POLYMER SIMULATION USING NVIDIA CUDA

By

Josh Lohse

Advisor: Mircea Teodorescu

Department of Computer Engineering

University of California Santa Cruz

December 2014

Abstract

The applications of new polymers and elastic materials has grown over the years. The issue is there is not many modeling software packages that can help predict behavior of a new polymer before an experiment is conducted. There currently is a package that can do this but it is very slow and is not powerful enough to handle larger sets of particles. This thesis provides a software package based on the previous packages equations but moves the calculations onto the gpu to provide faster computation times while being able to increase the complexity of the simulation. The method of computation of the simulation is done with Smooth particle Hydrodynamics(SPH) and the program is written using NVIDIA CUDA in C++.

Keywords: CUDA, Smooth Particle Hydrodynamics, Polymer

Contents

Chapter 1

Introduction

Polymers and other viscoelastic materials are becoming more commonly seen in micro scale applications. These polymers and elastomers may be understood in a macro sense for large applications but there is no reliable and fast simulation of these materials in the micro scale. The current simulation techniques require days and sometimes up to a week to complete [1].

The goal of this thesis is to provide a quicker simulation based on the previous CPU code and equations. The GPU is the most likely candidate for this because of its low cost over parallel CPUs and its inherent ability to handle massive numbers of simultaneous parallel computations.

Using CUDA, which is NVIDIAs architecture and language extension libraries, this software can simulate a more complex problem than the CPU code used before while increasing performance. The CUDA sdk includes many examples include the particle simulation which is the base of this software. The framework is left mostly unchanged but most of the kernel calculations are all modified to use the smoothed-particle hydrodynamic(SPH) equations.

1.1 Purpose and Goal

The purpose is to create a GPU accelerated application using the SPH equations. The software will be used to simulate polymers in the micro scale, such as polymer fibers that emulate the setae or hair like structure on the hands of geckos. One of the major goals is to use this software to look at the adhesive properties of these polymers.

1.2 Usability

The software should not need to modified to be run with different parameters. The program uses a text file which contains all of the parameters that the program uses during the simulation. This includes particle groups with their parameters, global constants, and simulation parameters such as data collection on and what type of integration method etc. This file is read by the program at startup so the program doesn't have to be recompiled each time and no clunky GUI is used.

There are two main ways to view the results of the program, the first is the graphics which is the existing opengl implementation from the sample, and the second is the data that is dumped at intervals from the GPU. The graphical view is the fastest and offers a real time view of the simulation while its running. The data collection moves some or all of the arrays of interest from the GPU memory to system memory and then written to the hard disk for analysis after the simulation run.

1.3 Future modifications

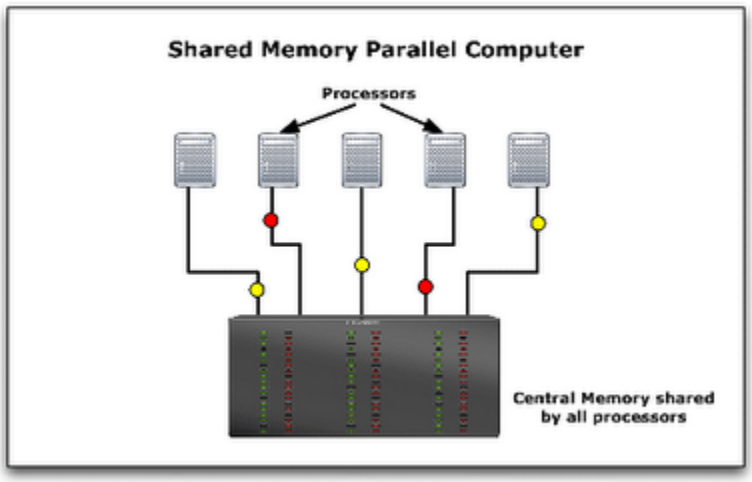
The software must be easy to use as well as be easily modified for future expansion. This requires good documentation and clear design approaches that limit the number of points to modify per new feature. For example, to add a constant only three areas are modified:

1. The function that checks an input line from the file
2. The struct containing the constants for either the group or global
3. The GPU kernel using the constant

The various integration and sph kernel methods are kept in classes to make adding new methods easier and neater than functions.

Chapter 2

Parallel Computation and GPUs



This section discusses parallel computing, its history and the advent of the GPU in scientific computing. The focus of this section is NVIDIA CUDA specifically its architecture and the C++ libraries.

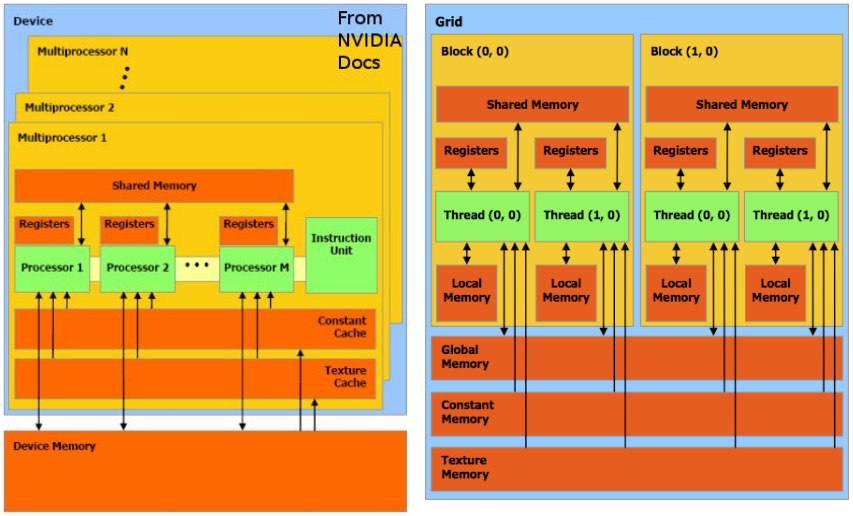
2.1 Parallel Processing

The concept of parallel computation and multiprocessor systems are nothing new with the advent of GPUs. Many of the first machines were single instruction single data(SISD) but most systems are now multiple instruction multiple data(MIMD). These classifications were created by Michael Flynn and are a part of what is called Flynn's Taxonomy [1]:

- Single Instruction, Single Data(SISD)
- Single Instruction, Multiple Data(SIMD)
- Multiple Instruction, Single Data(MISD)
- Multiple Instruction, Multiple Data(MIMD)

GPUs or whats known as General-purpose computing on Graphical Processing Units(GPGPU) fall under the SIMD category which makes them perfect for repetitive tasks such as graphics and image processing. This also makes it a good fit for a particle simulation.

2.2 GPU Structure



The NVIDIA GPU architecture particularly CUDA was first released in 2007 and began support at the 8000 series of GPUs. At its most basic the GPU is many smaller core that are paired in groups called streaming multiprocessors(SMs) and the size depends on the generation. Each of these SMs can contain up to 8 single precision(SP) and one double precision(DP) cores which correspond to threads at the program level [2]. Since the GPU is focused on many small simultaneous calculations more area of the chip or transistors are used for arithmetic logic units.

The major downside of the GPU architecture is the lack of large caches which means that if the program needs a memory access then that access is very expensive. Even more expensive is an access to global memory which could negate the benefits of using the GPU in the first place.

2.3 NVIDIA CUDA

NVIDIA CUDA as mentioned before is a hardware model, but it is also a programming model as well. The goal of CUDA is to make GPU computation easier to program since early adaptations required the programmer to adapt opengl to write programs.

The benefit to CUDA in the application of SPH is that it fits the GPU parallel programming model perfectly since the system is a very large number of particles, sometimes in excess of one million which is easily handled by the existing simulation.

The simplification that CUDA makes allows for easy adaptation of existing C code to be run on GPUs. The only caveat is that the code must be fairly simple and cannot contain any complex data structures.

Chapter 3

Particle Simulation

This software is based on the CUDA particle simulation example given in the CUDA sdk. This simulates a large number of particles, mainly >32k, colliding with one another inside a fixed sized box shown in fig 1.

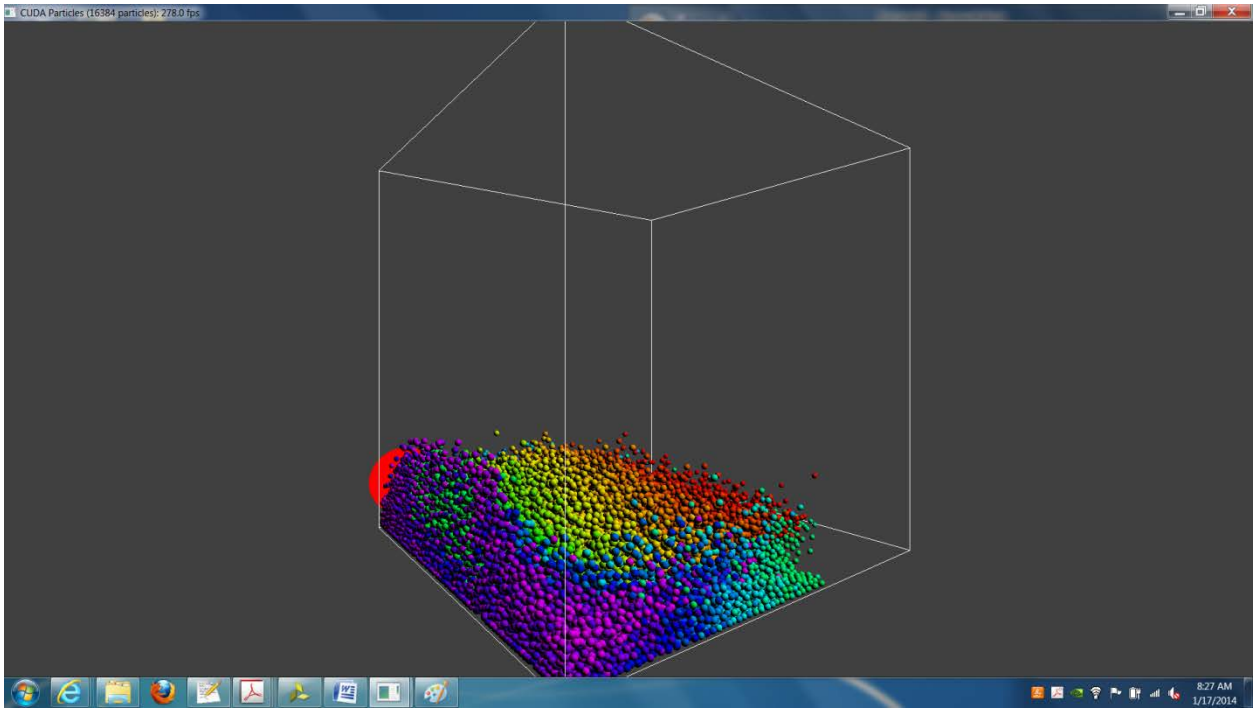


Figure 1, Particle simulation with 32k particles

This is used to demonstrate the power of the GPU when it comes to massively parallel operation, or basically when each particle is run on its own thread. This example is not completely grounded in reality, but its close as will be shown later. For the needs of the project goal this is a very good starting point because of the problem is very similar and can be adapted relatively easily.

3.1 Structure

There are three steps that this simulation goes through each frame or time step:

1.)Integration

2.) Data Structure sorting

3.) Collision calculation

The first step is the integration which in the original program uses the Euler method:

```
vel += params.gravity * deltaTime;
vel *= params.globalDamping;
pos += vel * deltaTime;
```

Each particles position is updated depending on the last time steps resulting velocity vector. This function also detects any collision between the threads particle and any of the six bounding walls. If there is then it applies a boundary damping constant to the current velocity:

```
if (pos.x > 1.0f - params.particleRadius){
        pos.x = 1.0f - params.particleRadius;
        vel.x *= params.boundaryDamping;
}
```

The integration step is later modified to allow the user to choose between different methods of integration which is covered later.

Now that each particle is updated as far as position how does the program establish a collision? The approach taken is to sort the particles in such a way that any particle within a certain distance of the threads particle will be placed next to it in the sorted array. The last part is important because if the program was to update the original position array then it would be impossible to keep track of each particles velocity unless this array was changed as well. To increase performance the divide and conquer approach is taken by dividing the 3d cube into smaller cubes about the size of a particle as shown in figure 2.

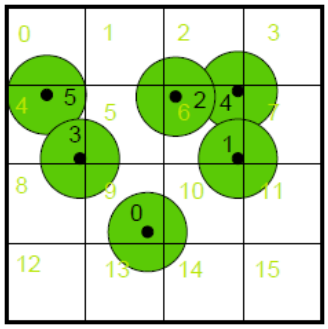


Figure 2, Grid data structure inside cube, cells hold particles

Each time deltaTime occurs the grid is re-calculated, ie each particle is placed in a maximum of four different cells. The program also offers a different approach which uses a hash table to organize the particles. The two approaches depend on what hardware is used, the first method only works for newer generations(GTX 200 series and above).

The algorithm has three steps:

1.) Calculate hash values of particles
2.) Sort particles by fast-radix sort
3.) Find start of each cell in list

The hash values in this program end up just being the cell id that that particle is in.

```
int3 gridPos = calcGridPos(make_float3(p.x, p.y, p.z));
uint hash = calcGridHash(gridPos);

gridParticleHash[index] = hash;
gridParticleIndex[index] = index;
```

Two functions are called, the first is calcGridPos which takes the x,y,z cordinate and finds the cell number. The second is calcGridHash which just calculates the index of where the particle is entered into the global array.

The second step calls a library function for CUDA to implement a fast-radix sort. This sorts the particles by both the cell and particle number. At the end of the sort, the array is scanned to store where each cell ends and begins. This is done to make the collision detection and calculation quicker due to faster memory accesses.

The final step is to detect collisions and if there are calculate the force exerted. Each particle is its own thread so only the force on the current particle is calculated and not for the others. The first function call is to collideD()which iterates over all neighboring cells to the cell that the particle is currently in. This means it iterates 3x3x3 cells around the current. This is to check if a particle collides with other particles in other cells, this is the case where the particle maybe on the edge of the cell.

```
for (int z=-1; z<=1; z++){
        for (int y=-1; y<=1; y++){
                for (int x=-1; x<=1; x++){
                        int3 neighbourPos = gridPos + make_int3(x, y, z);
                        force += collideCell(neighbourPos, index, pos, vel, oldPos,
                        oldVel, cellStart, cellEnd);
                }
        }
}
```

The function collideSphere() is called inside collideCell() when two spheres are inside of a specified constant distance. The collideSphere() function is where each pair of particles is collided to get a force which is added to a running sum. This sum is then added to the velocity vector of the current particle and then returned. The list of equations is listed below:

$$dist = length(relPos)$$
$$\overrightarrow{\boldsymbol{relpos}} = \overrightarrow{\boldsymbol{posB}} - \overrightarrow{\boldsymbol{posA}}$$

$$\overrightarrow{\boldsymbol{norm}} = \frac{\overrightarrow{\boldsymbol{relPos}}}{dist}$$

$$\overrightarrow{\boldsymbol{relVel}} = \overrightarrow{\boldsymbol{velB}} - \overrightarrow{\boldsymbol{velA}}$$
$$\overrightarrow{\boldsymbol{tanVel}} = \overrightarrow{\boldsymbol{relVel}} - \overrightarrow{\boldsymbol{norm}} \cdot (\overrightarrow{\boldsymbol{relVel}} \llcorner \overrightarrow{\boldsymbol{norm}})$$

$$x = collideDist - dist$$
$$\zeta = Damping\ ratio\ (0 \rightarrow 1.0)$$
$$k = spring\ constant\ (0 \rightarrow 1.0)$$
$$\tau = shear\ constant\ (0 \rightarrow 0.1)$$
$$\alpha = attraction\ (0 \rightarrow 0.1)$$
$$g = gravity\ constant\ (0.001 \rightarrow 0.003)$$

$$\overrightarrow{\boldsymbol{force_{i\ on\ 0}}} = -kx \cdot \overrightarrow{\boldsymbol{norm}} + \zeta \cdot \overrightarrow{\boldsymbol{relVel}} + \tau \cdot \overrightarrow{\boldsymbol{tanVel}} + \alpha \cdot \overrightarrow{\boldsymbol{relPos}}$$

$$Total\ Force\ on\ Particle_0 = \sum_{i=1}^{N} \boldsymbol{force_{i\ on\ 0}}$$

These equations are contained in collideSpheres() function and the resulting velocity after integration is listed below:

$$\overrightarrow{\boldsymbol{pos_{t+\Delta t}}} = \overrightarrow{\boldsymbol{pos_t}} + GD(\overrightarrow{\boldsymbol{v_t}} \cdot \Delta t) + GD(G \cdot \Delta t^2)$$

$$\overrightarrow{\boldsymbol{force_{i\ on\ 0}}} = -kx \cdot \overrightarrow{\boldsymbol{norm}} + \zeta \cdot \overrightarrow{\boldsymbol{relVel}} + \tau \cdot \overrightarrow{\boldsymbol{tanVel}} + \alpha \cdot \overrightarrow{\boldsymbol{relPos}}$$

$$\vec{\boldsymbol{v}}_{\boldsymbol{t+\Delta t}} = GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}} + G \cdot \Delta t) + \sum_{i=1}^{N} \boldsymbol{force_{i\ on\ 0}}$$

3.2 Data Flow

Flow charts showing steps of each timestep are shown in figures 3 and 4.

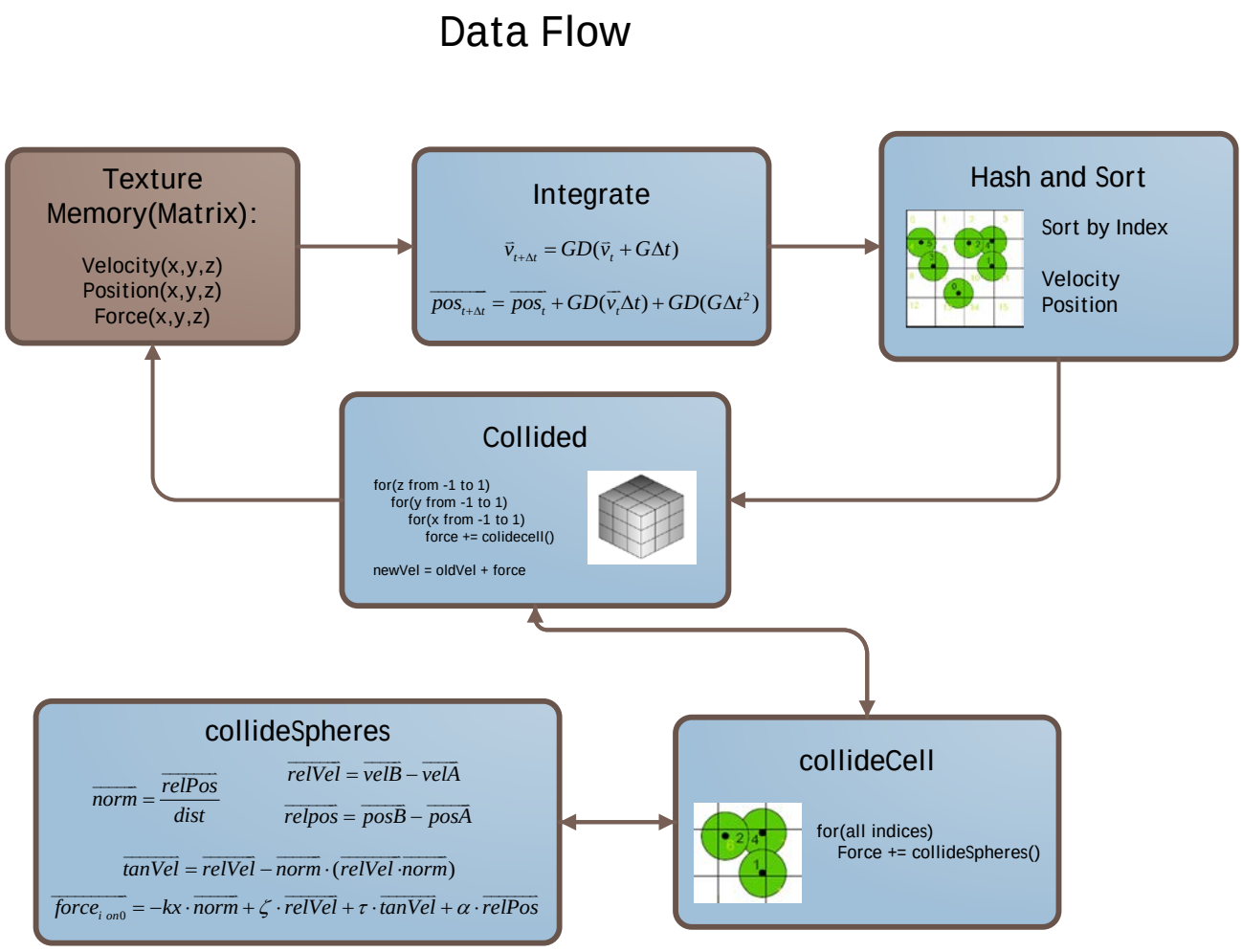


Figure 3, Function call flow, collideSpheres is called each time a collision is detected

The CUDA kernel functions first start off in the position integration function which updates the position on the previously calculated velocity. These positions are then passed to the functions in Hash and Sort which sort particles based on their location, so any particles that are inside or near cell 1 are group together. The indecies for the start and end of cells in the sorted arrays are also kept in arrays to be passed onto the Collided functions. These functions calculate the collisions on each particle within a cell. It iterates over all 26 cells surround the particle the thread is looking at.

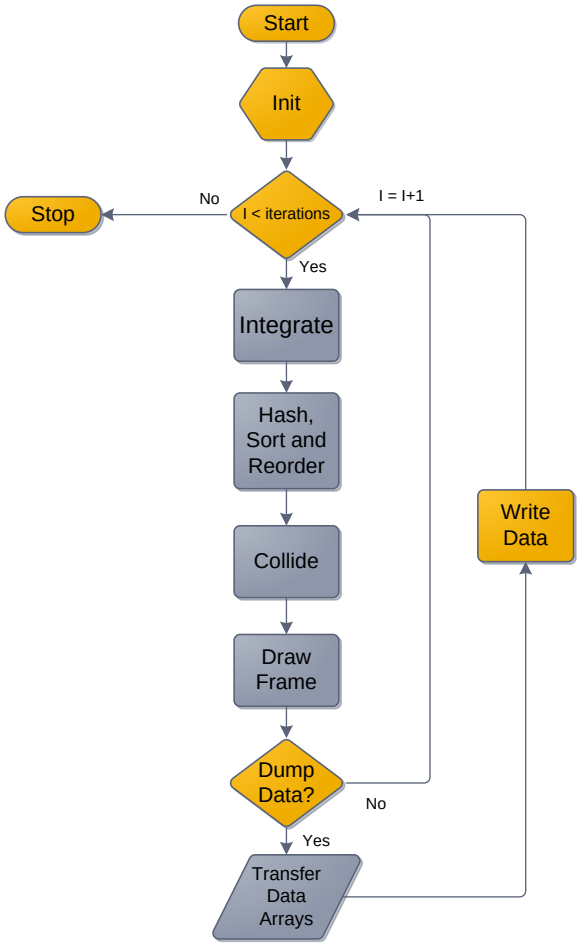


Figure 4, Flow chart showing same function calls but also data collection

3.3 Testing

In order for a program like matlab to read in the data, the format needed to be changed to something like:

Time	X	Y	Z	X	Y ...
0.0	0.5	0.5	0.0	-0.5	-0.5...
0.1	0.49	0.49	0.0	-0.49	-0.49...

This is fairly basic C++ I/O operation so I won't go over this, but the major issue is when and where to output the data to a file. This has to be done after the calculation, but before the display. This means that the arrays are dumped every calculation cycle which is not efficient but it works for this test.

The simulation setup I chose to go with is one of the most simple while still requiring gravity. This involves two particles equal distance apart as well as height and velocity. The setup for this simulation is added to a case statement in the reset() function in particlesystem.cpp as shown below:

```
case CONFIG_TEST:
{
	int p=0;
	int v=0;
	m_hPos[p++] = -0.5f;
	m_hPos[p++] = 0.5f;
	m_hPos[p++] = 0.25f;
	m_hPos[p++] = 1.0f; // radius

	m_hPos[p++] = 0.5f;
	m_hPos[p++] = 0.5f;
	m_hPos[p++] = 0.25f;
	m_hPos[p++] = 1.0f; // radius

	m_hVel[v++] = 0.01f;
	m_hVel[v++] = 0.0f;
	m_hVel[v++] = 0.0f;
	m_hVel[v++] = 0.0f;

	m_hVel[v++] = -0.01f;
	m_hVel[v++] = 0.0f;
	m_hVel[v++] = 0.0f;
	m_hVel[v++] = 0.0f;
}
```

The coordinate system for this program has its origin at the center of the floor of the box. The 'y' direction is height while the 'z' direction is depth from the original viewing position. For the test all constants are left as the default values except the time-step is set to 0.01.

```
float timestep = 0.1f;
float damping = 1.0f;
float gravity = 0.0003f;

float collideSpring = 0.5f;;
float collideDamping = 0.02f;;
float collideShear = 0.1f;
float collideAttraction = 0.0f;
```

The path of the particles is shown in figure 1, 2, 3, and 4.

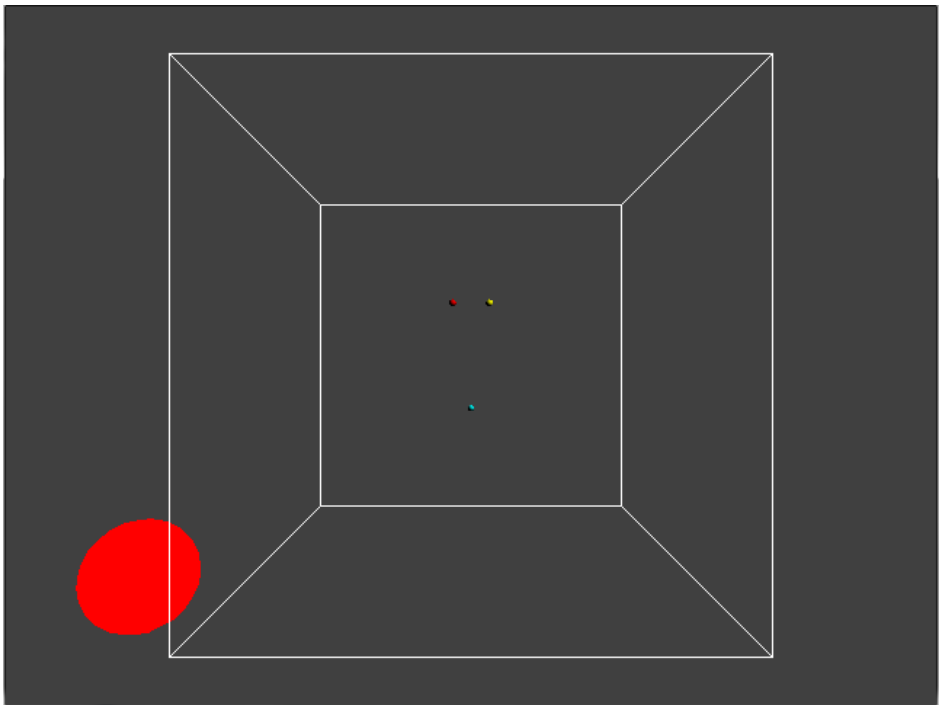


Figure 4, Particles just before collision with equal velocities

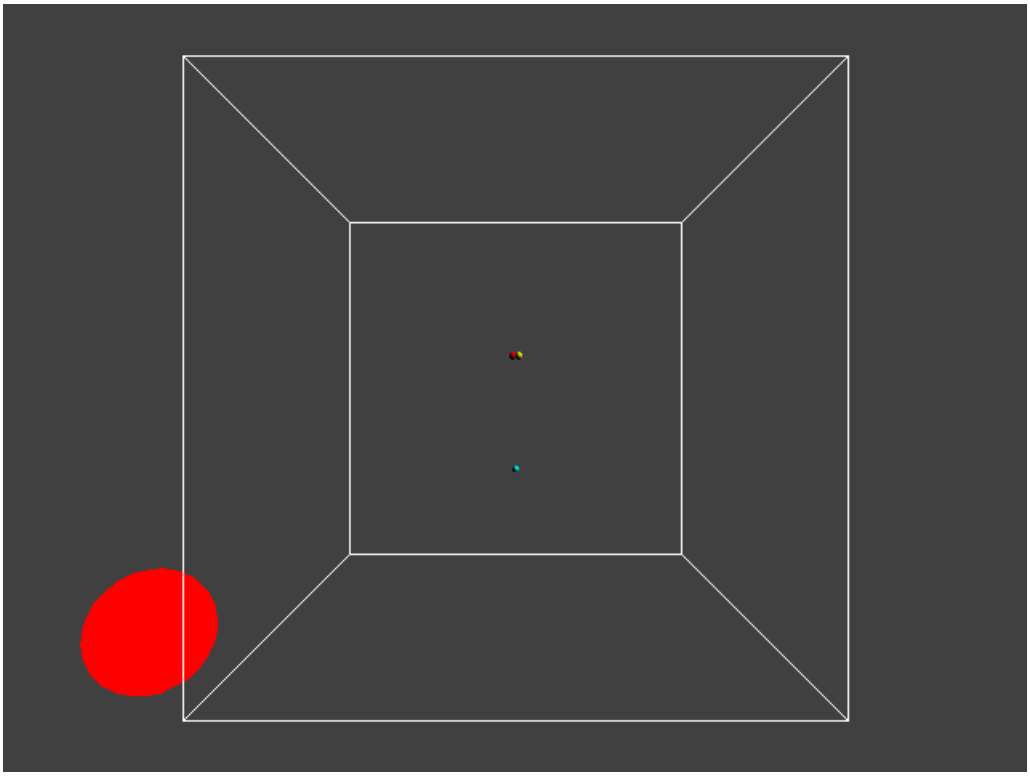


Figure 5, Particles at collision point

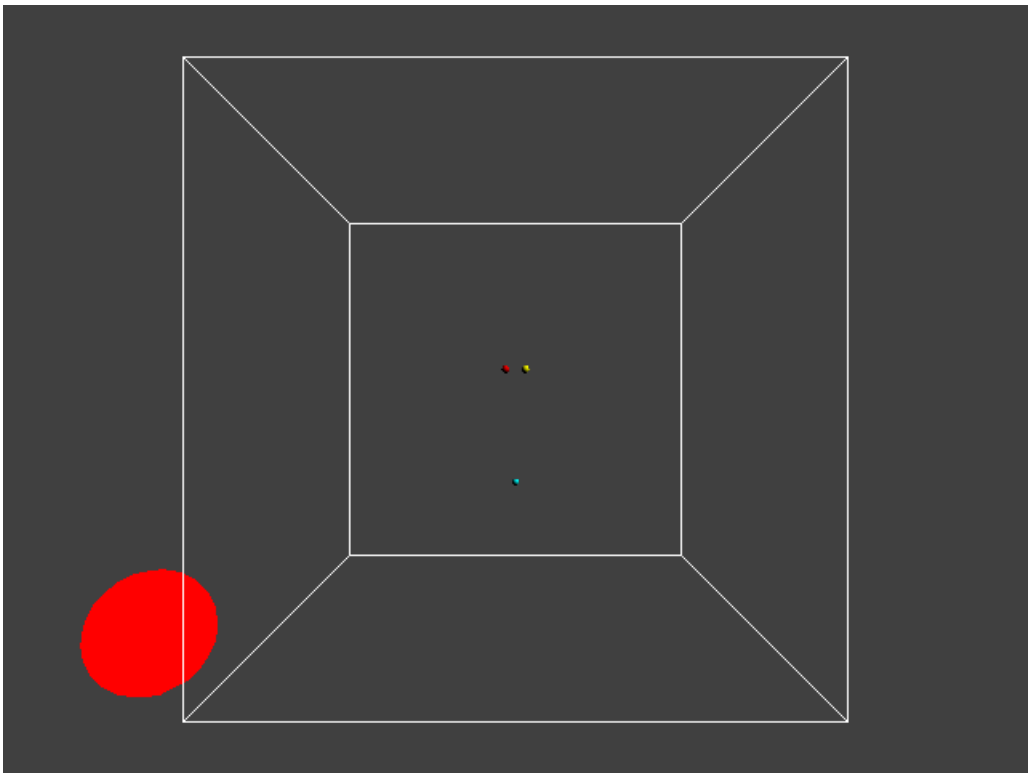


Figure 6, Particles just after collision at which the simulation ends

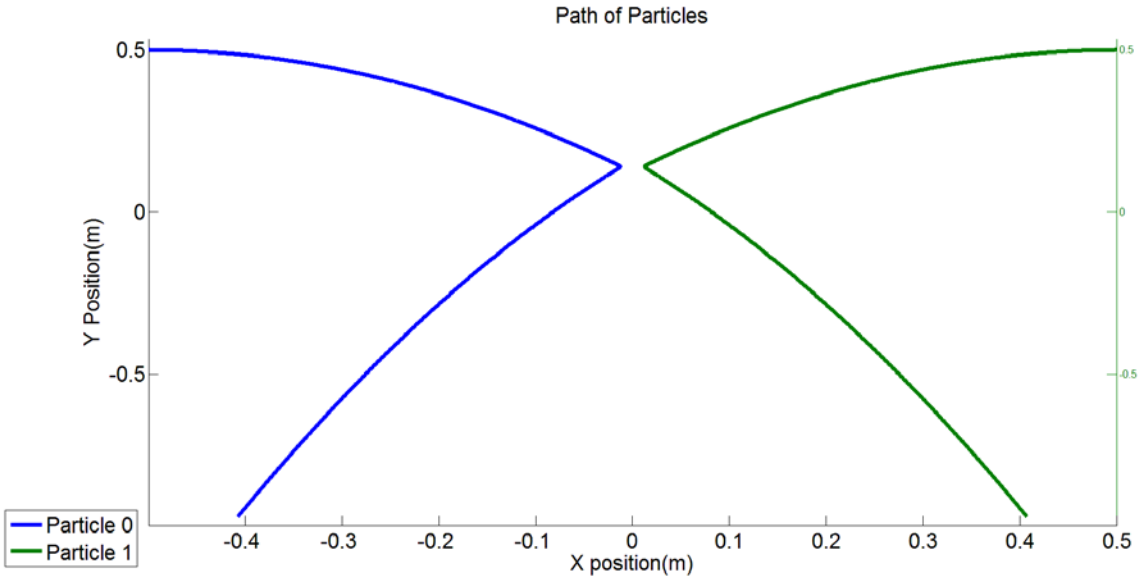


Figure 7, data retrieved from simulation showing the path of the particles plotted in MATLAB

The next thing looked at is the velocity and force data which were also dumped at the same time as the position data. I separated these data sets into three files each containing data for each

particle. The result of the velocity in x are shown in figure 5, velocity in y shown in figure 6, and the total force on particle 0 are shown in figure 7.

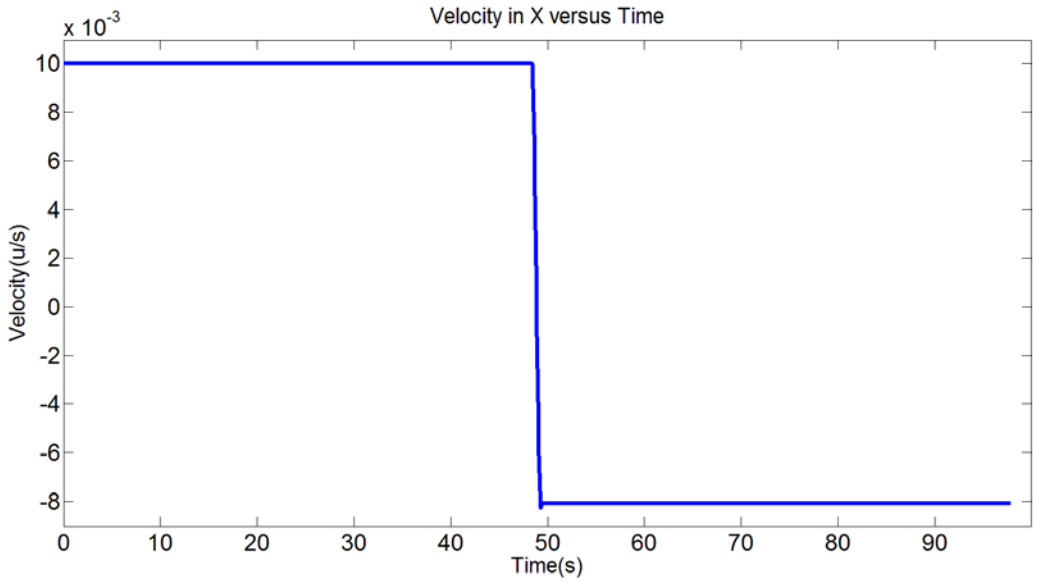


Figure 8, Velocity in the x direction reverses and drops in magnitude due to collision damping

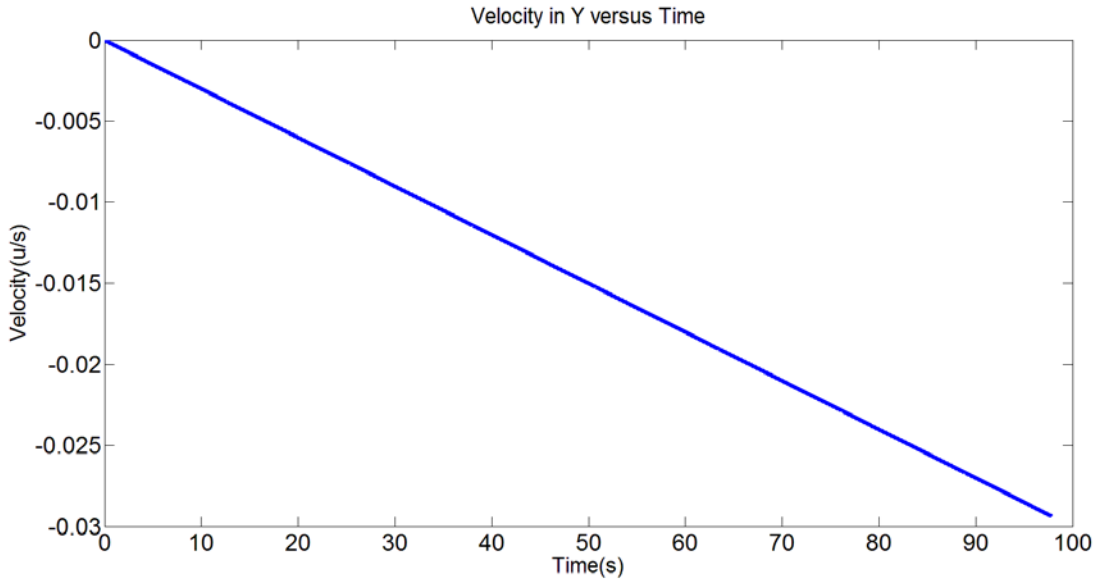


Figure 9, velocity in the vertical direction increases linearly as expected

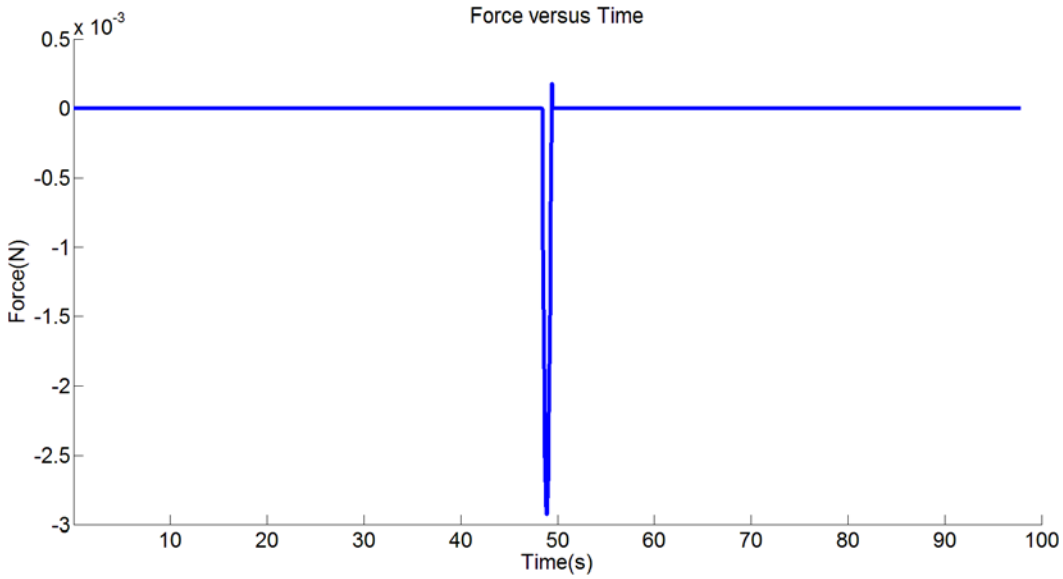


Figure 10, collision force on particle 0 showing dip only during collision

Chapter 4

Init File

The goal of this software is to provide an easy to use package to model various polymers, so instead of working with a clunky GUI a script of initfile is used. This program is C++ and windows based so a script is not the best option, so instead a simple text file is read by the program before the simulation begins.

The first step was to establish what sort of variables is the user likely to want to change easily and quickly. This may include the ability to turn the data collection or graphics on or off depending on what is most important. The other set is simulation constants both global as well as group specific. This means that two separate group are needed, the global settings and the group settings.

The parser must be able to distinguish between the two, so the global settings are listed first and the groups second. The other issue is how to distinguish between group which was solved by the syntax of "[group name]". The parser must first check to see if the next line it read contains a '[' at char 0, if it does than create a new group and append it to the group vector. A list of goals for the parser are listed below:

1. The user will never have to touch code to change existing constants and settings
2. If the user needs to add a constant, make it easy to understand and implement, few places to change as possible.
3. Stable under user error, spacing doesn't affect parsing and errors are thrown for unrecognized constants
4. Allows comments in files anywhere on a line denoted by '#'

The first two goals are the most important because while the user may have an understanding of code, it still takes time to figure out a poorly written section of code. The solution is to use two types of structs, one is for the global parameters which all groups share and a group struct which all the particles in a particular group share. The global struct is just the existing simulation parameters struct which was used for particle collision constants, but now just contains global constants like gravity and air resistance. The group struct is new but not any different except that it's contained in a vector with the other groups.

The program only has to be modified in three place in order to add new simulation constants. The first is the struct itself which is just adding the entry. The second is the function void set_key_value(group& g, GroupParams& gp, vector<string>& pair) which takes a pair of strings, the left and right side of the assignment, and sets that constant to the value specified. This function is a long if else-if structure so all that needs to be done is to add another statement. The third is the kernel itself which is where the calculations occur.

4.1 GroupParams

This program allows multiple groupings of particles for say different shapes which increases the complexity of data structures associated with that group. Originally the program uses a single group of parameters for every particle in the simulation. This will not work since the number of groups would have to be hardcoded into the code which is far from user friendly. The solution is to have a general struct which contains the constants and then an array of pointers to these structs. This allows any number of groups to be formed without the user worrying about the code.

This new array creates a problem because transferring is nontrivial when dealing with external devices, GPUs included. The following two function calls handle the allocation and copying:

```
cudaMalloc((void**)&d_gparams_array,
            10 * sizeof(GroupParams));

checkCudaErrors(cudaMemcpy(d_gparams_array, gparams_array,
                10 * sizeof(GroupParams),
                cudaMemcpyHostToDevice));
```

This is only an array of struts' which contain primitives so the allocation and copying are simple. If the struct where to contain an array, then a separate allocation and copy must be made for each struct in the original array.

4.2 Input Checking

The parser is a completely custom set of function that handles white space, comments and input error. The steps the parser completes for each line is shown below:

1. Check for empty line, if it is then skip
2. Remove whitespace
3. If after the whitespace removal the line is empty then skip
4. Read first character, if it's a '[' start a new group and skip next step
5. Split the line in half by the '=' and send to the set value function. This function is a if-else tree that checks to see if the input matches existing constants, if it does then check and apply the input value to it in the struct.

To provide a user friendly environment all possible corner cases of user input error have been checked. Any error in input may be difficult to debug if it occurs since these are passed to a simulation where the behavior may not be know before it is run.

These functions are run before the particle system is initialized so a large array is kept with the group number for each particle. This array is passed at the time of construction of the particle system. This was done because the particle count must be read before the particle system is constructed.

4.3 How the GPU uses these

The data is transferred into memory at location `gparams_array` from the host, but this pointer still needs to be passed when the first kernel call is made. This pointer is passed from function to function until it reaches the collideSpheres() which uses this pointer.

The data for the simulation is stored in arrays of four dimensional vectors. For some reason CUDA only supports float4's which could be if you need the fourth dimension to be time. Since each particle is part of a group an index into the group_params array is needed. This index could be stored in a separate array but this is wasteful both in memory allocation as well as an extra memory access per thread. The four dimension is not used by the existing functions so this can be used to store the index. The first thing collideSpheres() does is read that index and access the struct at that index in group_params_array[].

Chapter 5

Modifying the Code

Adaptations and modifications to the program are one of the most important things this software needs to take into account. This means that the code needs to be documented fully and the ways to modify it take into account the user may not understand CUDA that well or at all. To solve this problem all of the CUDA specific memory allocation has been taken care of and all that is left is straightforward copy and paste of existing working code. Another concept is to limit the points of modification down to least number possible. The leads to fewer compilation issues which may be unsolvable by a user who is not familiar with the code or CUDA memory allocation.

5.1 How to add to the Init File

As stated before there are two areas in the init file, the global area and the group area. The first is the group area which is the most complicated to add to. This is because the values in this area may be used on the gpu or just in the cpu code.

The first area to modify is to add the variable to the SimParams struct which is contained in the `particles_kernel.cuh` file:

```
struct SimParams{
    float3 colliderPos;
    float  colliderRadius;

    float3 gravity;
    float globalDamping;
    float particleRadius;

    uint3 gridSize;
    uint numCells;
    float3 worldOrigin;
    float3 cellSize;

    uint numBodies;
    uint maxParticlesPerCell;

    float spring;
    float damping;
    float shear;
    float attraction;
    float boundaryDamping;

    kernel kernel_type;
```

```
    Type integration_type;

    uint timestep;
};
```

The global struct should only contain values that are shared by all groups so that the group struct is reduced in size.

The second area to modify is the parser which is contained in `particles_kernel.cuh` which is where main is as well. The function `void set_key_value(group& g, GroupParams& gp, vector<string>& pair, SimParams* m_params)` is a long if-else tree which checks the current input line from the init file. All that needs to be added is another else-if statement checking for the variable name and then set the corresponding variable in one of the structs:

```
else if(key == "Variable Name"){
       struct.variable = value;
}
```

If the new addition is to create some new functionality such as switch between different functions then the variable needs to be handled in a different way. If the value represents some enum such as the kernel or integration type, then that value is used by the cpu in a switch statement to call different CUDA kernels.

The kernel and integration selections are done in seperate functions called `kernel get_kernel_type(string input) and Type get_integration_type(string input)` which do the same thing as the other function but shortens the if statements and cleans things up.

5.2 Data and Arrays

The existing program contained only two data arrays, one for position and the other the velocity. These values are stored for each particle in its corresponding index using float4s which are vectors. The reason for four dimensions is because of memory alignment and the fourth slot being useful.

CUDA requires the programmer to allocate these arrays in a C style on the host before the code is run. This means that there needs to be two pointers, one for the host and the other for the gpu. This simulation requires saving or dumping the data to the hardrive which follows the same functions as initializing or allocating the array:

```
cudaMalloc((void**)&d_gparams_array, 10 * sizeof(GroupParams));
checkCudaErrors(cudaMemcpy(d_gparams_array, gparams_array, 10 sizeof(GroupParams),
                           cudaMemcpyHostToDevice));
```

These two function calls are to allocate and copy the GroupParams array where each index contains the constants for each of the groups. The index into this array is contained in the fourth dimension of the position float4. A list of the existing GPU data pointers are listed below:

m_hPos
m_hVel
m_hForce

m_dPos
m_dVel
m_dForce

To add a new array onto the gpu there are six places to modify in particleSystem.cpp and one in particleSystem.h. The first is to add the two pointers, one for the host and one for the device, into the protected area of particleSystem.h under the corresponding comments. The second is the _initialize(int) method which will allocate the space on the host and the GPU. The first two lines to add are:

```
m_hNewPointer = new float[m_numParticles*4];
memset(m_hNewPointer, 0, m_numParticles*4*sizeof(float));
```

This will allocate the host storage so if and when the array is copied back to be dumped it has a place to go. To allocate the GPU space the following line is added:

```
allocateArray((void **)&m_dNewPointer, memSize);
```

Make sure to use the pointer that contains a 'd' which corresponds with the device. Much like C garbage collection is a must so the new arrays must be deleted and freed by adding two lines to the _finalize() function:

```
delete [] m_hNewPointer;
freeArray(m_dPointer);
```

The method dumpParticles() handles copying the various arrays of interest from the GPU to the host memory space and then written to the hard drive. A new text file needs to be opened, the array copied to is host pointer and the data is written to the corresponding file:

```
ofstream newFile;
newFile.open("newdata.txt", fstream::app);
copyArrayFromDevice(m_hNew, 0, &m_cuda_posvbo_resource, sizeof(float)*4*count);

for each particle:
posFile   << stepNum * stepSize << setw(12) << m_hNew[i*4] << setw(12) <<
       m_hNew[i*4+1] << setw(12) << m_hNew[i*4+2] << setw(12) << m_hNew[i*4+3] <<
       setw(12);
```

This data can be written at any interval, every dt or every hundred, and can be shut off all together to increase real time graphical performance.

The two functions that set and copy these arrays also need to be modified to include cases for each type. These types are enums that are declared in the particleSystem class, so new ones need to be added for each one. These functions are not necessary, but they do clean up the code somewhat:

```
case NEWARRAY:
     hdata = m_hNewArray;
     ddata = m_dNewArray;
     cuda_vbo_resource = m_cuda_posvbo_resource;
     break;
```

These functions can be used for arrays that are not numParticles long because they require a length variable.

If the array to be added needs to be initialized then its best to either create a new function that does that or use the initGrid() function which initializes the particles position and velocity based on the init file. This must occur before the copy to the device because the host array is modified first then copied. The GPU data cannot be modified by the host in any way, except with the thrust library which is slow.

5.3 Integration Methods

The existing software uses a simple euler method for position integration at the start of each time step or frame. This is adequate as a starting point but more accurate methods may be needed later. To accomplish this the user can now specify an integration type in the init file and the program will run that function. The user will add a new integration method within the integration files and will add a call to that function inside the existing switch statement. This provides as little modification as possible.

5.3.1 Adding Integration Methods

This is much like previous ways to add new functionality. The first step is to add the parameter to the initfile, the SimParams struct and the set key value function in particles.cpp. This is the same as any of the other new global params. A new enum should be added to the list in Type which is contained along with all of the other enums which is in particles_kernel.cuh.

The next step is to add to the switch statement contained in the integrate() function which uses the enum that was set in the previous step. A new case should be added and a call to the kernel function for the new integration type. All of the other integration methods are contained in integration.cuh. If a new parameter is needed that is not contained in params then it must be sent at the function call or be added to the SimParams struct using the method previously stated.

5.4 SPH Kernel Functions

The PySPH simulation makes uses of various different kernel function for the user to use. These include the following:

```
CUBIC_SPLINE
GAUSSIAN
QUINTIC_SPLINE
WENDLAND_QUINTIC_SPLINE
HARMONIC
M6_SPINE
W8
W10
REPULSIVE
POLY6
```

The currently implemented kernel function and gradient are the Gaussian because it does not require a neighbor list. The current algorithm for sorting as mentioned before groups particles by location respective to the cell they are in or near. This is calculated each frame of the simulation because it is a particle simulation so a particles position can change dramatically. For the future this will change because it will be simulating more solid objects so a particles position doesnt change radically with respect to its neighbors.

The existing Gaussian function written in Cython is over complicated and highly inefficient. This is due to the fact that the author decided to create an array, initialize it, fill it with the values and then copy this array into the original array that was already in memory and will be used by other functions. This is a common theme throughout all of the code so most of the time spent was on modifying the code to make it

5.4.1 Adding SPH Kernel Methods

The first step is virtually identical to adding an integration method. All of the enums are already defined in the particles_kernel.cuh and there is currently no switch statement to switch between what kernel function is called.

The second step is to add the CPU and equivalent GPU kernel function. particleSystem.h contains the switch statements for the two types, gradient and function, where the function are contained in cpukernel.h and .cpp. The declarations are contained in the .h and the definition in the .cpp. The GPU kernel functions reside in sphkernel.cuh and .cu which are __device__ functions since they will be called in particles_kernel_impl.cuh. If these function need to be called by the host then use __global__ instead of __device__.

5.5 GPU Kernel Calls

Adding kernel functions is the simplest task to modify this program once the general structure explained in a previous section is understood. The first level is the update() function which goes down the list of functions in the particleSystem.cuh. These include:

```
void integrateSystem(float *pos,
                     float *vel,
                     float deltaTime,
                     uint numParticles);

void calcHash(uint  *gridParticleHash,
              uint  *gridParticleIndex,
              float *pos,
              int    numParticles);

void reorderDataAndFindCellStart(uint  *cellStart,
                                 uint  *cellEnd,
                                 float *sortedPos,
                                 float *sortedVel,
                                 uint  *gridParticleHash,
                                 uint  *gridParticleIndex,
                                 float *oldPos,
                                 float *oldVel,
                                 uint   numParticles,
                                 uint   numCells);

void collide(float *newVel,
             float *newForce,
             float *sortedPos,
             float *sortedVel,
             uint  *gridParticleIndex,
             uint  *cellStart,
             uint  *cellEnd,
             uint   numParticles,
             uint   numCells,
             GroupParams* d_gparams_array);
```

These are the existing functions that call the GPU kernels for every time step or frame. Adding a new kernel is fairly straightforward but there are a few guidlines and rules:

1. The kernels need all of the pointers to data that will be used in the calculation
2. Use the existing calculation for the number of blocks and number of threads
3. Try to avoid using the thrust libraries at all cost, they are quite slow for something of this size.
4. If the existing particle sort method is used then make sure all calculations are done before, during or right after the collide kernel.
5. Objects cannot be passed to a kernel, but a object array pointer can. Objects should be avoided on the GPU.

The structure for a kernel call is as follows:

```
calcHashD<<< numBlocks, numThreads >>>(gridParticleHash,
                                       gridParticleIndex,
                                       (float4 *) pos,
                                       numParticles);
```

This is not much different from a normal function call except for the block and thread count which shouldn't be worried about. If a new kernel needs the group information then remember to send the particle group array with the kernel call.

All of the current kernels reside in particles_kernel_impl.cuh, but new functions are not limited to that. Just remember that the global constants struct called params resides in this file so make sure to make another copy or send it directly with the call.

Chapter 6

Future Work

The current version of this program provides a good base for future work. Most of the more difficult modifications that require knowledge of CUDA as well as months of reading the code have been taken care of. This leaves a good base for someone who doesn't need to know the details of CUDA or the program to add new features or kernels.

Most of the important existing code as been commented, this excludes the opengl code which requires no modifications, and all of the new code has extensive commenting. The existing code contained no commenting including no doxigen style function comments or explanation of how the algorithms work. These have been added to the particle system and CUDA code.

Future work would include continuing to translate the existing cython code from PySPH starting at the pressure, position and density functions which make use of the kernel functions. These are quite complex mathematically which requires reviewing literature and other code to fully understand these functions.

References

[1] Cramer Nick, Mircea Theoderescu. Analysis of the Viability of Smoothed Particle Hydrodynamics for the Study of Micro-Scale Systems

[2] NVIDIA. *CUDA Programming Guide 3.1*, 2009. URL http://www.nvidia.com/object/cuda_get.html.

[3] Øystein Eklund Krog. GPU-based Real-Time Particle Hydrodynamics. Technical Report 6/2010, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science, Trondheim, 2010.

[4] Ramachandran, P., and Kaushik, C. “Pysph: A python framework for smoothed particle hydrodynamics”.