2011

# Real-Time Particle Systems in the Blender Game Engine

Ian Johnson

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

# REAL-TIME PARTICLE SYSTEMS IN THE BLENDER GAME ENGINE

By

IAN JOHNSON

A Thesis submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2011

Ian Johnson defended this thesis on August 24, 2011.

The members of the supervisory committee were:

Gordon Erlebacher
Professor Directing Thesis

Tomasz Plewa
Committee Member

Anter El-Azab
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with the university requirements.

I dedicate this thesis to my parents David and Corine, who taught me both explicitly and implicitly the value of education. All my life they have fostered my curiousity and encouraged me to explore, for which I am eternally grateful.

# ACKNOWLEDGMENTS

This thesis was made possible by the help and support of many people both near and far. First and foremost I must thank my Advisor, Dr. Gordon Erlebacher for his relentless support and encouragement. Dr. Erlebacher has an infectious love for improvement and were it not for his help, both the code and my understanding would be far short of where they are today. I thank him for training me to think like a scientist.

I must thank Dr. Anter El-Azab and Dr. Tomasz Plewa not only for taking the time to be in my committee but for being excellent educators who have broadened and deepend my understanding of scientific computing.

I would also like to thank my colleagues and friends in the Department of Scientific Computing who have spent countless hours in the vizlab discussing, helping and playing. Thank you Evan, Andrew, Nathan, Myrna, Steve and Olmo.

This project would not have been possible without the help and support of the wonderful Blender community, thank you Moguri, dfelinto, Mike Pan and everyone in the IRC channel, BlenderArtists and BlenderNation who helped and encouraged me.

Finally I would like to thank my girlfriend Michelle for her support, including but not limited to, making coffee for me all those late nights.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Advances in computational power have lead to many developments in science and entertainment. Powerful simulations which required expensive supercomputers can now be carried out on a consumer personal computer and many children and young adults spend countless hours playing sophisticated computer games. The focus of this research is the development of tools which can help bring the entertaining and appealing traits of video games to scientific education.

Video game developers use many tools and programming languages to build their games, for example the Blender 3D content creation suite. Blender includes a Game Engine that can be used to design and develop sophisticated interactive experiences. One important tool in computer graphics and animation is the particle system, which makes simulated effects such as fire, smoke and fluids possible. The particle system available in Blender is unfortunately not available in the Blender Game Engine because it is not fast enough to run in real-time.

One of the main factors contributing to the rise in computational power and the increasing sophistication of video games is the Graphics Processing Unit (GPU). Many consumer personal computers are equipped with powerful GPUs which can be harnassed for general purpose computation.

This thesis presents a particle system library is accelerated by the GPU using the OpenCL programming language. The library integrated into the Blender Game Engine providing an interactive platform for exploring fluid dynamics and creating video games with realistic water effects. The primary system implemented in this research is a fluid simulator using the Smoothed Particle Hydrodynamics technique for simulating incompressible fluids such as water.

The library created for this thesis can simulate water using SPH at 40fps with upwards

of 100,000 particles on an NVIDIA GTX480 GPU. The fluid system has interactive features such as object collision, and the ability to add and remove particles dynamically. These features as well as phsyical properties of the simulation can be controlled intuitively from the user interface of Blender.

# CHAPTER 1

# INTRODUCTION

Video games have seen remarkable growth in recent times, and are one of the largest entertainment industries [3]. Interactive games are played by children and adults of all ages, on specialized consoles, on workstations and even telephones. Although the pervasiveness of games in our culture is not in dispute, their effect is [4]. Parents raise concern over the amount of time children spend playing games, and what their children may be learning from the content in these games. The aim of this research is to help develop games which encourage interest in science and teach scientific concepts, methodology and thought processes. Applying video game technology to education is an ongoing area of research and practice [5].

Tools for building games are varied and many, ranging from complex 3D modeling programs, sophisticated physics simulators and design authoring programs. Many professional tools costing thousands of dollars are used by the gaming industry today, while there exist competitive free and open source projects supported by everyone from hobbyists to major studios. These free and open source projects are quickly catching up to their professional counterparts and often have an extensive community surrounding their use and development. Educators wishing to develop games are not likely to have the large financial resources required to produce a successful game, but using the free tools available fun and captivating games can be constructed. These tools include the Blender 3D content creation suite and game engine, the Bullet physics engine, audio authoring programs such as Audacity and the 2D media editor GIMP.

One tool missing from this free toolbox is an interactive fluid simulator. Fluid simulations have been increasingly used by the film and entertainment industry for better special effects in movies and television, and the game industry has been following suit. Fluid simulation is computationally expensive and difficult to achieve at the speeds needed in game development.

Recent developments in graphics programming and hardware have made it feasible to simulate convincing fluids at frame rates suitable for games. The GPU was developed as a way to accelerate the 3D graphics rendering used in PC video games and has been a huge commercial success. A modern high-end consumer GPU has the same computational power as a supercomputer from 10 years prior. Most methods from Computational Fluid Dynamics developed for supercomputers have already beeen ported to the consumer desktop or laptop.

The goal of this research is to develop an open source tool implementing the Smoothed Particle Hydrodynamics (SPH) method for fluid simulation that can be used in developing educational games. First an overview of game development with the Blender software suite is given in Chapter 2. Chapter 3 introduces the concepts behind GPU computing. Particle systems in general are covered in Chapter 4 followed by the theory behind the SPH method in Chapter 5. The implementation details of the project are given in Chapter 6, and the results of the effort described in Chapter 7. The thesis concludes with an overview of future work planned for the project in Chapter 8.

## 1.1 Related Work

Interactive fluid simulation for use in games draws heavily from Computational Fluid Dynamics (CFD) and Computer Graphics research. The use of CFD methods for film and animation spans two decades [6] and has improved in performance tremendously. One of the most common techniques for interactive simulations of fluids is the Smoothed Particle Hydrodynamics (SPH) method because of its relatively low cost and high flexibility. It was adapted for use in computer animation by Desbrun et al. in 1996 [7].

Blender, a popular 3D content creation suite, and the software used as a platform for this thesis includes two fluid simulators, one based on SPH [8] and one based on the Lattice Boltzmann Method [9]. These simulators are not designed to run in real-time, and are not available in the Blender Game Engine.

Smoothed Particle Hydrodynamics works by representing a fluid as many particles, each with physical properties such as mass and volume, and defines rules for interactions between the particles based on the Navier-Stokes equations. The accuracy of SPH depends largely on the number of particles used to simulate a body of fluid, as does the performance. The literature shows a clear trend in increasing the number of particles that are able to be simulated in real-time so that more convincing fluid simulations can be achieved.

In 2003 Muller et al. used SPH to achieve 5000 particles at 5 frames per second [10]. Others implemented interactive SPH simulations with various improvements and additions to the algorithm with similar performance [11][12]. With the advent of GPU computing, researchers capitalized on the parallel implementations of SPH and were able to achieve real-time results with 16,000 particles [13]. Many types of particle systems have been accelerated with the GPU, and several issues important to general particle systems such as accessing values of neighboring particles are important in SPH [14]. The CUDA programming language made general purpose programming on the GPU more accessible and researchers were able to achieve upwards of 60 frames per second for simulations involving 60,000 interacting particles [15]. Recently SPH was implemented in CUDA with complex simulations able to handle 128,000 particles in real-time and simple simulations with upwards of 500,000 particles [16]. Real-time simulation is only part of the goal of making realistic interactive fluid animations. Techniques for rendering the surface of a fluid have been studied along with improvements to SPH in CUDA produced convincing animations with 75,000 particles [17].

For further historical reviews of SPH implementations see Hoetzlein et. al [18] and Krog's Master's Thesis [16].

# CHAPTER 2

# BLENDER

## 2.1 Educational Game Development

Educational games differ from other types of video games only in their objective, that players learn as a result of playing. Educational games attempt to take advantage of the engaging aspects of video game mechanics to entice players into learning. It follows that improving the mechanics available for creating video games will expand the tool-set for creating educational games.

The most common elements of a video game include a two or three dimensional environment that a player can manipulate. For many games the artistic representation of the environment is important, as well as audio effects and possibly interaction with other players over a network connection. Players must be able to interact with the environment, often using a keyboard and mouse or a specialized controller. The program that controls this environment and provides the aforementioned functionality is generally referred to as a Game Engine. Often times, the resources that the Game Engine manipulates, such as the artistic content and audio files, are not created by the same program and one must organize these resources and prepare them for use with the Game Engine.

Currently there are many free and commercial tools for creating video games, including game engines, various two and three dimensional content creation suites as well as audio and video editing programs. Professional game design studios generally have teams of experts for each of these pieces of software working together. An educator may not have these resources at their disposal and thus it is desirable to seek a solution where most of the functionality can be found in a single program, thereby reducing the costs of learning and integrating various components. Due to the relatively limited resources most educators have access to, it is desirable to have a tool that is free.

To these ends Blender [1] was chosen as the basis for this project. Blender is a powerful 3D content creation suite which has a built-in Game Engine. Blender has been in development for over a decade, with millions of users worldwide and a vibrant community of developers improving it daily. Blender is an Open Source project, meaning that anyone is free to download the source code and modify it. There is a large community of volunteers, students who spend their summer funded by the Google Summer of Code initiative, as well as full-time developers paid by the non-profit Blender Institute who continually contribute code to

[1] `http://www.blender.org`

the project. In addition to a development community, there is a large community of users who post tutorials on Blender's many features along with several active mailing lists and forums where Blender users give and receive support.

## 2.2 Blender Game Engine

The version of Blender used in this document is 2.57. The transition from version 2.4x to 2.5x involved a complete overhaul of the User Interface as well as many important internal changes which exclude the possibility of backwards compatibility for games created in 2.5x.

The Blender Game Engine is tightly integrated with the Blender software, and takes advantage of many of the content creation and 3D modeling features. The starting point for any game is a Scene, which will usually contain various Objects. These Objects can be manipulated in several ways by the game designer, exposing them to interaction with the user, defining rules for physical behavior or interaction with other Objects. The easiest way to create these Objects is with the Blender interface, which provides convenient methods for constructing primitives as well as powerful tools for manipulating individual vertices.

This section gives a brief overview of the core features of the Blender Game Engine. Many in-depth and step-by-step tutorials are available for free on the internet.

### 2.2.1 Logic Editor

The Logic Editor interface gives access to Game Properties and Logic bricks for each Object in the Scene. Logic bricks are the entry point for controlling Objects in the Blender Game Engine, while Game Properties allow Objects to have variables associated with them. Logic Bricks are divided into three classes: Sensors, Controllers and Actuators.

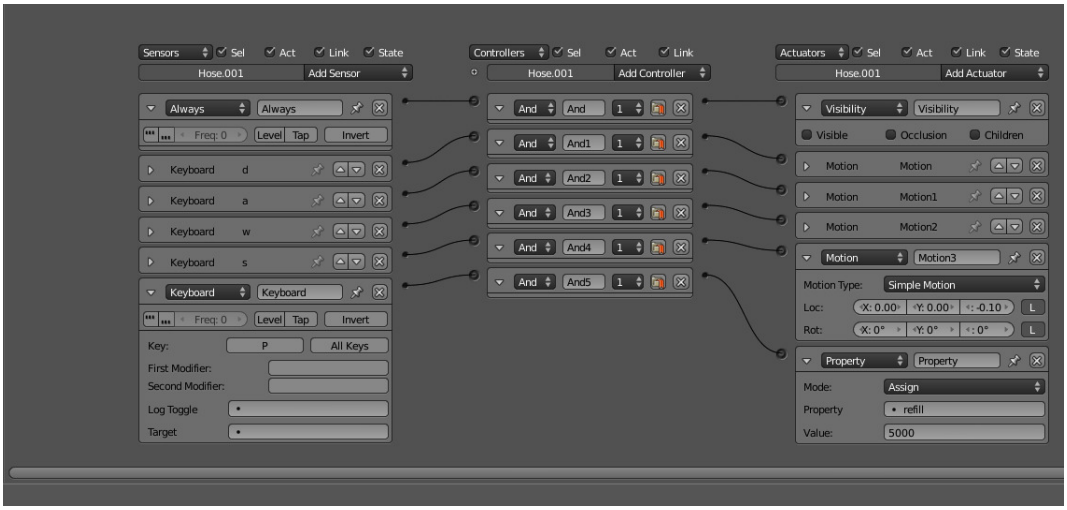


Figure 2.1: Blender Game Engine Logic Bricks

Sensors are widgets through which the Game Engine receives inputs from the user as well as from other objects or events. For example, to receive input from the keyboard one would add a *Keyboard* Sensor, or similarly a *Mouse* Sensor. Other examples of Sensors include collisions between Objects or the Always Sensor, which is always active. The basic unit of time in the Game Engine is a frame, so every frame each Sensor will be evaluated. When the Sensor is triggered, it will release an impulse to any Controller attached to it.

Controllers allow the game creator to combine the impulses from Sensors with Boolean logic in order to determine the action to take. For example an *And* controller to determine if both the *shift* and *A* keys have been pressed. If a controller is activated there are two possible consequences; either one or more Actuators are activated, or a Python script is executed.

Actuators provide functionality for manipulating Objects such as moving or rotating them, changing their properties or even destroying or adding new Objects. While many interesting game functionalities can be realized using actuators, any complex endeavor requires Python Scripting for practical development.

### 2.2.2 Python Scripting

The Blender Game Engine, like the rest of the Blender software provides a comprehensive Python scripting interface to the base functionality. Anything possible with Logic Bricks is also possible with Python, with the added benefit of extended functionality. Python scripts are used to make complex user interfaces within games, manipulate objects and their properties as well as incorporate functionality from outside libraries.

### 2.2.3 Physics

Rigid body physical simulation is provided by the Bullet Physics library [2]. Rigid body physics allow for many interesting effects and convincing interactions.

The Blender software suite provides two fluid simulators, one developed as an extension to Blender's particle system and the other as an internal library. Unfortunately neither of these fluid simulators is available in the game engine since they are unable to perform at interactive speeds.

### 2.2.4 Other Features

The game engine includes a host of other features which can be utilized to enhance a game but are not immediately relevant to this thesis. These features include animations for controlling objects and characters, video textures for playing video within a game, a networking module and support for a variety of inputs. These features undergo continuous improvement within the core code as well as through the addition of Python modules.

[2] `http://bulletphysics.org`

# CHAPTER 3

# GPU COMPUTING

## 3.1 OpenGL

OpenGL is a cross-platform and open industry standard for programming two and three dimensional graphics [19]. The most common use of OpenGL is for programming on dedicated graphics hardware. OpenGL is supported by all of the major graphics cards makers, operating systems and game development studios.

Dedicated graphics hardware is generally a combination of special processors and memory designed for the kind of vector processing commonly encountered in graphics programming. One of the primary tasks in graphics programming is rasterization, which is the process of representing a three-dimensional scene as a set of discrete two-dimensional pixels. While there are various techniques for rasterization, an underlying theme is the necessity for a large amount of independent calculations being done for each pixel. These operations include many linear algebra and other floating point math operations useful for two and three-dimensional geometry and effects [20].

As graphics hardware has become more sophisticated, more control of this parallel infrastructure has been given to the programmer. General programming became possible with the advent of *shaders*, small programs which could be executed by the graphics hardware to perform more sophisticated computational operations than available through the standard interface. Shaders were named for their primary use of shading and lighting objects, which often times calls for the computation of sophisticated mathematical and physical models. OpenGL provides a programming language called GLSL for writing shaders, which allows the programmer to manipulate vertices, colors, textures, and recently geometry all in parallel [20].

## 3.2 General Purpose GPU Programming

With the advances in graphics hardware, most notably with the introduction of shaders, the GPU began to share similarities with the vector processing architectures used in supercomputers such as CRAYs and the Connection Machine. Researchers began using GPUs to compute problems suitable for a parallel vector processor at a fraction of the cost of a supercomputer. The speed benefits of using a GPU over a CPU for highly parallel problems at a low cost has generated increasing interest in using the GPU for general purpose

computing [21].

## 3.3 GPU Programming Languages

Programming languages available for programming GPUs have evolved over the last decade from specialized graphics languages into general purpose programming languages such as CUDA and OpenCL [21]. These languages are introduced in the following sections.

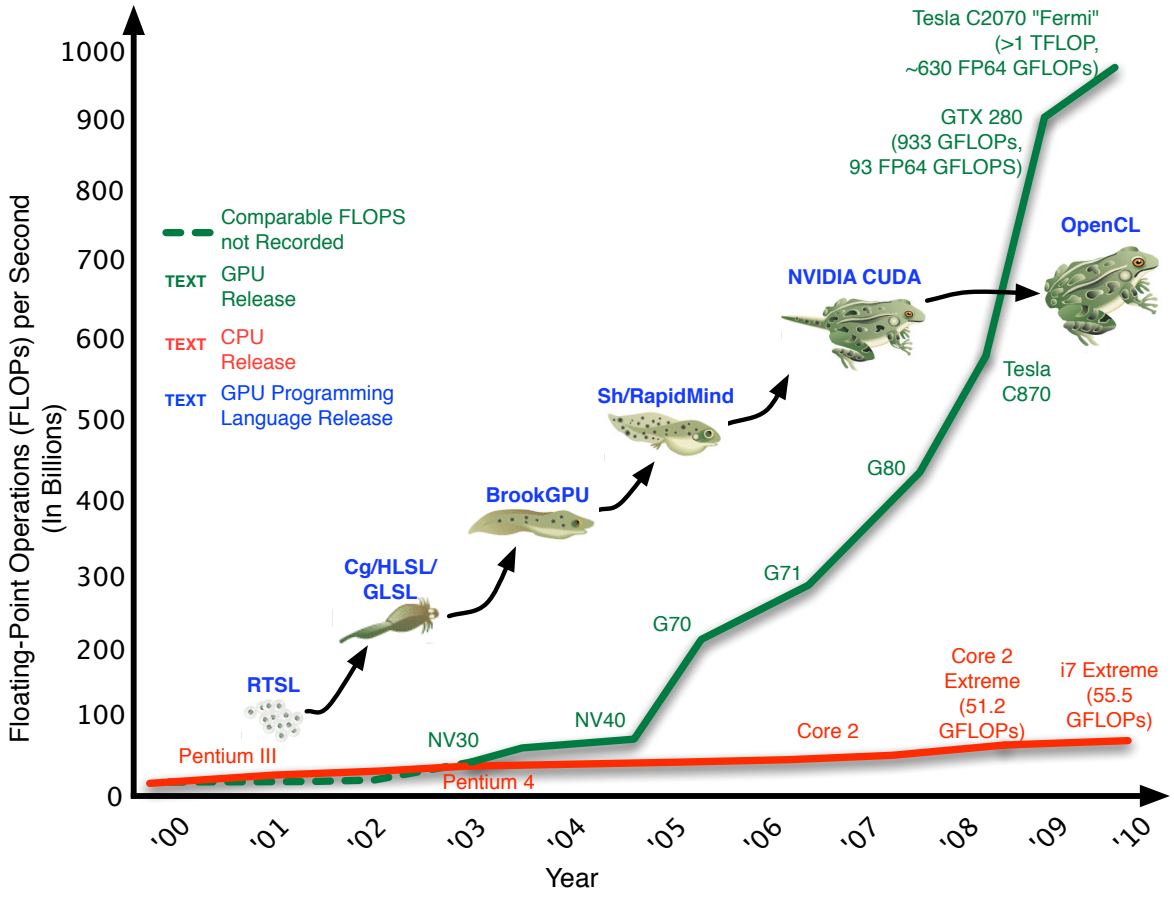


Figure 3.1: GPU Programming Language Evolution [1]

### 3.3.1 CUDA

In 2006, graphics card manufacturer NVIDIA created the CUDA architecture for general purpose computing on NVIDIA GPUs. CUDA is programmed with the 'C for CUDA' language, which abstracts access to the GPU resources. CUDA provides familiar programming techniques such as template programming and memory access and manipulation with

pointers. CUDA is leading the GPGPU market in performance and features; however due to its proprietary nature and lack of availability on competing platforms such as AMD or Intel, it was not considered as an option for this project.

### 3.3.2 OpenCL

In 2009, OpenCL was released as an open standard for a parallel computing language and runtime API by Khronos, an industry-run consortium that also controls the OpenGL standard [22]. The OpenCL specification provides for both the OpenCL runtime and the OpenCL programming language. The OpenCL runtime is designed for parallel programming in heterogeneous environments, providing mechanisms for dealing with multiple and varied compute devices such as CPUs and the many types and generations of GPUs. The OpenCL programming language is a derivative of C99, giving it familiar syntax in an otherwise unfamiliar context for many programmers.

OpenCL is more recent and less mature than CUDA. Additionally it is designed by a consortium rather than a single company so innovation is adopted rather than pushed. OpenCL still lacks several features such as templates and memory addressing which are very useful for complex programming projects. These drawbacks are outweighed by the stability of an open standard supported by the largest entities in the computing industry. Additionally we expect that OpenCL will continue to improve over the next few years and continue to incorporate successful features from other platforms.

## 3.4 Architecture

The idea behind the architecture of a Graphics Processing Unit is to have many smaller floating point processors operate on a large amount of data in parallel. This is because many graphics operations involve doing similar computations on a large amount of independent data. The way this is achieved in general is through a memory hierarchy that allows each processor to optimally access needed data. The OpenCL memory model is given in Figure 3.2

The figure shows four main types of memory: *global*, *constant*, *local* and *private*. *Global* memory is available to all the processors (also known as compute units) and is the slowest type of memory in the hierarchy. *Constant* memory is read-only and is usually cached for fast access. *Local* memory is specific to each compute unit. While much faster than *global* memory local *memory* cannot be used to communicate between compute units. Compute units are further divided into workers, which can be thought of as individual threads each with their own *private* memory. Threads are executed in batches called work-groups, and each thread in a work-group has access to the same bank of *local* memory, as well as access to all of *global* memory.

Threads, or work items, in a work-group are executed simultaneously and in parallel, unless the kernel being executed contains *branching* code such as if statements. In the event of branching, threads for each branch will execute serially, limiting the effectiveness of the GPU. Since *local* memory access is normally at least an order of magnitude faster than *global* memory access, *local* memory is often used as a user-controlled cache. A common

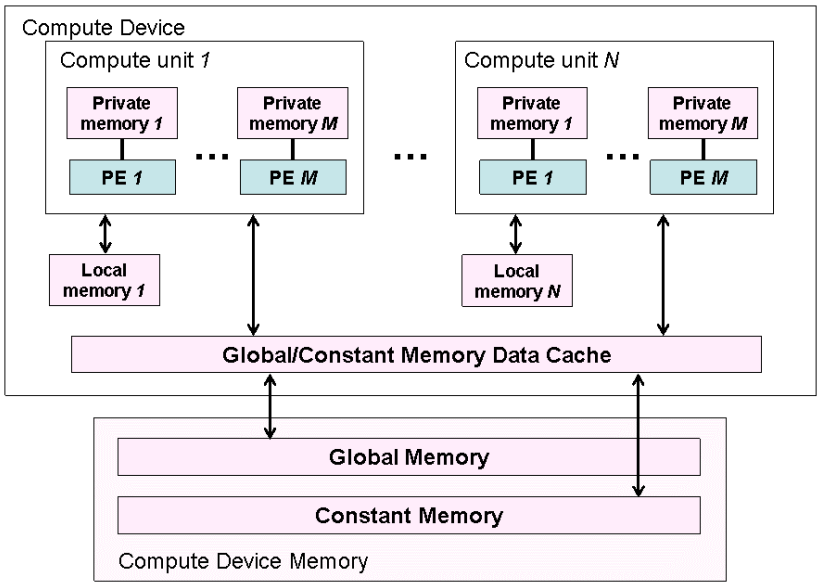


Figure 3.2: OpenCL Memory Hierarchy Diagram [2]

technique is to copy frequently used *global* memory to *local* memory with one thread, synchronizing the threads with a barrier and then processing the *local* memory in parallel.

*Local* and *global* memory are consistent at a work-group barrier, meaning all threads will see the same value in memory after a barrier is placed. *Global* memory is not guaranteed to be consistent across work-groups so it is important to avoid memory conflicts within a single kernel launch.

### 3.4.1 OpenCL Runtime

In order to execute programs written in OpenCL, a programmer must first use the OpenCL runtime API to set up a *context*. An OpenCL context provides the programmer with a way to manage *devices* such as a GPU or CPU. The programmer may send memory as well as programs to the context through the use of a *commandqueue*. Commands in the command queue may be executed synchronously or asynchronously at the programmers behest. The most important items manipulated in the command queue are *kernels* and $buffers$. Kernels are the OpenCL programs that will be executed on a particular device while buffers are the memory that the kernels can take as arguments (i.e. arrays). Kernels are compiled at runtime by OpenCL, allowing the same code to be run on different device architectures without any guidance from the programmer.

# CHAPTER 4

# PARTICLE SYSTEMS

## 4.1 Particle Systems

A Particle system is a system composed of points in space whose behavior is defined by rules that act on those points. Most commonly used in graphics and video games to simulate special effects like fire, smoke, water and explosions particle systems are also used in computational fluid dynamics, astrophysics and materials science.

All particle systems share some fundamental properties, first they must be able to represent points in 1-, 2- or 3-dimensional space. For the purpose of this research 3 dimensions will be used, with the location of each particle denoted with the coordinates `x`, `y` and `z` in a Cartesian coordinate system. Each particle is conceptualized as a unique individual in the system, usually representing some small piece of a larger whole. In the fluid simulation of this research, each particle represents some small volume of water, but for a fire effect the particle serves as an animated paint brush. In both cases each particle will have some associated properties unique to it, such as its density in the water simulation, or its color in the fire effect.

A particle system is not useful unless the properties of it's particles change in time, therefore it is important to have the concept of a `time-step`. The time-step determines how fast a system changes over time. It is important to distinguish between simulation time and real-time. One can simulate the formation of a galaxy over a million years, or the creation of a protein in nanoseconds and render each as a 30 second movie. We will discuss simulation time in terms of `updates` to the system, where each update advances the simulation time one time-step. The creator of the system can determine how many updates to perform for some unit of real-time. This is usually measured in frames per second, where a frame is the rendering of a 3D scene to be displayed. A desirable frames per second (fps) for a video game is 60fps, with 30fps or above considered real-time. For reference, movies are often displayed at 24fps.

Each particle has a location and some properties associated with it, and these can all change in time. Most particle systems follow physical rules, where the location of each particle is updated according to forces acting upon it. In these systems properties can include velocity, mass, temperature and other physical concepts. The forces used to update the locations of the particles can be external to the system, such as gravity or force fields (like wind) or internal, meaning the particles exert force on each other based on their

properties. The forces are used to update the particle locations according to Newton's laws of motion.

Particle systems are especially interesting because simple rules governing the behavior of each particle can lead to complex behavior of the whole system. Particle systems have been classified based on how the particles interact. Non-interacting particle systems are simply systems wherein one particle can not affect another, commonly used to simulate fireworks. Long range interacting particle systems are systems where most of the particles interact with every other particle, such as in gravitational $N$-body systems. Short range interacting particle systems are systems such as the fluid simulator implemented in this research, where each particle interacts with neighbors in a small region around it [14].

## 4.2 Introduction to the Framework

There are many aspects to particle systems that are common to different simulations, such as particle location and particle rendering. In order to create an extensible tool, this project is designed as an object-oriented software framework for general particle systems. The primary motivation for this is to avoid duplicating work when creating different types of simulations.

We first define an interface for the user of the particle system to interact with, the `RTPS` class. This class initializes a particle system with some initial settings and exposes functions to update and render the particles. It also allows access to it's internal system object, which has functions for interacting with the system directly. These functions are general for all systems, but may be implemented differently by different types of systems. For example inserting particles to the system, or setting up a "hose" to spray particles.

The rendering of particles is handled separately from the simulation so that different particle systems may take advantage of the available rendering classes. It is also easy to add a new rendering algorithm by defining a new rendering class derived from the `Render` class.

Someone wishing to add a new type of particle system does not have to spend much energy on infrastructure but can focus on implementing the rules which govern the behavior of their particles, or can add new functionality to interface with some external projects. However it is likely that users will want to integrate particles into their existing game mechanics, and the framework has been designed with this in mind.

# CHAPTER 5

# FLUID SIMULATION

## 5.1 Computational Fluid Dynamics

The study of fluid simulation is known as Computational Fluid Dynamics, which encompasses mathematical models for fluid behavior and numerical methods for implementing these models. The most well-known model for describing the behavior of fluids is given by the Navier-Stokes equations. These equations model a fluid by considering the physical quantities mass-density, pressure and velocity as continuous fields and describing their relationships over time with differential equations.

The numerical methods for solving these differential equations are many and varied. These methods can usually be classified as one of two types, Eulerian and Lagrangian. Eulerian methods describe a fluid in terms of space, while Lagrangian methods describe a fluid in terms of material. Essentially, in an Eulerian method one looks at a region in space and watches how much fluid moves in and out of the region, whereas in a Lagrangian method one watches the material properties of a small volume of fluid and tracks its movement through space.

In general, an Eulerian method discretizes an area of space with a grid, where each grid cell represents a small volume through which a fluid can pass. The accuracy of Eulerian methods is largely dependent on the resolution of this grid. Smaller cells generally imply a more accurate result but at a higher computational expense. This presents a problem when one considers that the grid must be stored in the limited memory of a computer, so there is a secondary cost to making a higher resolution grid. This cost is compounded if only a relatively small area of the grid contains fluid, since naturally one would want to use more and smaller cells in that area and not store the unused cells in other parts of the domain. Another related concern is when a fluid undergoes a large deformation where additional resolution is necessary to accurately resolve the behavior. One would then like to dynamically provide higher resolution in the area of the deformation while leaving the rest of the grid intact. While many Eulerian methods address these concerns with adaptive mesh techniques, the computational costs involved are not amenable to real-time simulations.

Lagrangian methods naturally remove the dependence on a spatial grid since they are based in the material description of the fluid. A common way to keep track of the material properties of a fluid in a Lagrangian method is by using particles, where each particle has a location in space, represents a volume of fluid and stores material properties. Particles then

interact with each other over time, transfer material properties and are displaced according to forces that depend on the type of fluid considered. Since the location of the particles is not fixed, and the fluid is represented only by the particles, the problem of limited storage is distinct from the problem of resolution. With particle-based methods resolution is increased by adding more particles, but without the spatial restriction.

## 5.2 Navier-Stokes

The Lagrangian formulation of the Navier-Stokes equation for an incompressible isothermal viscous fluid is given by
1) The continuity equation:

$$\frac{d\rho}{dt} = -\rho \boldsymbol{\nabla} \cdot v. \tag{5.1}$$

where $\rho$ represents the density of the fluid and $v$ represents the velocity;

2) The momentum equation:

$$\frac{dv}{dt} = -\frac{1}{\rho}P + \frac{\mu}{\rho}\nabla^2 v + f. \tag{5.2}$$

where $\mu$ represents the dynamic viscosity and $f$ represents external forces (such as gravity).

## 5.3 Smoothed Particle Hydrodynamics

The key idea behind Smoothed Particle Hydrodynamics lies in the integral approximation of a field function [23]. The integral representation of a field function, $A$, is given by

$$A(r) = \int_{\Omega} A(r')W(r - r', h)dr'. \tag{5.3}$$

Here $r$ is a point in space, and $W$ is a kernel function with a smoothing radius $h$, and $\Omega$ is the volume that contains $r$. If $W$ is the Dirac Delta function, the integral representation reduces to the exact value of $A(r)$. Otherwise it is known as a kernel approximation.

The integral representation can be discretized by a particle approximation, which is the summation of particles in a neighborhood:

$$A(r) = \sum_j A_j V_j W(r - r_j, h). \tag{5.4}$$

where $A_j$ is the field value at coordinate $r_j$ and $V_j$ is the volume of particle $j$. It is important to note that for a fluid, the volume of a particle is given by its mass divided by its density:

$$V = \frac{m}{\rho}. \tag{5.5}$$

The particle approximation of a field function in SPH is given by:

$$A(r) = \sum_j \frac{m_j}{\rho_j} A_j W(r - r_j, h). \tag{5.6}$$

The gradient of a field function is given by:

$$\nabla A(r) = \sum_j \frac{m_j}{\rho_j} A_j \nabla W(r - r_j, h). \tag{5.7}$$

Since $A_j$ is assumed constant in the volume $V_j$, only the kernel function $W$ is affected by the gradient. Similarly the Laplacian of a field function is given by

$$\nabla^2 A(r) = \sum_j \frac{m_j}{\rho_j} A_j \nabla^2 W(r - r_j, h). \tag{5.8}$$

### 5.3.1 Formulation

Applying the SPH approximations to the Lagrangian formulation of the Navier-Stokes equations is done in two steps. By keeping the mass fixed for each particle the continuity equation can be omitted, instead the density of each particle is approximated:

$$\rho_i = \sum_j \rho_j \frac{m_j}{\rho_j} W(r_i - r_j, h) \tag{5.9}$$

using equation 5.6 which can be simplified to

$$\rho_i = \sum_j m_j W(r_i - r_j, h). \tag{5.10}$$

The momentum equation

$$\frac{dv}{dt} = -\frac{1}{\rho}\nabla P + \frac{\mu}{\rho}\nabla^2 v + f \tag{5.11}$$

can be separated into three components on the right hand side:

$$\frac{dv}{dt} = F^{pressure} + F^{viscosity} + f^{external} \tag{5.12}$$

with

$$F^{pressure} = -\frac{1}{\rho} P \tag{5.13}$$

and

$$F^{viscosity} = \frac{\mu}{\rho}\nabla^2 v. \tag{5.14}$$

To approximate the pressure force, an equation of state is used to give weak compressibility, rather than solve the Poisson pressure equation for near incompressibility, to reduce computational cost. The equation of state is given as

$$P = k(\rho - \rho_0) \tag{5.15}$$

Where $k$ is a gas stiffness constant and $\rho_0$ is the rest density. The pressure term can be approximated as

$$F_i^{pressure} = -\frac{1}{\rho_i} \sum_j P_j \frac{m_j}{\rho_j} \nabla W(r_i - r_j, h). \tag{5.16}$$

However this does not produce a symmetric force and violates the action-reaction law. A symmetric approximation of the gradient is used instead:

$$F_i^{pressure} = -\frac{1}{\rho_i} \sum_j \frac{P_i + P_j}{2} \frac{m_j}{\rho_j} \nabla W(r_i - r_j, h). \tag{5.17}$$

The SPH formulation of the viscosity term is given as follows

$$F_i^{viscosity} = \frac{\mu}{\rho_i} \sum_j (v_j - v_i) \frac{m_j}{\rho_j} \nabla^2 W(r_i - r_j, h). \tag{5.18}$$

### 5.3.2 Kernel Functions

The kernel functions used in the integral representations have a large impact on the accuracy and consistency of the method. Kernel functions and their derivatives must have compact support, having a value of 0 outside of the smoothing radius. The behavior of the kernel functions and their derivatives can also strongly impact the behavior of the particles, so care is taken when selecting a kernel [24].

The following kernels were selected for their performance in [16].
For density approximation the $W_{poly6}$ kernel is used

$$W_{poly6}(r, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |r|^2)^3 & 0 \le |r| \le h \\ 0 & otherwise. \end{cases} \tag{5.19}$$

For the pressure term the gradient of the $W_{spiky}$ kernel is used

$$W_{spiky}(r, h) = \frac{15}{\pi h^6} \begin{cases} (h - |r|)^3 & 0 \le |r| \le h \\ 0 & otherwise \end{cases} \tag{5.20}$$

with the gradient given by

$$\nabla W_{spiky}(r, h) = -\frac{45}{\pi h^6} \frac{r}{|r|} \begin{cases} (h - |r|)^2 & 0 \le |r| \le h \\ 0 & otherwise. \end{cases} \tag{5.21}$$

For the viscosity term the Laplacian of the $W_{viscosity}$ kernel is used $W_{viscosity}$

$$W_{viscosity}(r,h) = \frac{315}{2\pi h^3} \begin{cases} -\frac{|r|^3}{2h^3} + \frac{|r|^2}{h^2} + \frac{h}{2|r|} - 1 & 0 \leq |r| \leq h \\ 0 & otherwise \end{cases} \tag{5.22}$$

with the Laplacian given by

$$\nabla^2 W_{viscosity}(r,h) = \frac{45}{\pi h^6} \begin{cases} h - |r| & 0 \leq |r| \leq h \\ 0 & otherwise. \end{cases} \tag{5.23}$$

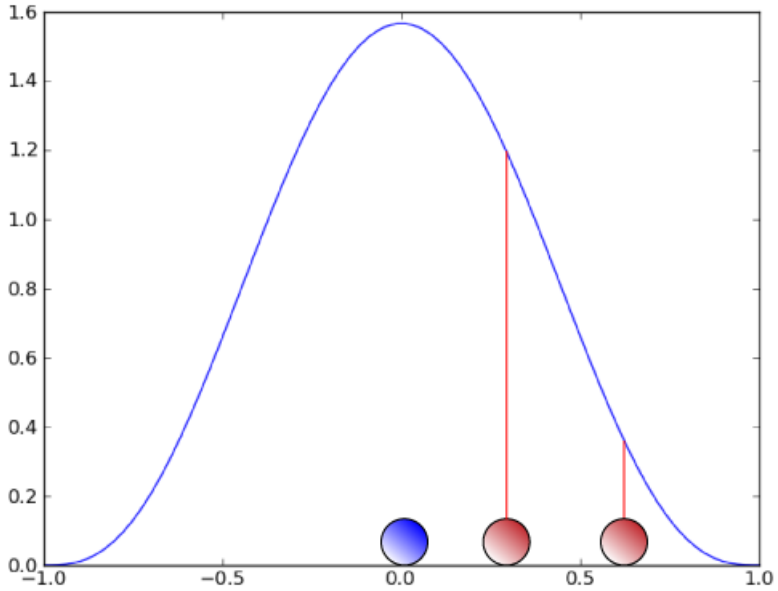


Figure 5.1: $W_{poly6}$ kernel in 1 dimension

### 5.3.3 Collisions

Collisions are handled by exerting a repulsion force on particles which collide with a surface. The repulsion force is given by

$$F^{repulsion} = (stiffness * distance - dampening * n \cdot v)) * n \tag{5.24}$$

where $n$ is the normal of the surface, $v$ is the velocity of the particle, *distance* is the distance of the particle to the surface and $stiffness$ and *dampening* are user-defined parameters to control the behavior of the repulsion force [16].

**Domain.** For convenience, the fluid is considered to exist in a domain, where each wall of the domain is a surface against which the particles can collide. The walls are defined as minimum or maximum values in each axis, with a corresponding normal along the axis pointing into the domain.

**Triangles.** Collisions with arbitrary triangles are calculated are performed by using an algorithm for intersecting a line segment with a triangle. The line segment is created by adding the velocity scaled by a distance parameter to the particle position. The technique uses barycentric coordinates to test if the point where the segment intersects with the plane of the triangle is within the triangle [25].

## 5.4 Integration

The SPH method provides the forces that describe the change in velocity over time. Two modifications are made to these forces to increase stability. The first is a speed limit, which clamps the magnitude of the force to a user-defined speed.

$$F = \begin{cases} s * \frac{F}{|F|} & \text{if } |F| > s \\ F & otherwise \end{cases} \tag{5.25}$$

where $s$ is the user-defined speed limit and $F$ is the force vector.

The second modification is a smoothing of the velocity given by

$$v_i = v_i + \epsilon \sum_{j \neq i} 2m_j \frac{(v_j - v_i)}{(\rho_i + \rho_j)} W(r_i - r_j, h) \tag{5.26}$$

where $\epsilon$ is a user-defined parameter between 0 and 1.

### 5.4.1 Time Integration

To advect the particles in time, the second-order Leap-Frog integration method is used. The new position and velocity each time step is given by

$$x_{i+1} = x_i + v_i dt + \frac{a_i}{2} dt^2, \tag{5.27}$$

$$v_{i+1} = v_i + \frac{(a_i + a_i + 1)}{2} dt, \tag{5.28}$$

where $i$ denotes the current time-step, $x$ is the position vector, $v$ is the velocity vector and $a$ is the acceleration vector.

# CHAPTER 6

# IMPLEMENTATION

## 6.1 The RTPS Library

In this section we discuss the various classes that make up the Real-Time Particle System framework. Filenames will be denoted as such `rtpslib/RTPS.h` and refer to the directory structure of the code in Appendix section A.1.

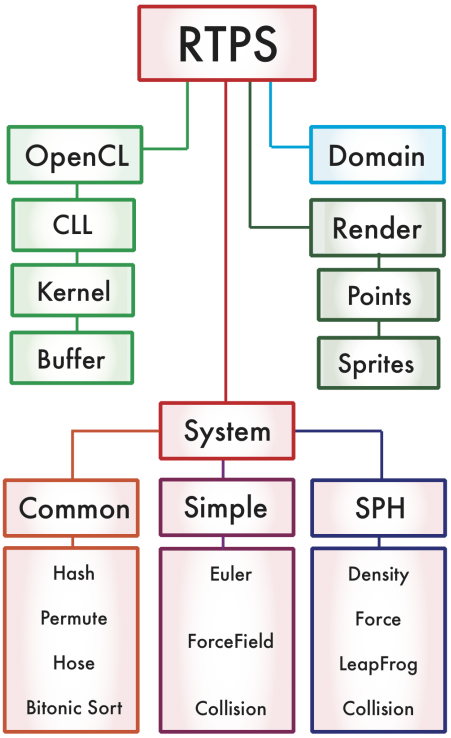


Figure 6.1: Code organization for the RTPS library

### 6.1.1 RTPS

The entry point into the framework is the `rtpslib/RTPS.h` header file. This file defines the RTPS class that a user of the library instantiates in order to run a particle-based simulation. The Application Programmers Interface (API) is relatively simple, giving the user an object that only exposes two methods: An `update` function and a `render` function. These two functions abstract internal logic for updating the simulation and display of the particles. The class constructor accepts either no arguments or a `RTPSettings` object. This object contains many settings including the type of simulation to instantiate, rendering options as well as simulation parameters.

The typical use case (with default settings) is envisioned as follows:

```
#include "RTPS.h"
using namespace rtps;
//... initialization phase ...
RTPSettings settings();
RTPS ps(settings);

//... run loop ...
while(true)
{
    //... event handling and game logic
    ps.update()
    //... rendering phase
    ps.render()
}
```

The class is implemented in `rtpslib/RTPS.cpp`.

### 6.1.2 RTPSettings

Options and parameters for the system are stored in a private STL Map object defined in `rtpslib/RTPSettings.h` and accessed through public getter and setter methods. The available settings are not determined by this class, but by the implementation of each system. This is desirable since different systems will offer different combinations of settings and rather than creating, implementing and inevitably changing new constructors, the proper combination of parameters can be set with calls to the setter functions. This is especially convenient when developing for a user interface such as in Blender where the configuration of settings is not known until runtime. An example of specifying the available settings for SPH can be found in `rtpslib/system/SPHSettings.h`.
Setting an option with an RTPSettings instance:

```
settings->SetSetting("sub_intervals", 1);
```

Getting an option with an RTPSettings instance:

```
int sub_intervals =  settings->GetSettingAs<float>("sub_intervals");
```

### 6.1.3 System

Interaction with the simulation is provided through the `system` object which has various methods on the type of simulation specified. The most important methods are those that allow the user to add particles to a system, namely `addBox`, `addBall` and `addHose`. For the fluid simulation there is also the `loadTriangles` method which allows the user to pass in triangles for the fluid particles to collide with.

These methods are declared in the `System.h` header file which defines an Abstract Class `System`. The methods are implemented in the specific system classes such as `SPH` and `Simple`.

### 6.1.4 Common Structures

Utilized throughout the code is the `float4` struct, which are used to represent coordinate positions of the particles, as well as many of the particle properties associated with the particles such as density, velocity and force. The functionality of the `float4` class match the OpenCL `float4` type as closely as possible. We have overloaded several operators for constructing and doing arithmetic and have provided a print function. This functionality allows the developer to quickly construct data to interact with the system. Being able to reimplement an OpenCL routine on the CPU is also convenient for debugging logic since it makes printing output much simpler. There is also the `int4` struct, which is a vector of four integers, for more convenient communication with OpenCL. The declarations and definitions for these structures and related functions are found in `rtpslib/structs.h` and `rtpslib/structs.cpp`

### 6.1.5 Domain

The `Domain` class calculates and stores the parameters for a regular grid on the interior of a cube. The cube serves as a bounding domain for the particles in a simulation, whereas the grid is used to accelerate the neighbor search for particles. It is important to note that the grid is not stored as a set of nodes or cells, only the minimum and maximum of the cube along with the size and number of cells in each dimension.

The `Domain` class is found in `rtps/domain/Domain.h` and `rtps/domain/Domain.cpp` alongside `rtps/domain/IV.h` and `rtps/domain/IV.cpp` which define several helper functions for inserting particles at regularly spaced intervals based on grid parameters.

## 6.2 Nearest Neighbor Search

The Nearest Neighbor Search (NNS) is essential to the SPH simulation because it allows for the efficient updating of a particle's forces based on surrounding particles. The NNS is implemented in two distinct phases, preparation and lookup. The preparation phase prepares the data structures needed at the beginning of each update, while the lookup phase is performed by any routine which needs access to nearest neighbors.

### 6.2.1 Preparation

The first step in the NNS preparation is to create an integer hash value for each particle to be sorted. This hash value is created by overlaying a uniform grid on the domain, calculating the cell which contains the particle and then calculating a one dimensional index from the three-dimensional cell index [16].

**Hash.** The cell index is calculated as follows:

```
int4 cell_index = (pos - grid_min) * grid_delta;
```

where `grid_min` is a `float4` representing the minimum coordinate of the grid, obtained from the domain. `grid_delta` is a `float4` that contains the width of the cell in each dimension. The cell width is twice the smoothing length of a particle. In this way a particle will only be able to interact with particles in the 26 surrounding cells.

The hash is calculated as follows:

```
int gx = cell_index.x
int gy = cell_index.y
int gz = cell_index.z
int hash = (gz*grid_res.y + gy) * grid_res.x + gx;
```

where `grid_res` is an `int4` that contains the number of cells in each dimension.

The hash for each particle is stored in an integer array by the `Hash` class defined in `rtpslib/system/common/Hash.cpp` with the OpenCL kernel implemented in `rtpslib/system/common/cl_src/hash.cl`.

**Sort.** The hash array is then sorted using the Bitonic Sort implementation provided by NVIDIA[26]. The C interface to the OpenCL code was replaced with a C++ interface defined in `rtpslib/system/common/BitonicSort.cpp`. The Bitonic Sort implementation can only operate on arrays or subsets of arrays with power of two lengths, so the sort always operates on the number of particles rounded up to the nearest power of two. The sort is performed on the hash values of each particle, with an array of indices being permuted along with the hashes.

**Permute.** Once the hash and indices array are sorted, the arrays for particle position, velocity, veleval and color are permuted according to the reordered index array. This leaves all the particles and their associated values sorted according to their spatial position. The `Permute` class is defined in `rtpslib/system/common/Permute.cpp` with the OpenCL Kernel implemented in `rtpslib/system/common/cl_src/permute.cl`.

**Cell Indices.** The next step in the preparation is to update two arrays that keep track of which cells are populated with particles. These two arrays are `cell_indices_start` and `cell_indices_end`, where the former has a value of -1 if no particles are present in the cell, or the index into the particle array of the first particle in the cell. The latter contains the index into the particle array of the last particle contained in a cell. The `CellIndices` class is defined in `rtpslib/system/common/CellIndices.cpp` with the OpenCL Kernel implemented in `rtpslib/system/common/cl_src/cellindices.cl`.

The `cellindices` kernel utilizes local memory to decrease global memory accesses in the process of setting the `cell_indices` arrays. The C++ code allocates a local memory array equal to the workgroup size plus 1. This array is populated by the current particles hash and then the first thread sets the first element of the array to the hash of the previous particle.

```
uint tid = get_local_id(0);
sharedHash[tid+1] = hash;

if (index > 0 && tid == 0)
{
    // first thread in block must load previous particle hash
    uint hashm1 = sort_hashes[index-1] < ncells ? sort_hashes[index-1] : ncells;
    sharedHash[0] = hashm1;
}
```

The kernel waits for all threads to finish this computation with a memory fence:

```
barrier(CLK_LOCAL_MEM_FENCE);
```

The accelerated part of the cell index assignment is given by

```
if (index > 0)
{
    if(sharedHash[tid] != hash)
    {
        cell_indices_start[hash] = index;
        cell_indices_end[sharedHash[tid]] = index;
    }
}
```

This updates the appropriate start and end indices if the hash between neighboring particles is different.

### 6.2.2 Lookup

A neighbor lookup is performed by calculating the grid cell of a particle and iterating over all of the particles in that cell and its 26 neighbors. The grid cell and corresponding hash are calculated with the same routines used in the preparation. The neighbor search is composed of three functions, the first two found in `rtpslib/system/sph/cl_src/cl_neighbors.h` and the last found in the `Force` and `Density` kernels.

**IterateParticlesInNearbyCells.** This function loops over the 26 cells neighboring the cell of the particle being queried. For each cell it calls the IterateParticlesInCell function.

**IterateParticlesInCells.** This function first determines the start index into the particle array from the current cell's hash and the `cell_indices_start` array. If the value is not $-1$ then the end index is obtained from the `cell_indices_end` array. If the hash of the cell is greater than the maximum hash size the routine exits early to avoid accessing non-existent array elements. The routine then iterates over all of the particles from the start index to the end index and calls the ForNeighbor function.

**ForNeighbor.** This function now has access to the original particle being queried as well as the index of a neighboring particle. The code for the previous two routines is contained in a header file included by the kernel that implements the ForNeighbor function. This allows for kernels with different functionality to reuse the neighbor search code. Different kernels may require access to different arrays within the ForNeighbor routine. In order to use different arguments with the same code, `#define` macros are used to specify the arguments of each function and the values passed through. An example of this is provided in Appendix section A.4.

## 6.3 Particle Insertion and Deletion

Particles can be dynamically inserted into and deleted from the simulation by taking advantage of the hashing and sorting functionality. The number of particles in the system is tracked by the `SPH` instance. Calculations in the simulation are only done on the first *num* elements of the relevant arrays (such as position, velocity, force, etc.) where *num* is the current number of particles. The values in the position array for indices greater than *num* are set to a location that will always place them outside of the grid (e.g., a positive multiple of the grid maximum). Any particle outside of the grid will receive a hash value greater than the maximum hash value, thus when the particle array is sorted, the active particles are always first in the array.

### 6.3.1 Insertion

Particles are inserted by setting the values of the position, velocity and color arrays immediately after the *num* index and incrementing *num* by the number of particles added. When the particles are sorted, the new particle positions will be inside the grid and will be part of the new collection of particles to be used in the simulation.

### 6.3.2 Deletion

Deletion of particles is accomplished by moving particles outside of the grid. When the `Hash` kernel calculates the hash for a particle that falls outside of the grid, it assigns the maximum hash value to that particle. The number of active particles is updated by taking advantage of the `CellIndices` kernel. The value of the element at the maximum hash value of the `cell_indices_start` after the `CellIndices` kernel execution gives the start index of the particles outside the grid and conversely, the number of particles which are inside the grid. The number of particles is then set to this value and the deleted particles are no longer used in the simulation.

## 6.4 Parameters

Two structures are defined that hold all of the parameters for the system, one is the `GridParams` structure which holds all the parameters relevant to the Domain and spatial grid. The second is `SPHParams`, which holds all the parameters relevant to the SPH simulation. These structures are populated by the `SPH` and `SPHSettings` classes and passed to all

of the OpenCL kernels as constant memory structures. The `SPHParams` structure is listed in Appendix section A.2.

### 6.4.1 Calculated Parameters

A subset of the physical and simulation parameters are automatically calculated based on one user-defined parameter, the maximum number of particles. The per-particle volume is calculated by dividing a set volume by the maximum number of particles:

```
float VP = 2 * .0262144 / max_num;                    //Particle Volume [ m^3 ]
```

The mass of each particle is the product of the rest density and the particle volume:

```
float mass = 1000. * VP;                               //Particle Mass [ kg ]
```

The rest distance between particles is a fraction of the particle radius:

```
float rest_distance = .87 * pow(VP, 1.f/3.f);    //rest distance between particles [↩
    m ]
```

The smoothing distance is simply twice the rest distance:

```
float smoothing_distance = 2.0f * rest_distance; //interaction radius
```

The simulation is scaled to the coordinate space of the user-specified Domain by relating the volumes of each particle to the volume of the Domain:

```
float domain_vol = (dmax.x - dmin.x) * (dmax.y - dmin.y) * (dmax.z - dmin.z);
float simulation_scale = pow(.5f * VP * max_num / domain_vol, 1.f/3.f);
```

These calculations are specified in `rtpslib/system/SPHSettings.cpp` [16].

## 6.5 SPH Force Calculations

The force calculations defined by the SPH formulation are implemented in three distinct phases: density, force and collision.

### 6.5.1 Density

The density of each particle is calculated by the summation over its neighbors as given by the SPH formulation in equation 5.10. The density calculation only depends on the position of the particles and the particle mass. In this implementation the particle mass is uniform for all particles and is thus passed in as a constant. The density calculation is controlled by the `Density` class defined in `rtpslib/system/sph/Density.cpp` and the OpenCL kernel is defined in `rtpslib/system/sph/cl_src/density.cl`. The density kernel is included in Appendix section A.4.

### 6.5.2 Force

The force calculation is controlled by the `Force` class defined in `rtpslib/system/sph/Force.cpp` and the OpenCL kernel is defined in `rtpslib/system/sph/cl_src/force.cl`.

Pressure and Viscosity forces of the SPH formulation are calculated, as well as a smoothing factor known as XSPH.[16] The XSPH calculation is given as follows:

```
    float Wijpol6 = Wpoly6(r, sphp->smoothing_distance, sphp);
    float4 xsph = 2.f * sphp->mass * Wijpol6 * (velj-veli)/(di+dj);
    xsph += xsph * (float)iej;
```

where `veli` and `velj` are the velocities at particles $i$ and $j$ respectively and `di` and `dj` are the densities at particles $i$ and $j$ respectively. The value of $iej$ is given by

```
int iej = index_i != index_j;
```

which is used to nullify any force or XSPH contributions from a particle with itself. Additional care must be taken if two particles have the same position since the calculation of the $W_{spiky}$ kernel derivative involves the division of the distance between particles. A divide-by-zero error is avoided by setting the distance used in the kernel to the maximum between the distance and a small epsilon, $10^{-6}$.

### 6.5.3 Collision

The routines that calculate the repulsion force contributions from collisions are specified in `rtpslib/system/sph/cl_src/cl_collision.h`.

Collisions with the boundary are computed by the kernel in `rtpslib/system/sph/cl_src/collision_wall.cl` and controlled by the `CollisionWall` class in `rtpslib/system/sph/Collision_wall.cpp`. A collision with the boundary is determined by a simple distance query. The collision test with the bottom of the domain is given as an example:

```
float diff = sphp->boundary_distance - (p.z - gp->bnd_min.z);
if (diff > sphp->EPSILON)
{
    float4 normal = (float4)(0.0f, 0.0f, 1.0f, 0.0f);
    force += calculateRepulsionForce(normal, v, sphp->boundary_stiffness,
            sphp->boundary_dampening, diff);
}
```

Here `p` is the particle position, `v` is the particle velocity and `gp->bnd_min.z` is the z coordinate of the bottom of the domain.

Collisions with triangles are computed by the `collision_triangle` kernel in `rtpslib/system/sph/cl_src/collision_tri.cl` and controlled by the `CollisionTriangle` class in `rtpslib/system/sph/Collision_triangle.cpp`.

Triangles are defined by their three vertices and a normal in a `Triangle` structure:

```
typedef struct Triangle
{
    float4 verts[3];
    float4 normal;
} Triangle;
```

The triangles are stored in a vector of `Triangle` structures in C++ and passed to OpenCL as an array. The `collision_triangle` kernel is executed with one thread per-particle. The first thread in each workgroup copies a section of the triangle array into local memory. After synchronizing the threads with a local barrier, each thread iterates over the list of triangles and determines particle-triangle collisions with a ray-triangle intersection test [25]. If a collision is detected, the force on a particle is modified according to the repulsion force formula. Once all of the triangles in the section have been iterated over, the next section of triangles is copied to local memory and processed. This is repeated until all of the triangles have been checked for collisions.

## 6.6 Integration

Integration of the forces and velocities of the particles are computed by the `leapfrog` kernel in `rtpslib/system/sph/cl_src/leapfrog.cl` and controlled by the `LeapFrog` class in `rtplslib/system/sph/LeapFrog.cpp`.

In the following code snippets, the following variables will be referred to: `p` is the position of a particle, `v` is the velocity and `f` is the force.

Gravity is first added to the z component of force of each particle. The magnitude of the force is scaled to a maximum speed given by the `velocity_limit` parameter:

```
float speed = length(f);
if (speed > sphp->velocity_limit)
{
    f *= sphp->velocity_limit/speed;
}
```

The force is then integrated to update the velocity. The velocity is effectively smoothed by the XSPH calculated in the `force` kernel adjusted by a user-defined parameter. The position is then updated by integrating the velocity.

```
float4 vnext = v + dt*f;
vnext += sphp->xsph_factor * xsph_s[i];
p += dt * vnext;
```

The LeapFrog integration is calculated and stored for use in the force and XSPH calculations in the `force` kernel.

```
float4 veval = 0.5f*(v+vnext);
```

## 6.7 OpenCL

The framework provides several classes to abstract access to OpenCL routines. The underlying OpenCL functionality is provided by the Khronos C++ header files [22].

The purpose of RTPS OpenCL classes is convenience for the developer, hiding tedious and repetitive behavior and providing sensible default behavior. The classes are not intended to be a general replacement for the official OpenCL interface, and focus primarily on the functionality required in this project.

### 6.7.1 CLL

The fundamental OpenCL class provided by the library is the `CLL` class defined in `rtpslib/opencl/CLL.cpp`. When this class is instantiated, it creates and stores the OpenCL context and command queue to be used by the `RTPS` and `System` classes. The class also provides routines for building programs and loading kernels.

### 6.7.2 Kernel

The `Kernel` class abstracts the creation and use of OpenCL kernels. Instantiating a `Kernel` class will load a kernel from source using the context from the provided `CLL` instance. The primary purpose of this class is to abstract kernel execution, simplify the process of setting kernel arguments, automate the timing of kernel executions and hide the command queue syntax.

The `Kernel` class is defined in `rtpslib/opencl/Kernel.cpp`.

### 6.7.3 Buffer

The `Buffer` class abstracts the creation and use of OpenCL buffers. Buffers can be created from OpenGL vertex buffer objects or STL vectors. Convenience methods are provided for copying data to and from the GPU.

The extent of the simplification can be seen by comparing the code to copy a standard vector to the GPU with the Khronos C++ API:

```
std::vector<float> a;
... //initialize a
err = queue.enqueueWriteBuffer(cl_a, CL_TRUE, 0, array_size, &a[0], NULL,
    &event);
```

with the utility classes provided by this project

```
std::vector<float> a;
 ... //initialize a
cl_a.copyToDevice(a);
```

The `Buffer` class is defined in `rtpslib/opencl/Buffer.cpp`.

### 6.7.4 CL/GL Interoperability

A key performance consideration with GPU computing is memory transfer. As discussed earlier there is a large difference in speed between *local* and *global* memory, there is an even larger difference in the cost for transferring memory between the host (CPU) memory and the device (GPU). Due to this cost, it is desirable to perform as many operations as possible on the GPU without transferring data between the host and the device.

A key feature in OpenCL programming for games is the ability to share a context with OpenGL, allowing OpenCL to directly access and manipulate OpenGL memory objects. If for example the particle positions are stored in an OpenGL memory object for rendering, an OpenCL kernel can modify those positions directly without the need to transfer the memory between OpenGL and OpenCL.

### 6.7.5 Issues

In the course of developing this project several small issues have been raised whose solutions were not trivial. This section describes those issues and the design decisions made to address them.

**Kernel Compilation Caching.** OpenCL is designed to compile kernels at runtime, and once those kernels are compiled, the binary version is cached for the context in which it is run. This cuts down on the cost of running a kernel after the first compilation. An issue arises with the Apple implementation of OpenCL when using include directives in OpenCL code. If a kernel has been executed and cached by the Apple runtime, changes to files included in the kernel will not be reflected in the following run, indicating that the cache is not updated. Included files (such as header files) are often used to isolate functionality that is common to several routines. It is desirable to develop this functionality in one place and have all the code that uses it updated at once. Because of the caching issue, this style of development was not possible. The solution put forth in this project was to use the C++ compiler to preprocess the OpenCL files after making changes to the headers. This generates OpenCL source files with everything in a single file so that the kernel cached by OpenCL was forced to update.

**float4.** The OpenCL 1.0 specification defines the `float4` type as a vector of four floating point numbers. Since the simulation in this project is designed to be three-dimensional, it is the most convenient data type for storing arrays of coordinates. The OpenCL 1.1 standard does define a `float3` type, but as of this writing the 1.0 standard is still the most widely available implementation, so we use `float4` for backwards compatibility. It is also convenient to package many variables into a single array for use on the GPU, and as such one may utilize the unused components of 3-D vectors for storing some scalar values.

**GL Context Sharing on the Mac.** The Apple implementation of OpenCL uses a specific extension for OpenGL context sharing that requires a specific way of creating an OpenCL context. The C++ headers provided by Khronos did not allow for this manner of Context creation, leading to the following modification of *cl.hpp* provided by Khronos:

```
Context(cl_context_properties* properties, cl_int* err = NULL)
    {
```

```
        cl_int error;
        object_ = ::clCreateContext(
            properties, 0,
            0,
            NULL, NULL, &error);

        detail::errHandler(error, __CREATE_CONTEXT_FROM_TYPE_ERR);
        if (err != NULL) {
            *err = error;
        }

    }
```

### 6.7.6 Timing

When developing a performance critical application it is important to know how much time subroutines are taking to execute. OpenCL provides a mechanism for timing the execution time of a kernel on the GPU, which is essential since kernels may be executed asynchronously from the command queue. As part of the convenience classes provided by this project for OpenCL, executing a kernel automatically times its execution and returns the time in milliseconds.

### 6.7.7 Debugging Routines

It is often desirable to output the values of a variable while debugging code; however this is currently not simple with OpenCL. We have implemented a pattern using a set of macros in the kernel source code as well as two dedicated arrays with corresponding GPU buffers. When debugging a kernel the programmer will pass the two buffers as the final arguments, with the first buffer being an array of `float` and the second buffer being an array of `int4`. Inside the kernel the user will have the following macros defined (we include these in our cl_macros.h header)

```
#define DEBUG_ARGS   , __global float4* clf, __global int4* cli
#define DEBUG_ARGV   , clf, cli
```

Here, `DEBUG_ARGS` is added to the kernel's list of parameters and to the argument list of any functions subject to debugging. `DEBUG_ARGV` is used when calling functions with the debug arguments. This way one can easily extend or change the debugging arrays without the tedious task of adding and subtracting arguments to kernel and function definitions, and when done debugging one can simply redefine the macros to be empty.

Once the programmer has these arrays available inside of the kernel and updates their contents in OpenCL, the next step is to return these arrays to the CPU for printing and other analysis. The `Buffer` class provides a `copyToHost` routine that makes this straight forward. Simply initialize an empty vector of the same type as the array, with size equal to the number of elements to be copied from the GPU. Then pass this vector to the routine:

```
//where mybuffer was instantiated like Buffer<float4>(...)
std::vector<float4> tmp(100);
mybuffer.copyToHost(tmp);
```

```
//or to copy 100 elements starting at the 500th element (offset of 500)
mybuffer.copyToHost(tmp, 500);
```

The vector `tmp` will then be populated with the data from the GPU and can be printed and manipulated using all of the familiar CPU functions available to the programmer.

## 6.8 CMake

CMake is a cross-platform, open-source build system [27]. CMake abstracts the process of generating Makefiles for the compilation of large projects. This abstraction handles the differences between operating systems and compilation tool chains. CMake is used in this project to compile and link the RTPS library as well as standalone test applications. By using CMake, the project can be compiled with relative ease on Windows, Linux and Mac OS X.

CMake is also used by the Blender project as one of the supported build systems for compilation. This was a strong for choosing CMake for the RTPS library. Since both projects use CMake, it is straightforward to integrate the two. This is accomplished by placing the RTPS library in a subdirectory of the Blender source tree and changing the Blender $CMakelists.txt$ file to add the subdirectory. With this configuration, compiling Blender also compiles the RTPS library. By adding a few more modifications to the Blender CMake configurations, RTPS could be included and linked against in Blender source code.

## 6.9 OpenGL

Rendering of the simulation is controlled by the `Render` class or one of its subclasses. The `Render` class provides basic OpenGL rendering functionality through the use of `Vertex Buffer Objects` (VBO), which are OpenGL Buffer types for storing coordinate based information. With the use of OpenCL/OpenGL context sharing the positions of particles and their associated color can be stored in a VBO and manipulated as OpenCL buffers without copying or transferring data. The `Render` class manages the OpenGL buffers for the position and color properties of the particles. The class also maintains a reference to the RTPS settings object which contains settings related to the rendering. The Render class is defined in `rtpslib/render/Render.h`.

### 6.9.1 Points

The most basic form of rendering provided simply displays each particle as a filled in circle with a fixed radius. This choice allows the developer to see the behavior of the fluid with minimal overhead. The rendering is accomplished using the OpenGL function $glDrawArrays$ with the `GL_POINTS` option and specifying the position and color VBOs with $glVertexPointer$ and $glColorPointer$ respectively. The Point rendering functionality is implemented in `rtpslib/render/Render.cpp`.

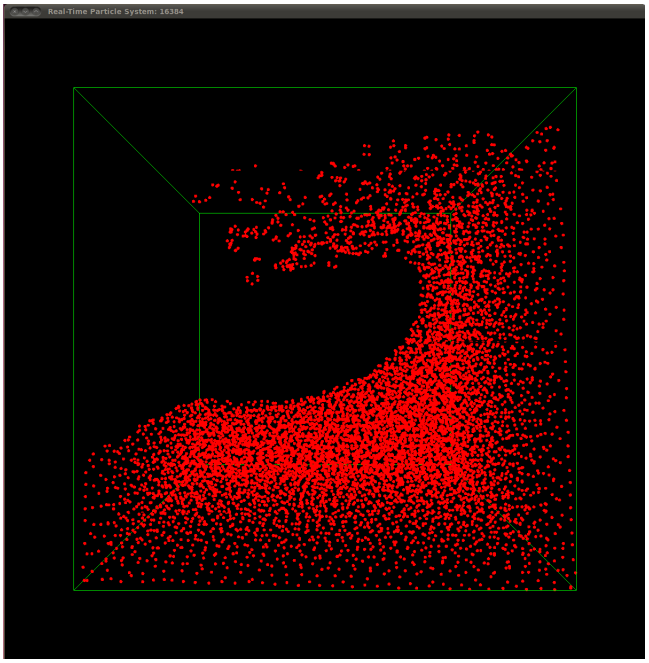


Figure 6.2: Fluid rendered as Points in standalone application

### 6.9.2 Point Sprites

A common strategy to render particles is with a technique known as *billboarding* [14]. In this technique a small texture is rendered to a quad perpendicular to the camera direction. With this technique it is possible to improve upon the point rendering as well as fake some volumetric effects with low computation cost.

The `SpriteRender` class implements billboarding using the `GL_POINT_SPRITE` functionality in OpenGL which will create a textured quad of a specified size centered at each coordinate in the position VBO. GLSL vertex and fragment shaders are used to manipulate the appearance of the point sprites. The default shaders use artificial lighting to fake the appearance of spheres. Another set of shaders is provided in the code for loading of an image for use as a texture on the billboards. The `SpriteRender` class inherits from `Render` and overrides the *render* function. The class is implemented in `rtpslib/render/SpriteRender.cpp` and the shaders are in `rtpslib/render/shaders`.

## 6.10 Blender Source Modification

In order to use the RTPS library inside the Blender Game Engine, the source code for Blender must be modified. The manner in which RTPS is exposed to Blender is through a construct known as a Modifier. The purpose of a Blender Modifier is to enable custom functionality on a Blender Object, usually resulting in a modification of the Object's data. In the Blender Game Engine all modifiers on an Object are *Applied* at the start of a game and are *Updated* every frame. The functionality of *Apply* and *Update* depend on the

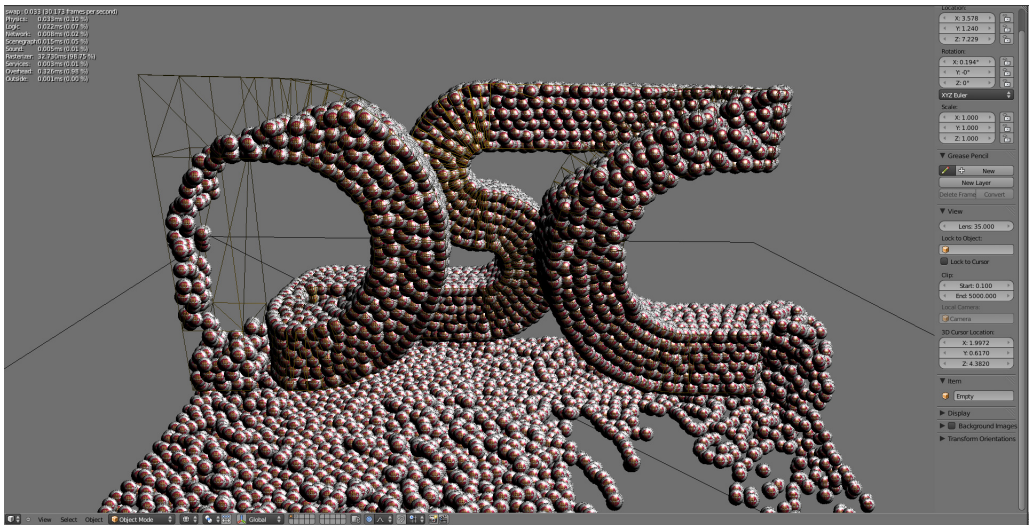

Figure 6.3: Fluid rendered as PointSprites in Blender

Modifier, and in the case of RTPS, it is desirable to instantiate the particle system when the Modifier is *Applied* and to calculate the next timestep when *Updated*.

Blender Modifiers are specified using a C structure. A Modifier's data can be exposed to the Blender User Interface through Blender's Data API and Python scripting interface. The Blender Data API is a way to programmatically access data structures used by Blender without needing to know the details of how they are stored [28]. The User Interface for Blender is controlled with Python, and the Python/RNA API gives access to the Data API so that the values and functionality of a data structure can be manipulated in the User Interface.[1]

### 6.10.1 User Interface

For this project the `RTPS Modifier` was created [29]. The data stored in the modifier is the configurations and run-time options for the RTPS library. These options are exposed to the game designer in a panel and displayed according to a Python script that calls the RNA API. The RNA code specifies aspects of the user interface such as the minimum and maximum values allowed for an input, or the options available in a drop-down menu. The appearance of the user interface elements is automatically created using the defaults provided by Blender, which are suitable for the current configuration.

### 6.10.2 Apply

The values from the User Interface are passed to the RTPS library when the Modifier is *Applied* at the start of the game. The control flow is as follows, the Game Engine iterates

[1]More information on Blender's internal architecture is available at `http://www.blender.org/development/architecture/`.

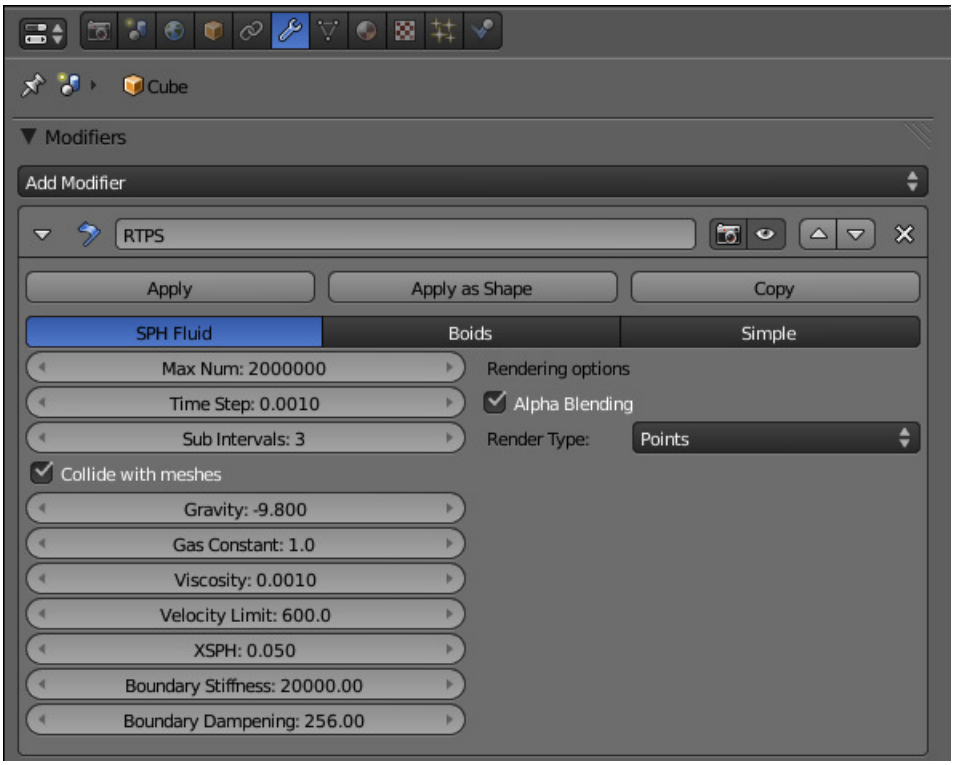


Figure 6.4: RTPS Modifier UI panel.

through all of the objects in the scene and if the object has a Modifier the Modifier's apply routine is called. This logic is extended to check explicitly for the `RTPS Modifier`, and if present initialize an RTPS instance.

The initialization involves passing in the values of the Modifier struct to the RTPS settings instance as well as constructing a Domain object from the axis aligned bounding box of the object which the Modifier is Applied to.

### 6.10.3 Update

The *Update* routine works similarly to the *Apply* routine with respect to control flow. The *Update* routine is called for every frame of the game, rather than only once at the start. In addition to calling the update routine of the RTPS instance, this is where most of the interaction between the Game Engine and the library takes place.

**Collision.** Objects to be used for collision detection are dealt with in the update routine by checking for a specific game property. If the boolean value of the *collider* property is true, the faces of the object will be collected in a vector and passed to the RTPS instance as Triangle objects. This does assume that the mesh of the object is composed purely of Triangles.

**Emission.** Objects which emit particles are also handled in the Update routine. The objects are detected and handled depending on which Game Properties are present. Cur-

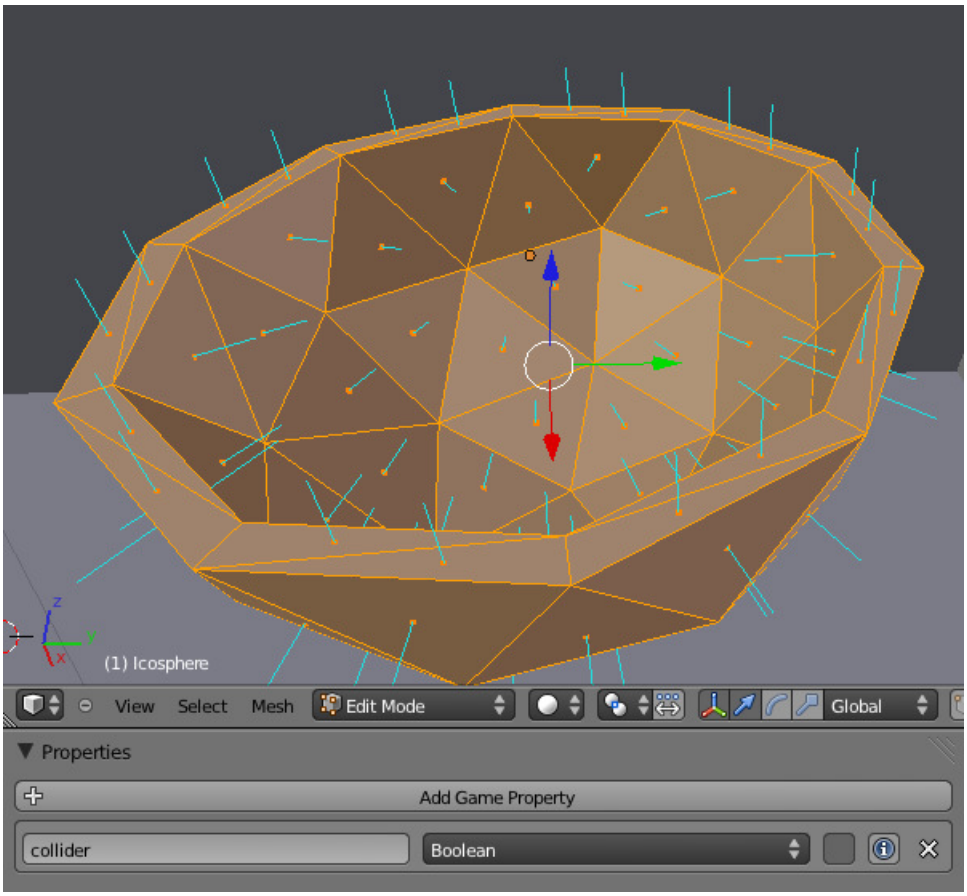


Figure 6.5: Collision Object

rently, there are two types of emitters, Blob and Hose. A Blob emitter will fill a user-specified box with a user-specified number of particles. The spacing of the particles is determined by the relation of the particle radius to the domain size. This means that a user may attempt to fill the volume with more particles than will fit within the domain, in which case only the number of particles which fit will be emitted. If the user requests more particles than the maximum allowed, no particles will be produced. The Blob emitter is determined in blender by the presence of the *num* property. Every frame, if the *num* property is greater than zero, *num* particles are emitted from the volume given by the Object's axis aligned bounding box and the *num* is set to zero. In this way it is possible to provide interactive emissions of particles by using the Logic Bricks to change the *num* property during game play.

The Hose emitter will spray a stream of particles in the direction and speed of a user-specified velocity vector. The radius of the nozzle is specified as a multiple of the particle spacing. The orientation of the Hose is determined by the y-axis of the object being modified. The Hose has a set number of particles it will emit, and each frame, a disc of particles is emitted at a rate proportional to the time step and the velocity. The number of particles emitted at each frame is subtracted from the total number available to the Hose until there

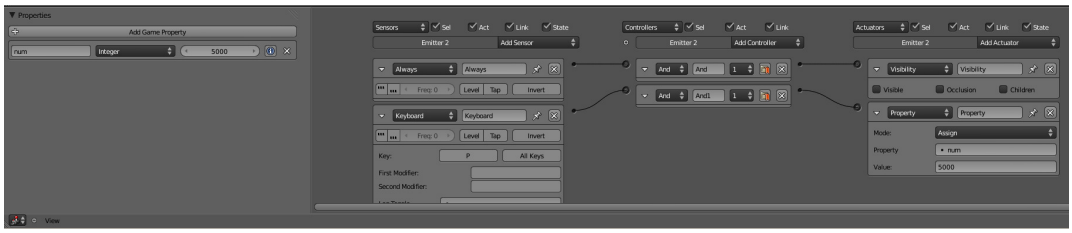


Figure 6.6: Logic panel showing Blob emitter

are no more particles left, at which point the Hose will stop spraying. The positions of the particles emitted in the disc are also slightly randomized (by one particle spacing) along the velocity vector as well as along the plane of the disc, giving a more natural depiction of spray.

A Hose is specified by the presence of the boolean *hose* game property. The value of *hose* determines whether or not the Hose is activated. The rest of the functionality for the Hose is controlled by the *num*, *radius* and *speed* properties. The *num* property specifies the total amount of particles available to the hose, the *speed* property gives the initial velocity of the particles emitted from the hose while the *radius* determines the Hose width.

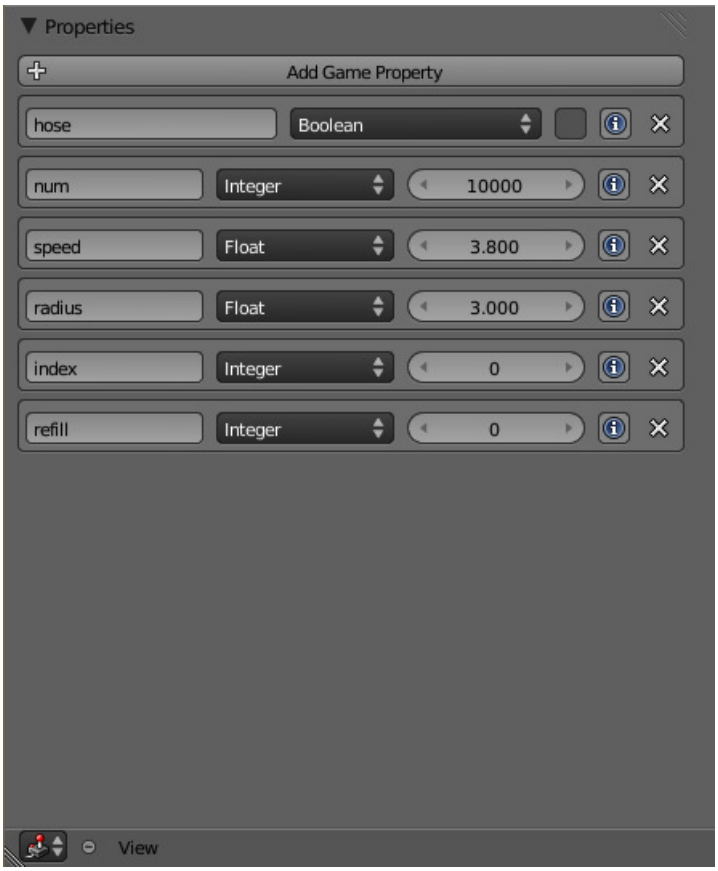


Figure 6.7: Logic panel showing Hose emitter

### 6.10.4 Orientation

In Blender objects are positioned in a global coordinate system, with data (such as vertices) specified in a local coordinate system relative to the object's center. In the RTPS library particles are positioned in a coordinate system relative to the Domain. In order for Blender and RTPS objects to interact, it is essential that objects can be represented in one of the two coordinate systems. This is accomplished by using Blender's global coordinate system when passing information from Blender to RTPS. To use the global coordinate system for an object, one multiplies a given local coordinate by the global rotation and transformation matrices maintained by the Game Engine for the object. This is done for the Domain, as well as Collider and Emitter objects, making all spatial coordinates in RTPS the Blender global coordinates.

### 6.10.5 Render

Rendering of the RTPS particles in the Blender Game Engine is accomplished by taking control of the way the Object with the `RTPS Modifier` is displayed. The Game Engine manages a list of all Objects to be rendered and calls various OpenGL rasterization routines depending on properties and materials of the Object. In order to render the RTPS particles the *render* method of the RTPS instance must be called. In the Game Engine rasterization code, a conditional statement was added to check for the presence of the `RTPS Modifier`. If present, the normal rendering is bypassed and the *render* routine of the RTPS instance is called instead.

### 6.10.6 Files

For a complete list of Blender source files modified by the project please refer to Appendix section B

# CHAPTER 7

# RESULTS

The result of this project is a functional prototype of an interactive fluid simulator. A game designer can use the modified version of Blender to add fluids into a game built with the Blender Game Engine. Basic functionality in the simulation and game logic have been achieved, laying the groundwork for more complex physical phenomenon as well as more interesting interactions.

## 7.1 Performance

The performance of the project was benchmarked on two modern GPUs: an NVIDIA GTX 480 and an AMD FirePro V7800. The GTX 480 has 480 CUDA Cores and 1.5GB of video memory while the FirePro V7800 has 1440 Stream processors and 2GB of video memory. The number of cores and processors do not directly map to performance, as can be seen from the timings presented in Figure 7.1.

In addition to benchmarking the project, the CUDA SPH implementation from Krog[16] was run on with the GTX 480. As shown in Figure 7.1, RTPS is not yet as efficient as SimpleSPH. Comparisons with CPU implementations were not carried out as the literature has shown the wide gap in performance makes the CPU non-competitive [18][16].

Frames per Second (fps) is an important metric for most video game players. A reasonable number for an interactive 3D game is 60fps, although 30fps is sometimes deemed acceptable. Note that movies play at 24fps. The performance of the simulations are compared in frames per second, while analysis of the routines within the simulation is measured in milliseconds. Updating one frame takes 17ms to obtain a frame rate of 60fps, and 33ms at 30fps. In the following timings, when using fps a higher number is desirable while when measuring in milliseconds a lower number is preferred.

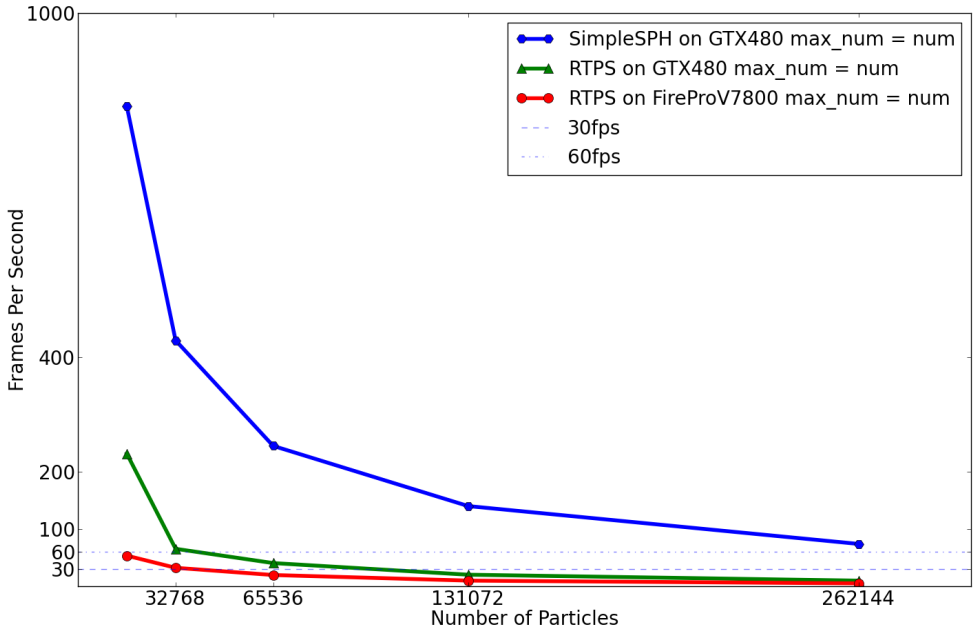


Figure 7.1: Performance of RTPS compared with SimpleSPH. Here the maximum particle setting for RTPS is the same as the number of particles used for timing

For the timings in 7.1 the only parameter varied is the maximum number of particles. For each case the maximum number of particles is emitted and the timing of each function call is averaged over 1000 iterations.

One logical use scenario involves setting a large maximum number of particles and only using a relatively small subset of them. This makes the radius of each particle smaller if the domain size remains fixed, allowing the user to get a more detailed simulation. It turns out that this use case is much more efficient than setting the maximum number of particles to the amount to be used as can be seen in Figure 7.2.

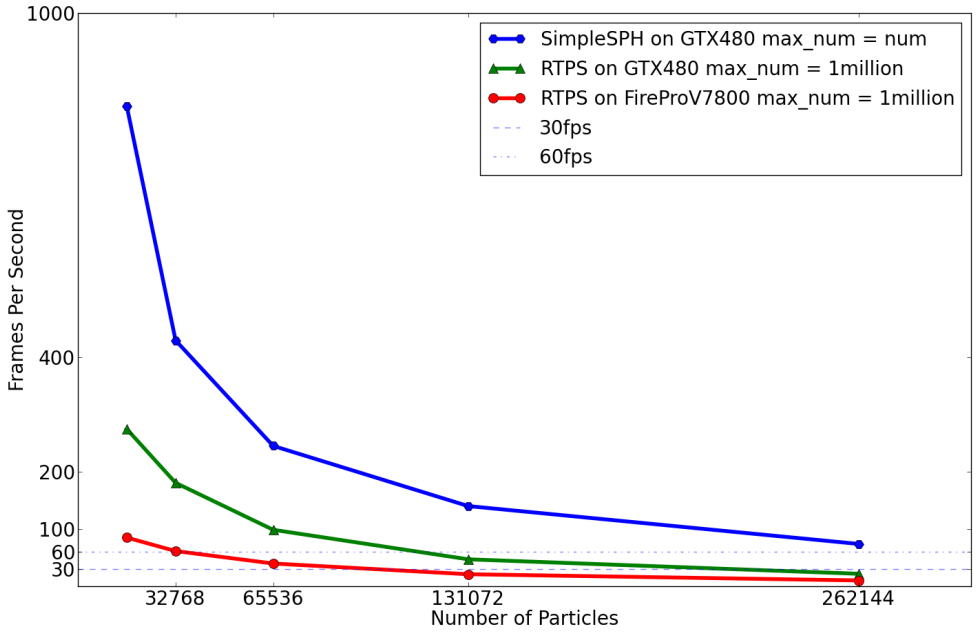


Figure 7.2: Performance of RTPS with the maximum number of particles setting equal to 1 million. The timings for RTPS are collected using the given number of particles.

One sees from Figure 7.3 that the Density and Force calculations are by far the most expensive routines. These routines have the most memory accesses, and the more neighbors that each particle must calculate, the more memory accesses these routines use. It is important to note that the Sorting routine contributes a non-trivial amount of time to the computation. One also a linear increases in cost as the number of particles increase past 16384. Below this number, increasing the particles has a negligible effect on the cost. This is most likely due to the overhead of launching the kernels overshadowing the small amount of memory accesses and computations required by such a small amount of particles.

Figure 7.4 shows the proportion of time taken by the most expensive kernels on each video card using 262144 particles. The proportions are similar for each card, with the sort more expensive on the AMD hardware.

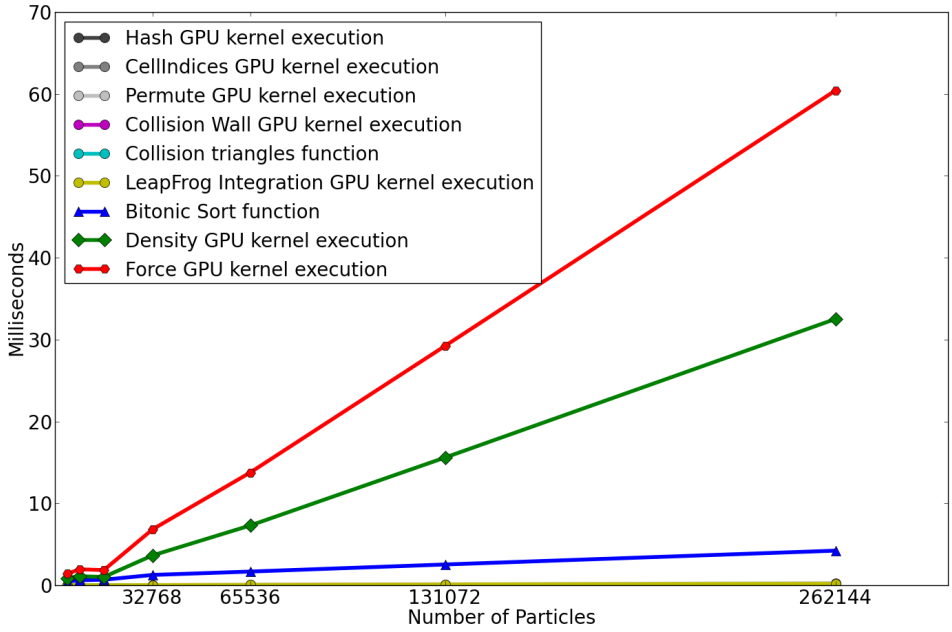


Figure 7.3: GPU timings for each kernel

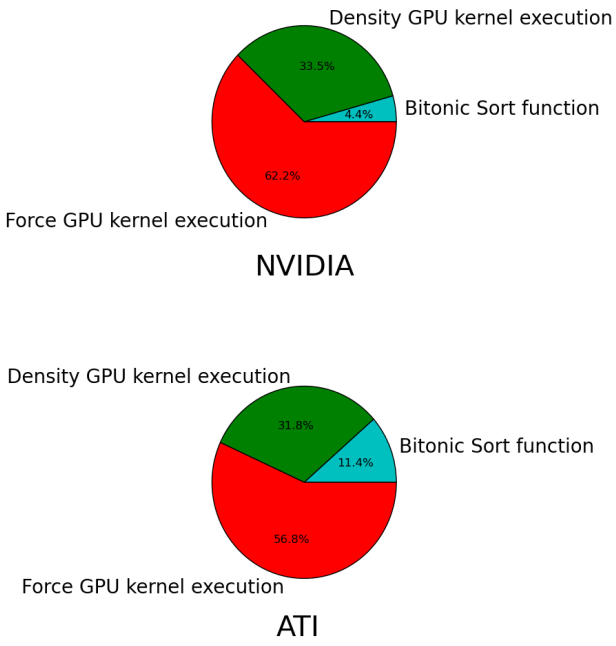


Figure 7.4: Proportion of GPU timings for the most expensive kernels on each card

## 7.2 Data Structures

The data structures used on the GPU are given as follows:

```
//Depend on number of particles
Buffer<float4>      cl_position_u;
Buffer<float4>      cl_position_s;
Buffer<float4>      cl_color_u;
Buffer<float4>      cl_color_s;
Buffer<float4>      cl_velocity_u;
Buffer<float4>      cl_velocity_s;
Buffer<float4>      cl_veleval_u;
Buffer<float4>      cl_veleval_s;
Buffer<float>       cl_density_s;
Buffer<float4>      cl_force_s;
Buffer<float4>      cl_xsph_s;
Buffer<unsigned int>         cl_sort_hashes;
Buffer<unsigned int>         cl_sort_indices;
Buffer<unsigned int>         cl_sort_output_hashes;
Buffer<unsigned int>         cl_sort_output_indices;

//Depend on number of grid cells
Buffer<unsigned int>         cl_cell_indices_start;
Buffer<unsigned int>         cl_cell_indices_end;
```

Using the standard size of single precision floats and unsigned ints of 4 bytes, each particle requires $45 * 4$ or 180 bytes of memory. Each grid cell requires 8 bytes of memory. With this configuration a simulation with 65,536 particles requires a grid with 12,167 cells for a total of 11.34MB. A simulation with 1,048,576 particles requires a grid with 132,651 cells and a total of 181MB. These numbers do not come close to saturating the 2GB of memory available on many modern GPUs.

## 7.3 Community

An important aspect of this project is adoption by the game design community. Acceptance by the Blender community entails benefits such as user generated tutorials and developer support. The project has enjoyed a preliminary amount of success in this regard, being featured on the primary news source for the Blender community.[1]

Builds were compiled for the Windows, Mac and Linux operating systems and distributed to members of the community for testing. These members have already provided valuable feedback, crash reports and even demos.[2]

The project has also received support from developers with regards to integrating the RTPS library with Blender. As the code improves and the integration becomes tighter it is expected that more developers will be willing and able to help.

[1] http://www.blendernation.com/2011/01/03/fluids-in-real-time-with-opencl/

[2] http://www.youtube.com/watch?v=s2QBRazykEA

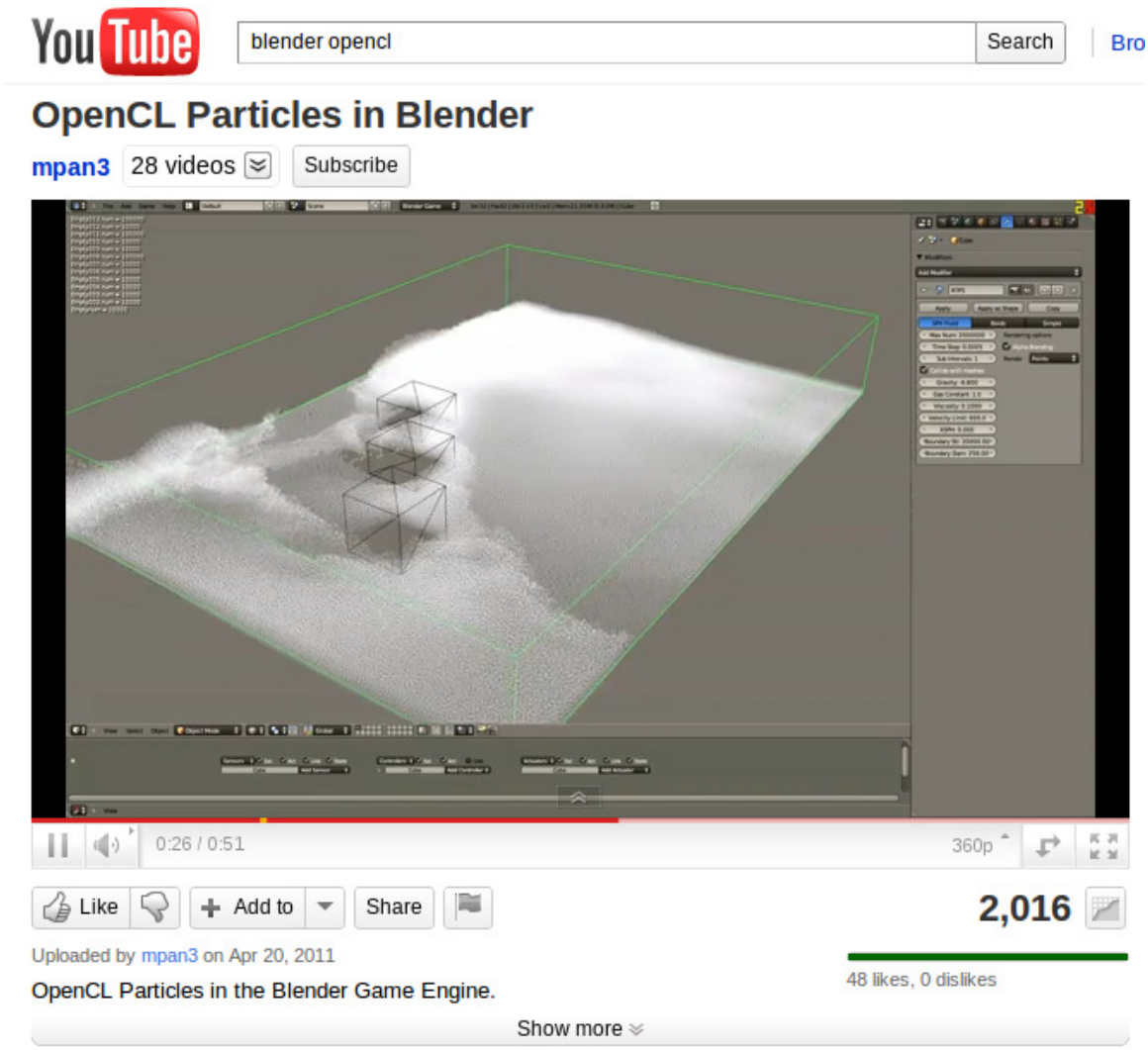


Figure 7.5: Artist demo

## 7.4 Illustrative Examples

The following figures demonstrate various possible simulations using the RTPS framework inside the Blender Game Engine. Figure 7.6 shows a simple dam break simulation with collision against boxes inside the domain. Here approximately 10% of the maximum number of particles are used and the simulation runs at 60fps.

Figures 7.7 and 7.8 show a water flooding simulation where over 100k particles are emitted through several hoses and then collide against cubes placed within the domain. The maximum number of particles was set to 1,000,000 for a higher resolution simulation at the cost of real-time. This simulation averaged 10fps.

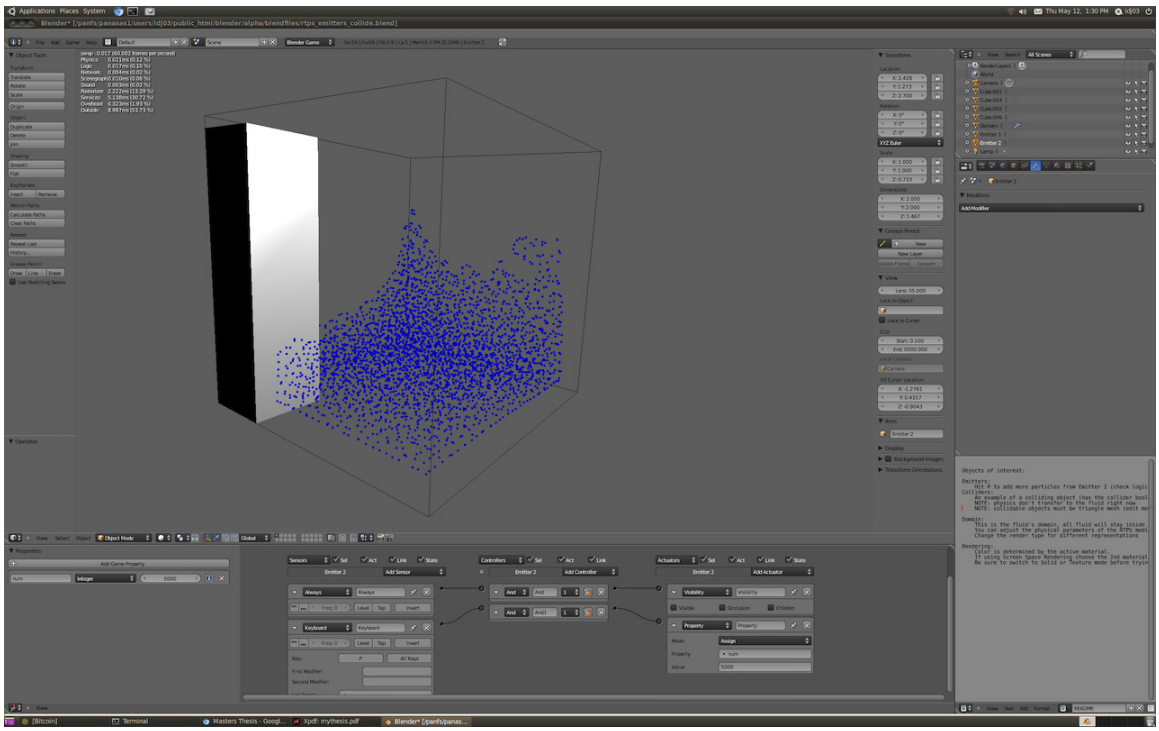

Figure 7.6: A Dam break simulation. The maximum number of particles is set to 65k. Physical parameters are set as follows: Gravity = -9.8, Gas Constant = 1.0, Viscosity = .001, Velocity Limit = 600, XSPH = .2

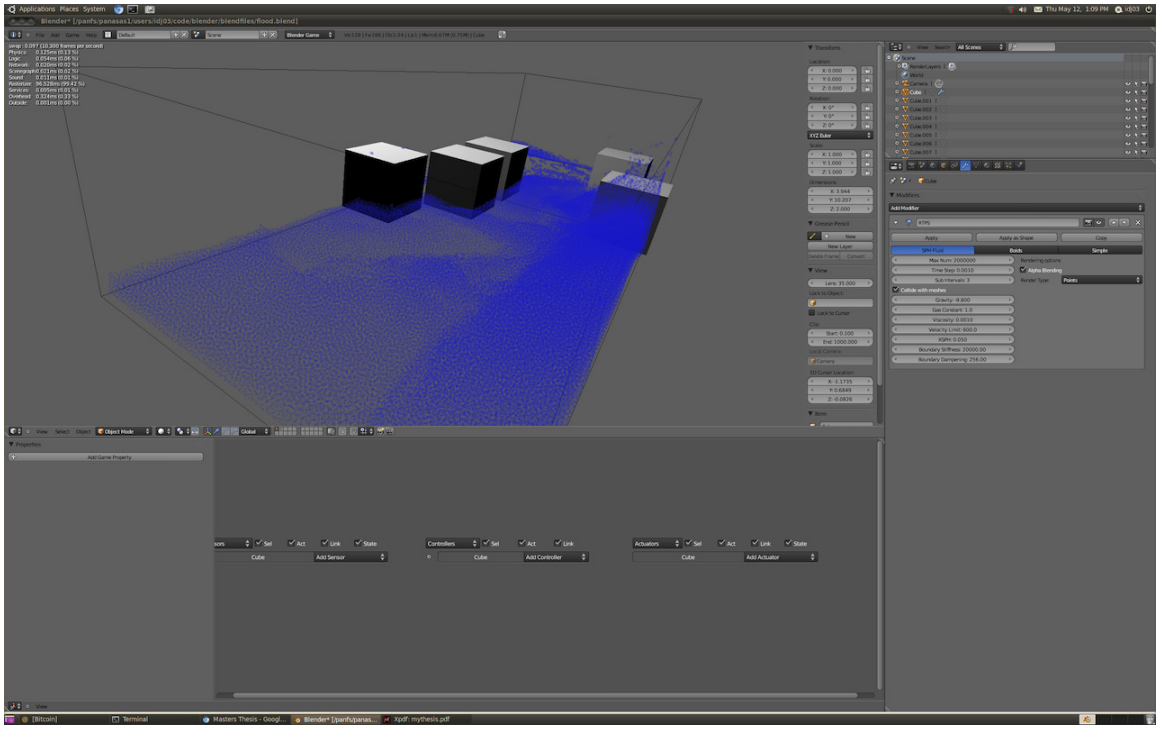

Figure 7.7: Flooding simulation, 100k particles emitted from hoses and colliding with boxes. The maximum number of particles is set to 1,000,000. Physical parameters are set as follows: Gravity = -9.8, Gas Constant = 1.0, Viscosity = .001, Velocity Limit = 600, XSPH = .05

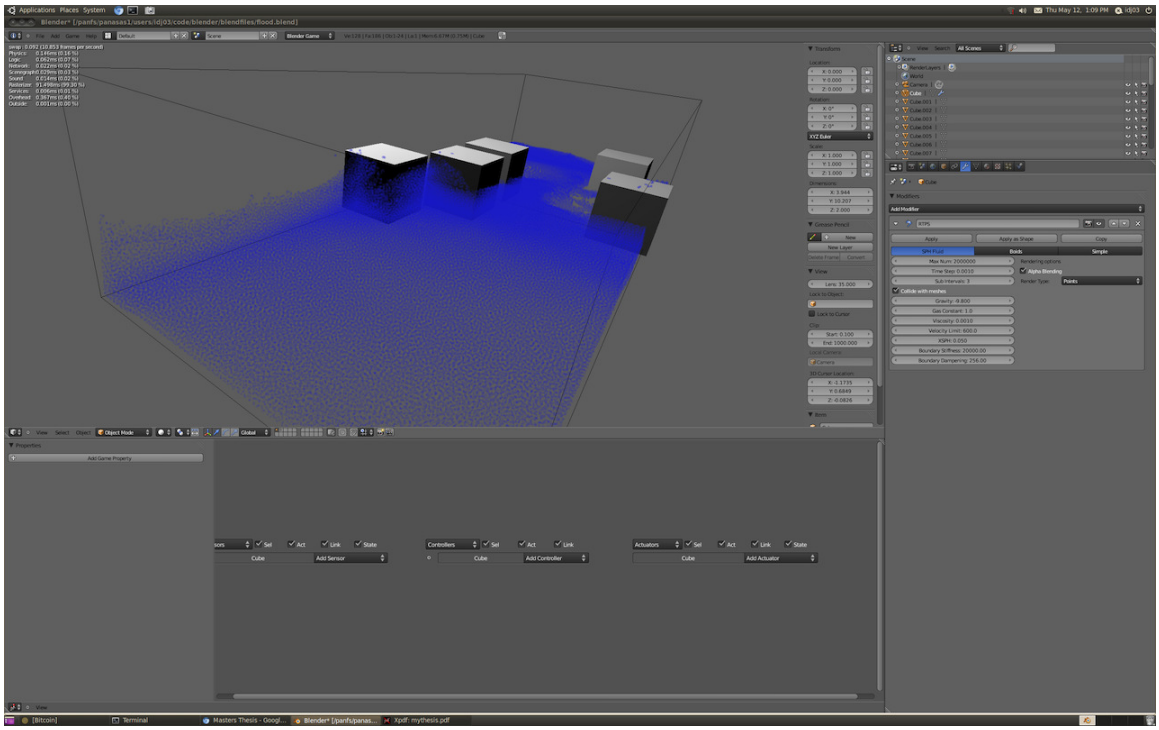

Figure 7.8: Later timestep in same flooding simulation as Figure 7.7

Figure 7.9 shows a viscous fluid being emitted from a hose. A high viscosity gives a more honey-like appearance and reduces waves and splashes in the fluid.

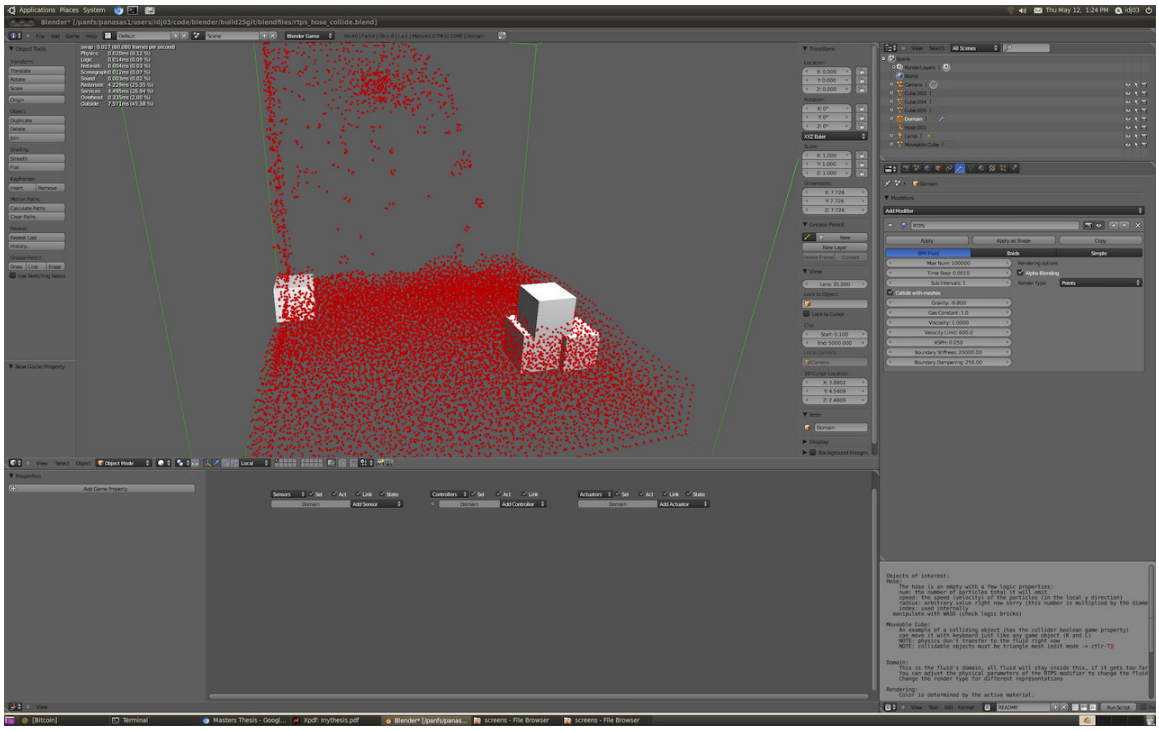

Figure 7.9: Demonstrating the Hose emitter with high viscosity. The maximum number of particles is set to 100k. Physical parameters are set as follows: Gravity = -9.8, Gas Constant = 1.0, Viscosity = 1.0, Velocity Limit = 600, XSPH = .2

### 7.4.1 Videos

Videos have been recorded throughout the process of this research and are available on the web. The videos serve as progress reports of the project throughout its development, each highlights new features or explains design decisions made. These videos are listed in reverse chronological order from most recent to oldest.

http://vimeo.com/21743717
http://vimeo.com/19794084
http://vimeo.com/17906099
http://www.youtube.com/watch?v=YvQFgY4kY68
http://www.youtube.com/watch?v=kXi70eaClnQ
http://www.youtube.com/watch?v=lP6zfAVm7B0

# CHAPTER 8

# FUTURE WORK

## 8.1 Optimization

The project demonstrates real-time performance on modern graphics hardware, however there is always room for improvement. Several bottlenecks in the code have known methods for elimination. Possibly the greatest speedup would involve improving the neighbor search. A technique known as Z-indexing would allow for neighbor lookups without storing the cell-indices array and consequently cutting a significant amount of memory accesses [17].

Another known bottleneck is the sorting algorithm used in the process of preparing the neighbor search. Researchers have demonstrated a highly efficient implementation of the Radix sort in CUDA [30] which can be ported to OpenCL.

Collision detection is currently implemented in a brute force manner. Data structures must be investigated which can handle particle collisions with objects at various scales. One such data structure is Hierarchical Spatial Subdivision, an extension of the current grid-based method for particle interaction [31]. Triangle-based collisions are desirable for their ability to handle general objects but are not necessary for all use cases. Implementing various collision primitives such as spheres, cylinders and axis aligned bounding boxes will allow for interesting interactions in game play at a lower performance cost.

## 8.2 Rendering

For the purpose of fluid effects in a game, it is important to consider how the fluid is displayed. This is especially difficult in real-time, as traditional rendering methods such as marching cubes have not yet been implemented at suitable speeds. Two approaches are being considered. First a screen-space technique that uses curvature flow to give the impression of a fluid surface [32]. Another approach is to extract the surface particles using data from the simulation and performing GPU ray casting on the much smaller subset of points [17]. Volume rendering techniques for the GPU have received increasing interest and may present an opportunity for real-time rendering [33].

It is also desirable to support effects for other kinds of fluids such as smoke and other particle system effects such as fire. For these techniques a variety of blending modes and custom textures must be made available to the user. Sorting particles with respect to depth from the camera is an essential task for enabling proper alpha blending used by some of

these effects.

## 8.3 Blender

The current integration with Blender gives a solid foundation for future improvements. One important goal is to move the user interface from the Modifiers to its own Particle System panel. This is desirable from the perspective of current Blender users who are already familiar with the existing particle system. Along these lines, another goal is to integrate with the existing particle system as much as possible, both in user interface as well as functionality. Further improvements to the user interface will include a special modifier for emitting particles from objects, eliminating the error prone and tedious method of adding custom game properties.

Another important goal is the creation of a Python interface to the RTPS library within Blender and the Game Engine. Python scripting gives the game designer powerful control over the behavior of their game and exposing RTPS functions to Python will allow for new and creative uses.

Currently collision detection and interaction with objects in the Game Engine do not use all of the available functionality present. The code could be made cleaner by utilizing existing classes for interacting with Objects as well as improved communication with the Bullet physics engine.

## 8.4 SPH

There are a host of improvements that can be made to the simulator itself, including better physical accuracy, more physical phenomena and interesting combinations of effects. There are several features available in the literature that can be readily implemented such as smoothing of field variables and improved smoothing kernels for better stability [24], surface tension and visco-elastic effects for more convincing and varied fluids [8]. Simulating multiple fluids interacting at once as well as phase changes based on temperature is also desirable [34]. SPH can also be extended to simulate granular materials such as sand [35].

A very important goal for the future is better interaction with solid boundaries and rigid objects. Semi-analytic boundary conditions have been put forth which allow for more physically accurate and computationally feasible collisions.[36] [37] Interaction with rigid bodies has been explored by representing the solid objects as a set of particles that are subject to fluid-like forces with rigid constraints.[38] Fluids have also been coupled with cloth simulations with nice results.[39]

# APPENDIX A

# CODE

## A.1 Download

The code for the entire modified Blender project is available for download at the following url
`https://github.com/enjalot/BGERTPS/tree/ianthesis`
If git[40] is installed, the following command will download the code:

```
git clone git://github.com/enjalot/BGERTPS.git
git checkout ianthesis
cd BGERTPS
git submodule init
git submodule update
```

The code for just the RTPS library is available for download at the following url
`https://github.com/enjalot/EnjaParticles/tree/ianthesis`
If git[40] is installed, the following command will download the code:

```
git clone git://github.com/enjalot/EnjaParticles.git
git checkout ianthesis
```

Build instructions for each platform are provided in the README and INSTALL files in the root directory of both source trees.

## A.2 SPHParams

The structure used to pass physical and simulation parameters to OpenCL.

```
#ifdef WIN32
#pragma pack(push,16)
#endif
typedef struct SPHParams
    {
        float mass;
        float rest_distance;
        float smoothing_distance;
```

```
        float simulation_scale;

        //dynamic params
        float boundary_stiffness;
        float boundary_dampening;
        float boundary_distance;
        float K;        //gas constant

        float viscosity;
        float velocity_limit;
        float xsph_factor;
        float gravity; // -9.8 m/sec^2

        float friction_coef;
        //next 4 not used at the moment
        float restitution_coef;
        float shear;
        float attraction;

        float spring;
        //float surface_threshold;
        //constants
        float EPSILON;
        float PI;       //delicious
        //Kernel Coefficients
        float wpoly6_coef;

        float wpoly6_d_coef;
        float wpoly6_dd_coef; // laplacian
        float wspiky_coef;
        float wspiky_d_coef;

        float wspiky_dd_coef;
        float wvisc_coef;
        float wvisc_d_coef;
        float wvisc_dd_coef;


        //CL parameters
        int num;
        int nb_vars; // for combined variables (vars_sorted, etc.)
        int choice; // which kind of calculation to invoke
        int max_num;

 } SPHParams
#ifndef WIN32
    __attribute__((aligned(16)));
#else
        ;
        #pragma pack(pop)
#endif
```

## A.3 Cell Indices Kernel

The OpenCL Kernel for computing the start and end indices pointing into the particle array for each grid cell.

```
#include "cl_macros.h"
#include "cl_structs.h"
```

```
__kernel void cellindices(
                            int num,
                            __global uint* sort_hashes,
                            __global uint* sort_indices,
                            __global uint* cell_indices_start,
                            __global uint* cell_indices_end,
                            //__constant struct SPHParams* sphp,
                            __constant struct GridParams* gp,
                            __local  uint* sharedHash   // blockSize+1 elements
                            )
{
    uint index = get_global_id(0);
    uint ncells = gp->nb_cells;

    uint hash = sort_hashes[index];
    //don't want to write to cell_indices arrays if hash is out of bounds
    if( hash > ncells)
    {
        return;
    }
    // Load hash data into shared memory so that we can look
    // at neighboring particle's hash value without loading
    // two hash values per thread
    uint tid = get_local_id(0);
    sharedHash[tid+1] = hash;  // SOMETHING WRONG WITH hash on Fermi

    if (index > 0 && tid == 0)
    {
        // first thread in block must load neighbor particle hash
        uint hashm1 = sort_hashes[index-1] < ncells ? sort_hashes[index-1] : ncells↩
            ;
        sharedHash[0] = hashm1;
    }

#ifndef __DEVICE_EMULATION__
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

    // If this particle has a different cell index to the previous
    // particle then it must be the first particle in the cell,
    // so store the index of this particle in the cell.
    // As it isn't the first particle, it must also be the cell end of
    // the previous particle's cell

    //Having this check here is important! Can't quit before local threads are done
    //but we can't keep going if our index goes out of bounds of the number of ↩
        particles
    if (index >= num) return;

    if (index == 0)
    {
        cell_indices_start[hash] = index;
    }

    if (index > 0)
    {
        if(sharedHash[tid] != hash)
        {
            cell_indices_start[hash] = index;
            cell_indices_end[sharedHash[tid]] = index;
        }
    }
    //return;
```

```
    if (index == num - 1)
    {
        cell_indices_end[hash] = index + 1;
    }
}
```

## A.4 Density Kernel

The OpenCL Kernel for computing the density of each particle.

```
//These are passed along through cl_neighbors.h
//only used inside ForNeighbor defined in this file
#define ARGS __global float4* pos, __global float* density
#define ARGV pos, density

#include "cl_macros.h"
#include "cl_structs.h"
//Contains all of the Smoothing Kernels for SPH
#include "cl_kernels.h"

inline void ForNeighbor(//__global float4*  vars_sorted,
                        ARGS,
                        PointData* pt,
                        uint index_i,
                        uint index_j,
                        float4 position_i,
                        __constant struct GridParams* gp,
                        __constant struct SPHParams* sphp
                        DEBUG_ARGS
                       )
{
    int num = sphp->num;
    float4 position_j = pos[index_j] * sphp->simulation_scale;
    float4 r = (position_i - position_j);
    r.w = 0.f; // I stored density in 4th component
    // |r|
    float rlen = length(r);

    // is this particle within cutoff?
    if (rlen <= sphp->smoothing_distance)
    {
        // return density.x for single neighbor
        float Wij = Wpoly6(r, sphp->smoothing_distance, sphp);

        pt->density.x += sphp->mass*Wij;
    }
}
//Contains Iterate...Cells methods and ZeroPoint
#include "cl_neighbors.h"

//------------------------------------------------------------
// compute forces on particles

__kernel void density_update(
//                        __global float4* vars_sorted,
                        ARGS,
                        __global int*    cell_indexes_start,
                        __global int*    cell_indexes_end,
                        __constant struct GridParams* gp,
```

```
                        __constant struct SPHParams* sphp
                        DEBUG_ARGS
                        )
{
    // particle index
    int nb_vars = sphp->nb_vars;
    int num = sphp->num;
    //int numParticles = get_global_size(0);
    //int num = get_global_size(0);

    int index = get_global_id(0);
    if (index >= num) return;

    float4 position_i = pos[index] * sphp->simulation_scale;

    //debuging
    clf[index] = (float4)(99,0,0,0);
    //cli[index].w = 0;

    // Do calculations on particles in neighboring cells
    PointData pt;
    zeroPoint(&pt);

    //IterateParticlesInNearbyCells(vars_sorted, &pt, num, index, position_i, ↩
        cell_indexes_start, cell_indexes_end, gp,/* fp,*/ sphp DEBUG_ARGV);
    IterateParticlesInNearbyCells(ARGV, &pt, num, index, position_i, ↩
        cell_indexes_start, cell_indexes_end, gp,/* fp,*/ sphp DEBUG_ARGV);
    density[index] = sphp->wpoly6_coef * pt.density.x;
    /*
    clf[index].x = pt.density.x * sphp->wpoly6_coef;
    clf[index].y = pt.density.y;
    clf[index].z = sphp->smoothing_distance;
    clf[index].w = sphp->mass;
    */
    clf[index].w = density[index];
}
```

# APPENDIX B

# MODIFIED FILES

A list of Blender source files modified to support RTPS library. A comprehensive list can also be generated with the help of a svn-patch.

```
CMakeLists.txt (lines 167-216) to add rtps library as include and link target
build_files/cmake/
    macros.cmake (lines 115-118 and 155-160)
source/creator
    CMakeLists.txt (modify install directives, lines 316-323 and 430-437)

//game engine stuff
source/gameengine/

    Converter/
        CMakeLists.txt
        BL_BlenderDataConverter.cpp (check for RTPS modifier so the game engine ↩
            modifier constructor knows what to do)
        BL_ModifierDeformer.h (define HasRTPSModifier and give property m_bIsRTPS)
        BL_ModifierDeformer.cpp (meat of the interface, Apply instantiates and ↩
            Update is responsible for updating the system)

    Rasterizer/
        CMakeLists.txt
        RAS_MaterialBucket.h (where we add our RTPS object to the mesh slot)
        RAS_MaterialBucket.cpp (setup, cleanup, and bypass transforming our system ↩
            with OpenGL)
        RAS_OpenGLRasterizer/
            CMakeLists.txt
            RAS_ListRasterizer.cpp (where we call RTPS->render();) (also move glew.↩
                h include up)
            RAS_OpenGLRasterizer.cpp (move glew.h up)
            RAS_VAOpenGLRasterizer.cpp (move glew.h up)

    BlenderRoutines/
        CMakeLists.txt
    Ketsji/
        CMakeLists.txt
        KX_Dome.h (moved glew.h include above the other includes to avoid conflicts↩
            )
        KX_KetsjiEngine (moved KX_Dome.h above the other includes because it ↩
            includes glew.h)

    Physics/Bullet/
        CMakeLists.txt
```

```
        CcdPhysicsEnvironment.cpp (moved glew.h up)
    VideoTexture/
        CMakeLists.txt
        Texture.cpp (moved glew.h up)

//custom modifier stuff ( http://enja.org/2010/05/24/blender-creating-a-custom-↩
    modifier/ )
source/blender/
    blenkernel/
        BKE_modifier.h
    makesdna/
        DNA_modifer_types.h (defines struct and type)
    makesrna/
        RNA_access.h
        intern/
            rna_modifier.c
    modifiers/
        CMakeLists.txt
        MOD_modifiertypes.h
        intern/
            MOD_rtps.c
            MOD_util.c

    blenlib/
        BLI_path_util.h     (add BLENDER_RTPS constant as a runtime path)
        intern/path_util.c

release/scripts/ui/
    properties_data_modifier.py (Python UI)

intern/opencl
    //the opencl header files from khronos with slight modification

intern/memutil/intern/
    MEM_CacheLimiterC-Api.cpp (had to change #include <malloc.h> to #include <↩
        stdlib.h> for mac

extern/rtps
    //use the git submodule command to get this directory: (execute these commands ↩
        from the base directory)
        git submodule init
        git submodule update
```

# BIBLIOGRAPHY


[1] Evan Bollig. *CENTROIDAL VORONOI TESSELATION OF MANIFOLDS USING THE GPU*. PhD thesis, Florida State University, 2009.

[2] Khronos Group. OpenCL Specification, 2011.

[3] Y. B. Kafai. Playing and Making Games for Learning: Instructionist and Constructionist Perspectives for Game Studies. *Games and Culture*, 1(1):36–40, January 2006. ISSN 1555-4120. doi: 10.1177/1555412005281767. URL `http://gac.sagepub.com/cgi/doi/10.1177/1555412005281767`.

[4] C Ferguson. Evidence for publication bias in video game violence effects literature: A meta-analytic review. *Aggression and Violent Behavior*, 12(4):470–482, July 2007. ISSN 13591789. doi: 10.1016/j.avb.2007.01.001. URL `http://linkinghub.elsevier.com/retrieve/pii/S1359178907000055`.

[5] E. R. Hayes and I. a. Games. Making Computer Games and Design Thinking: A Review of Current Software and Strategies. *Games and Culture*, 3(3-4):309–332, July 2008. ISSN 1555-4120. doi: 10.1177/1555412008317312. URL `http://gac.sagepub.com/cgi/doi/10.1177/1555412008317312`.

[6] Jie Tan and XuBo Yang. Physically-based fluid animation: A survey. *Science in China Series F: Information Sciences*, 52(5):723–740, May 2009. ISSN 1009-2757. doi: 10.1007/s11432-009-0091-z. URL `http://www.springerlink.com/index/10.1007/s11432-009-0091-z`.

[7] Mathieu Desbrun and M.P. Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation*, volume 96, pages 61–76. Citeseer, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.3447&rep=rep1&type=pdf`.

[8] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, page 219, 2005. doi: 10.1145/1073368.1073400. URL `http://portal.acm.org/citation.cfm?doid=1073368.1073400`.

[9] Nils Thurey, Klaus Iglberger, and Ulrich Rude. Free Surface Flows with Moving and Deforming Objects for LBM Free Surface Flows with LBM.

[10] Matthias Müller and David Charypar. Particle-based fluid simulation for interactive applications. *Proceedings of the 2003 ACM*, 2003. URL http://portal.acm.org/citation.cfm?id=846298.

[11] Daniel Kallin. Real-Time Large Scale Fluids for Games. 2009.

[12] M. Kelager. Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics. *Department of Computer Science, University of Copenhagen*, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.6640&rep=rep1&type=pdf.

[13] Takahiro Harada, Seiichi Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Computer Graphics International*, pages 63–70, 2007. URL http://individuals.iii.u-tokyo.ac.jp/~{}yoichiro/report/report-pdf/harada/international/2007cgi.pdf.

[14] Pyarelal Knowles. GPGPU Based Particle System Simulation. *Star*, pages 1–33, 2009.

[15] Alexander Seizinger. Numerical Simulation of Particle Agglomerates. 2010.

[16] Ø ystein Eklund Krog. GPU-based Real-Time Snow Avalanche Simulations. (June), 2010.

[17] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive SPH Simulation and Rendering on the GPU. *Computer*, pages 1–10, 2010.

[18] R.C. Hoetzlein and Tobias Höllerer. Analyzing Performance and Efficiency of Smoothed Particle Hydrodynamics. *sph-sjtu-f06.googlecode.com*. URL http://sph-sjtu-f06.googlecode.com/files/sph_paper.pdf.

[19] Khronos Group. OpenGL, March 2011. URL http://www.opengl.org.

[20] David Luebke. How GPUs Work. *Computer*, 40(2):96–100, February 2007. doi: 10.1109/MC.2007.59.

[21] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):1–33, 2007.

[22] Khronos Group. OpenCL, March 2011. URL http://www.khronos.org/registry/cl/.

[23] J J Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68 (8):1703–1759, August 2005. URL http://stacks.iop.org/0034-4885/68/i=8/a=R01?key=crossref.c562820df517a049ca7f11d0aefe49b4.

[24] M. B. Liu and G. R. Liu. Smoothed Particle Hydrodynamics (SPH): an Overview andRecent Developments. *Archives of Computational Methods in Engineering*, 17(1): 25–76, February 2010. ISSN 1134-3060. doi: 10.1007/s11831-010-9040-7. URL http://www.springerlink.com/index/10.1007/s11831-010-9040-7.

[25] C Ericson. *Real-time collision detection*. Morgan Kaufmann, 2005.

[26] NVIDIA. OpenCL Sorting Networks, 2011. URL `http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html#oclSortingNetworks`.

[27] Kitware. CMake, March 2011. URL `http://www.cmake.org`.

[28] Blender Foundation. Blender, April 2011. URL `http://wiki.blender.org/index.php/Dev:2.5/Source/Architecture/DataAPI`.

[29] Ian Johnson. Blender: Creating a Custom Modifier, March 2011. URL `http://enja.org/2010/05/24/blender-creating-a-custom-modifier/`.

[30] Duane Merrill and Andrew Grimshaw. Revisiting sorting for gpgpu stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[31] Mickael Pouchol, Alexandre Ahmad, Benoit Crespin, and Olivier Terraz. A Hierarchical Hashing Scheme for Nearest Neighbor Search and Broad-Phase Collision Detection. pages 45–59, 2009.

[32] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*, page 91, 2009. doi: 10.1145/1507149.1507164. URL `http://portal.acm.org/citation.cfm?doid=1507149.1507164`.

[33] Roland Fraedrich, Stefan Auer, and Rüdiger Westermann. Efficient high-quality volume rendering of SPH data. *IEEE transactions on visualization and computer graphics*, 16(6):1533–40, 2010. ISSN 1077-2626. doi: 10.1109/TVCG.2010.148. URL `http://www.ncbi.nlm.nih.gov/pubmed/20975195`.

[34] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, page 237, 2005. doi: 10.1145/1073368.1073402. URL `http://portal.acm.org/citation.cfm?doid=1073368.1073402`.

[35] Nathan Bell, Yizhou Yu, and Peter J. Mucha. Particle-based simulation of granular materials. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, page 77, 2005. URL `http://portal.acm.org/citation.cfm?doid=1073368.1073379`.

[36] Sauro Manenti, Mario Gallati, Stefano Sibilla, Giordano Agate, and Roberto Guandalini. SPH Modeling of Solid Boundaries Through a Semi-Analytic Approach. *Engineering*, 5(1):1–15, 2011.

[37] Takahiro Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics in complex shapes. In *Proc. of spring conference on computer graphics*, pages 26–8, 2007. URL `http://individuals.iii.u-tokyo.ac.jp/~{}yoichiro/report/report-pdf/harada/international/2007sccg.pdf`.

[38] Takahiro Harada, Masayuki Tanaka, and Seiichi Koshizuka. Real-time Coupling of Fluids and Rigid Bodies. *Simulation*, pages 1–13, 2007.

[39] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Real-time Fluid Simulation Coupled with Cloth. *Theory and Practice*, 2007.

[40] Linus Torvalds. Git, April 2011. URL `http://git-scm.com`.

# BIOGRAPHICAL SKETCH

Ian 'enjalot' Johnson was born on the 6th of October, 1985 in Leiden, Netherlands. Ian grew up in Tallahassee, Florida where he became interested in computers at a young age. One chapter of his youth involved lucrative yet misguided efforts in hacking video games. Upon reaching some semblance of maturity Ian has channeled his creative energies into constructive projects and now enjoys building tools that help people create video games.

Ian's research interests include computer graphics, scientific visualization, and interactive education. When not studying, he enjoys spending time with his loved ones, teaching those around him and traveling and eating local foods.

Ian currently lives in Tallahassee, Florida but can be found from anywhere in the world by visiting his web-site `http://enja.org`.