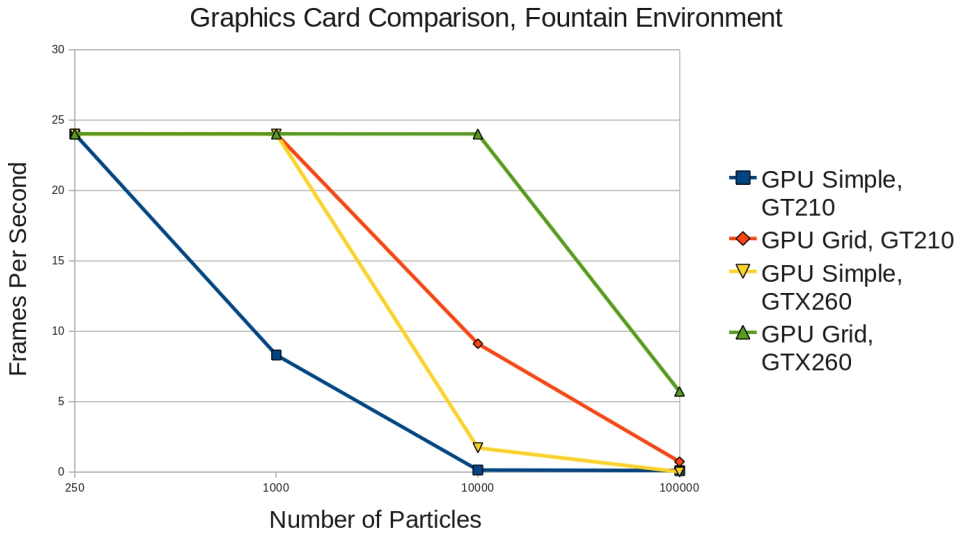


Fig. 20. GT210 vs GTX260 comparison in fountain environment. Frames per second limited to a maximum of 24.

CHAPTER V

CONCLUSION AND FUTURE WORK

We have developed a GPU accelerated SPH fluid simulation that operates both in an OpenGL environment, and as a plug-in for the Houdini software package. Our implementation makes use of a grid based spatial partitioning scheme to accelerate fluid calculations, but also implements a scheme without spatial partitioning for efficiency comparison. Our implementation provides customizable viscosity and pressure calculations for the fluid. It also allows for GPU calculation of external forces such as gravity and drag. It allows for the computation of collisions against simple environment boundaries using implicit surfaces on the GPU as well.

Our implementation was developed by implementing the same algorithms on both the GPU and CPU, allowing for easy comparison of performance between the two. In both the OpenGL and Houdini environment, the GPU algorithm is able to run the simulation in a smaller amount of time.

This work can be extended both by improving the SPH simulation and implementation, and by improving the Houdini HDK implementation, as explained below. Also, the OpenGL implementation could be extended in several ways.

The SPH simulation used does not include the most recent algorithm advancements. Newer advancements, such as the adaptive particle size sampling and predictive-corrective imcompressible schemes described in Section II.1.2 would likely improve the efficiency of the algorithm. The adaptive particle size sampling scheme allows for fewer particles in the sampling. The predictive-corrective incompressible scheme allows for the use of large simulation timesteps, which in turn would require fewer calculations for each simulation frame.

The SPH algorithm on the GPU could also be implemented in a more efficienct manner. The implementation currently stores all particle and grid information in global memory on the GPU. While this works, a more efficient scheme could be implemented that makes use of shared memory for each processor block. For example, a processor block could be allocated to calculate the necessary information for one grid cell. This would include initially copying all particle information for that grid cell and neighboring grid cells into that processor block's shared memory, and then using the information from shared memory to perform the fluid calculations. Once processing is complete on the particles in that grid cell, the particle information would be transferred back to global memory for future calculations.

One other addition that would be very beneficial would be to use the GPU for collision calculations with polygonal faces. Currently the GPU based simulation only calculates collisions with implicit surfaces.

The Houdini HDK implementation could also be improved. Currently it makes use of the Houdini Particle Operator networks. This allows integration with other Houdini aspects, but a more dynamic implementation could be created that makes direct use of Houdini Dynamic Operators. This would allow more 'seamless' coupling with other dynamic objects in the scene.

The OpenGL implementation could be improved in several ways. The most obvious improvement would be to add a particle surface extraction functionality to this code, and render an actual fluid surface in a nicely rendered environment instead of just the particles. Another improvement could be to integrate this simulation into part of an immersive visualization environment [12].

The fluid simulation framework presented in this thesis is a good example for accelerated implementations of various algorithms using Nvidia GPUs and CUDA. The strong object oriented design and the class implementation used could be applied

to most parallelizable algorithms. Furthermore, it could serve as an example for future projects that want to use Houdini's HDK as a means for visualizing simulations.

REFERENCES

[1] B. Adams, M. Pauly, R. Keiser, and L. Guibas, "Adaptively sampled particle fluids," in *ACM SIGGRAPH 2007 papers*, ser. SIGGRAPH '07. New York, New York, USA: ACM, 2007. http://doi.acm.org/10.1145/1275808.1276437

[2] Dreamworks Animation, "How To Train Your Dragon," Film, 2010.

[3] R. Gingold and A. Monaghan, "Smoothed particle hydrodynamics: theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–398, 1977.

[4] W. Hong, D. House, and J. Keyser, "Adaptive particles for incompressible fluid simulation," in *The Visual Computer (Proceedings of Computer Graphics International 2008)*, 2008, pp. pp. 535–543.

[5] F. Liu, T. Harada, Y. Lee, and Y. Kim, "Real-time collision culling of a million bodies on graphics processing units," in *ACM SIGGRAPH Asia 2010 papers*, ser. SIGGRAPH ASIA '10. New York, New York, USA: ACM, 2010, pp. 154:1–154:8. http://doi.acm.org/10.1145/1866158.1866180

[6] L. Lucy, "Smoothed particle hydrodynamics," *Annual Review of Astronomy and Astrophysics*, vol. 30, pp. 543–574, 1992.

[7] J. Monaghan, "A numerical approach to the testing of the fission hypothesis," *The Astronomical Journal*, vol. 82, pp. 1013–1024, 1977.

[8] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '03.

Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 154–159. http://dl.acm.org/citation.cfm?id=846276.846298

[9] NVIDIA Corporation, "Parallel Programming and Computing Platform - CUDA - NVIDIA," http://www.nvidia.com/object/cuda_home_new.html, 2012.

[10] NVIDIA Corporation, "Thrust: A CUDA Library," http://research.nvidia.com/news/thrust-cuda-library, 2012.

[11] NVIDIA Corporation, "World Leader in Visual Computing Technologies - NVIDIA," http://www.nvidia.com, 2012.

[12] F. Parke, "Lower cost modular spatially immersive visualization," in *Proceedings of the 2nd International Conference on Virtual Reality*, ser. ICVR'07. Berlin: Springer-Verlag, 2007, pp. 142–146. http://dl.acm.org/citation.cfm?id=1770090.1770107

[13] D. Pnueli and C. Gutfinger, *Fluid Mechanics*. Cambridge Univ. Press, New York, 1992.

[14] Side Effects Software, "HDK: Houdini Development Kit," http://www.sidefx.com/docs/hdk11.0/hdk_intro.html, 2011.

[15] Side Effects Software, "Houdini - 3D Animation Tools," http://www.sidefx.com, 2011.

[16] B. Solenthaler and M. Gross, "Two-scale particle simulation," *ACM Trans. Graph.*, vol. 30, no. 4, pp. 81:1–81:8, Aug. 2011. http://doi.acm.org/10.1145/2010324.1964976

[17] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible sph," in *ACM SIGGRAPH 2009 papers*, ser. SIGGRAPH '09. New York, New York, USA: ACM, 2009, pp. 40:1–40:6. http://doi.acm.org/10.1145/1576246.1531346

[18] Warner Brothers Pictures, "Inception," Film, 2010.

[19] H. Yan, Z. Wang, J. He, X. Chen, C. Wang, and Q. Peng, "Real-time fluid simulation with adaptive sph," *Comput. Animat. Virtual Worlds*, vol. 20, no. 23, pp. 417–426, Jun. 2009. http://dx.doi.org/10.1002/cav.v20:2/3

VITA

Mathew Allen Sanford
College of Architecture
Texas A&M University
C108 Langford Center
3137 TAMU
College Station, Texas 77843-3137
mat@viz.tamu.edu

Education

M.S., Visualization,	Texas A&M University, August 2012
B.S., Computer Engineering,	Texas A&M University, December 2004