SPH using the NVIDIA CUDA Particle Simulation Example

Quick Start Guide

Rough Draft

Josh Lohse

Introduction

This program provides the basis for a Smoothed Particle Hydrodynamics simulation specifically for viscoelestic material and polymers. This guide is intended to give a brief overview of how to set up the environment and how to modify the code. If more details on existing functions are needed consult the code or the larger document.

Requirements

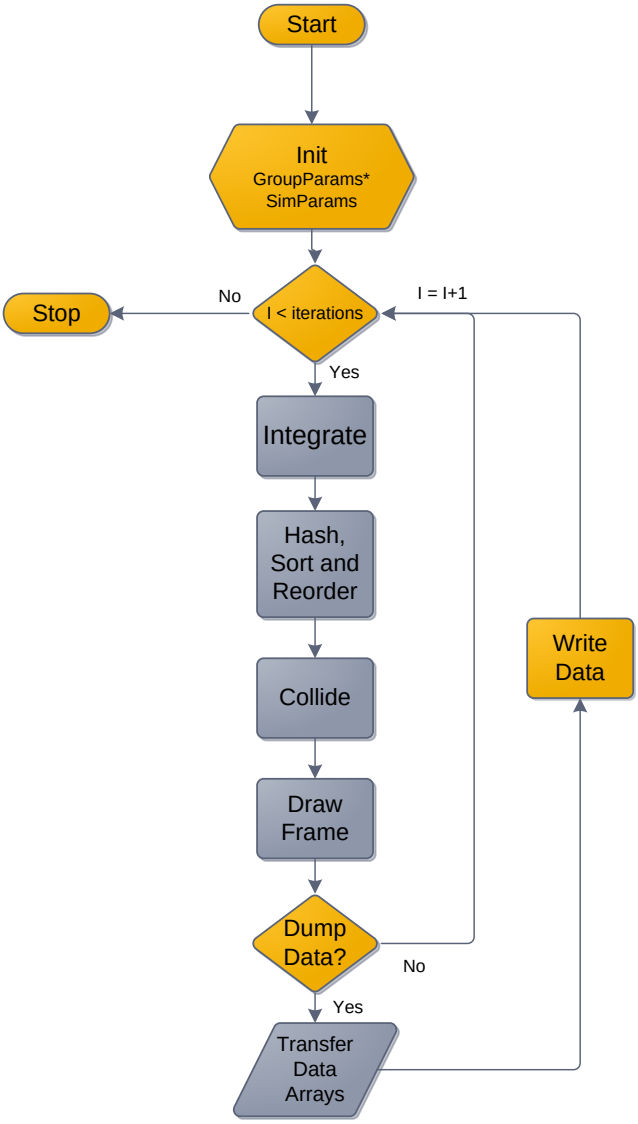
CUDA is a relatively new programming and hardware model where the initial release was in 2007 alongside the 8xxx series of GPUs. As such this program requires some features that were added much later to the hardware so only 9xxx and newer cards are supported. The program has a check for the hardware so it will not run if it doesn't detect a compatible card.

The version of the CUDA sdk that was used at the time of development was version 6.0 where 6.5 was released near completion. Newer version can be used because they only add features rather than remove them. Older versions are not supported because new functions have been added as well as atomic operations and direct memory access to GPU global memory.

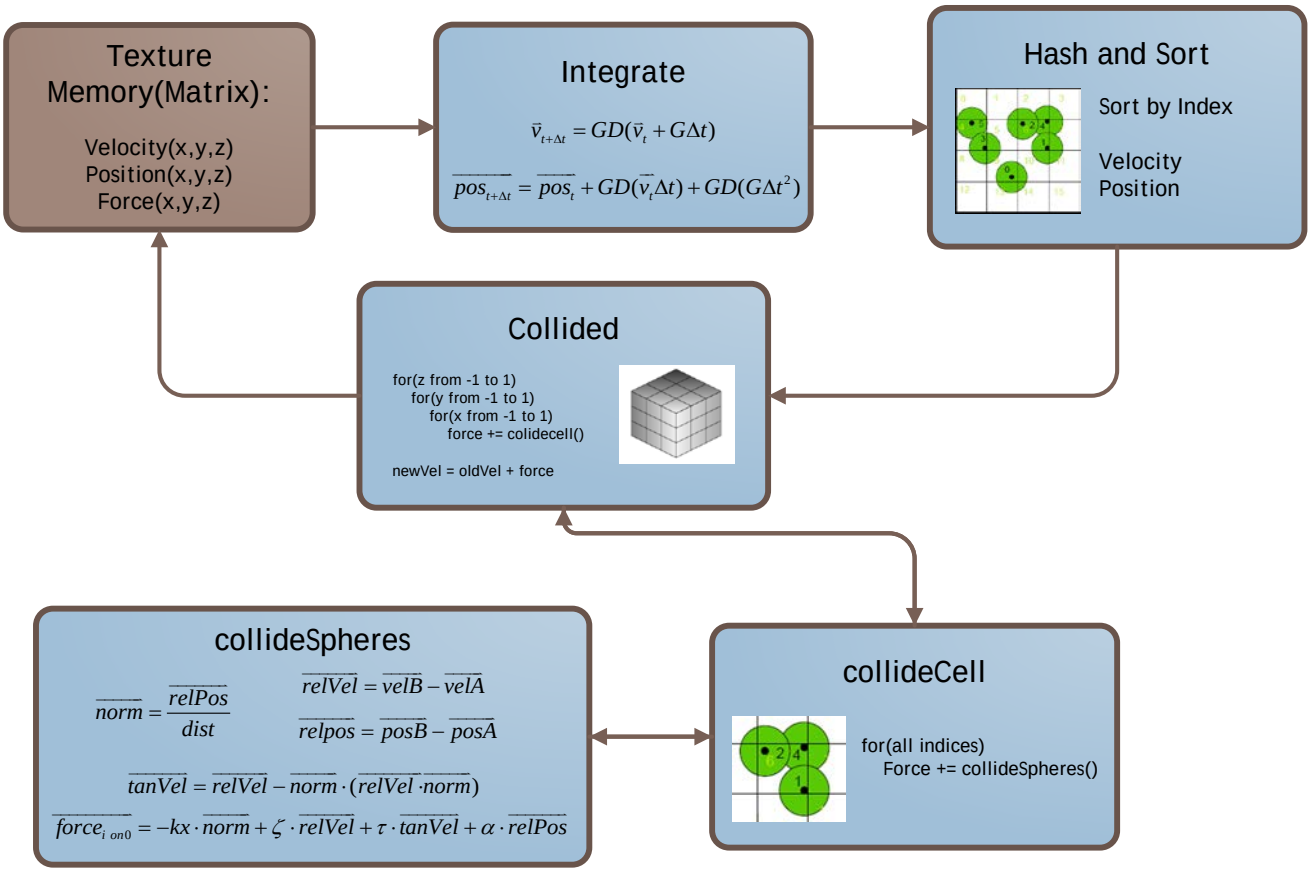
The NVIDIA sdk requires a version of visual studio to be installed, every version from 2007 are supported. It is preference but 2010 seems to have the best support at the time of writing this document but may have changed with 6.5.

General Flow Chart

The flow chart shown below shows the general function call flow as well as where the major data structures and their pointers are held:



Data Flow



These are flow charts for the existing structure, so anytime a new kernel function is added it should be in the center portion.

A word about pointers

CUDA may say it supports C++ but in reality it doesn't. It does support classes on the GPU but it is wasteful and unnecessary. This means that where ever possible focus on the calculation with just simple function calls and try and eliminate any complex data structures.

CUDA requires the programmer to allocate arrays just like in C. The only difference is that it must be done in the host code before the first kernel call that requires that allocation. So this means that you must create a pointer in the host code, use cudaMalloc using that pointer and then send this pointer to the kernel function that needs it. If the program requires this array to be passed back to the host then two pointers are required, one for the host and the other for the device. Most pointers are stored in the ParticlesSystem class as will be explained later.

CUDA Data Type and Functions

One of the nice features of CUDA are the included math functions as well as the 3d and 4d vectors. CUDA by default support 4d(float4) vectors since it is more efficient in memory, but this program also had 3d vectors denoted by float3. These also have overloaded operations associated with them including all of the basic operators as well as dot products.

To make a float4 or float3 just call make_float3(x,y,z) or make_float4(x,y,z,w). The fourth dimension in the program is used as an index into the GroupParams array, so basically it's a group identifier for each particle.

Modifications

The goal of this work was to make modifications as simple as possible. This means the number of points in the code to add new definitions is kept to the minimum. A chart of the general modifications is listed below:

Adding global constants:

partinit.txt

add constant to file following
direction at top

particles_kernel.cuh

add constant to SimParams

particles.cpp

add else if statement to
set_key_value() at bottom

if value is float use atof, else use atoi

Adding integration methods:

partinit.txt

add statement to file following directions at top

particles_kernel.cuh

if new sph kernel or integration method then add to corresponding enum

add enum variable to SimParams

particles.cpp

add else if statement to set_key_value() at bottom

add to switch statement in get_integration_type()

particleSystem_cuda.cu

add to switch statement in integrate() including kernel call following the euler case

integration.cuh

add kernel function, if new array is needed follow adding array directions below

Adding sph kernel:

partinit.txt

add statement to file following directions at top

particles_kernel.cuh

if new sph kernel or integration method then add to corresponding enum

add enum variable to SimParams

particles.cpp

add else if statement to set_key_value() at bottom

add to switch statement in get_kernel_type()

cpukernel.h & cpukernel.cpp

add function declaration to .h and function definition to .cpp, code should be identical to gpu code

particleSystem.h

add to switch statements in run_gradient() and run_function() with functions being ones in cpukernel.h

gradient function should be same for all types

gpu kernel function calls

a switch statement that uses the passed enum needs to call the corresponding gpu function in sphkernel.cuh

Adding group constants:

partinit.txt

add constant to file following direction at top

particles_kernel.cuh

add constant to GroupParams

particles.cpp

add else if statement to set_key_value() at bottom

if value is float use atof, else use atoi

Adding new Arrays:

particleSystem.h

add pointer to protected area under GPU data using the convention m_d*xxx*

if array is copied back to host add a host pointer to protected area using the convention m_h*xxx*

add enum of array name in ParticleArray

particleSystem.cpp

In *_initialize()*:

allocate host storage if needed using m_hPos as example

allocate device storage using allocateArray()

In *_finalize()*

delete host array if it exists

use freeArray() to free GPU array

In *update()*

pass gpu pointer to function in particleSystem_cuda.cu

In *dumpParticles()*

if array needs to be dumped to a file then add the following:

new ofstream object
copyArrayFromDevice()
new formatted ofstream redirection in for loop
close the ofstream

In *getArray()*

add case for new array using enum added

In *setArray()*

add case if array is initialized to something other than zero, otherwise no case is needed. Initilization of array should be done before or in the reset()

Adding new CUDA kernel functions

Kernel functions are the easiest thing to add to the program since the least amount of steps or knowledge is required. These functions can reside in new files which should be .cuh because that is the only thing that seems to work. Make sure you create the file with visual studio because there are some properties it changes to make sure the C++ compiler ignores these files.

The current program structure has the functions in particleSystem_cuda.cu call the CUDA kernels, so if you need a new kernel function you need to add another function in this file as well. These functions are then called by the update() function in particleSystem.h. The two required lines are:

```
uint numThreads, numBlocks;
computeGridSize(numParticles, 256, numBlocks, numThreads);
```

These lines calculate the block size as well as the total number of threads which is based directly with the number of particles being simulated. These values are used in the kernel function call as shown below:

```
yourKernelFunction<<< numBlocks, numThreads >>>((float4 *) pos,
                                                (float4 *) vel,
                                                deltaTime,
                                                numParticles);
```

Make sure to pass all of the pointers to arrays you will need in the calculation. The cast (float4 *) is needed for any array that contains vectors of the type float4.