

UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO VICENTE PIRES DOS REIS FILHO

GPU HistoPyramid Based Fluid Simulation and Rendering

NITERÓI

2014

UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO VICENTE PIRES DOS REIS FILHO

GPU HistoPyramid Based Fluid Simulation and Rendering

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science.
Topic Area: Visual Computing.

Advisor:

Prof. D.Sc. Marcos de Oliveira Lage Ferreira

NITERÓI

2014

JOÃO VICENTE PIRES DOS REIS FILHO

GPU HistoPyramid Based Fluid Simulation and Rendering

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science.
Topic Area: Visual Computing.

Approved by:

Prof. D.Sc. Marcos de Oliveira Lage Ferreira / IC-UFF
(Advisor)

Prof. D.Sc. Leandro Augusto Frata Fernandes / IC-UFF

Prof. D.Sc. Waldemar Celes / Departamento de
Informática - PUC-Rio

Niterói, December 1st, 2014.

To my parents, who have given me endless support and inspiration.

Acknowledgments

First and foremost I would like to thank my advisor, Marcos de Oliveira Lage Ferreira, who has supported me throughout the development of this work, for all his advices, patience, commitment and for being ready to help at all times. Thank you for always being so encouraging and friendly.

To Esteban Walter Gonzalez Clua, for all the opportunities, invaluable advices and specially for helping me getting started on my Masters degree. Thank you for believing in my work and for always being there to help and offer support during this whole journey.

To all the other researchers and faculty members who have shared their time, experience and knowledge in order to contribute in the best way possible to the education of the students.

To Teresa Cancela, Viviane Aceti and all the university staff, for being so kind and helpful.

To CNPq, for providing a scholarship.

To my parents, for their unconditional love and for being there for me in every moment of my life: from the happiest to the hardest ones. Thanks for guiding me so well through the challenges of life.

To all my family and friends, who have encouraged me to pursue my objectives.

To God, for giving me the opportunity to come to this world and the curiosity needed to learn and improve myself. And also for making me healthy and strong enough to keep seeking what I believe.

Resumo

Algoritmos que mapeiam cada elemento de entrada para um único elemento de saída podem facilmente utilizar todo o massivo poder de processamento paralelo oferecido pelas GPUs atuais. Por outro lado, algoritmos que precisam seletivamente descartar entradas ou mapeá-las para um ou mais elementos de saída, processo que é chamado de compactação e expansão de *stream*, são difíceis de serem implementados na GPU. Neste trabalho, uma estrutura de dados chamada *Histogram Pyramid* (também conhecida como *HistoPyramid*) será apresentada como uma solução eficiente para implementar esse tipo de reorganização de dados na GPU.

O *Marching Cubes* é um algoritmo tradicional de computação gráfica que pode se beneficiar dessa estrutura de dados. Ele é usado para extrair uma malha de triângulos a partir de uma função implícita. Ele pode ser usado, por exemplo, como uma maneira rápida para se visualizar simulações de fluidos baseadas no método de *Smoothed Particle Hydrodynamics*. Este trabalho vai explorar uma implementação do algoritmo *Marching Cubes* baseada em *HistoPyramid* e também compará-la com a implementação equivalente utilizando o *geometry shader*. Um *solver* de fluidos que roda completamente na GPU também será apresentado. Ele usa a *HistoPyramid* para deixar partículas vizinhas ordenadas de maneira próxima na memória, o que é essencial para uma boa performance de simulação. Ambos os algoritmos podem ser utilizados em conjunto, formando uma biblioteca completa para simulação e renderização de fluidos em GPU. Técnicas avançadas do OpenGL 4.3, como os *compute shaders*, foram usadas para maximizar a performance e a flexibilidade dos algoritmos desenvolvidos.

Palavras-chave: *Histogram Pyramid*, *Marching Cubes*, *Smoothed Particle Hydrodynamics*.

Abstract

Algorithms that map each input element to a unique corresponding output can easily harness all the massive parallel processing power offered by today's GPUs. On the other hand, algorithms that need to selectively discard inputs or map them to one or more output elements, what is called stream compaction and expansion, are difficult to implement on the GPU. In this work, a data structure called Histogram Pyramid (also known as HistoPyramid) will be presented as a solution to implement this kind of data re-organization efficiently on the GPU.

The Marching Cubes is a traditional computer graphics algorithm that can benefit from this data structure. It is used to extract a triangle mesh from an implicit function. It can be used, for instance, as a fast way to visualize fluid simulations based on the Smoothed Particle Hydrodynamics method. This work will explore a HistoPyramid implementation of the Marching Cubes algorithm and also compare it with its geometry shader based counterpart. A fluid solver that runs fully on the GPU will also be presented. It also takes advantage of the HistoPyramid in order to sort neighbor particles close together in memory, which is essential for good simulation performance. Both algorithms can be used in conjunction, forming a complete GPU accelerated framework to simulate and render fluids. Advanced OpenGL 4.3 techniques, like compute shaders, were used to maximize the performance and flexibility of the algorithms developed.

Keywords: Histogram Pyramid, Marching Cubes, Smoothed Particle Hydrodynamics.

Contents

Acronyms	viii
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Contributions	3
1.2 Dissertation Structure	4
2 Related Work	5
2.1 Smoothed Particle Hydrodynamics on GPUs	6
2.2 Fluid Rendering	7
2.3 The HistoPyramid Data Structure	8
3 The HistoPyramid Data Structure	9
3.1 Data Structure Description	10
3.2 Data Structure Initialization	12
3.3 Building Phase	14
3.4 Traversal Phase	15
3.5 Implementation Details	19
4 Fluid Simulation	21
4.1 Smoothed Particle Hydrodynamics	21

4.2	Implementation Details	26
4.3	GPU Implementation	29
5	Fluid Rendering	33
5.1	The Marching Cubes Algorithm	33
5.2	GPU Implementation	37
5.2.1	Geometry Shader Implementation	38
5.2.2	HistoPyramid Based Implementation	39
6	Results and Discussion	43
6.1	Methodology of the Experiments	43
6.2	Experiments	44
6.2.1	SPH Comparison: CPU vs GPU Implementation	45
6.2.2	Marching Cubes Comparison: Geometry Shader vs HistoPyramid Implementation	46
6.2.3	Combined Experiment	48
7	Conclusions and Future Work	50
7.1	Future Work	51
	Bibliography	53

Acronyms

CFD	:	Computational Fluid Dynamics
CPU	:	Central Processing Unit
CUDA	:	Compute Unified Device Architecture (Nvidia)
GLSL	:	OpenGL Shading Language
GPGPU	:	General-Purpose Computation on Graphics Processing Units
GPU	:	Graphics Processing Unit
HP	:	Histogram Pyramid
MC	:	Marching Cubes
OpenGL	:	OpenGL Graphics Library
SPH	:	Smoothed Particle Hydrodynamics
Texel	:	Texture Element

List of Figures

1.1	The classic dam break flow simulation.	2
3.1	Sample input and output streams.	11
3.2	Sample 1D HistoPyramid.	12
4.1	A dam break animation.	23
4.2	Comparison between the ideal gas equation and the Tait equation.	24
4.3	Surface tension influence in a zero gravity environment.	25
4.4	Uniform grid used to accelerate the neighbour particle lookups.	26
5.1	A 2D scalar field with an isoline highlighted in red.	34
5.2	Comparison of the midpoint method with linear interpolation.	36
5.3	Marching Cubes unique cases [21].	36
5.4	A mesh created by the Marching Cubes algorithm of a zero gravity fluid drop.	37
6.1	Three dam break simulations performed by the SPH algorithm using a varying number of particles.	45
6.2	Mesher generated by the Marching Cubes algorithm with varying grid resolutions.	47
6.3	A plot with pairs of parameters that can be used to create animations that run consistently at 30 frames per second.	49

List of Tables

6.1	SPH comparison: average execution times for various particle counts over 2,000 simulation steps.	46
6.2	Marching Cubes comparison: average execution times for various grid resolutions over 2,000 rendered frames. The draw indirect technique was enabled for the HistoPyramid version.	48
6.3	HistoPyramid Marching Cubes comparison: average execution times for various grid resolutions over 2,000 rendered frames. Using standard read-back (RB) vs the draw indirect technique (DI).	48
6.4	Configurations of the simulation and rendering HistoPyramid algorithms that were able to achieve a smooth 30 frames per second animation.	48

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is an increasingly popular research area in the last few decades. This is due to the many applications it has in the most diverse fields, such as engineering, medicine, illustration and entertainment. For instance, among other things, it is useful for simulating airflow around an aircraft, blood flow in the human body and to create realistic water bodies both in movies and interactively in games.

One of the biggest challenges of this area comes from the complexity of the physical behavior of fluids. To model convincingly the way a fluid moves and reacts to external forces in the real world requires complex mathematics and plenty of processing power. This fact severely limits the quality and scale of real-time interactive fluid simulations.

In the last years, graphics processing units (GPUs) have gotten progressively more flexible and programmable. This has allowed the processing of a wide range of applications to be accelerated by them. But despite current generation GPUs offering extremely high performance, often in the range of a few teraflops (trillion floating point operations per second), they have a unique massively parallel architecture, which makes certain algorithms really hard to implement on them in an efficient manner.

Some architecture limitations include the need to use specialized parallel data structures and deal with possible race conditions, reduced global memory throughput if coalescent access patterns are not used while reading or writing data, and also weak divergent control flow performance [32].

On the other hand, there are the so called embarrassingly parallel algorithms, which are more straightforward to implement on massively parallel processors. Examples of such algorithms are most graphics and image processing tasks (GPUs were designed for them after all). Grid based fluid simulations are also embarrassingly parallel. But in spite

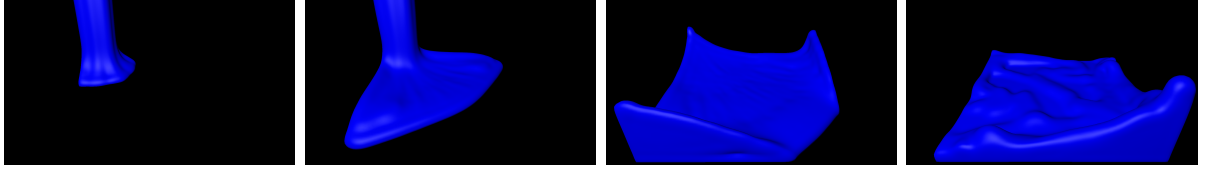


Figure 1.1: The classic dam break flow simulation. Time passes from left to right.

of being a better fit for the current GPUs architecture, grid based methods need huge amounts of memory to simulate large fluid bodies and in addition they are only able to handle a limited region of space. In other words, they only operate on a fixed domain.

Particle based simulations are another way to model fluid flows. They generally employ a method called Smoothed Particle Hydrodynamics (SPH) [24], use less memory and allow the fluid to exist in an infinite domain, where it can move freely to anywhere in space. The SPH method also trivially adapts to topological changes in the fluid surface. Despite these advantages, they are very hard to implement efficiently on GPUs because they exhibit non-coalescent memory access patterns, specially when particles need to search for their neighbours. To alleviate this situation, particles need to be spatially sorted into memory. A lot of research have been done in this area in the last few years and there are some successful SPH implementations on the GPU [14, 8, 20]. In this work, a particle sorting solution using the HistoPyramid data structure will be proposed.

It is also important to consider how to render the simulation data in a realistic, convincing and precise way. There are many ways to do it: some methods are screen space based, some are mesh based, among others. For this work the Marching Cubes algorithm [22], which is a mesh based solution, was chosen. Figure 1.1 illustrates how the visualization looks like. This method has some advantages. For instance, the generated mesh represents the geometry of the fluid surface and it also can be post-processed in many ways.

Both the particle sorting and the Marching Cubes algorithms can be seen as stream compaction and expansion problems, which are hard to implement well on the GPU. Graphics processors usually process data in arrays or grids, and the output count is exactly the same as the input count. However, in this kind of problems, each input element can be either discarded or used to calculate one or more outputs. This is clearly a hard type of task to parallelize, as a naive implementation will almost always not be able to evenly balance the computing load among the GPU threads, which is detrimental performance wise.

This work will explore a data structure called Histogram Pyramid (also known as HistoPyramid) and the associated parallel algorithms used to build and traverse it in order to implement algorithms that need this kind of data re-organization efficiently on the GPU. In particular, both a particle sorting algorithm and a Marching Cubes version that take advantage of the HistoPyramid will be presented. This will lead to a complete fluid framework with both simulation and rendering being executed fully on the GPU.

Furthermore, aiming to maximize the performance and flexibility of the algorithms developed, advanced OpenGL 4.3 techniques like compute shaders and the draw indirect idiom were used.

1.1 Contributions

The main contributions of the present work are:

- A compute shader based 3D HistoPyramid implementation. It can process input elements from a 3D array of data.
- A GPU bucket sort algorithm based on the 3D HistoPyramid data structure and tailored for sorting SPH neighbouring particles into GPU memory to allow fast coalesced memory access.
- A weakly compressible SPH simulator that runs fully on the GPU, using compute shaders and the aforementioned bucket sort.
- A Marching Cubes implementation based on the 3D HistoPyramid.
- A comparison between the HistoPyramid based Marching Cubes and a more traditional GPU version of it that uses the geometry shader to perform the stream compaction and expansion.
- An application that puts together the SPH simulator and the Marching Cubes renderer, providing a complete real-time fluid simulator.

1.2 Dissertation Structure

The remainder of this dissertation is structured as follows. Chapter 2 provides an overview of previous and related works. Chapter 3 will describe the HistoPyramid data structure and the algorithms needed to build and traverse it. Next, Chapter 4 will talk about fluid simulation, explaining in detail the SPH method that was used in this work and also how it was implemented on the GPU. After that, Chapter 5 will explore fluid rendering techniques, giving special attention to the Marching Cubes algorithm. Chapter 6 discusses the results obtained in this work. Finally, Chapter 7 concludes this dissertation and also proposes future improvements to this work.

Chapter 2

Related Work

The Navier-Stokes equations are the basis used to describe the dynamics of fluids. There are two approaches to solve them and track the motion of a fluid: the Eulerian viewpoint and the Lagrangian viewpoint [2].

The Eulerian approach is grid based and looks at fixed points in space to see how measurements of fluid quantities, such as density and velocity, at those points change in time. Foster and Metaxas [11] employed this approach to successfully model the motion of a hot, turbulent gas. Later, Stam [34] proposed an unconditionally stable model which still produced complex fluid-like flows even when large simulation time steps were used. His work was an important step towards real-time simulation of fluids. More recently, Chentanez and Müller [3] presented a new Eulerian fluid simulation method that uses a special kind of grid, called the restricted tall cell grid, to allow real-time simulations of large scale three dimensional liquids. Chentanez and Müller [4] also presented a GPU friendly purely Eulerian liquid simulator that is able to conserve mass locally and globally.

The Lagrangian approach is particle based and it is an irregular discretization of the continuum. It measures fluid quantities at particle locations and the particles freely move with the fluid flow. Examples of Lagrangian methods are the Moving-Particle Semi-Implicit (MPS) [19, 30] and the Smoothed Particle Hydrodynamics (SPH), which was developed by Gingold and Monaghan [13] and independently by Lucy [23] mainly for astronomy simulations of large gas dynamics. The SPH method uses smoothing kernels for the Navier-Stokes discretization. It has been used in the most diverse research fields.

For instance, Müller et al. [26] created an interactive blood simulation for virtual surgery to allow medical training. In fact, Müller et al. [24] were the first to develop a fast enough SPH solver that was suitable to be used in interactive systems and allow user

manipulation. Many of the interactive SPH implementations that followed were based on their work, including the model used in the present work.

In the field of superficial flows over terrains, Kipfer and Westermann [18] employed the SPH model to create physically realistic simulations of rivers in real-time, allowing it to be used in games and virtual reality environments. Regarding fluid-solid interactions, Solenthaler et al. [33] described a SPH based model to simulate liquids and deformable solids and their interactions. The model described by their work is also useful to simulate multi-phase flows (liquid-solid), as was also demonstrated by Hu and Adams [16].

One challenging problem that particle based fluid simulations have to deal with is high compressibility and oscillations, which are undesired properties that look unnatural and particularly strange in liquid simulations. Becker and Teschner [1] proposed to use the Tait equation to make a weakly compressible fluid formulation with very low density fluctuations. They have also addressed surface tension, which plays an important role to handle small details in the simulations.

2.1 Smoothed Particle Hydrodynamics on GPUs

Recently, some researchers started to study ways to implement the SPH algorithm on GPUs. The simulation can be made many times faster by exploring their massive computational power. However, due to their unique architecture, this is not a straightforward task. The algorithm to find the neighbours of a given particle, which is necessary to calculate the forces acting on it, is specially hard to implement on the GPU. Harada et al. [14] were one of the first able to implement a SPH simulation running fully on the GPU.

Silva Junior et al. [9, 8] developed a heterogeneous system to leverage both GPU and multi-core CPU processing power for real-time fluid and rigid body simulation supporting two-way interactions. Krog [20] investigated how to create an interactive snow avalanche simulation, using the Nvidia CUDA technology to program the GPU. He started with a simple SPH model, which was suitable for simulating low-viscosity incompressible Newtonian fluids (e.g. water-like fluids). Then he made a more complex SPH model, which is more accurate and has support for simulating non-Newtonian fluids with viscosities determined by empirical models. Using this model he was able to simulate flowing avalanches on a terrain at interactive frame rates.

2.2 Fluid Rendering

Modeling the physics that govern the movement of a fluid is only half of what is necessary to make a complete simulation. Another important aspect is how to create images from the abstract mathematical representation of the fluid. This is addressed by the research of fluid rendering algorithms, which calculate how light interacts with the fluid surface and interior.

There are a number of ways to render a fluid. Some are more realistic and show a greater amount of details; others have faster performance and thus are more suitable for real-time applications. This work is based on a traditional and straightforward approach called isosurface extraction. A GPU version of the Marching Cubes algorithm is used to create a 3D mesh representing the fluid surface.

The Marching Cubes algorithm was originally developed by Lorensen and Cline [22] in 1987 as a way to create triangle models of constant density surfaces from 3D medical data. This algorithm is widely used until today and many extensions and improvements have been proposed by several authors. Newman and Yi [27] have done a survey of the most representative derived works. Custódio et al. [7] implemented a variant of Marching Cubes that tries to generate topologically correct manifold meshes for any input data.

Müller et al. [25] proposed an alternative to render particle-based fluid models called screen space meshes. This method allows the generation and rendering of surfaces defined by the boundary of a 3D point cloud. It first creates a 2D screen space triangle mesh that later is transformed back to 3D world space for the computation of lighting and shading effects. It is worth noting that this algorithm only generates visible surfaces and offers view-dependent level of detail for free.

More recently, van der Laan et al. [36] presented a screen space rendering approach that is not based on polygonization and as such avoids tessellation artifacts. It alleviates sudden changes in curvature between the particles to create a continuous and smooth surface and prevent it from looking “blobby” or jelly-like. Also, like the screen space meshes method, it only renders visible surfaces and has inherent view-dependent level of detail.

Other techniques that are suitable for fluid visualization are Surface Raycasting [17], Volume Rendering [12] and Ray-Tracing [29].

2.3 The HistoPyramid Data Structure

The HistoPyramid has been successfully employed to implement many stream compaction and expansion problems on the GPU. Dyken et al. [10] presented a reformulation of the Marching Cubes algorithm as a data compaction and expansion process and took advantage of the HistoPyramid to make an implementation that outperformed all other known GPU-based isosurface extraction algorithms in direct rendering for sparse or large volumes at the time their work was published. It only required OpenGL 2.0 and in order to do so it used a flat 3D layout to store the HistoPyramid data, i.e., a large tiled 2D texture to represent the 3D scalar field and output count data. The present work Marching Cubes implementation is based on Dyken's approach. However, it utilizes modern OpenGL 4.3 features (e.g. compute shaders) to implement a truly 3D HistoPyramid.

Ziegler et al. [38] showed how a HistoPyramid can be utilized as an implicit indexing data structure, allowing them to convert a sparse 3D volume into a point cloud entirely on the graphics hardware. Their method can be used to create visual effects, such as particle explosions of arbitrary geometry models. It is also able to accelerate feature detection, pixel classification and binning, and enable high-speed sparse matrix compression.

As far as the author knows, there is no article in the literature discussing how to apply the HistoPyramid in the context of sorting neighbour particles for the SPH algorithm.

Chapter 3

The HistoPyramid Data Structure

Some algorithms process each input element always generating a unique corresponding output. However, in many other cases, the ability to selectively discard inputs or map them to one or more output elements is required. This type of processing is called stream compaction and expansion and is difficult to implement properly on the GPU, as a naive implementation will almost always not be able to evenly balance the computing load among the GPU threads and so will not take advantage of its full processing power.

The HistoPyramid [10], short for Histogram Pyramid, is a data structure that allows efficient implementation of stream compaction and expansion algorithms on parallel architectures. Many problems may be modeled in this way and can benefit from this approach, as discussed in the related works section. In this work, the capabilities of this data structure will be leveraged to implement both an efficient GPU particle sorting method for a fluid simulator and also the Marching Cubes polygonization algorithm that is used to create a visualization of the fluid surface.

Instead of iterating through the input elements and discarding them or generating their corresponding output straight away, an algorithm must be split in two main execution phases in order to be able to use the HistoPyramid. After the data structure is allocated appropriately in GPU memory, the first phase kicks in and fills it using a predicate function to determine each input element multiplicity, i.e., how many outputs each input will generate. After that, a parallel reduce operation is performed and calculates the total number of outputs. Then, the second phase allocates one GPU thread per output element. Each thread traverses the HistoPyramid to find out for what input element and for which of its outputs it is responsible for. Then it can solve the original problem for that particular item and write the result in an appropriate location of the GPU memory.

3.1 Data Structure Description

The HistoPyramid is a stack of textures that looks a lot like a standard mipmap pyramid [37] and also exhibits similar memory access patterns, thus taking full advantage of all GPU optimizations related to mipmapping, such as the dedicated texture caches.

The array of input elements, also known as the input stream, may be an 1D, 2D or 3D array. The texture that sits at the base level of the HistoPyramid must have the same number of dimensions as the input stream and the length of each of its sides must be at least as large as the same side is in the input array. In addition, the length of all of its sides must be equal and also a power of two. Input arrays with asymmetric or non power of two side lengths require appropriate padding to be applied (often it is enough for the predicate function to return zero for the output count of the extra non-existent elements). The texel count of the base level represents the maximum number of input elements that a given HistoPyramid can handle. The value contained in each texel of the base level encodes the output count of a given input element.

Similarly to the layout of a traditional mipmap stack, at each subsequent level of the HistoPyramid the texture dimensions are halved until the top of the stack is reached. There resides a texture with a single texel, which is called the top element. The value stored in the top element is the final output stream length, i.e., the total number of elements contained in the output stream.

Note that it is actually possible to process a 3D array of input elements with a 2D HistoPyramid by using what is called a flat 3D layout [15], i.e., 3D data laid out as slices tiled in a 2D texture. However, this approach adds extra burden and complexity to the implementation, as extra code is necessary to remap the texture coordinates. Hence, in spite of being a useful technique for older hardware, it is not really needed anymore. OpenGL 4.3 hardware and newer offers not only compute shaders which can natively dispatch 3D work groups to the GPU but also the ability to perform arbitrary image store operations that allow shaders to directly write data to a specific texel of a 3D texture [6]. These modern OpenGL features allow a more straightforward implementation of a truly 3D HistoPyramid.

Another interesting layout modification is called the 2D vec4-HistoPyramid [38]. Instead of using a single channel texture and storing only one value per texel, a four channel RGBA texture is used to store four values per texel. This halves the texture sizes along both dimensions, allowing bigger HistoPyramids to be built under the same hardware

limitations (each GPU has a maximum texture size limit). It also cuts texture fetches by a factor of 4 during the HistoPyramid processing and on top of that, given the fact that graphics hardware is extremely optimized to fetch RGBA values, it generally improves the performance of the algorithm.

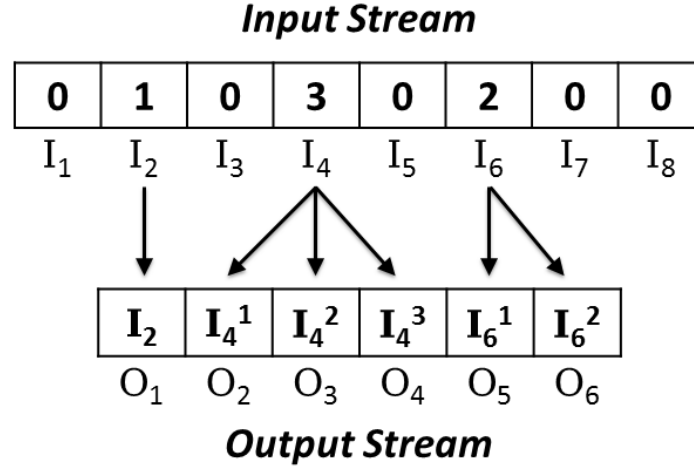


Figure 3.1: Sample input and output streams.

Figure 3.1 shows an example of input stream and also the corresponding output stream it generates after processed. It can be observed that the first element of this input stream allocates zero outputs for itself. It means this element is discarded, or in other words, that stream compaction will be performed. The second element allocates exactly one output, which means that stream passthrough will be performed and the stream size will not be affected by this particular case. On the other hand, the fourth input element allocates three outputs, thus performing stream expansion.

Looking by the side of the output stream, after the HistoPyramid traversal is performed, some conclusions can be reached. For instance, the source used to calculate the first output element is the second input element. Other observation is that the fifth output element is the first output generated by the sixth input element; and the sixth output element comes from the second output created by the sixth input element.

Figure 3.2 shows the HistoPyramid that was built and used while processing the example above. Each value of the base level represents the output count of a given input element. At each new level, the texture size is halved and the output counts are summed. This process continues until the top level is reached. This last level only contains one element which is the total output count of the algorithm. Or in other words, it is the size that the output stream will have.

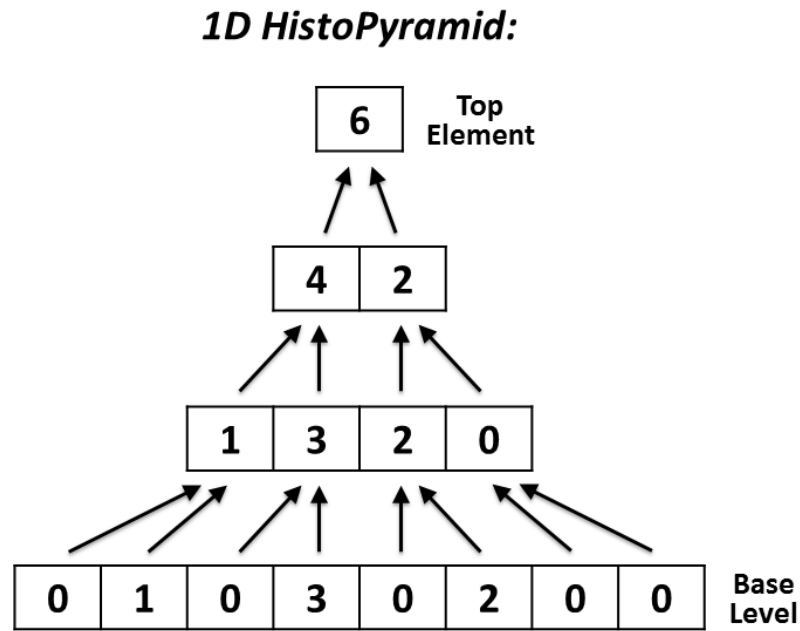


Figure 3.2: Sample 1D HistoPyramid.

Note that in this simplistic 1D example, only two elements of the previous levels are summed to generate an element of a subsequent level, which only halves the texture size. In the 2D case, a 2×2 square is considered (4 elements) and both the texture width and height are halved. So the next level of a 2D pyramid actually has only one fourth of the number of elements of the previous one. Finally, in the 3D case, a $2 \times 2 \times 2$ cube is taken into account (8 elements) and the depth of the texture is also halved. Therefore, each subsequent level of a 3D pyramid only has one eighth of the element count of its previous level.

3.2 Data Structure Initialization

Algorithm 1 describes the steps that are necessary to initialize the HistoPyramid data structure on GPU memory. First, all the compute shaders that are required to build and traverse the data structure should be compiled and loaded into GPU memory. Next, the largest length of all of the input stream dimensions must be determined. Then, based on this, a length is chosen for the side of the base level of the HistoPyramid. It must be a power of two and ensure that all of the input stream elements will fit into the data structure. After that, it is possible to calculate how many levels the HistoPyramid will have by using a simple logarithmic formula.

Algorithm 1 Data Structure Initialization

```

1: Load the required compute shaders on the GPU;
2:  $largestDimension \leftarrow \max(inputWidth, inputHeight, inputDepth) - 1$ ;
3:  $hpBaseSideLength \leftarrow 1$ ;
4: while  $hpBaseSideLength < largestDimension$  do
5:    $hpBaseSideLength \leftarrow hpBaseSideLength \times 2$ ;
6: end while
7:  $hpLevels \leftarrow (\ln(hpBaseSideLength) \div \ln(2)) + 1$ ;
8: Allocate the HistoPyramid texture on the GPU memory;
   Tex Dimensions =  $hpBaseSideLength \times hpBaseSideLength \times hpBaseSideLength$ ;
   Tex Mipmap Levels =  $hpLevels$ ;
9: Fill the HistoPyramid texture base level with zeros; {using a compute shader.}
10:  $\Rightarrow$  The HistoPyramid is now ready to be used.

```

Finally, the HistoPyramid texture can be allocated on the GPU memory with the appropriate dimensions and the given number of mipmap levels. Details about how the texture allocation was implemented are given in Section 3.5.

The last step required to run before the HistoPyramid is ready to be used is the execution of a compute shader to write zeros to the whole base level of the texture. Due to the requirements of the data structure (all sides being of equal length and also a power of two length), the number of entries in the HistoPyramid may be greater than the number of input elements. In this case, as was noted before, it is enough to apply proper padding, which generally means making the predicate function return zero for the output count of the extra non-existent elements. However, for performance reasons, the predicate function is not even run for the extra elements at the building phase. So, in order to everything work correctly, the texture must be entirely filled with zeros from the very beginning. This manual step is necessary because usually there are no guarantees about the memory returned by the graphics driver and it may contain random data from previously used memory.

It is possible to calculate the fraction of the HistoPyramid base that is actually being used to process the input stream elements by using the following formula:

$$f = \frac{e}{m},$$

where f is the used fraction, e is the number of elements present in the input stream and

m is the maximum element count supported by the HistoPyramid. In the 3D case, for instance, $m = s^3$, where s is the length of the side of the base level of the texture.

3.3 Building Phase

After the data structure is properly initialized, the first execution phase can run to update its contents and finally build the HistoPyramid. Algorithm 2 describes this process. At first, a compute shader is dispatched to update the base level by performing a simple task in parallel for each input element: applying the predicate function and writing the result at the correct location of the texture.

The predicate function is unique to each stream expansion and compaction problem and it calculates the multiplicity of each input element, i.e., how many output elements each input element will generate.

Next, a compute shader is dispatched for each level starting from level 1 (which is the first above the base level) up to the top level in order to update its contents. To do so, it performs a parallel reduction operation on each previous level. This reduction involves summing the values of 2, 4 or 8 elements of the previous level, according to how many dimensions the HistoPyramid has.

Algorithm 2 HistoPyramid Building Phase

```

1: Dispatch a compute shader to update the HistoPyramid base level;
   {The work done in parallel by the shader is equivalent to the following loop;}
2: for all input elements do
3:   Apply the predicate function to each element to calculate its multiplicity;
4:   Write the result into the appropriate location of the base level of the texture;
5: end for
   {The following steps are executed regularly on the CPU;}
6:  $currentLevel \leftarrow 1$ ;
7: while  $currentLevel < hpLevels$  do
8:   Dispatch a compute shader to update the current level of the texture by performing
      reduction on the previous one;
9:    $currentLevel \leftarrow currentLevel + 1$ ;
10: end while

```

At this point, all the HistoPyramid levels up to the top element are updated and ready to be traversed during the second phase of the algorithm. Also, the value of the top element is now the total output count of the algorithm.

3.4 Traversal Phase

This phase balances the workload evenly among the GPU threads. Given that the total number of output elements is already known, a buffer can be allocated on GPU memory to hold the output stream and it is possible to dispatch a compute shader to execute the desired algorithm with enough threads to fully utilize the GPU processing resources. Algorithm 3 shows in detail all the steps that are performed during the traversal phase.

In order to spawn the appropriate number of compute threads, the top element of the HistoPyramid must be accessed. Traditionally, the CPU needed to read back this value and then issue a rendering command. However, a readback almost always stall the graphics pipeline, forcing the CPU to wait idle while the GPU finishes all its processing. Fortunately, there is a newer approach available in modern graphics cards which yields better performance and completely avoids any readbacks. It is called indirect compute dispatch and allows the GPU to source the parameters used to dispatch the compute shader from a buffer on its own memory. Further details about this technique are explained in Section 3.5.

Each GPU thread will execute the work necessary to generate a single output element. This work begins with the traversal of the HistoPyramid data structure in order to determine for what input element and for which one of its outputs each thread will be responsible for. In other words, traversal is performed once for each output element. It starts by initializing some important variables: *currentLevel* stores at which level of the HistoPyramid the traversal currently is and it is initialized pointing to one level below the top level of the data structure. The top level can be skipped because it always contains only one element and so there is only one way to traverse it. Traversal goes recursively down and finishes when then base level is reached. *currentTexCoord* points to a specific texel in the current level. Conceptually, it starts pointing to the center of the single texel at the top level. However, as the presented algorithm actually begins the traversal in the following level, it is initialized pointing to the middle of the elements one level below the top. In addition, as this work uses integer texture coordinates instead of normalized ones, it is not really possible to point to a location between texels. So what is actually stored is

the integer texture coordinate of the element immediately to the left of where the traversal is targeting. In the 2D and 3D versions, the coordinate stored is from the element immediately to the left, top and front of the real coordinate. The *currentTexCoord* variable keeps being updated during the traversal and, at the end, it points to the input element that generated this output.

Each instance of the compute shader that is executed has a unique linearly increasing index, that is stored in the *outputElementIndex* variable and is used to point to a specific entry in the output stream. The *localIndex* variable is also initialized to this value, but it keeps getting updated during traversal. In the end, in case of stream expansion, it tells which one of the multiple outputs should be processed by this thread. In case of stream passthrough, its value is always 0 (zero).

In the 1D HistoPyramid, at each level visited, two texels must be read. They are labeled *a* and *b* and their values are used to define two ranges named *A* and *B*. And so the traversal may follow two different paths, according to which range the *localIndex* falls into. Then, both the *currentTexCoord* and the *localIndex* variables must be updated conforming to the chosen path. *currentTexCoord* points to the new direction in which the traversal will proceed and is also multiplied by 2, as integer texture coordinates are being used and the next level will have twice as many texels in each dimension. There is an exception: at the base level this multiplication by 2 is not needed. In addition, the value of the start of the chosen range is subtracted from the *localIndex*.

This process continues recursively until the base level is reached. When this happens, the traversal is complete and both *currentTexCoord* and *localIndex* have their final values, so the compute thread can finally perform the calculations specific to the algorithm been executed to generate a given output element and then write it to the appropriate location of the output stream on the GPU memory.

Note that at each level of a 2D or 3D HistoPyramid that is visited, respectively, four or eight texels must be read. In the 2D case the texels are read and labeled in the following order:

a	b
c	d

Accordingly, four ranges *A*, *B*, *C* and *D* are defined. The 3D case is similar, with eight ranges defined and the frontmost layer of texels always read first.

Algorithm 3 HistoPyramid Traversal Phase

```

1:  $hpTopLevel \leftarrow hpLevels - 1$ ;
2: Dispatch a compute shader to generate the output stream;
   {One GPU thread is spawned to process each output element.}
3: for all output elements do
4:    $currentLevel \leftarrow hpTopLevel - 1$ ;
5:    $currentTexCoord \leftarrow 0$ ;
6:    $localIndex \leftarrow outputElementIndex$ ;
7:   while  $currentLevel \geq 0$  do
8:     Fetch texel  $a$  from the HistoPyramid texture at the  $currentLevel$  and using
        $currentTexCoord$  as the texture coordinate;
9:     Fetch texel  $b$  from the HistoPyramid texture at the  $currentLevel$  and using
        $currentTexCoord + 1$  as the texture coordinate;
10:    Set the following ranges:
         $A = [0, a)$ 
         $B = [a, a + b)$ 
11:    if  $localIndex$  falls into  $B$  then
12:       $currentTexCoord \leftarrow currentTexCoord + 1$ ;
13:       $localIndex \leftarrow localIndex - a$ ;
14:    end if
15:    if  $currentLevel \neq 0$  then
16:       $currentTexCoord \leftarrow currentTexCoord \times 2$ ;
17:    end if
18:     $currentLevel \leftarrow currentLevel - 1$ ;
19:  end while
   {The traversal is finished by now and at this point the variable  $currentTexCoord$ 
   references the input element that gave origin to this output. Also, in case of stream
   expansion,  $localIndex$  now tells which of the outputs should be processed by this
   thread.}
20: Calculate this output element according to the algorithm being executed.
21: Write the result to the appropriate location of the output stream array.
22: end for

```

For the sake of clarity, the 2D ranges are laid out as follows:

$$\begin{aligned} A &= [0, a) \\ B &= [a, a + b) \\ C &= [a + b, a + b + c) \\ D &= [a + b + c, a + b + c + d) \end{aligned}$$

Also, as an illustrative example, let's examine step by step the traversal for the third output element from Figure 3.1. By looking at the HistoPyramid in Figure 3.2, it is possible to determine that *currentLevel* is initialized with the value 2, *currentTexCoord* starts as 0 and the *localIndex*, which begins with the same value of the *outputElementIndex*, is equal to 2. The two texels from level 2 form the ranges:

$$\begin{aligned} A &= [0, 4) \\ B &= [4, 6) \end{aligned}$$

The *localIndex* falls into range *A* and the traversal can descend to level 1 and proceed to the left, where the two texels read have values 1 and 3, respectively. They define the following new ranges:

$$\begin{aligned} A &= [0, 1) \\ B &= [1, 4) \end{aligned}$$

At this time, *localIndex* falls into range *B* and the traversal should continue to the right. The *currentTexCoord* is updated to 2 and the *localIndex* is updated to 1. Finally, the traversal reaches the base level. The two texels read (0 and 3) form the ranges:

$$\begin{aligned} A &= (0, 0) \\ B &= [0, 3) \end{aligned}$$

Again, *localIndex* falls into range *B*. The *currentTexCoord* is updated to 3 and *localIndex* finishes with the value 1. Now the traversal is complete and it is possible to infer that the third element from the output stream comes from the second output (as indicated by *localIndex*) of the fourth element from the input stream (as indicated by the *currentTexCoord*). Please note that the actual implementation uses zero based indexes and that is why a *localIndex* value of 1 actually means the second output.

3.5 Implementation Details

In this work, all textures are allocated using the *glTexStorage3D* method (or its 1D or 2D counterparts), which was introduced by OpenGL 4.2 and allows the allocation of the so called immutable textures. This type of texture has all of its storage allocated up-front (including everything needed for all of its mipmap levels). Thus, its format and dimensions are immutable at runtime. Only the contents of the image may be modified. This helps to lessen the graphics driver runtime validation overhead and therefore leads to improved application performance [6]. Its contents must be specified by the *glTexSubImage3D* method, as invoking *glTexImage3D* would allow the user to alter the dimensions or format of the texture, and so it is an illegal operation on this kind of texture.

Another useful technique is the indirect compute dispatch. As discussed before in Section 3.4, it allows the traversal phase to be implemented in a more efficient way by avoiding a CPU readback that stalls the graphics pipeline. Note that apart from the performance gains on dispatching the work for the GPU, everything in shader execution behaves identically as the traditional approach. Instead of reading back the top element of the HistoPyramid by using the *glGetTexImage* method and then calling *glDispatchCompute* passing the number of work groups as a parameter, a slightly different code setup is required. First, a tiny extra buffer called the dispatch indirect buffer must be allocated on GPU memory at program initialization. Also, a small extra compute shader must be executed before the traversal phase. It is dispatched with a single work group containing only one element (i.e. it will only be executed once) and its sole task is to read the top element from the HistoPyramid texture and populate the dispatch indirect buffer directly from the GPU. Now the traversal phase can be dispatched, but this time by calling the *glDispatchComputeIndirect* method. This method does not require the CPU to know in advance the number of work groups that will be executed. At the right time, the GPU itself will source this information from the buffer that resides in its own memory.

There are also indirect draw calls that source their execution parameters from draw indirect buffers. If the output stream will contain vertices from a mesh and the user just wants to render it without saving the data to a buffer, an indirect draw call may be executed in place of the compute shader to perform the traversal phase. The implementation is very similar to the compute case. For instance, the *glDrawArraysIndirect* method should be used as an alternative to *glDrawArrays* [6].

It is possible that multiple shader invocations that were previously dispatched to be queued simultaneously for asynchronous execution on the GPU. In some situations, like in the reduce operation of the HistoPyramid building phase, it is critical to ensure the proper ordering of memory operations with the objective of avoiding data access hazards (e.g. a read after write hazard, where one pass of the algorithm tries to read data of a previous pass that has not yet been written to memory, and it ends up getting incorrect junk data). The *glMemoryBarrier* command was used to define barriers ordering the memory transactions, thus avoiding this kind of problem [6].

Both 1D and 2D HistoPyramids can be easily implemented on older hardware using standard render-to-texture approaches. A 3D HistoPyramid could also be implemented in a similar way. However, its efficiency would be greatly reduced. That is because each slice of the data structure would have to be rendered individually, one at a time, slice by slice, requiring a huge number of render target switches, which is extremely expensive performance-wise. On the other hand, compute shaders along with the image load/store ability of OpenGL 4.2 (which lets shaders read from and write to arbitrary texels of a texture) allow a more efficient way of dispatching this type of computing work to the GPU.

In addition, it is possible to make an implementation using normalized texture coordinates instead of integer coordinates. An integer texture coordinate i can be converted into a normalized texture coordinate c by using the following formula:

$$c = \frac{i + 0.5}{n},$$

where n is the length of the texture in a given axis. The opposite conversion can be done as follows:

$$i = (c \cdot n) - 0.5$$

Chapter 4

Fluid Simulation

The Smoothed Particle Hydrodynamics (SPH) method [24] was the Computational Fluid Dynamics (CFD) model chosen for this work. It is a Lagrangian approach, which means it is based on particles whose locations are used to measure various fluid properties. The particles can move freely with the fluid flow, which brings some advantages to this method such as the ability to naturally deal with topological changes without the need for any kind of special treatment and also the ease to simulate flows with a free boundary. In addition, this method has a plausible adaptation to parallel architectures, as will be discussed in Section 4.3.

4.1 Smoothed Particle Hydrodynamics

This section will explain the basics of the SPH model described by Müller et al. [24], like the discrete representation of the fluid which is approximated by particles and the use of smoothing kernels to interpolate fluid properties across space. Moreover, some extensions and modifications proposed by the current work will be presented.

The SPH method represents the physical model of the fluid through a particle system, which is a finite set of elements. Each particle is located at a discrete position of the space and contain certain properties, like mass, density, pressure and velocity.

The initial condition of the particles (starting value of their properties) may be changed in order to create different flow animations. However, all the particles should have the exact same mass and for the sake of simulation stability, this mass should remain constant throughout the whole simulation. The initial positions depend on the geometric shape of the domain. The particles may be distributed over a regular grid inside the space

of the domain or in random positions. Or even using other sampling techniques.

Boundary conditions are enforced during the simulation. They define how to handle particles that eventually leave the simulation domain. Furthermore, it is important to note that the foundations of SPH lie in interpolation theory. In that sense, it allows quantities of a field that are only defined at discrete positions to be evaluated at any position in space. To do this, the SPH distributes these quantities over a local neighbourhood of defined radius using the so called smoothing kernels. These kernels are mathematical functions employed during the update of the particles properties at every new step of the simulation and they define how each particle interacts with its neighbours.

The fluid motion is described by the SPH through the Navier-Stokes equations [24]:

$$\rho \frac{D\vec{v}}{Dt} = \rho \left(\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \vec{\nabla} \vec{v} \right) = -\vec{\nabla} p + \rho \vec{g} + \mu \Delta \vec{v}, \quad (4.1)$$

where \vec{v} stands for velocity, ρ is the density, p is the pressure, \vec{g} means any external forces (like gravity) and μ is the viscosity of the fluid.

For instance, to calculate the density the following equation is used:

$$\langle \rho(x) \rangle = \sum_{j=1}^N m_j W(x - x_j, h), \quad (4.2)$$

where N is the particle count, m_j is the mass of particle j and W is the smoothing kernel employed by the SPH to perform interpolation. Pressure and viscosity forces are calculated in an analogous manner to the density. However, for each particle property a different smoothing kernel can be chosen, always aiming to improve the simulation stability.

The differential equations that represent the physical laws governing the simulation are solved using the Leapfrog integration scheme [31, 5].

One improvement made over the model described by Müller's work [24] is in how the pressure field is evaluated. Before the forces caused by the pressure gradient can be updated, the pressure at each particle location must be known. Originally, to calculate the pressure first the density at the location of each particle was obtained through equation 4.2 and then the pressure could be computed with a modified version of the ideal gas equation:

$$p = k(\rho - \rho_0), \quad (4.3)$$

where k is a gas constant that depends on the temperature and ρ_0 is the rest density. However, this formulation results in a rather high compressibility of the fluid which regularly creates unrealistic flow animations. In contrast, the current work uses the Tait equation which enforces very low density variation and is efficient to compute. The equation is used as proposed by Becker [1] in his weakly compressible SPH model:

$$p = b \left(\left(\frac{\rho}{\rho_0} \right)^\gamma - 1 \right), \quad (4.4)$$

with $\gamma = 7$ and where b is a pressure constant that governs the relative density fluctuation $\frac{|\rho - \rho_0|}{\rho_0}$. The full procedure to determine values for b is a bit involved and can be found in Becker's paper [1].

Figure 4.1 shows a common CFD test scenario called the dam break. It consists of a column of fluid falling under the force of gravity.

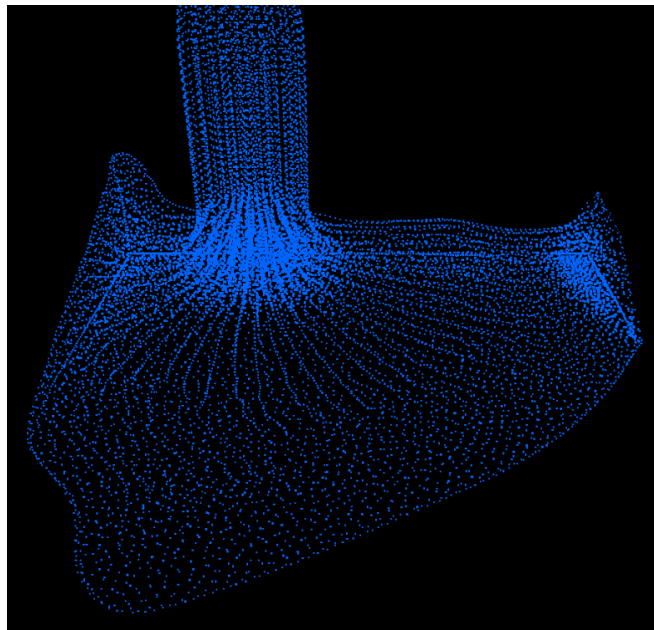


Figure 4.1: A dam break animation.

The dam break is an excellent example to show the benefits of the Tait equation for the pressure update. Figure 4.2 makes a comparison of the animation performed with both equations. It is clear that with the Tait equation the density fluctuations in the fluid are much lower at all times, which yields a way more natural and realistic animation.

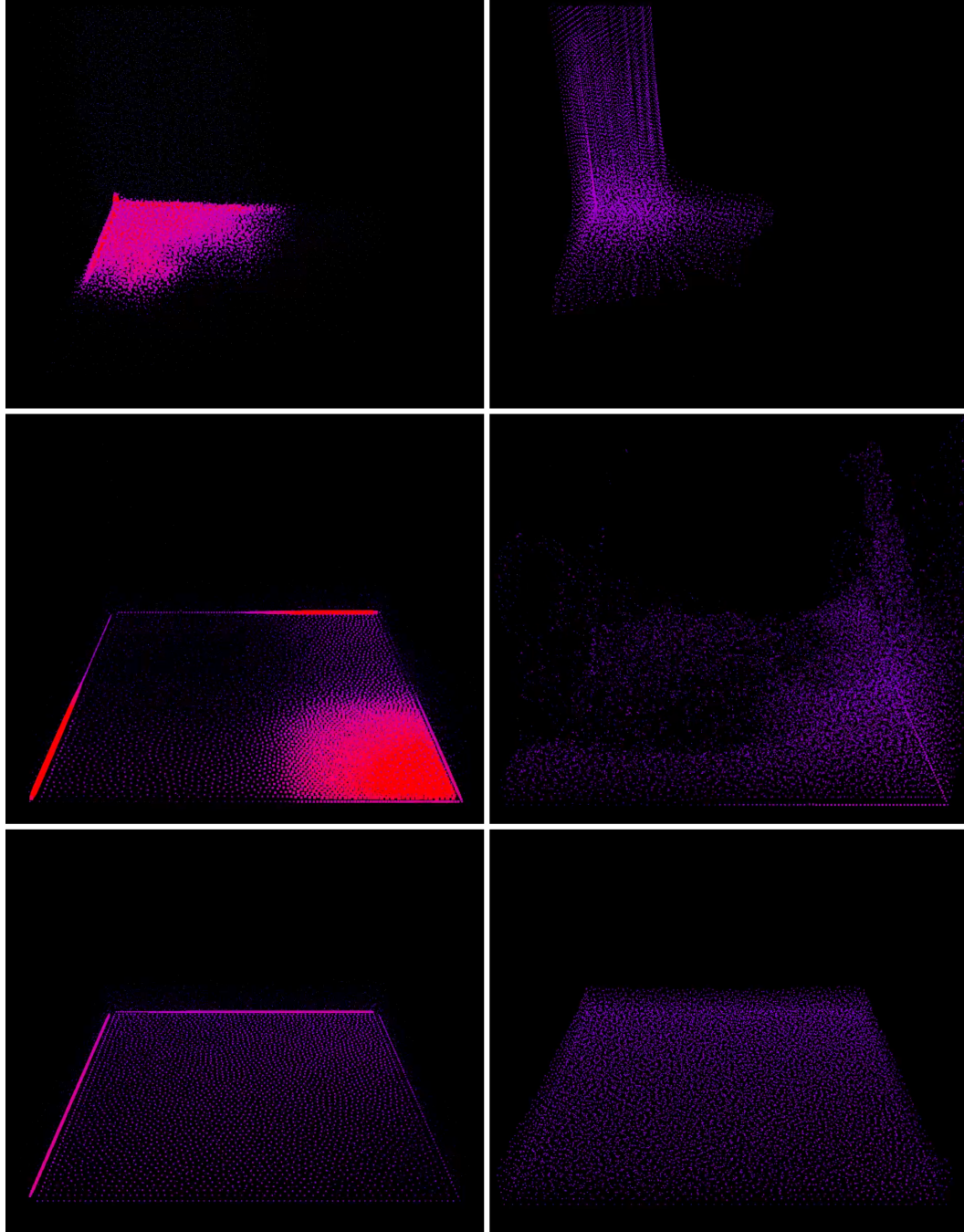


Figure 4.2: Comparison between the ideal gas equation (left column) and the Tait equation (right column). Time advances from top to bottom. The first row depicts the beginning of the flow; the middle one shows the splash and the bottom row illustrates the fluid at rest in the end of the animation. Density is color coded (bright red for high density particles and a pale blue for low density ones). The size of the particle points is also greater in highly compressed areas.

Another improvement over the original model is the addition of the surface tension force. It plays a key role to add finer details to flow animations, especially in thin sheets of fluid. The surface tension approach chosen is also based on Becker's work [1]. It relies on cohesion forces and is inspired by a microscopic view, since on a molecular level the surface tension arises due to attractive forces between molecules. These attractive forces are scaled using the smoothing kernel W as a weighting function:

$$\frac{D\vec{v}_i}{Dt} = -\frac{\kappa}{m_i} \sum_{j=1}^N m_j W(x_i - x_j, h) \cdot (x_i - x_j), \quad (4.5)$$

where κ is a constant that controls how strong the surface tension will be, m_i and m_j are the masses of particles i and j , N is the particle count, x_i and x_j are the positions of particles i and j , respectively, and h is the smoothing kernel radius.

This method allows the surface tension computation in an efficient manner, does not rely on the simulation of a second fluid phase and is well suited for free surface problems with high curvatures.

Figure 4.3 illustrates the surface tension in action. The initial state of the fluid is a cube floating in zero gravity. As the simulation progresses and the fluid reaches equilibrium, a perfectly round sphere is formed. Different values of b for the Tait equation result in spheres of various densities.

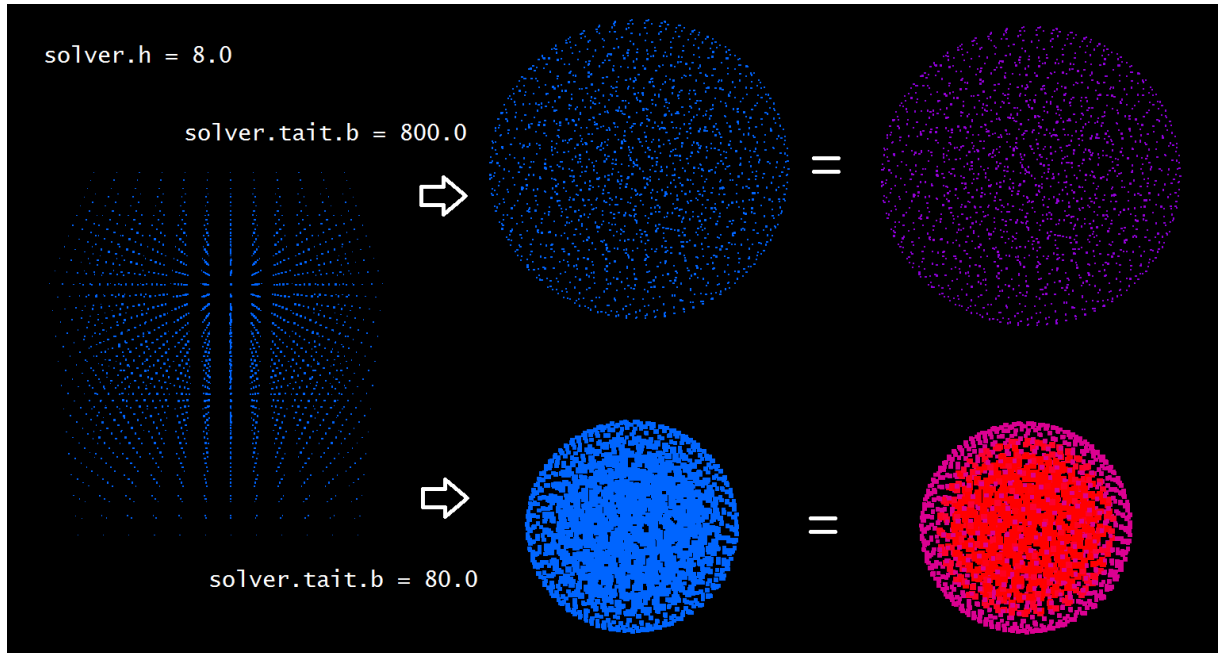


Figure 4.3: Surface tension influence in a zero gravity environment. The solver had the smoothing kernel radius set to 8 and different values of b were tested for the Tait equation.

4.2 Implementation Details

The simulation domain can be either infinite allowing the particles to go anywhere or closed with the shape of a rectangular parallelepiped. In the latter case, a boundary condition that makes particles bounce off the domain walls is applied.

Neighbour particle lookups are an essential part of the SPH algorithm and a specialized data structure is used to accelerate this task. The fact that the smoothing kernels have compact support (i.e. a limited range), which is defined by the radius h , allows the particles to be distributed over a uniform grid containing cubic cells of side h . In this way, all the particles that interact with a given particle i and thus are needed to calculate its physical quantities will be either on the same cell as particle i is or in the immediate neighbour cells. This scheme reduces the computational complexity of finding the neighbour particles from $\mathcal{O}(n^2)$ to $\mathcal{O}(nm)$, where n is the total particle count and m is the average particle count per cell of the uniform grid. In general, n is much greater than m . Figure 4.4 shows a visualization of the data structure described above.

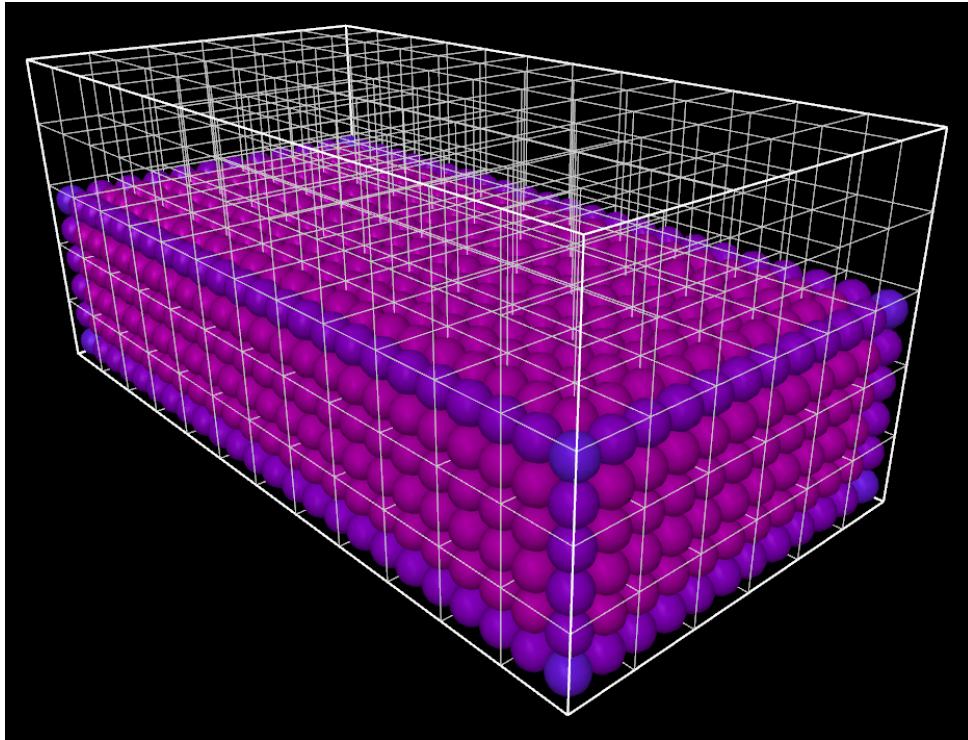


Figure 4.4: Uniform grid used to accelerate the neighbour particle lookups.

The uniform grid is a finite data structure and so it cannot natively handle particles in an infinite domain. In order to do so, a technique called Spatial Hashing [35] must be used. This technique enables sparse fluids in large environments by employing a hash function to map an infinite abstract 3D grid into a finite concrete grid. This work

uses a very intuitive hash function that maps an abstract cell (x, y, z) to a concrete cell $(x \% w, y \% h, z \% d)$, where $\%$ is the modulus operator, w is the width, h is its height and d is the depth of the concrete grid. Neighbour searching is fast and simple as an abstract neighbour cell $(x + dx, y + dy, z + dz)$ maps to the concrete cell $((x + dx) \% w, (y + dy) \% h, (z + dz) \% d)$. Sometimes cells will contain particles that are far away from each other in the domain. However, this does not pose a problem because as the smoothing kernels have limited support, the contributions from far away particles to any calculations end up being canceled.

The simulation is initialized as shown in Algorithm 4. Then, all steps described in Algorithm 5 are executed for each iteration of the SPH method.

Algorithm 4 SPH simulation initialization.

```

1: Initialize two copies of the uniform grid data structure;
2:  $dx \leftarrow 1.4$ ;  $\{dx \text{ is the initial spacing between particles.}\}$ 
3:  $x \leftarrow y \leftarrow z \leftarrow dx \div 2$ ;
4:  $max_x \leftarrow 48$ ;
5:  $max_y \leftarrow 20$ ;
6:  $max_z \leftarrow 24$ ;
7: while  $z < max_z$  do
8:   while  $y < max_y$  do
9:     while  $x < max_x$  do
10:      Create particle at position  $(x, y, z)$ ;
11:      Allocate the particle in the first uniform grid;
12:       $x \leftarrow x + dx$ ;
13:    end while
14:     $y \leftarrow y + dx$ ;
15:  end while
16:   $z \leftarrow z + dx$ ;
17: end while
18:  $time \leftarrow 0$ ;  $\{\text{Time that the simulation is running in milliseconds.}\}$ 

```

Algorithm 5 SPH method iteration.

```

1: while simulating do
2:    $\gamma \leftarrow 40$ ; { $\gamma$  is the integration timestep.}
3:    $time \leftarrow time + \gamma$ ; {Advances 40 milliseconds in time.}
4:   for every cell of the first uniform grid do
5:     if the current cell is not empty then
6:        $L \leftarrow$  List of neighbour particles;
7:       for each particle  $P$  contained in the current grid cell do
8:         UpdateParticle( $P, L$ ); {See Algorithm 6 for details.}
9:       end for
10:    end if
11:  end for
12:  for every cell of the first uniform grid do
13:    if the current cell is not empty then
14:      for each particle  $P$  contained in the current grid cell do
15:        Reallocate the particle in the second uniform grid;
16:      end for
17:      Clear this cell in the first uniform grid;
18:    end if
19:  end for
20:  Switch the first grid with the second one;
21: end while

```

Algorithm 6 details how each particle is updated. It is now that the smoothing kernels described in Section 4.1 are used to update the physical properties of each particle. In order to do so, a list of neighbours of each particle being updated is built. All particles within the distance of the smoothing kernel radius are considered to be neighbours. They are the ones which will have an impact on the calculations. A common length chosen for the smoothing radius is 4. In addition, the length of the side of each cube of the uniform grid data structure is usually the same as the smoothing kernel radius because it makes the search for neighbours faster and more intuitive.

Algorithm 6 Method UpdateParticle (particle, list of neighbours).

- 1: $P = \text{particle}$;
 - 2: Update the density of P ;
 - 3: Update the pressure at P ;
 - 4: Calculate the pressure force;
 - 5: Calculate the gravity force;
 - 6: Calculate the viscosity force;
 - 7: Calculate the surface tension force;
 - 8: Calculate the acceleration of P by summing all the forces that are acting over P ;
 - 9: Update the velocity of P based in the acceleration of P ;
 - 10: Update the position of P based in the velocity of P ;
 - 11: Enforce the boundary conditions;
-

4.3 GPU Implementation

The SPH algorithm needs massive arithmetic power to run with good quality and at an acceptable performance level, especially when it is being used to create real time animations and simulations. Thus, it is a good fit to be executed by the GPU. On the other hand, as the average number of neighbours per each particle varies a lot, it is difficult to come up with an efficient GPU SPH implementation. This variation makes it harder not only to balance the processing workload properly across the GPU execution resources, but also to make an efficient and simple data structure to gather each particle neighbours. Such a data structure should be easily accessible in parallel by multiple threads without requiring any kind of expensive synchronization and ideally it also should not use any pointers, which do not perform well with the GPU memory architecture. The HistoPyramid presents itself as a GPU friendly data structure which is suitable to make an efficient particle sorting implementation that allows quick neighbour searching.

The GPU based SPH is initialized in a similar way to its CPU counterpart. However, instead of using two copies of an uniform grid data structure, two linear memory buffers are allocated on the GPU. Each of these buffers will contain an array of particle structs. These arrays are used in a ping pong fashion as the uniform grids were. That is, in a given iteration of the simulation, the particles will be read from the first array, updated and then sorted and copied into the second array. After that, the arrays are swapped so that everything is ready for the next iteration. The particle struct just defines a standard and tightly packed format for storing all particle properties in memory (e.g. position,

velocity, density, etc). In the very beginning, a compute shader is dispatched to seed the initial particles, which are also sorted and have their densities calculated for the first time before the simulation is able to start.

Algorithm 7 describes a high level overview of one SPH iteration on the GPU.

Algorithm 7 SPH method iteration on the GPU.

```

1: while simulating do
2:    $\gamma \leftarrow 40$ ; { $\gamma$  is the integration timestep.}
3:    $time \leftarrow time + \gamma$ ; {Advances 40 milliseconds in time.}
4:   Dispatch a compute shader to update the particles;
     {One GPU thread is spawned to process each particle.}
5:   Dispatch a compute shader to handle the domain boundary conditions;
6:   Swap the particle buffers;
7:   Sort the particles for neighbour search; {See details in Algorithm 8.}
8: end while

```

Steps 4 and 5 of the above algorithm are performing exactly the same work that is done by the CPU on Algorithm 6. After the particles are updated, they might have moved and thus they cannot be considered to be spatially sorted anymore, so the particle buffers need to be swapped. The buffer that previously contained the array of spatially sorted particles is now bound to the GPU as the unsorted one and vice-versa. This is done in preparation to the last step which consists of finally sorting the particles from the unsorted buffer and writing the results into the other buffer. In essence, the particles are simultaneously sorted and copied from one buffer to the other. After this process is finished, everything is ready to proceed to the next simulation iteration.

The compute shader that update the particles is able to search for each particle neighbours by looking at three data structures: a 3D HistoPyramid, a 3D offsets texture and the sorted particles buffer. The 3D HistoPyramid dimensions are equal to the uniform grid dimensions. Each element at the base of the HistoPyramid stores the particle count of a given grid cell. The offsets texture is an auxiliary texture pyramid with the same dimensions of the HistoPyramid that encodes in which memory location of the sorted particles buffer the first particle of a given grid cell is stored.

These data structures make it possible to easily find every particle contained in a grid cell. It is as simple as reading from the sorted buffer as many particles as is determined by the particle count in the HistoPyramid base, starting from the memory location of the

first particle contained in that cell, as indicated by the offsets texture. To gather all the neighbours of a given particle is a matter of collecting all the particles from the nearby grid cells.

In order for all this to work properly, first these data structures must have been built and updated. In addition to that, the particles must already have been sorted. Algorithm 8 details how these processes are done.

Algorithm 8 Preparing the required data structures for particle sorting.

- 1: Fill the HistoPyramid base level with zeros;
 - 2: Update the HistoPyramid base level;
 - 3: Reduce the HistoPyramid;
 - 4: Update the offsets texture;
 - 5: Sort the particles and also copy them from the unsorted buffer to the sorted one;
-

At the beginning, the HistoPyramid base level is cleared by having zeros written to all its elements. Next, a compute shader is dispatched to update the HistoPyramid base. One GPU thread is spawned per particle with the sole job of determining in which grid cell each particle falls and then using atomic addition to increase by one the corresponding element in the HistoPyramid base. After this, the remaining levels of the HistoPyramid are built as usual by reducing the previous levels.

The HistoPyramid traversal used in this algorithm, however, is performed with a slightly different objective than usual. Here it is done incrementally in order to update the offsets texture level by level, starting from the top level and continuing until the base is reached. For each level of the offsets texture that is being updated, both its previous level and the HistoPyramid are used to calculate the new offsets. When this process finishes, each element of the base level of the offsets texture points to the exact location of the sorted particles buffer that will store the first particle of a given grid cell. The remaining particles of each grid cell are stored adjacently and immediately after the first one.

Finally, a compute shader can be dispatched to effectively sort the particles. One GPU thread is spawned for each particle on the unsorted buffer. The grid cell that each particle belongs to is determined. By looking at the appropriate location of the offsets texture it is possible to know where in the sorted particles buffer the particles belonging to a given grid cell should be stored. In addition to this, atomic additions are used to find a unique location to store each particle, as multiple threads may try to simultaneously

allocate space for different particles lying in a same cell. Then, with a definitive location resolved, the particle can be copied to the sorted buffer straight into the right place.

After this is done, the sorted particles buffer contains all the fluid particles again and is ready to be used in the next iteration of the simulation.

Chapter 5

Fluid Rendering

The Marching Cubes algorithm is used to create a triangle mesh representing the fluid surface. Later, this mesh can be rendered using traditional shading techniques. This algorithm is a well known method used to create polygonal surfaces approximating an isosurface from a scalar field. As the SPH particles define a density scalar field for the fluid being simulated, this method is suitable to create a graphic visualization of the fluid model used in this work.

Among other techniques available to create images for a SPH simulation, this one presents some advantages, such as outputting a regular triangle mesh which can be rendered and manipulated by a wide range of existing computer graphics algorithms. Additionally, it can be adapted in a relatively straightforward way to run in parallel architectures and the resolution of the extracted mesh can be easily adjusted to make a trade-off between quality and performance, which helps in the creation of real-time animations that look the most realistic way that is possible within a minimum acceptable frame rate. Section 5.1 will explain the basics of the algorithm and Section 5.2 will detail two viable GPU implementations for it. The first one is older and based on geometry shaders and the second one is a new approach based on the HistoPyramid data structure.

5.1 The Marching Cubes Algorithm

This algorithm is grid based and the grid is usually uniform. The main objective of the algorithm is to convert an implicit representation of a mesh into a parametric one. The fluid density scalar field can be seen as an implicit representation of its surface while the generated triangle mesh can be seen as a parametric representation.

First of all, it is important to define what is an isosurface. Consider a function $f(x, y, z)$ that defines a scalar field in 3D space. An isosurface S is the set of points that satisfies the following implicit equation:

$$f(x, y, z) = c,$$

where c is a constant called isovalue. Figure 5.1 shows an example of isolines, which are the analogous to isosurfaces in the 2D space.

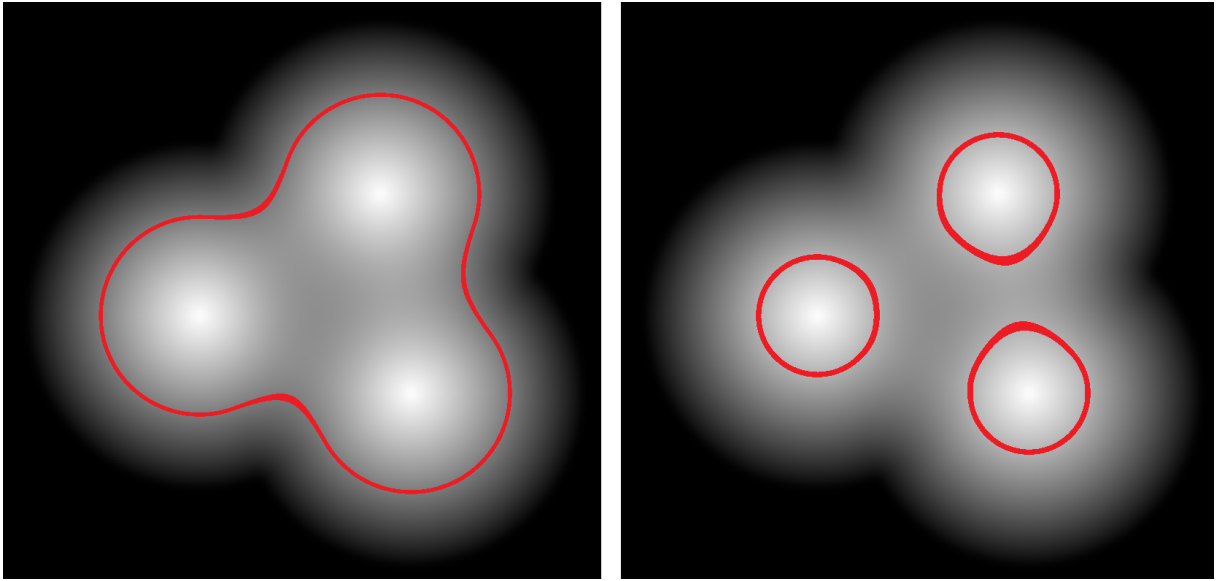


Figure 5.1: A 2D scalar field with an isoline highlighted in red. On the left the isovalue is set to 0.2 and on the right it is set to 0.45.

Algorithm 9 lists the steps performed by the Marching Cubes isosurface extraction procedure.

Algorithm 9 Marching Cubes steps.

- 1: Set the isovalue as desired;
 - 2: Sample the function $f(x, y, z)$ at all vertices of a 3D uniform grid;
 - 3: **for** every vertex of the uniform grid **do**
 - 4: Test if the vertex is inside or outside of the isosurface;
 - 5: **end for**
 - 6: **for** every cube of the uniform grid **do**
 - 7: **if** the surface intersects the current cube **then**
 - 8: Approximate the surface inside the current cube by a set of triangles;
 - 9: **end if**
 - 10: **end for**
-

The algorithm starts by evaluating the function $f(x, y, z)$ at the position of every vertex of a 3D uniform grid. Next, each vertex undergoes a test to verify if it is inside or outside of the isosurface. If the value sampled at the vertex position is greater than the chosen isovalue, then the vertex is considered to be inside of the isosurface. On the other hand, if the sample value is less than the isovalue, the vertex is considered to be on the outside.

The final step is to visit every cube of the uniform grid and verify if the surface intersects it. This is done by checking every edge of a given cube. If any edge of the cube has a vertex inside the isosurface and the other one on the outside, then the edge must be intersecting the surface. And thus, this cube also intersects the surface.

Triangles are created inside all the cubes that intersect the surface. Every cube has 8 vertices that can be either inside or outside the isosurface. The state of each vertex can be encoded in one bit. The state of all 8 vertices of a cube can be encoded in a byte. So there are 256 basic ways in which the surface can interact with any given cube. This byte is used to search a lookup table in order to retrieve the appropriate triangles that must be generated for each case. This table maps each case to a set of edges. Each three edges that are present for any case encode a triangle that must be created. Each vertex of this triangle must lie in one of the encoded edges. There are multiple methods to determine the exact position where each vertex will be created. The simplest one is called midpoint and consists of just creating each vertex exactly at the middle of an edge. This yields a very poor approximation of the original surface. The most widely used method is the linear interpolation, which considers the scalar field values at each vertex of a given edge to determine the optimum location to create the vertex. Figure 5.2 shows a comparison between these two methods using as an example a 2D scalar field. Note that in this case the Marching Squares algorithm is used and lines are created instead of triangles. However, the same principles from the 3D case apply.

It is important to observe that there are only 15 unique Marching Cubes cases. It is due to the fact that there are a lot of symmetries (like reflections and mirroring) among the 256 original cases. Figure 5.3 illustrates these 15 unique cases.

Figure 5.4 shows the Marching Cubes algorithm being used to render a zero gravity fluid drop.

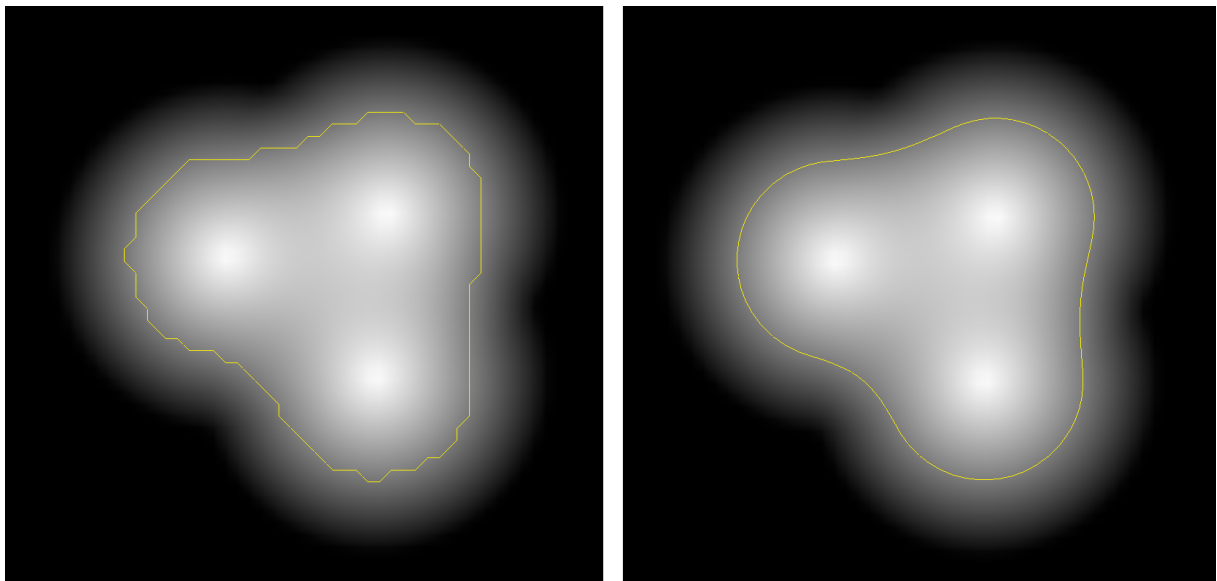


Figure 5.2: A 2D scalar field with an isoline highlighted in yellow. On the left the line segments were created using the midpoint method. On the right, linear interpolation was used. Usage of the latter method clearly results in a better approximation of the contour.

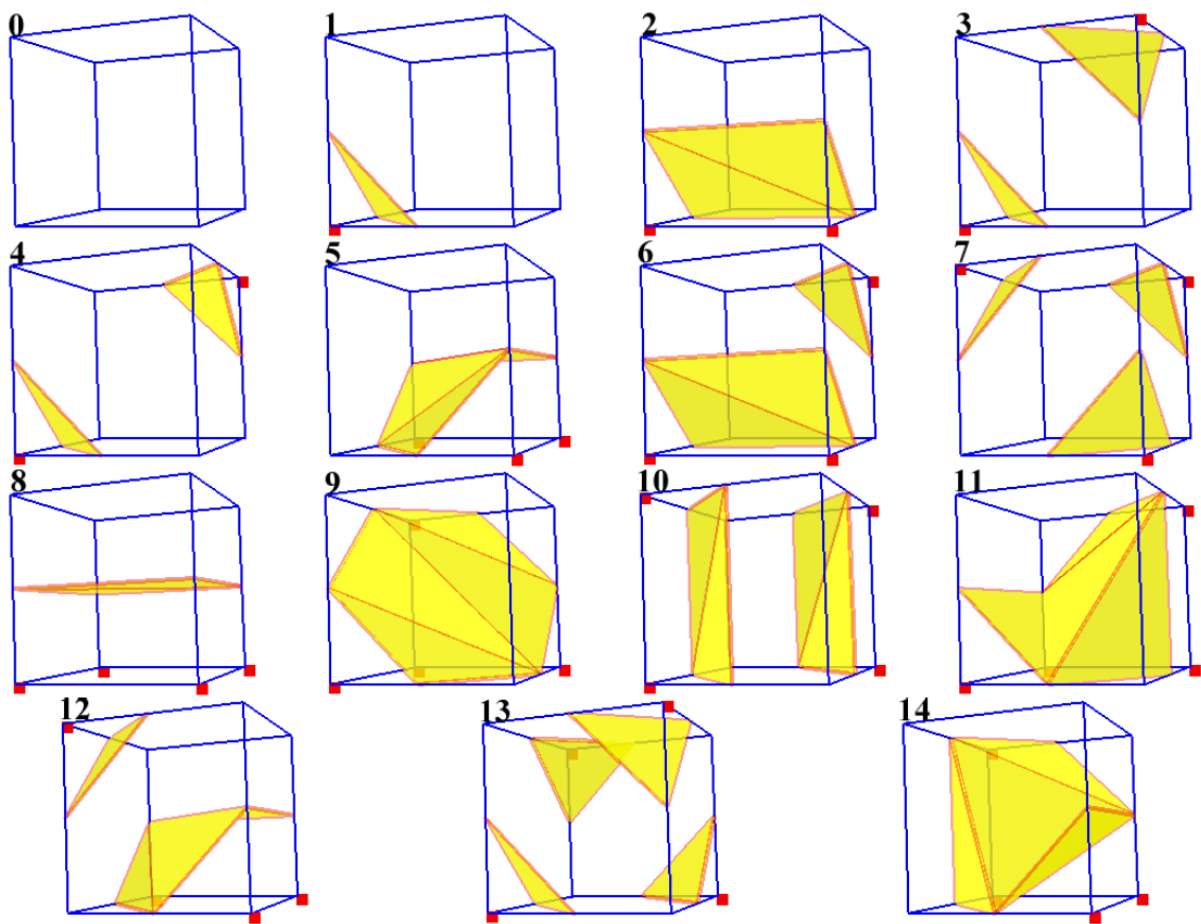


Figure 5.3: Marching Cubes unique cases [21].

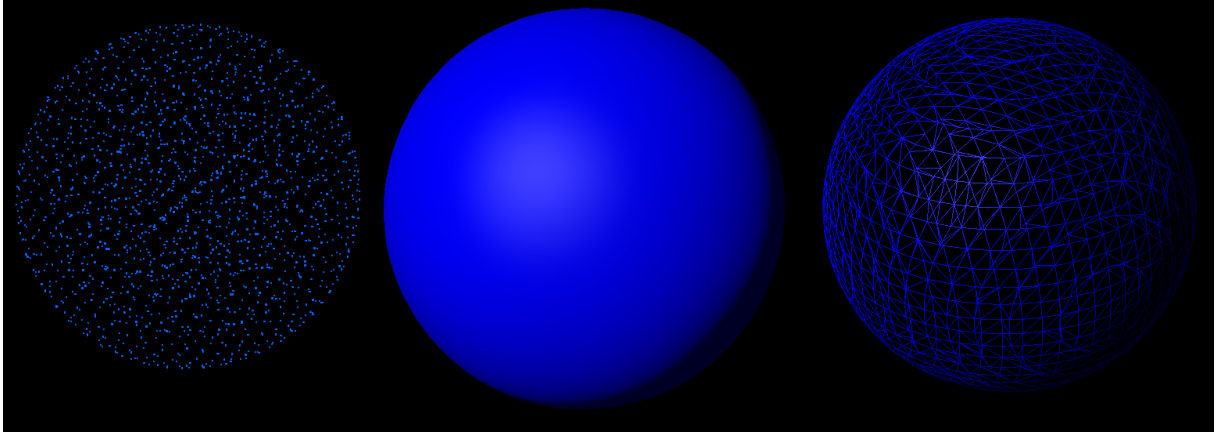


Figure 5.4: A mesh created by the Marching Cubes algorithm of a zero gravity fluid drop. Left: original SPH particles. Middle: the shaded mesh. Right: a wireframe view.

Some of the 256 basic cases have multiple possible triangulations. Extra calculations must be performed to solve these ambiguous cases and additional support lookup tables need to be queried in order to determine the correct triangulation for these cubes. Using only the standard lookup table, which is only able to encode one default triangulation for each one of these ambiguous cases, may eventually lead to small cracks and inconsistent topology in the generated meshes. In spite of this, it was the approach chosen for this work. The ambiguity resolution process may help create topologically correct meshes, however, as some cases require extensive calculations to be disambiguated properly, it also comes with a huge computational cost. In particular, the processing load required by each cube is highly imbalanced, which makes it much harder to create a good parallel implementation. Significant efforts were put into adapting existing disambiguation procedures to be more GPU friendly, but a full topologically consistent implementation was left as an improvement for a future work. A detailed discussion about this topic can be found in Custódio et al. [7].

5.2 GPU Implementation

At first it seems that it is straightforward to parallelize the Marching Cubes algorithm. It appears that it is enough to distribute all the cubes that have to be processed evenly among the available processors. However, for each cube a different case from the lookup table may apply, leading to zero, one or more triangles being generated for that cube. This variability in the number of triangles outputted per cube creates an imbalance in processing load that makes it harder to create an efficient parallel implementation. There are two main approaches to handle this workload on the GPU, as will be explained next.

5.2.1 Geometry Shader Implementation

The first approach relies on the built-in geometry shader and takes advantage of its capability to output zero or more primitives, effectively discarding or amplifying the geometry that is passed into it.

Algorithm 10 describes the steps required by this approach. First, a 3D texture must be allocated on the GPU memory to hold the 3D scalar field values. This texture may be updated either manually by uploading data from the CPU or by a compute shader running on the GPU itself. In addition, the case table is also uploaded to the GPU as a texture. Then, a draw call is made instructing the GPU to render one point for each cube that will be processed. Specially designed vertex and geometry shaders are active during this draw call.

Algorithm 10 Geometry Shader Marching Cubes Implementation.

- 1: Set the isovalue as desired;
 - 2: Update the 3D scalar field texture; {This can be done via CPU or GPU.}
 - 3: Setup the appropriate shaders and upload the case table to the GPU as a texture;
 - 4: Dispatch a draw call to render one point for each cube to be processed;
 {The following steps are executed on the GPU for each point:}
 - 5: The vertex shader calculates the position of each cube in space and passes both the cube index and position to the geometry shader;
 {The following steps are executed by the geometry shader:}
 - 6: Fetch the scalar field values at each corner of the cube from the 3D texture;
 - 7: Calculates the position of each corner;
 - 8: Test if each vertex of the cube is inside or outside of the isosurface;
 - 9: Determine the Marching Cubes case;
 - 10: Look up the appropriate entry of the case table;
 - 11: **if** the surface intersects the current cube **then**
 - 12: **for** each set of 3 edges on the retrieved table entry **do**
 - 13: Create a triangle whose vertices lie in these 3 edges;
 - 14: **end for**
 - 15: **end if**
 {The following step is executed immediately after the geometry shader:}
 - 16: Use transform feedback to store the generated triangles into a buffer object for later usage or let the rasterization process continue in order to draw the triangles straight to screen;
-

The vertex shader starts by calculating the position of each cube in space and passes this information along with the cube index to the geometry shader, which is configured to take points as input and to produce triangles as its output. The first thing that the geometry shader does is to use the current cube index to fetch all the required scalar field values from the 3D texture. It also calculates the position in space of each corner of the cube. After this is done, each vertex of the cube is tested to find out if it is inside or outside of the isosurface. Then, the state of all the 8 vertices can be encoded in a byte in order to determine to which Marching Cubes case this particular cube belongs.

Now a lookup is performed in the case table. If the entry for the current case is empty, it means that the surface does not intersect the current cube and so the geometry shader instructs the GPU to discard this input primitive, not generating any triangles at all. This only happens if, at the same time, the 8 vertices are either all inside or all outside of the surface. For all the other cases, the surface intersects the current cube and the data retrieved from the table entry must be processed further.

For each set of three edges found on the table entry, a triangle must be generated. The vertices of this triangle must lie each somewhere in one of the edges from the set. The exact position where each vertex will be depends if the midpoint method or linear interpolation is being used, as was explained in Section 5.1.

In addition, normals are generated for each vertex using a finite difference method. This makes the generated mesh ready for the lighting calculations that will happen in the later stages of the graphics pipeline. Both forward and central differences can be used. The implementation of the former requires less calculations and thus yields a slightly better performance while the latter offers greater quality.

After the geometry shader execution finishes, the triangles that were created by it may have two destinations. They can either be stored into a buffer object on GPU memory through the transform feedback functionality or they can be passed down directly to the rest of the pipeline for immediate rasterization on screen.

5.2.2 HistoPyramid Based Implementation

The other approach to implement the Marching Cubes algorithm on the GPU is based on the HistoPyramid data structure that was presented in Chapter 3. Algorithm 11 lists the steps performed by this implementation.

Algorithm 11 HistoPyramid Based Marching Cubes Implementation.

- 1: Initialize the HistoPyramid data structure;
 - 2: Set the isovalue as desired;
 - 3: Update the 3D scalar field texture; {This can be done via CPU or GPU.}
 - 4: Setup the appropriate shaders and upload the case table to the GPU as a texture;
 - 5: Upload the auxiliary vertex count table to the GPU as a texture;
 {First HistoPyramid execution phase:}
 - 6: Dispatch a compute shader to update the HistoPyramid base level;
 {These steps are executed by the previously invoked compute shader:}
 - 7: **for** each cube to be processed **do**
 - 8: Fetch the scalar field values at each corner of the cube from the 3D texture;
 - 9: Test if each vertex of the cube is inside or outside of the isosurface;
 - 10: Determine the Marching Cubes case;
 - 11: Look up the appropriate entry of the auxiliary vertex count table;
 - 12: Store both the case number and how many vertices this cube will output in the
 correct place of the HistoPyramid base level;
 - 13: **end for**
 - 14: Perform reduction on the HistoPyramid;
 {Second HistoPyramid execution phase:}
 - 15: **if** the draw indirect technique is enabled **then**
 - 16: Dispatch a compute shader to setup the draw indirect buffer;
 - 17: **else**
 - 18: Read back the total vertex count from the top element of the HistoPyramid;
 - 19: **end if**
 - 20: Dispatch an appropriate draw call to render the entire triangle mesh;
 {The following steps are executed by the vertex shader:}
 - 21: Traverse the HistoPyramid in order to determine which vertex outputted from what
 cube should be processed;
 - 22: Retrieve the case number of the current cube;
 - 23: Look up the case table to find out the correct edge where this vertex should lie;
 - 24: Calculate the vertex position;
 {The following step is executed after the vertex shader and primitive assembly:}
 - 25: Use transform feedback to store the generated triangles into a buffer object for later
 usage or let the rasterization process continue in order to draw the triangles straight
 to screen;
-

The first step is to initialize the HistoPyramid data structure as was described by Algorithm 1. Then, in a similar way to the other implementation, a 3D texture must be allocated on GPU memory to hold the 3D scalar field values. As before, this texture may be updated either manually by uploading data from the CPU or by a compute shader running on the GPU itself. After this, both the case table and a new auxiliary vertex count table are uploaded to the GPU memory as textures. The vertex count table stores how many vertices each Marching Cubes case outputs.

Next, the first phase of the HistoPyramid execution starts. This is the building phase and its general steps were outlined in Algorithm 2. It starts by dispatching a compute shader in order to update the base level. One GPU thread is spawned for each cube that needs to be processed. Each processing thread fetches from GPU memory the scalar field values at the corners of the cube its responsible for. Then, it tests each vertex to see if it is inside or outside of the isosurface and determines which Marching Cubes case apply. After this, the appropriate entry of the vertex count table is retrieved in order to find out how many vertices will have to be created inside this cube to polygonize the surface. The last thing each GPU thread does in this phase is to store both the case number and the vertex output count it computed in the correct place of the HistoPyramid base level. Caching the case number in this way will avoid the need to recompute it later. And finally, to complete this phase parallel reduction is performed on the data structure, as was described earlier in Section 3.3.

Now, the second HistoPyramid execution phase can start. This phase is when the processing is actually done. One GPU thread is needed to process each output element, i.e., each vertex that will be generated for the final mesh. To determine the appropriate number of threads to spawn, the total vertex count must be retrieved from the top element of the data structure. As was discussed in Section 3.4, there are two ways to do this. The first one relies on the draw indirect technique, which yields better performance but only works on newer hardware. If the GPU supports this technique, then a compute shader is dispatched to setup the draw indirect buffer and after that an indirect draw call can be dispatched. Otherwise, as a fallback, the CPU reads back the top element from the HistoPyramid texture and dispatches a traditional draw call to render the appropriate number of vertices.

Regardless of the method that generated the draw call, it will spawn one GPU thread to process each vertex of the mesh. Each thread starts by traversing the HistoPyramid as was described in Algorithm 3. After the traversal is done, it knows which vertex

outputted from what cube it should process. The next step is to retrieve the case number of the cube, which was previously stored in the base level. Then, a lookup is performed in the case table to determine the correct edge where the vertex being processed should lie. Knowing this, it is possible to calculate the vertex position using either the midpoint method or linear interpolation. Also, the vertex normal is generated using the same finite difference method that was used in the geometry shader based implementation.

Finally, after the vertices have been processed they are assembled into triangles that can be either stored into a buffer object on GPU memory via the transform feedback mechanism or they can be passed down directly to the rest of the graphics pipeline for immediate rasterization on screen.

Chapter 6

Results and Discussion

Some experiments were performed with the objective of assessing the efficiency of the HistoPyramid based algorithms which were developed in this work. To make sense and compensate for the implementation efforts, these new algorithms must perform better and more efficiently than the ones that were previously available.

Although the algorithms were built to work together as a fully fledged fluid simulation framework, it is essential to first evaluate their efficiency separately. So, the initial set of experiments will focus on testing both the simulation and rendering algorithms as isolated from each other as possible. Later, a final experiment will assess the combined performance of the whole framework.

6.1 Methodology of the Experiments

All tests were carried on a computer with an Intel Core 2 Quad Q6600 CPU running at 2.4 Ghz, 4 GB of RAM and a Nvidia GeForce GTX 560 GPU with 1 GB of dedicated video memory. On the software side, the operating system used was the 64-bit edition of Windows 7. The Nvidia graphics driver version 335.23 was installed.

Each experiment consists of a previously specified scenario with well defined initial conditions. Most parameters remain fixed with a few exceptions. The ones that change are key parameters which are incrementally modified in order to create problems of increasing computational complexity. The objective is to make comparisons that illustrate how well each algorithm scales in relation to each other.

All scenarios are based on the dam break animation, which is a classic CFD test case as was discussed in Section 4.1. Every measurement was performed three times and the resulting timings were averaged. The specific setup of each scenario will be discussed in the remainder of this chapter.

An existing limitation is that the compute shader work group sizes were kept fixed across the tests. The heuristics described in a best practices guide by Nvidia [28] were investigated and used to pick sensible default values. One dimensional compute shaders are dispatched with a work group size of 256 and the three dimensional ones with work groups of $8 \times 8 \times 8$ elements. These values were selected striving to achieve good hardware occupancy, i.e., keeping the GPU processing resources as busy as possible, for all the compute shaders that were used.

Ideally the occupancy should be calculated on a shader by shader basis in order to determine the most optimized work group size for each compute dispatch. However, such a detailed analysis was beyond the scope of this work and thus was left as an exercise for a future work. The main reason behind this choice was that such a thorough study would require the use of specialized techniques and profiling tools on several parts of the code, which would be extremely time consuming. In addition, this kind of optimization is specific to each GPU architecture and must be redone for every other architecture.

The other types of shaders are free from this hassle, as they are automatically managed by the graphics driver.

6.2 Experiments

The first experiment is about fluid simulation and aims to compare a standard single-threaded CPU implementation of the SPH method with the GPU HistoPyramid based one. Then, a second experiment deals with the HistoPyramid Marching Cubes algorithm, evaluating how well it fares against the geometry shader based implementation. Finally, one last test addresses the combined performance of both GPU HistoPyramid based algorithms working together to simulate and render a complete fluid animation. Details about each experiment will be discussed next.

6.2.1 SPH Comparison: CPU vs GPU Implementation

The objective of this experiment is to verify if the proposed GPU SPH algorithm is indeed more efficient than a standard single-threaded CPU SPH implementation. To accomplish this, similar dam break simulations were executed using both algorithms and the average times they needed to compute each step of a simulation were measured and compared.

Figure 6.1 illustrates this experiment in action. Marching Cubes was disabled and the particles were rendered in the simplest way possible (i.e. like points) in order to interfere as little as possible with the processing loads. In this test, the variable parameter was the particle count and it was incrementally increased to see how well each algorithm could perform with the greater amount of computation necessary to run an animation smoothly in real-time. More particles were created by raising the height of the dam.

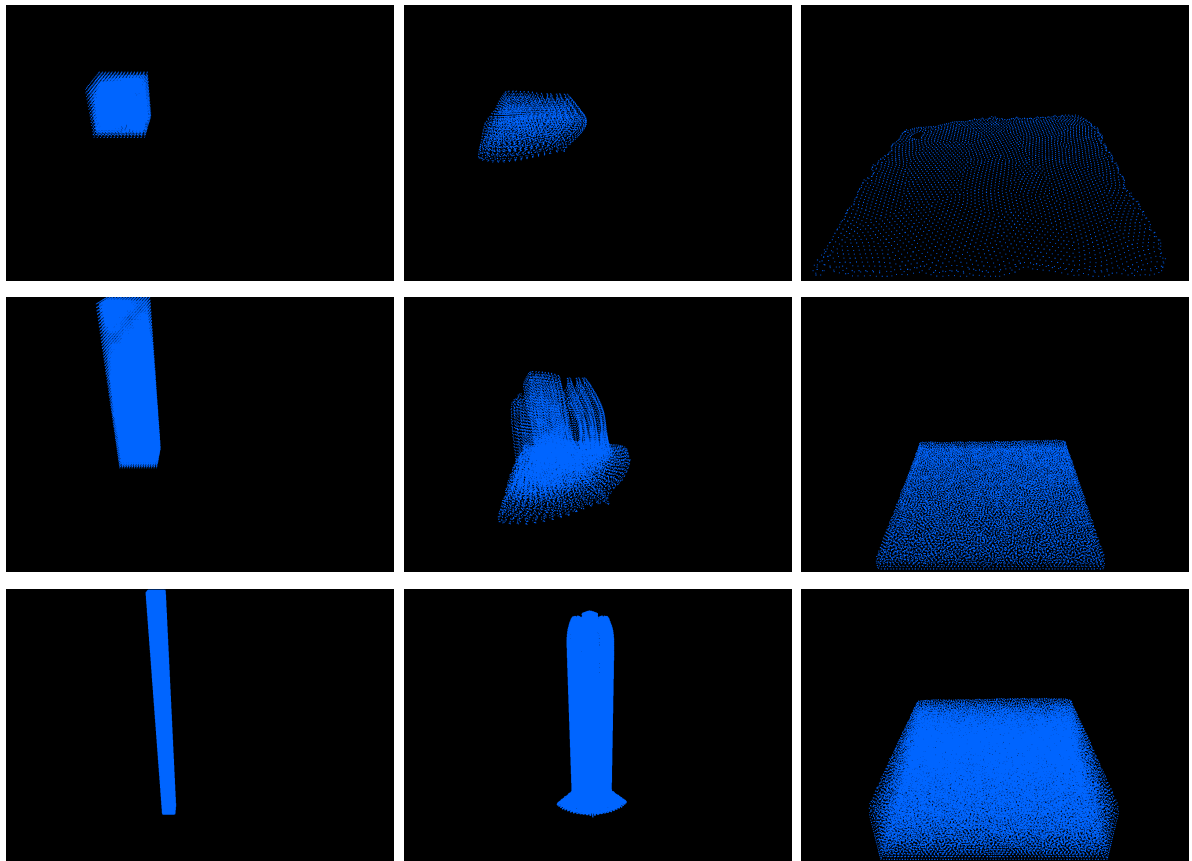


Figure 6.1: Three dam break simulations performed by the SPH algorithm using a varying number of particles. The one on the top row was made with only 4,096 particles. On the middle row, 16,384 particles were used and on the bottom row 65,536 particles were being animated. Notice that the highest the particle count is, the tallest the initial dam height must be. This is pictured on the left column. The middle column shows the fluid falling due to gravity and the right one depicts its resting state.

Table 6.1: SPH comparison: average execution times for various particle counts over 2,000 simulation steps.

Particle Count	CPU Time (ms)	GPU Time (ms)	Speedup
4,096	26.0	2.2	11.8×
8,192	70.2	3.2	21.9×
16,384	227.2	5.9	38.5×
32,768	494.1	10.2	48.4×
65,536	943.2	20.2	46.7×

The other simulation parameters were kept constant throughout the tests, including, for instance, gravity, viscosity, the integration timestep, the smoothing kernel radius, etc.

Table 6.1 lists the average time that each algorithm spent on a single simulation step. This average was taken over 2,000 steps, which is generally enough for the fluid to reach its resting state.

It is possible to observe that up to 32,768 particles, the greater the particle count, the more efficient the GPU algorithm turns to be. This is probably due to the hugely increased arithmetic power required to compute the force interactions between pairs of particles.

It was confirmed that both implementations are able to run simulations with up to 262,144 particles flawlessly.

6.2.2 Marching Cubes Comparison: Geometry Shader vs HistoPyramid Implementation

This test aims to confirm if the HistoPyramid based Marching Cubes is more efficient than the older, geometry shader based algorithm. To do this, a single frame of a fluid simulation with 16,384 particles was chosen to be polygonized by both of the algorithms using various grid resolutions to see how well they could cope with bigger scalar fields.

For each case tested, the fluid was rendered for a while and the average time needed to render each frame was measured. Note that as the simulation was frozen, the contents of the scalar field were not being updated. However, it was indeed being polygonized every frame. In addition, the wireframe mode was used in order to use as little fill rate as possible from the GPU. The focus was strictly on comparing the performance of the conversion of the scalar field into a triangle mesh. The draw indirect technique was enabled on the HistoPyramid implementation.

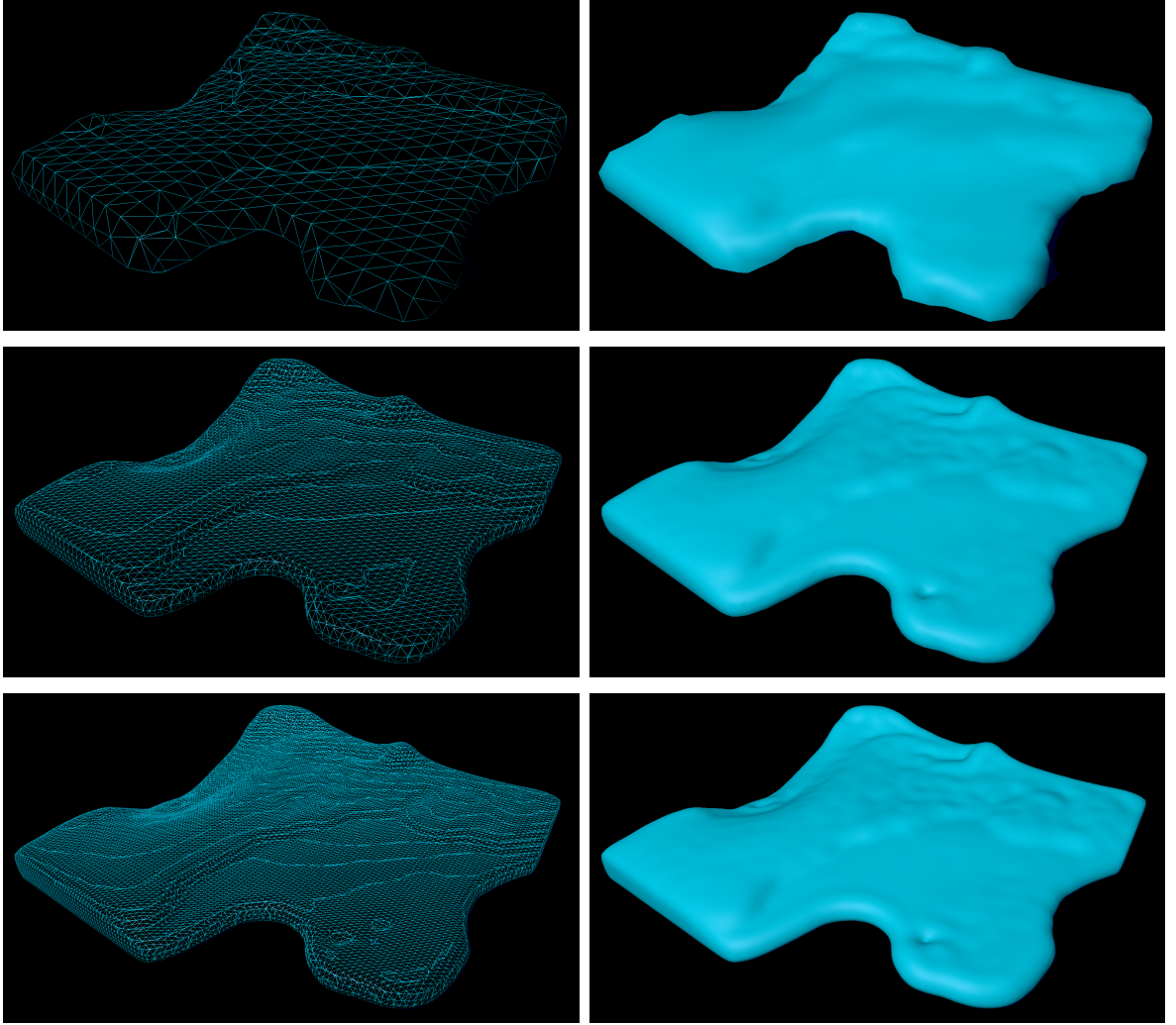


Figure 6.2: Meshes generated by the Marching Cubes algorithm during this experiment. The following grid resolutions were used: $35 \times 67 \times 35$ on the top row; $99 \times 195 \times 99$ on the middle row and $163 \times 323 \times 163$ on the bottom row. On the left, the polygonized mesh is shown in wireframe. On the right, a shaded version with per pixel lighting is presented.

Some of the meshes generated during the tests are shown in Figure 6.2, ranging from lower to higher resolution ones. Table 6.2 summarizes the results, comparing the efficiency of both Marching Cubes implementations. Except for the lowest resolution grid, significant speedups were obtained. In this case that requires a smaller amount of processing, the overhead of the HistoPyramid algorithm outweighed the benefits of the increased parallelism it offers and the execution ended up being slower.

Additional tests were done to analyze the impact of the draw indirect technique on the HistoPyramid version. Speedups ranging from 1.08 to 1.19 \times were obtained with the technique enabled, as can be observed in Table 6.3.

Table 6.2: Marching Cubes comparison: average execution times for various grid resolutions over 2,000 rendered frames. The draw indirect technique was enabled for the HistoPyramid version.

Grid Resolution	GPU GS Time (ms)	GPU HP Time (ms)	Speedup
$35 \times 67 \times 35$	0.65	0.97	$0.67\times$
$67 \times 131 \times 67$	2.21	1.09	$2.03\times$
$99 \times 195 \times 99$	5.92	1.54	$3.84\times$
$131 \times 259 \times 131$	12.82	1.89	$6.78\times$
$163 \times 323 \times 163$	22.22	2.40	$9.26\times$

Table 6.3: HistoPyramid Marching Cubes comparison: average execution times for various grid resolutions over 2,000 rendered frames. Using standard readback (RB) vs the draw indirect technique (DI).

Grid Resolution	GPU RB HP Time (ms)	GPU DI HP Time (ms)	Speedup
$35 \times 67 \times 35$	1.15	0.97	$1.19\times$
$67 \times 131 \times 67$	1.28	1.09	$1.17\times$
$99 \times 195 \times 99$	1.71	1.54	$1.11\times$
$131 \times 259 \times 131$	2.05	1.89	$1.08\times$
$163 \times 323 \times 163$	2.63	2.40	$1.10\times$

6.2.3 Combined Experiment

This experiment was concerned with running both the HistoPyramid based simulation and rendering algorithms together and investigating how to load balance the computational costs between them in order to achieve a smooth fully shaded real-time animation running at a fixed 30 frames per second. Increasing the processing required by one of the algorithms diminishes the headroom left for the other. Table 6.4 lists some configurations that yielded the target frame rate.

Table 6.4: Configurations of the simulation and rendering HistoPyramid algorithms that were able to achieve a smooth 30 frames per second animation.

Grid Resolution	Polygonized Cubes	Particle Count
$35 \times 147 \times 35$	168,776	73,728
$67 \times 247 \times 67$	1,071,576	62,464
$99 \times 243 \times 99$	2,324,168	40,960
$131 \times 195 \times 131$	3,278,600	24,576
$163 \times 151 \times 163$	3,936,600	15,104

Figure 6.3 shows a plot of the data from Table 6.4.

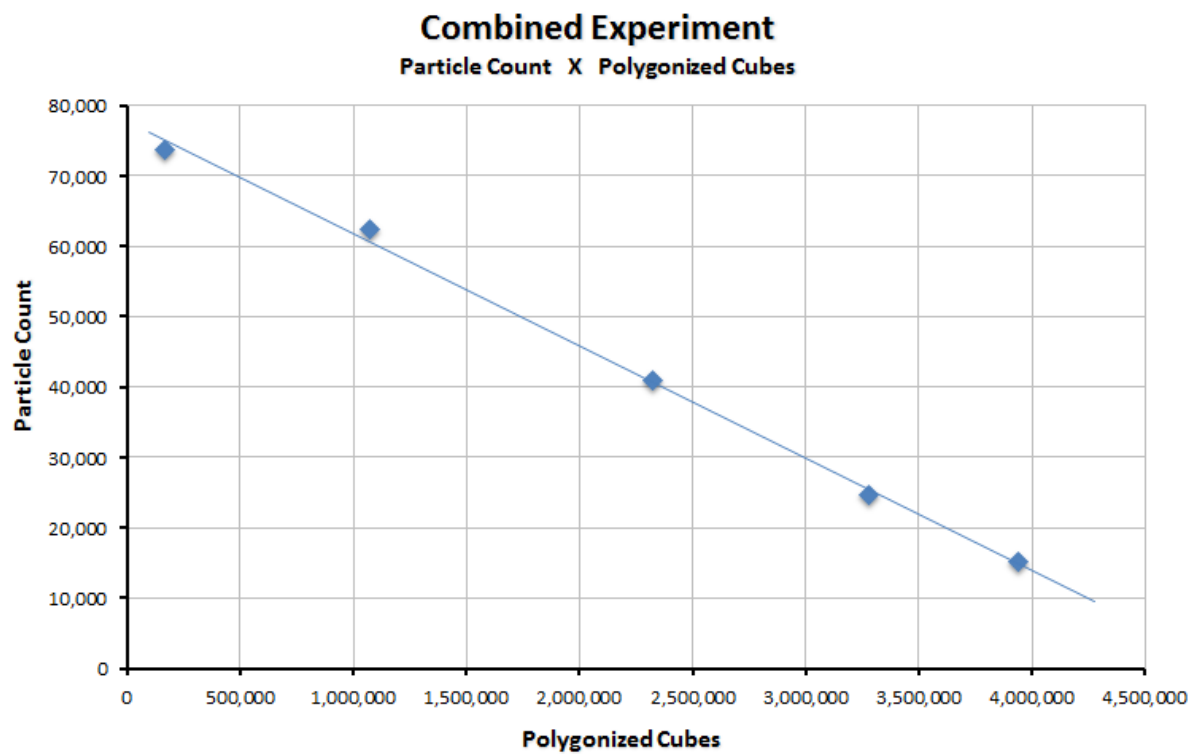


Figure 6.3: A plot with pairs of parameters that can be used to create animations that run consistently at 30 frames per second. A linear trend line is shown.

Chapter 7

Conclusions and Future Work

This work employed the Histogram Pyramid data structure in order to successfully implement an efficient GPU accelerated framework to simulate and render fluids. This data structure proved to be extremely useful as it enables an entire kind of new problems to be properly parallelized to run on the GPU. In particular, stream compaction and expansion algorithms can now take advantage of the massive computational power offered by modern GPUs.

For instance, the Smoothed Particle Hydrodynamics fluid simulation method greatly benefits from using the HistoPyramid. It requires a huge amount of arithmetic calculations and thus is a good candidate to be computed on the GPU, but there is a caveat to implement it efficiently. Neighbour particle lookups, which are an essential part of the SPH method, need to retrieve a variable number of particles from seemingly random memory positions. This adversely affects performance as GPUs need data accesses to be as coalescent as possible to perform well. The present work made an efficient implementation possible by using the HistoPyramid to implement a GPU bucket sort algorithm, which is able to store nearby particles close together in memory by spatially sorting them.

The Marching Cubes algorithm, which is used to polygonize the fluid surface, is another example of process that can be improved by the HistoPyramid. In the experiments realized, the geometry shader based Marching Cubes was consistently outperformed by the HistoPyramid based implementation, especially in large datasets. The only exception to this rule occurred when processing the smallest dataset tested, where the overhead of setting up the data structure was too big for the little amount of work being processed and the geometry shader implementation ended up being a bit faster.

It may be argued that the geometry shader should also be capable of handling any kind of stream compaction and expansion problem, as it is also capable of discarding and amplifying data. However, it was not designed to be adequate to all kinds of problems. It also has hard limits on the maximum output count per input element because of limited on-chip hardware buffers. In addition, some GPU architectures in spite of being able to execute geometry shaders cannot handle them very well.

Another challenge imposed by the Marching Cubes algorithm to the geometry shader architecture is that the amount of work per invocation varies greatly, resulting in extremely poor parallelism. Some shader instances will create a lot of triangles while others will output just a few or none. The HistoPyramid version creates a more balanced workload.

Beyond that, one more reason for why the HistoPyramid polygonization is faster is because it is able to completely skip all empty cells after it has determined during the building phase that they will not generate any outputs. This is especially beneficial because in most fluid simulations the majority of cells is either entirely inside or outside the fluid and thus empty.

The experiments have also shown that the draw indirect technique was able to deliver some small but noticeable performance gains.

Finally, by using compute shaders it was possible to create a truly 3D HistoPyramid implementation for the GPU in a flexible way that was not possible before by any other means. In spite of this, it is still possible to implement a 3D HistoPyramid on older hardware, although not so cleanly and only with the use of many workarounds.

7.1 Future Work

The current Marching Cubes implementation is not topologically consistent, which means that occasionally small cracks and holes may appear in the generated meshes. As discussed in the end of Section 5.1, significant efforts were put into solving this problem. The work by Custódio et al. [7] would help address this issue. However, an appropriate parallel version of the algorithms proposed by her must be devised. As far as the author knows, a GPU accelerated Marching Cubes algorithm that guarantees the generation of topologically correct meshes for any input data is still an open research problem.

Section 6.1 mentioned a limitation in how the work group sizes of the compute shaders were chosen. A possible solution to this lies in the implementation of a fully featured GPU profiler, such as the one described in the OpenGL Insights book [6]. This would allow precise measurements of individual parts of each algorithm, possibly exposing hotspots in the code that could be further optimized. By exploring and profiling varying work group sizes, it would be possible to verify in practice which ones work best for each compute shader. In addition, the execution configuration optimizations proposed by Nvidia [28] could be investigated in more depth. Sparse textures could possibly be used to improve HistoPyramid memory usage for the cases where the input array is asymmetric or has non power of two side lengths.

Furthermore, the rendering algorithms deserve more attention. In particular, the implementation of a more detailed lighting pipeline would substantially improve the quality of the images generated.

Bibliography

- [1] BECKER, M.; TESCHNER, M. Weakly compressible sph for free surface flows. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 209–217.
- [2] BRIDSON, R. *Fluid Simulation for Computer Graphics*. A K Peters, 2008.
- [3] CHENTANEZ, N.; MÜLLER, M. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph.* 30 (August 2011), 82:1–82:10.
- [4] CHENTANEZ, N.; MÜLLER, M. Mass-conserving eulerian liquid simulation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2012), SCA '12, Eurographics Association, pp. 245–254.
- [5] CHIU, C.-W.; CHUANG, J.-H.; LIN, C.-C.; YU, J.-B. Modeling highly-deformable liquid. In *International Computer Symposium* (2002).
- [6] COZZI, P.; RICCIO, C. *OpenGL Insights*. CRC Press, July 2012. <http://www.openglinights.com/>.
- [7] CUSTÓDIO, L.; ETIENE, T.; PESCO, S.; SILVA, C. Practical considerations on marching cubes 33 topological correctness. *Computers & Graphics* 37, 7 (2013), 840–850.
- [8] DA SILVA JUNIOR, J. R. Simulação computacional em tempo real de fluidos utilizando o método sph em ambiente heterogêneo cpu/gpu. Master's thesis, Universidade Federal Fluminense, 2010.
- [9] DA SILVA JUNIOR, J. R.; CLUA, E. W. G.; MONTENEGRO, A.; LAGE, M.; DE ANDRADE DREUX, M.; JOSELLI, M.; PAGLIOSA, P. A.; KURYLA, C. L. A heterogeneous system based on gpu and multi-core cpu for real-time fluid and rigid body simulation. *International Journal of Computational Fluid Dynamics (Print)* 26 (2012), 193–204.
- [10] DYKEN, C.; ZIEGLER, G.; THEOBALT, C.; SEIDEL, H.-P. High-speed marching cubes using histopyramids. *Computer Graphics Forum* 27, 8 (2008), 2028–2039.
- [11] FOSTER, N.; METAXAS, D. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 181–188.

- [12] FRAEDRICH, R.; AUER, S.; WESTERMANN, R. Efficient high-quality volume rendering of sph data. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (nov 2010), 1533–1540.
- [13] GINGOLD, R. A.; MONAGHAN, J. J. Smoothed particle hydrodynamics - theory and application to non-spherical stars. In *Royal Astronomical Society, Monthly Notices, vol. 181* (1977), pp. 375–389.
- [14] HARADA, T.; KOSHIZUKA, S.; KAWAGUCHI, Y. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International* (2007), pp. 63–70.
- [15] HARRIS, M. J.; BAXTER, W. V.; SCHEUERMANN, T.; LASTRA, A. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 92–101.
- [16] HU, X. Y.; ADAMS, N. A. A multi-phase sph method for macroscopic and mesoscopic flows. *J. Comput. Phys.* 213, 2 (2006), 844–861.
- [17] INÁCIO, R. T.; NOBREGA, T.; CARVALHO, D. D. B.; VON WANGENHEIM, A. Interactive simulation and visualization of fluids with surface raycasting. In *SIBGRAPI* (2010), IEEE Computer Society, pp. 142–148.
- [18] KIPFER, P.; WESTERMANN, R. Realistic and interactive simulation of rivers. In *GI '06: Proceedings of Graphics Interface 2006* (2006), Canadian Human-Computer Communications Society, pp. 41–48.
- [19] KOSHIZUKA, S.; OKA, Y. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear Science and Engineering Volume* 123, 3 (jul 1996), 421–434.
- [20] KROG, O. E. Gpu-based real-time snow avalanche simulations. Master's thesis, Norwegian University of Science and Technology, 2010.
- [21] LEWINER, T.; LOPES, H.; VIEIRA, A. W.; TAVARES, G. Efficient implementation of marching cubes cases with topological guarantees. *Journal of Graphics Tools* 8, 2 (december 2003), 1–15.
- [22] LORENSEN, W. E.; CLINE, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *COMPUTER GRAPHICS* 21, 4 (1987), 163–169.
- [23] LUCY, L. B. A numerical approach to the testing of the fission hypothesis. In *Astronomical Journal, vol. 82* (1977), pp. 1013–1024.
- [24] MÜLLER, M.; CHARYPAR, D.; GROSS, M. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 154–159.
- [25] MÜLLER, M.; SCHIRM, S.; DUTHALER, S. Screen space meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), SCA '07, Eurographics Association, pp. 9–15.

- [26] MÜLLER, M.; SCHIRM, S.; TESCHNER, M. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technol. Health Care* 12, 1 (2004), 25–31.
- [27] NEWMAN, T. S.; YI, H. A survey of the marching cubes algorithm. *Computers & Graphics* 30, 5 (oct 2006), 854–879.
- [28] NVIDIA CORPORATION. *CUDA C Best Practices Guide*, August 2014.
- [29] PHARR, M.; HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [30] PREMOZE, S.; TASDIZEN, T.; BIGLER, J.; LEFOHN, A.; WHITAKER, R. T. Particle-based simulation of fluids, 2003. EUROGRAPHICS.
- [31] PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.
- [32] RHU, M.; EREZ, M. The dual-path execution model for efficient gpu control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (Washington, DC, USA, 2013), HPCA '13, IEEE Computer Society, pp. 591–602.
- [33] SOLENTHALER, B.; SCHLÄFLI, J.; PAJAROLA, R. A unified particle model for fluid-solid interactions. *Computer Animation and Virtual Worlds* 18, 1 (2007), 69–82.
- [34] STAM, J. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (August 1999), pp. 121–128.
- [35] TESCHNER, M.; HEIDELBERGER, B.; MUELLER, M.; POMERANETS, D.; GROSS, M. Optimized spatial hashing for collision detection of deformable objects. In *In Proceedings of VMV'03* (2003), pp. 47–54.
- [36] VAN DER LAAN, W. J.; GREEN, S.; SAINZ, M. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 91–98.
- [37] WILLIAMS, L. Pyramidal parametrics. *SIGGRAPH Comput. Graph.* 17, 3 (July 1983), 1–11.
- [38] ZIEGLER, G.; TEVS, A.; THEOBALT, C.; SEIDEL, H.-P. On-the-fly point clouds through histogram pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)* (Aachen, Germany, 2006), L. Kobbelt, T. Kuhlen, T. Aach, and R. Westermann, Eds., European Association for Computer Graphics (Eurographics), Aka, pp. 137–144.