Fluid Simulation on Mobile GPU

Report submitted by

Rohan Das
2013CS10251

Laxman Jangley
2013CS10234

under the guidance of

Prof. Subodh Kumar
Dept. of CSE

in fulfilment of the requirements for the degree of

B. Tech. in Computer Science and Engineering

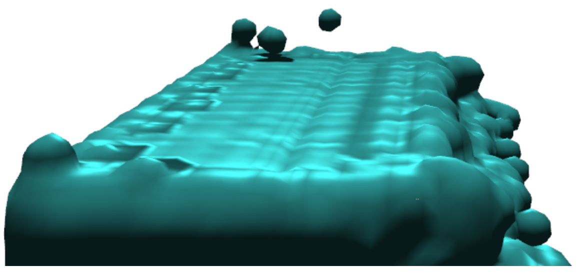
INDIAN INSTITUTE OF TECHNOLOGY DELHI

November 2016

ABSTRACT

In recent years, mobile devices have become increasingly more computationally powerful, while lowering cost at the same time. Given today's exploding popularity of mobile devices, in this project we investigate the possibilities for realtime fluid simulation on mobile GPUs.

At the present time, physically based simulation is widely used in computer graphics to produce animations. Even the complex motion of fluid can be generated by simulation. There is a need for faster simulation in real-time applications, such as games, virtual surgery and so on. However, the computational burden of fluid simulation is high, especially when we simulate free surface flow, and so it is difficult to apply fluid simulation to real-time applications. Moreover, mobile GPUs operate using significantly lesser energy than their desktop counterparts and have a low number of shader cores and low clock speeds.



Contents

Chapter 1

INTRODUCTION

In this project our approach involves 3 sections. First being physics, which is done using Smooth Particle Hydrodynamics (SPH), second comes fluid surface triangulation using Marching Cubes and finally we draw the surface triangulation mesh using the OpenGL Rasterizer. Our implementation is completely done in a GPGPU fashion in OpenGL..

1.1 Smooth Particle Hydrodynamics

SPH solves the motion of fluids by modelling them as discrete particles. Each of these particles has a spatial distance ("smoothing length") associated with it over which their properties are "smoothed" using a kernel function.

$$A(\vec{r}) = \textstyle\sum_j m_j A_j/\rho_j W(\vec{r} - \vec{r_j}, h)$$

$$\triangledown A(\vec{r}) = \textstyle\sum_j m_j A_j/\rho_j \triangledown W(\vec{r} - \vec{r_j}, h)$$

SPH uses an equation of state, in our case the Navier Stokes equation, and an integrator to solve for the system. We place the particles into bins for our fixed-radius (spatial distance) nearest neighbour search. The subsequent step is to compute pressures at each particle and thus compute force for each particle using the equation of state. Then, we integrate to update velocities and positions of the particles.

1.2 Marching Cubes

Marching Cubes is an algorithm used to extract polygonal meshes from iso-surfaces of 3-D scalar fields. The algorithm proceeds through each of the cells and determines the polygon(s) representing the shape of the iso-surface inside the cell.

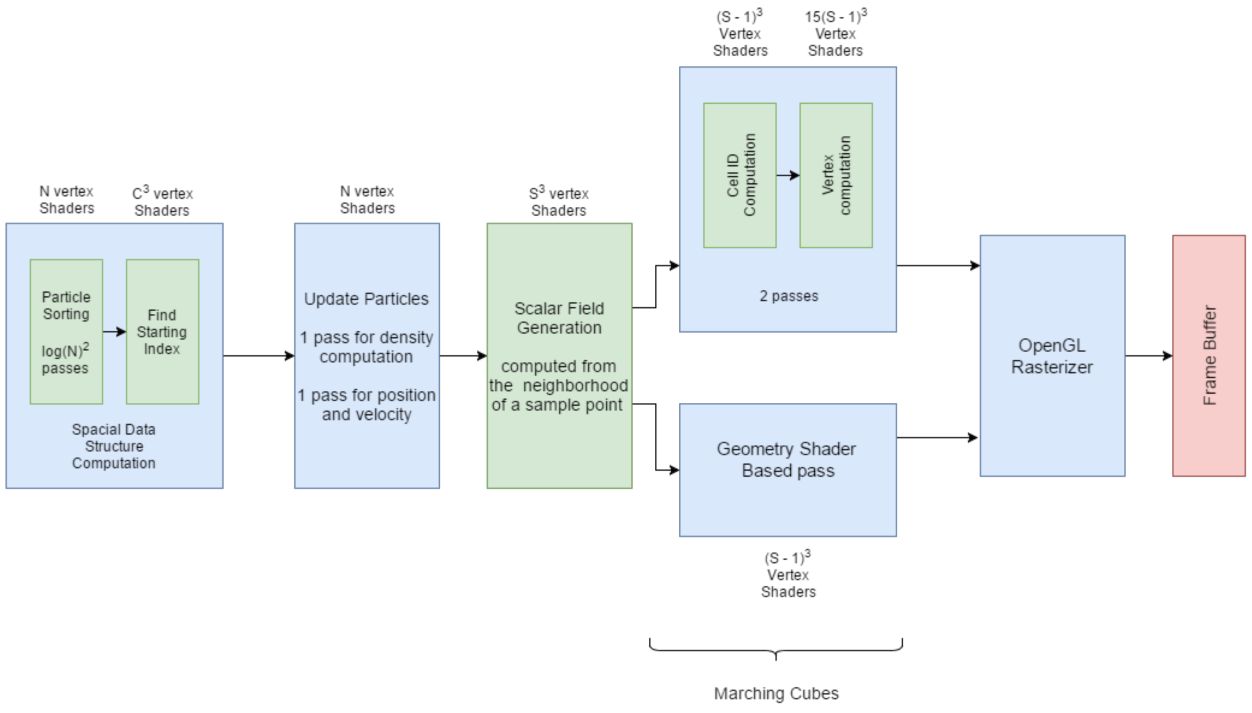
The polygons are determined by comparing scalar field values with the iso-value at each of the cell corners and setting the cell's bit to 1 if it is higher than the iso-value and 0 otherwise ,thus forming an 8 bit index to the type of cell through a precomputed lookup table for each of the $2^8 = 256$ configurations of the cell.

Chapter 2

THE PIPELINE

In simple terms the algorithm being followed is:

- Insert particles into bins.
- Compute pressure and forces.
- Update particles.
- Generate scalar field.
- Generate polygonal mesh.



The above figure gives an overview of our pipeline. Here the variables for the system are:

- **Particles :** The number of particles in the simulation (N).
- **Space Cells :** The space is divided into C uniform sections per axis which form C^3 cells that act as bins for our neighbourhood queries.
- **Sample Points :** Each axis is sampled at S points placed uniformly

2.1 Spatial Data Structure Computation

A spatial data structure is required for efficient fixed radius nearest neighbour search, as a naive O(n^2) implementation require a lot of redundant checking to find the particles that actually fall within the spatial length of the kernel used for SPH.

To create the spatial data structure we first define an ordering on the space cells and then sort the vertices according to the space cell using bitonic sort. This operation leaves us with an array with contiguous blocks corresponding to the individual cells.

Bitonic sort requires O($log^2(N)$) passes for each of the N vertex shaders for the particles.

In order to traverse cells we define an array indexed according to the cells which contains the beginning of the block with the lowest cell index greater than the current cell. A contiguous block's length can be found by subtracting it's entry from the next entry in this array. Thus with these two structures we have random access to the particles in a given cell.

The second array is constructed using C^3 vertex shaders, one for each cell, where each shader does a binary search in the sorted particles array.

2.1.1 Spatial query

Given a query point we first compute the nearest space cell corner and then we simply consider the 8 cells surrounding that corner to be the neighborhood of the query point.

2.2 Updating Particles

Particles are updated in 2 passes. The first is used to compute density and the second integrates velocities and acceleration to find the new positions and velocities respectively.

Density is updated by first doing a neighbourhood query on the particle of interest and then accumulating the sums corresponding to the SPH equations for each particle in the neighbourhood.

Each of these passes is done using N vertex shaders (one for each particle).

2.3 Scalar Field Generation

This step involves calculating the new scalar field values at each of the sample points. We run a vertex shader for each sample point (S^3 vertex shaders in all), each shader performs a neighbourhood query and accumulates the scalar field values and normals.

2.4 Marching Cubes surface triangulation

In this stage we apply the actual marching cubes algorithm to the discretely sampled scalar field we generated in the previous pass.

- **Classical Implementation :** In the first pass we run a vertex shader for each unit cubic cell in the sampled scalar field and find out its type ($(S-1)^3$ vertex shaders run in this stage). The second pass is done with rasterizer enabled and for each vertex as a maximum of 5 triangles is possible we run 15 vertex shaders per cell. Note that many vertex shaders will not yield any primitive as most cells will not lead to 5 triangles.
- **Geometry Shader Implementation :** This is exactly similar to the classical implementation except for the fact that now depending on the type of a cell a geometry shader is called which outputs the required number of primitives.

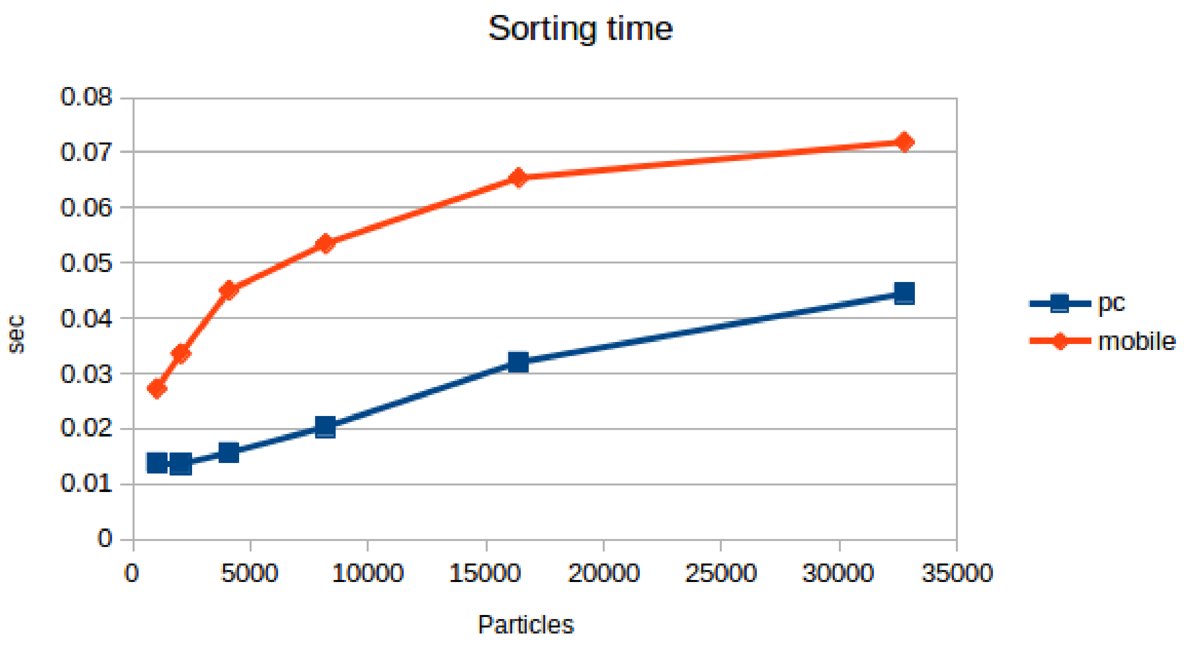
Chapter 3

Benchmarking

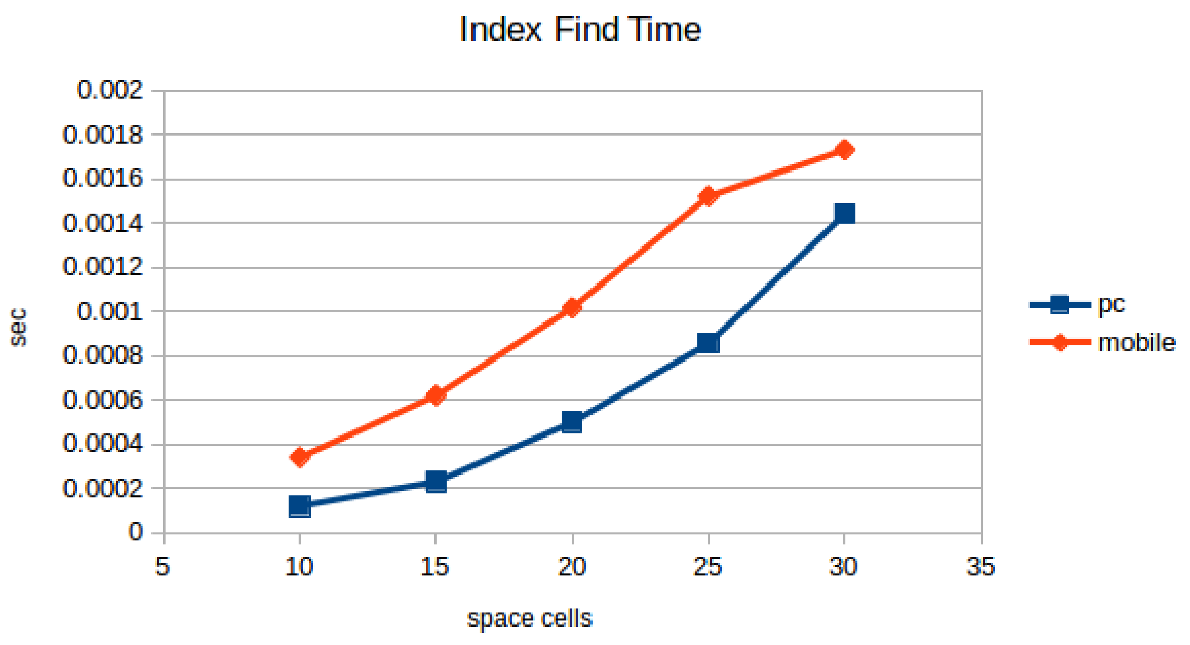
The program was tested on one handheld device GPU, the Mali T880 (12 shader cores running at 600MHz) and on a laptop GPU Intel HD 4400 (20 shader cores running at 200-1000MHz).

3.1 Construction of Spatial Data Structure

- **Sorting** The sorting stage is one of the major bottlenecks of our program. As seen from the graph given the time taken to sort increases with the number of particles. The performance is significantly better in the laptop GPU compared to the mobile GPU as it is more powerful.

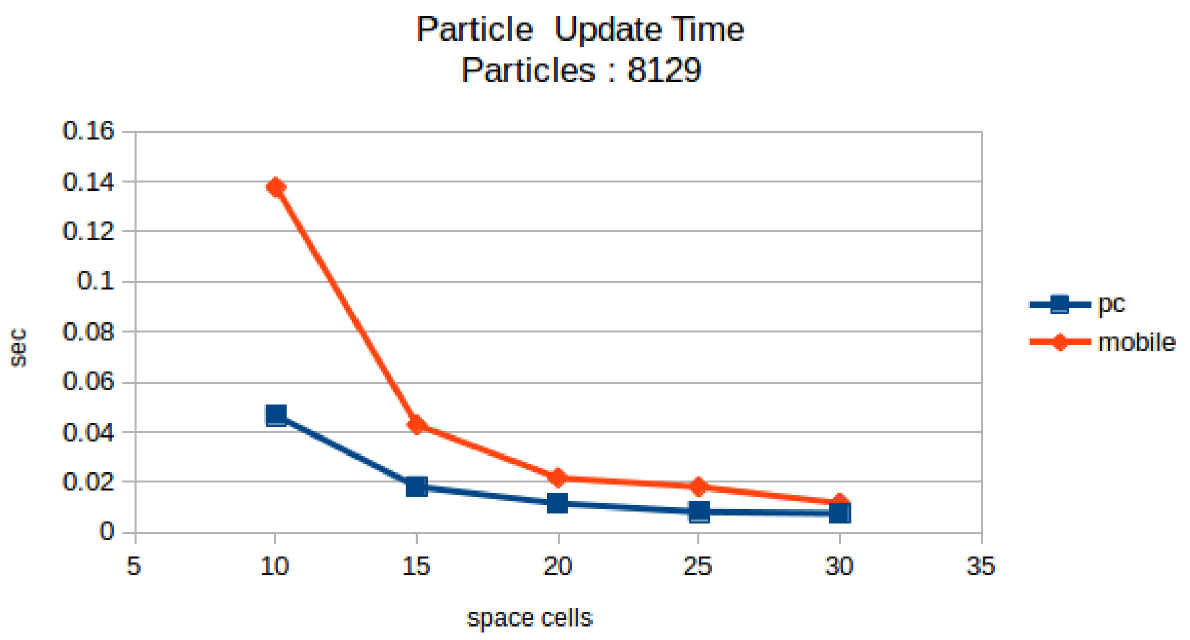


- **Index Searching** This step is a minimal cost step as binary search is employed by us in this stage. There is little difference between the laptop GPU and mobile GPU in terms of performance.

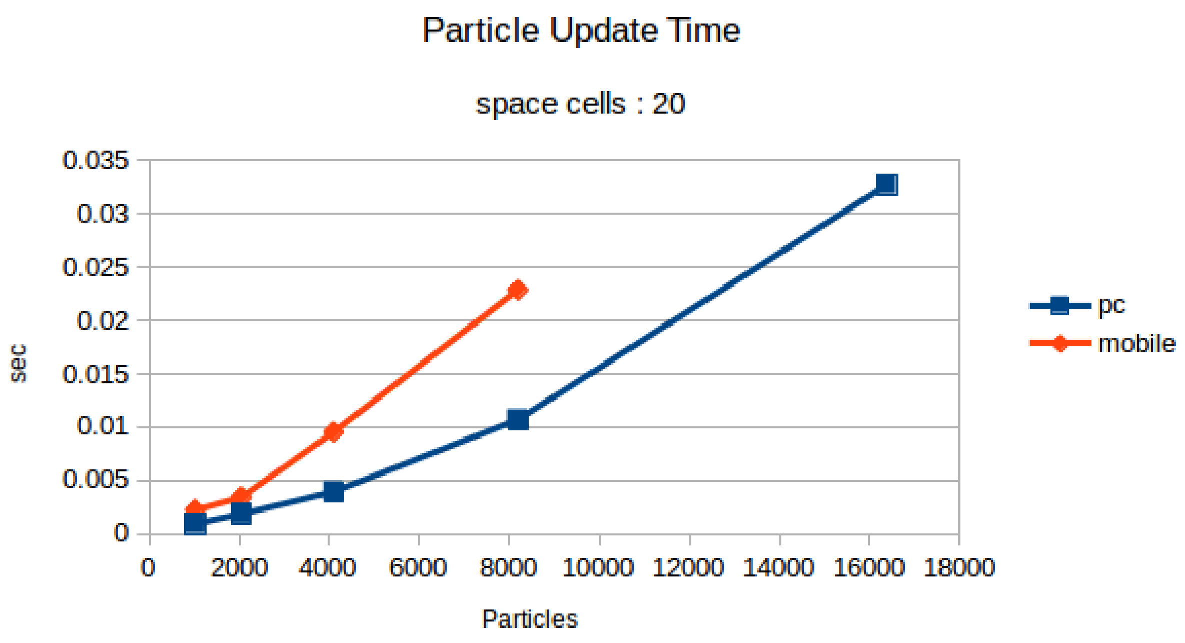


3.2 Updating Particles

The time taken to update particles decreases with increase in the number of space cells per axis. This is the expected trend as increasing the number of space cells would reduce the number of particles returned by the neighbourhood query. But this is a double edged sword as increasing the number of space cells would reduce the effect radius of the particles' interaction leading to inaccuracies in our model.

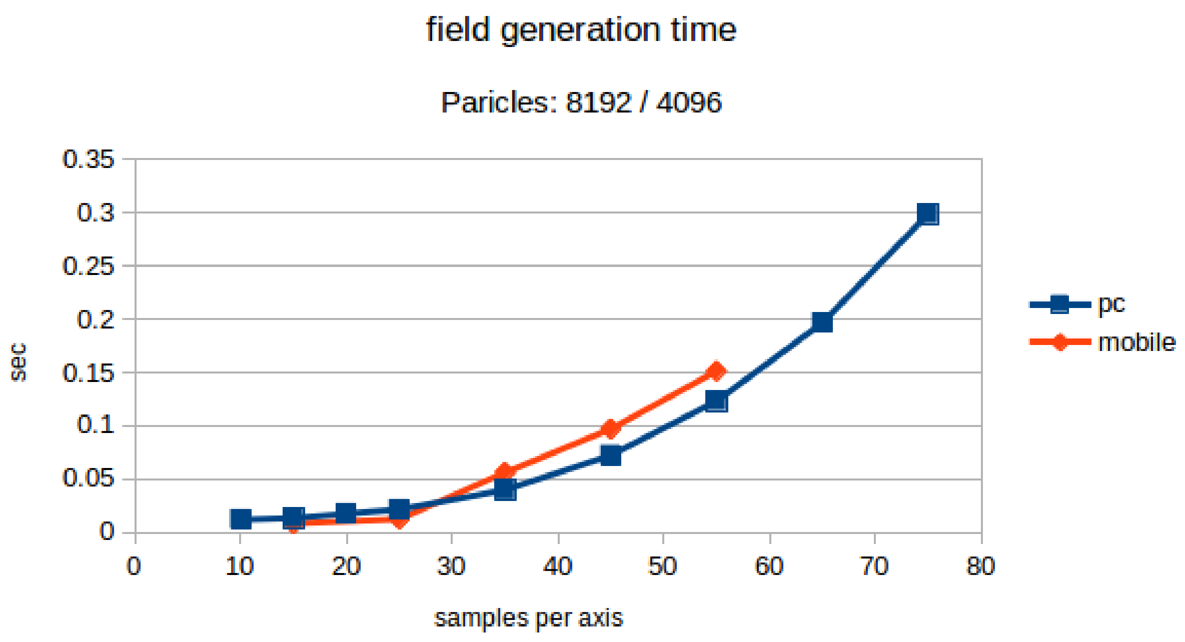


The time taken to update increases with the number of particles.



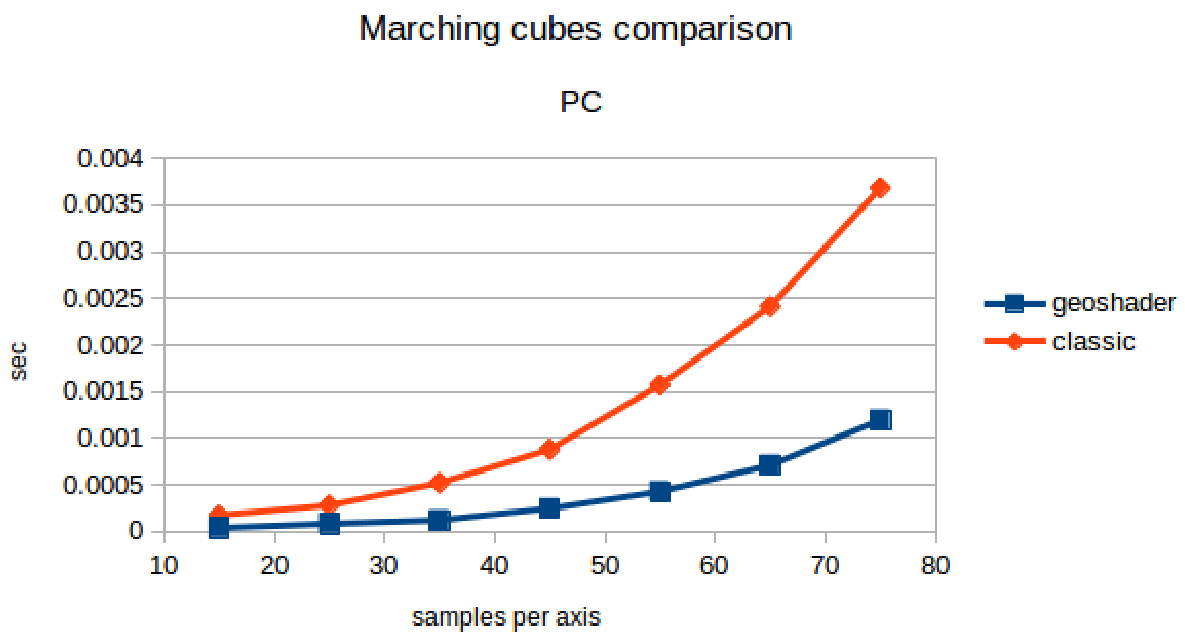
3.3 Scalar Field Generation

Scalar field generation is the primary bottleneck in our application. This is to be expected as we require running S^3 vertex shaders for each of the sampled grid points and do a neighbourhood query for the grid point.



3.4 Marching Cubes

The geometry shader implementation is faster and more scalable than the vertex shader implementation as we run only one geometry shader per cell and it emits the required number of primitives whereas the vertex shader implementation requires 15 vertex shaders to be executed per vertex.



Chapter 4

Conclusion

In this project we have implemented fully GPU based fluid simulation on a mobile device, but it should be noted that these devices are not suited for rendering realtime high resolution Marching Cubes as it requires high resolution sampling as well as sorting. In the current implementation we sort the whole array of particles, without any consideration for it's initial state. If we can improve this by using the fact that a particle's displacement won't be too big, we can speed the sorting stage by a lot. We also do not look at the state of the scalar field before updating it so we ignore any physical constraints that may ease our calculation of the same.

Chapter 5

References

- Benz, W., SPH, in ‘The Numerical Modeling of Nonlinear Stellar Pulsations’ ed. J. R. Buchler, Kluwer (1990)
- Monaghan, J. J., Particle Methods for Hydrodynamics, Comp. Phys. Reports (1985)
- Monaghan, J. J., SPH, ARA&A (1992)
- Lucy (1977) and Gingold & Monaghan (1977)
- Initial code samples taken from Mali-SDK tutorial