Thesis no: BCS-2014-06

Physically-based fluid-particle system using DirectCompute for use in real-time games

Jesper Hansson Falkenby

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Jesper Hansson Falkenby
E-mail: jehh11@student.bth.se

University advisor:
Petar Jercic
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context: Fluid-particle systems are seldom used in games, the apparent performance costs of simulating a fluid-particle system discourages the developer to implement a system of such. The processing power delivered by a modern GPU enables the developer to implement complex particle systems such as fluid-particle systems. Writing efficient fluid-particle systems is the key when striving for real-time fluid-particle simulations with good scalability.

Objectives: This thesis ultimately tries to provide the reader with a well-performing and scalable fluid-particle system simulated in real-time using a great number of particles. The fluid-particle system implements two different fluid physics models for diversity and comparison purposes. The fluid-particle system will then be measured for each fluid physics model and provide results to educate the reader on how well the performance of a fluid-particle system might scale with the increase of active particles in the simulation. Finally, a performance comparison of the particle scalability is made by completely excluding the fluid physics calculations and simulate the particles using only gravity as an affecting force to be able to demonstrate how taxing the fluid physics calculations are on the GPU.

Methods: The fluid-particle system has been run using different simulation scenarios, where each scenario is defined by the amount of particles being active and the dimensions of our fluid-particle simulation space. The performance results from each scenario has then been saved and put into a collection of results for a given simulation space.

Results: The results presented demonstrate how well the fluid-particle system actually scales being run on a modern GPU. The system reached over a million particles while still running at an acceptable frame rate, for both of the fluid physics models. The results also shows that the performance is greatly reduced by simulating the particle system as a fluid-particle one, instead of only running it with gravity applied.

Conclusions: With the results presented, we are able to conclude that fluid-particle systems scale well with the number of particles being active, while being run on a modern GPU. There are many optimizations to be done to be able to achieve a well-performing fluid-particle system, when developing fluid-particle system you should be wary of the many performance pitfalls that comes with it.

Keywords: Fluid-particle system, GPGPU, DirectCompute, fluid physics model

Table of Contents

1. Introduction

Games are increasingly getting more and more realistic with each year coming. Each year brings us games that are closer towards being photorealistic, games that use more advanced physics models and artificial intelligence that is more adaptable and advanced. What is hindering the goal towards realism is hardware performance and algorithm efficiency. Since the emergence of gaming, we are witnessing constant demand on performance, pushing hardware and calculations to the edge.

Fluid-particle systems are the perfect example of a typical hardware-pushing sub-system in a game, requiring complex algorithms and physics models. By using fluid-particle systems, a game could potentially move one step closer to the ultimate goal of realism, but also introduce several challenges, such as an increase in complexity and hardware requirements of the game, which may be too high for it to be worth implementing a system of such.

1.1 Aim and objectives

The general aim of this thesis is to make an attempt at proving that modern games have the ability to utilize particle systems for fluid simulations with the power of today's graphics cards, thus trying to motivate the reader of using fluid-particle systems in games. The objectives are to explore different methods of simulating fluid, and how optimizations to the fluid-particle system can be done to achieve a well-performing system with good scalability. The methods are then going to be measured in how well they are performing so that we are able to conclude on how feasible it is to utilize fluid-particle systems in real-time games.

1.2 GPU

The *Graphical Processing Unit*, the GPU, is typically used for rendering purposes in games. The GPU executes shaders performing lighting calculations and then outputs lit pixels to display on the screen. Besides this, the GPU is having an increased role in games for doing physics calculations, games are allowed to be more realistic than ever in a much larger environment using more complex physics models. Certain areas are benefitting more from having the processing power of a GPU, particle systems is one such an area. Particle systems are commonly used in games; it is often used to simulate environmental effects such as smoke and fire, but also other not-so-common effects like glass shatter and debris. Particle systems have always been very limited in the number of existing particles and the complexity of each particle.

1.3 GPGPU

With the rise of *General-purpose computing on graphics processor units*, GPGPU, complex physics models together with a great number of particles allows for advanced effects such as fluids, in real-time [1] [2]. Although this technique has been around for quite some time now, games of today are still having a lack of such mechanisms with poor efficiency models, if implemented at all. Only in recent years, games have started to show that particle systems can play a major role in them.

1.3.1 GPGPU - Microsoft DirectCompute

In recent years, shaders have not only been used for rendering purposes, but also for general purposes such as computing algorithms. With an *Application Programming Interface* (API), such as Microsoft DirectCompute, games have the possibility to utilize the power of the GPU for areas like artificial intelligence, physics and collision detection. To be able to utilize the GPU, DirectCompute introduces a shader known as the Compute Shader. This shader is similar to the other shaders used for rendering in the sense that it is programmed using the same language, the *High Level Shading Language*, but with the purpose of performing general computations for use in the application [10]. DirectCompute comes bundled with DirectX 11 and is available for the Microsoft Windows Vista operating system and later. It requires hardware that supports DirectX 10 and later [6].

DirectCompute enables me to perform GPGPU in an environment I have previous experience in, namely the DirectX 11 environment using Direct3D as a rendering API. This allows me to speed up the development process and as a result of it greatly increasing the chance of reaching the goal of this thesis.

1.4 Third-party libraries for particle physics calculations

The PhysX department of NVIDIA have recently presented a fluid-particle system interacting and colliding with other 3D-geometry while rendering this in real-time, running completely on the GPU with respectable performance [4]. The downside of PhysX is that it is closed-source and tied to their specific brand of GPUs.

The Bullet SDK is a fully-featured, open source physics library. Bullet has not been able to compete with competitors like PhysX, it has more often been used in a context of pre-rendering, such as a plugin for the Maya software by Autodesk. Recently the Bullet team have presented their next generation of the Bullet SDK, featuring a 100% GPU accelerated rigid body pipeline [5]. This next generation of Bullet makes for a very good candidate for particle systems such as fluid-particle ones. Although still not officially released, this version is available for the

general public and is in a very late stage of development. Because of the fact of it being unfinished and unstable, the choice was made to not utilize the SDK to run the fluid-particle system, but to use it more as a reference point instead.

1.5 Fluid-particle systems

The usage of fluid-particle systems in games are not very common. It more often occurs as an experiment used as a measurement of how well the system scales in regards of performance on a specific set of hardware. To achieve the realism required of fluid-particle systems, one has to implement a system with tens or even hundreds of thousands of particles to get the desired look. Each particle would act as a sphere, and then on this sphere one applies forces such as gravity and pressure to achieve motion of this particle. After doing this on all the particles in the system, one has to make them interact with each other, which would be the point of the whole particle system. The Figures 1 and 2 below demonstrate the two different steps of a typical fluid-particle system, the particle simulation and the fluid rendering.

Figure 1. Fluid particles as spheres [15] Figure 2. Fluid rendering [15]

So how do the particles interact with each other? Collision detection and the response from it. For each particle one finds its neighboring particles and check whether or not they are colliding with each other, and if they are, perform necessary steps accordingly. After the collision detection has been performed, the final step is to update the velocity of the particle and its position thereafter.

1.5.1 Fluid-particle systems - Fluid physics models

To be able to simulate a fluid-particle system and make it behave like an actual fluid, implementing an underlying fluid physics model for the system is essential. In large, the choice of fluid physics model dictates the behavior of the fluid in the simulation, with various parameters used to fine-tune the behavior, depending on the model of choice. While some are more popular than the other, one model that stands out and that has seen an increase of

popularity through recent years, though widely used in many other areas, is *Smoothed particle hydrodynamics*, often just referred to as SPH. The model originates from the late 70's, and has been widely used for different research areas such as astrophysics [21].

The second model brought up in this thesis is the *discrete element method*, or simply just DEM. This model also originates from the late 70's with the intention to model the behavior of soil particles under dynamic loading condition, though today, DEM is used to simulate granular materials [22]. This model is not intended to be used for fluid-particle systems, though previous studies have shown that it is possible, in a limited fashion [16].

2. Background

Ever since the dawn of video games, physics have been an integral part of the game. Whether it be pseudo-physics or physics based on the real world, it adds some amount of realism to the game, allowing the player to be able to relate to what is happening in it. Some games are completely reliant on the laws of physics, making it a complete necessity to invest time into developing a physics engine based on realism. Portal, a game developed by the game company Valve, uses realistic physics as a core gameplay feature. The player is given a set of puzzles of which are solvable by using the laws of physics. In an exemplary puzzle, the player encounters a pipe mounted on a wall, spitting out a blue, gel-like substance as seen in the figure below (Figure 3).

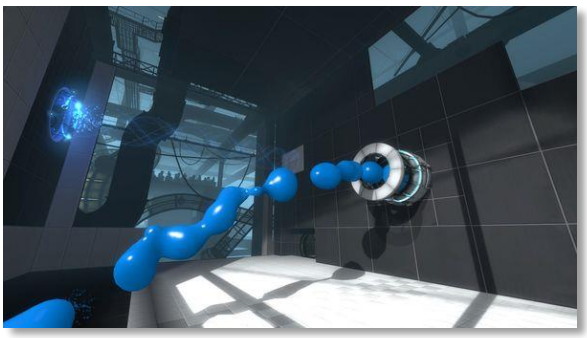

Figure 3. Puzzle solving using fluid physics in Portal 2 (2011) [13]

However, there are good reasons for using pseudo-physics instead of using physics to simulate the real world. One reasoning for this might simply be caused by a time constraint. Writing pseudo-physics specifically for your game where you don't strive for realism is less time consuming than if you would try to simulate the real world. There has also been, up until recent years, a hardware performance factor involved in this reasoning. To be able to simulate the real world, games need complicated algorithms, complicated algorithms requires powerful hardware.

2.1 Particle systems in games

Particle systems are part of the physics in games. As early as in 1962, when the 3rd video game ever *Spacewar* got released, one could witness a very primitive particle system, where the explosion of a ship caused debris particles to emit from the body of the ship [7]. This is to show that the ship actually got destroyed and the body of the ship turned into debris as a cause of it. To emphasize the simplicity of this, the game takes place in a 2-dimensional environment with black and white to color the world, and each debris from the ship is represented by a white pixel. The figure below (Figure 4) demonstrates the simplicity of particle systems used in early games.

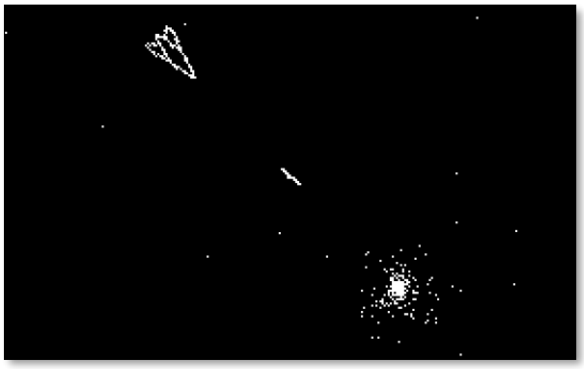

Figure 4. Particle effect in Spacewar simulating debris as the result of an explosion (1962) [11]

Fast-forwarding to the current generation of games, we can witness games that are close to being photorealistic, but still lack the realism contributed from real-world physics. The recently released game *Hawken* features real-time particle systems simulating up to 110,000 particles using sophisticated physics used to simulate different types of effects, such as debris and energy fields. The figure below (Figure 5) demonstrates one particular case where the particle system is simulating an energy field, charging the player with energy [12].

Figure 5. Particle effect in Hawken simulating an energy field (2013) [12]

The culprit here is, more often than you may think, particle systems. When you develop a game, you may stumble upon the limitations of particle systems. Achieving realistic particle systems requires a lot of processing power, especially when the particles are in great numbers, and when the calculations are performed on the central processing unit, the CPU. This may lead to developers being forced to limit their particle systems used in their games by either falling back to using pseudo-physics, in other words more simple physics calculations, or by limiting the amount of particles emitted, or, even worse, by completely dropping the particle system, leading to a possible decrease in realism and the liveliness of the game.

By summarizing the history of particle systems used in games, we have a standpoint of how particle systems have developed through the years, and thus we are able to approximate the potential of future particle systems that would potentially be used in future games.

2.1.1 Fluid-particle systems in games

As mentioned in the previous chapter, fluid-particle systems are not very common in games because of the performance cost that comes with it. Modern games compute particle effects such as smoke on the GPU to achieve good performance results with fluid dynamics to make the effect look realistic. Although not simulated in a game environment, the figure below (Figure 6) demonstrates the visual result of an exemplary particle system using fluid dynamics to simulate smoke.

Figure 6. Smoke particle system using fluid dynamics [14]

What this means is that games do not run actual fluid-particle systems because of the apparent performance cost.

In the next part of this thesis we present some results that has been provided from previous research. Based on these results we try to show how well fluid-particle systems might perform on the GPU, we also try to provide arguments for how reasonable it would be to implement fluid-particle systems for games, and then try to conclude on what kind of research is missing to be able to answer these types of questions.

2.2 Previous research

Previous work has shown that fluid-particle systems are possible to run on the GPU at an interactive frame rate. In the paper *Position Based Fluids* by Macklin and Müller, a method is presented by simulating fluid using *Smoothed Particle Hydrodynamics* as a physics model while performing collision detection against outside geometry [4]. The figures 7 and 8 below demonstrate the final results from their fluid-particle system research.

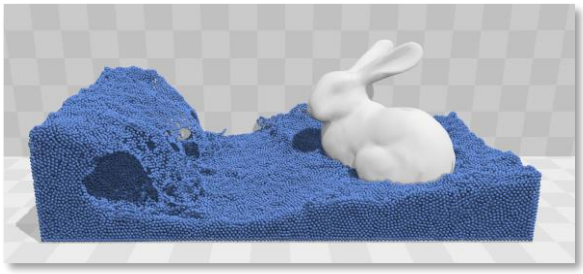

Figure 7. Underlying simulation particles [7]

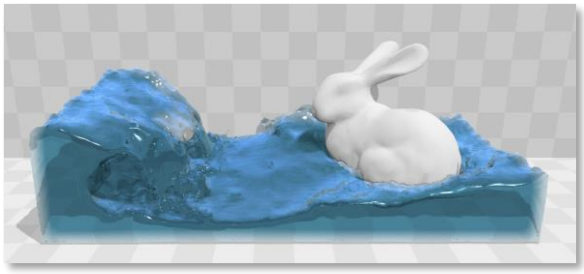

Figure 8. Real-time rendered fluid surface [7]

The following table (Table 1) are performance measurement results extracted from the paper.

Scene	Particles	Steps/frame	Iters/step	Time/step
Armadillo Splash	128k	2	3	4.2
Dam Break	100k	4	3	4.3
Bunny Drop	80k	4	10	7.8

Table 1. Performance results at a frame time of 16ms [7]

The results from Table 1 originate from three different scenarios, the *Armadillo Splash* scenario, the *Dam Break* scenario and the *Bunny Drop* scenario. In the figures above, Figure 7 and Figure 8, we can witness the visual results from the *Bunny Drop* scenario. In this particular scenario, the number of active particles is exactly 80k, or 80,000. This amount of particles is enough to achieve fluid realism, but for a smaller simulation space.

The paper also present results from breaking down the calculation of one frame. This is essentially an overview of how much time in percentage, from total execution time, a particular simulation step has required to be able to execute.

Step	Armadillo Splash	Dam Break
Integrate	1	1
Create Hash Grid	8	6
Detect Neighbors	28	28
Constraint Solve	38	51
Velocity Update	25	14

Table 2. Frame breakdown [7]

This breakdown gives us an inside look on what exact part needs the most of the execution time. In this experiment, the particles do perform collision detection against other meshes outside of the particle system. This collision detection is included in the *Constraint Solve* step, this gives us a hint of why the *Constraint Solve* part always peaks the list.

These results gives us an overview of what parts of a fluid-particle system you may have to spend that extra time on to optimize, so that it does not cause a bottleneck for the system and thus give us a more stable fluid-particle system overall.

2.3 Evaluation and measurement

Based upon the information from the previous part in this chapter, we see a real-time fluid-particle system and the performance results presented from it, showcasing the power of the GPU. They had a controlled amount of particles, of which they measured the time in milliseconds that it took for each calculation step to process, in the range of a fixed frame time of 16 milliseconds. Their test environment featured an enclosed space of rectangular shape with a pre-defined mesh inside to demonstrate the collision detection of the particles.

In another research paper, the author used a controlled amount of particles and then measured the time it took, in milliseconds, to process each frame. These factors were then taken into consideration when measuring the particle system running on the GPU, with collision detection and without. The collision testing done here was to test each particle against an underlying terrain, no particle inter-collision was executed [8].

This study of previous work shows that the most important factor to measure, when the particle systems are actually to be rendered, is the measurement of time each frame requires to process, in milliseconds.

3. Research questions

To be able to provide the arguments for the point of this thesis, the following research questions were formed:

1. How scalable are fluid-particle systems of today?

   A fluid-particle system is scalable in the sense that, for when a particle system increases its number of particles being active, a performance drop is most likely to be seen. The scalability answers the question of how much of an impact on performance the increase of particles has on the system. By measuring and ultimately finding out the scalability of a fluid-particle system, we are able to put this in the context of real-time games by arguing for whether or not the particle system might be applicable for games.

2. How big of a performance impact do physics calculations have on particle systems, meaning, how taxing is it for the GPU to perform physics calculations?

   By measuring how big of an impact a fluid-particle system with all the fluid physics calculations might have on the GPU, we get a perspective on how much the implementation of a fluid-particle system might have in an application as a whole. When you develop a real-time game, you might actually be able to tell how big of an impact a fluid-particle system would have on the performance of a game as a whole and thus be able to tell if the system is going to bottleneck the engine of the game and require further optimization.

4. Methodology

As stated in previous chapters, I will not utilize any third-party libraries for particle simulations, or even for the collision detection. This gives me the freedom I need to be able to achieve an optimal fluid-particle system for my particular case. Third-party libraries are often limited in the ways of how one can implement them in a system, meaning, the library could be specifically written for a certain rendering API, such as OpenGL, which makes it hard to implement for systems using other APIs. Some libraries could even be tied to a specific graphics card vendor, such as the PhysX library which is developed by NVIDIA.

For this thesis I have chosen to utilize the Direct3D rendering API together with the DirectCompute API. Both are developed by Microsoft Corporation and comes bundled with the DirectX 11 API. The fluid-particle system will be written from the ground up, allowing me to optimize where necessary and perform measurements on every part of the system.

To be able to answer the research questions of this thesis, I will control the amount of particles that are active during a specific experiment session to evaluate the scalability of the fluid-particle system. Scalability in this case means how much the amount of particles being active affects the amount of stress it has on the GPU to be able to execute all the required computations. The system is also flexible enough to be able to exclude the fluid physics calculations being run on the GPU, this is helpful for when answering the research question of how much of an impact these fluid physics calculations have on the GPU.

4.1 Frame rate and frame time calculation

The determining factor that will eventually enable us to conclude how well a particle system scales, is the average frame rate of our real-time fluid-particle application. The average frame time would also be provided, to follow the evaluation methods presented in the previous chapter.

For each experiment, the particle system is simulated during a period of 30 seconds. While running the experiment, we keep track of every new frame that has been rendered, measuring the time it took, in milliseconds, to render it. Thus, we are able to tell how many frames that were actually rendered during this period. After doing this, we may also calculate the average frame time. For each second that has passed, we may also measure how many frames that were rendered during this second, by doing this, we are also eventually able to tell the average frame rate of our experiment.

For the experiment to be considered stable, a minimum value of 30 frames per second, or frame rate, is set. The reason for this particular value is that when an applications goes below 30 in frame rate, the application starts to lose its interactivity, meaning that the amount of frames being rendered per second is not enough for the application to seem smooth, a form of choppiness is introduced.

4.2 Rendering

The particles are also finally rendered to be able to evaluate the result of the fluid simulation. As the goal of this thesis is not to provide a visually pleasing fluid-particle system, the particles are only rendered as spheres. The system is easily extendable to be able to render it using a more sophisticated fluid rendering method.

4.3 Target configuration

The experiments are going to be run on a modern computer system by today's standards. The target configuration is as follows:

- GPU: GeForce GTX 770 (using Shader Model 5.0)
- CPU: Intel® Core™ i7-2700K CPU @ 3.50GHz
- RAM: 16 GB DDR3-1333
- Operating System: Microsoft Windows 8.1

5. Implementation

The fluid-particle system is implemented using DirectX 11. The particles are rendered using the Direct3D 11 rendering API and the particle physics are computed in Compute Shaders using the DirectCompute GPGPU API.

5.1 Fluid physics models

The fluid-particle system implements two different methods for simulating fluid, having the ability to switch between them for comparison purposes.

5.1.1 Discrete Element Method (DEM)

The first method is based on the DEM physics model. This method requires less processing power from the GPU but is very limited in how you can achieve realism for fluid simulations. This first method calculates forces such as spring, damping, shear and attraction forces for every particle, based on the neighbors of this particle. Ultimately, this requires you to only iterate through every neighboring particle once.

The fluid-particle system based on the DEM physics model is divided into a seven-step process. The following pseudo-code (Figure 11) demonstrates the flow of this system and gives the reader an overview of how it is functioning.

Algorithm 1 Fluid-particle simulation - DEM method

```
1:  procedure ClearGridIndices
2:      for all grid indices i do
3:          clear grid index i
4:  procedure BuildGrid
5:      for all particles i do
6:          calculate grid hash from particle i position
7:          insert grid hash and particle i index to grid key-value pair list
8:  procedure SortGrid
9:      while grid key-value pair list != sorted do
10:         sort grid key-value pairs based on their grid hash into ascending order
11: procedure BuildGridIndices
12:     for all particles i do
13:         if particle i is start of cell j then
14:             set cell j start index to particle i
15:         if particle i is end of cell j then
16:             set cell j end index to particle i
17: procedure RearrangeParticles
18:     for all particles i do
19:         put particle i in sorted list from sorted key-value pair
20: procedure CalculateForces
21:     for all particles i do
22:         for all neighboring particles j do
23:             calculate particle i forces based on particle j forces
24: procedure Integrate
25:     for all particles i do
26:         update particle i velocity and position
```

Figure 11. Pseudo code for the fluid-particle system using DEM as the fluid physics model

5.1.2 Smoothed Particle Hydrodynamics

The second method is a more sophisticated one and results in a much more realistic result for fluid-particle systems. This method uses SPH as the physics model. This SPH-based method ultimately requires the particles to iterate through their neighbors twice. In the first iteration, every particle calculates its density based on every neighboring particles density and the distance between them. This calculated density is then taken into consideration in the next iteration which consists of calculating forces for the current particle.

Now, in this next iteration, we have the current particles density, which has been influenced by its neighbors. The remaining thing to do is to calculate forces for this particle. Based on the density of this particle and its neighbors we calculate forces such as pressure and velocity. We also want to enable us to customize the fluid, so we also send in some fluid properties to this calculation, such as viscosity.

The fluid-particle system based on the Smoothed Particle Hydrodynamics physics model is divided into an eight-step process. The following pseudo-code (Figure 12) demonstrates the flow of this system and gives the reader an overview of how it is functioning.

Algorithm 1 Fluid-particle simulation - Smoothed Particle Hydrodynamics

```
1:  procedure ClearGridIndices
2:      for all grid indices i do
3:          clear grid index i
4:  procedure BuildGrid
5:      for all particles i do
6:          calculate grid hash from particle i position
7:          insert grid hash and particle i index to grid key-value pair list
8:  procedure SortGrid
9:      while grid key-value pair list != sorted do
10:         sort grid key-value pairs based on their grid hash into ascending order
11: procedure BuildGridIndices
12:     for all particles i do
13:         if particle i is start of cell j then
14:             set cell j start index to particle i
15:         if particle i is end of cell j then
16:             set cell j end index to particle i
17: procedure RearrangeParticles
18:     for all particles i do
19:         put particle i in sorted list from sorted key-value pair
20: procedure CalculateDensities
21:     for all particles i do
22:         for all neighboring particles j do
23:             calculate particle i density based on particle j densities
24: procedure CalculateForces
25:     for all particles i do
26:         for all neighboring particles j do
27:             calculate particle i forces based on particle j forces
28: procedure Integrate
29:     for all particles i do
30:         update particle i velocity and position
```

Figure 12. Pseudo code for the fluid-particle system using SPH as the fluid physics model

The pseudo codes for both of the methods are only a very rough estimate of how the particle system is functioning. Its purpose is mostly to introduce the reader to the fluid-particle system. As you can see from the pseudo-code from both of them, we keep the particles in a grid of which we update on every timestep. This is so that we may quickly find every particle neighboring particles in our density and forces calculations. Instead of having to iterate through every particle in the entire system, we only have to iterate though every neighboring particle. To be able to quickly iterate through all the neighbors, we also keep track of where in the grid every cell begins and ends, this way we have access to every particle in a given cell. To give the reader a more detailed view of the system, the following headlines presents a detailed view of different key parts of the system.

5.2 Uniform grid spatial data structure

As stated before, the particles are kept in a uniform grid spatial data structure [19] that is updated on each timestep. The basic reasoning behind this is that if we limit the number of particles each particle has to access when performing the physics calculations as much as possible, we are able to achieve as good performance as possible, for this part.

We keep a key-value pair, the key being the grid hash and the value being the particle ID. The grid hash is basically the linear cell ID in our grid, calculated from what the current position of a given particle is. The grid is then sorted by grid hashes in ascending order. To give the reader a simplified look of the grid, Table 3 below demonstrates how the grid is structured in its unsorted state.

Index	Unsorted
0	(0, 1)
1	(1, 4)
2	(0, 6)
3	(4, 7)
4	(2, 3)
5	(2, 5)
6	(1, 2)
7	(3, 9)
8	(5, 8)

Table 3. Unsorted grid

What we see here is that the first step in the grid setup process is to actually insert the key-value pairs into the grid.

5.2.1 Grid sorting

As the key-value pairs are just being linearly inserted into the grid, we can almost guarantee that the list is not being inserted into a sorted order. So, for this grid to actually be useful, we do have to sort this grid based on their keys, which is, if you remember, the grid hashes. The table below (Table 4) demonstrates how the grid is structured in its sorted state.

Index	Sorted
0	(0, 1)
1	(0, 6)
2	(1, 2)
3	(1, 4)
4	(2, 3)
5	(2, 5)
6	(3, 9)
7	(4, 7)
8	(5, 8)

Table 4. Sorted grid

The algorithm used for sorting our grid is a parallel algorithm called *Bitonic mergesort* [20]. The algorithm is excellent for us to use as it is a parallel algorithm, leading to a very good performance potential when executed on the GPU. We parallelize the sorting process as much as we possibly can so that we minimize the execution time required by this process. Sorting is more often than not the culprit in a slow system, if we manage to speed up the sorting process as much as possible, we get a much faster system.

5.2.2 Grid index building

The next step in this whole grid building process is to keep track of where each and every grid hash starts and ends in our grid. The following table demonstrates a simplified look of how the cells keep track of what their start and end particle is. The table is based upon the results from above.

Grid hash	Start	End
0	0	1
1	2	3
2	4	5
3	6	6
4	7	7
5	8	8

Table 5. Grid start and end particle indices

5.3 Particle neighbor finding

As we build our grid from the ground up in every new frame, the particles have new, updated positions of which we must calculate their grid hashes from. The question is, how do we calculate the grid hash? There is a limited number of grid hashes available, dictated by the number of cells we have in our grid, as the grid hash is basically a linear cell ID. Based on the position of a particle, we first have to determine where in our grid, the grid position, this particle exists in. The grid clamps our particles center position to where the corresponding position of a cell is, assuming a particle may only exist in one cell. To be able to make this assumption, the size of one cell may only be as large as the size of our particle, in this case the diameter.

Now we are able to calculate the hash of this cell position, leaving us with a single number which is easily sortable.

Having this linear grid hash for every particle, iterating through every neighbor is now easily done by calculating the cell positions adjacent to the current cell of a particle, in the case of a 3-dimensional world, like in our case, the number of neighboring cells are eight. From these neighboring cell positions we calculate their grid hashes and finally we are able to access the particles by iterating through the grid start position and end position of this calculated hash.

6. Results

These results are based on three different grid size environments with a range of around 64,000 particles to a little bit over one million particles. The tests are executed independently of each other, meaning, one test environment could be to simulate the fluid-particle system with 64k particles using DEM, and another could be using the same method but with 128k particles instead, and so on. The tests were executed using the target configuration system specified in chapter 4.

The y-axis in our graphs represents what the average frame rate was during a period of 30 seconds of simulation time. The x-axis represents the number of particles active during the simulation.

The table below each graph provides us with more detailed information about how well the system performed. Several new aspects are introduced, such as the average frame time in milliseconds.

6.1 A comparison of using DEM and SPH

The following results are presented in such a way that the two fluid physics models are easily comparable with each other. As previously explained, the tests are executed independently of each other. The tests are distinguished from each other by these three factors:

- Grid size
- Particle amount
- Fluid physics model

The fluid physics models are being run using the same conditions; the environment of the experiment stays the same regardless of what method is being used.

6.1.1 64^3-dimensional grid (262,144 cells)

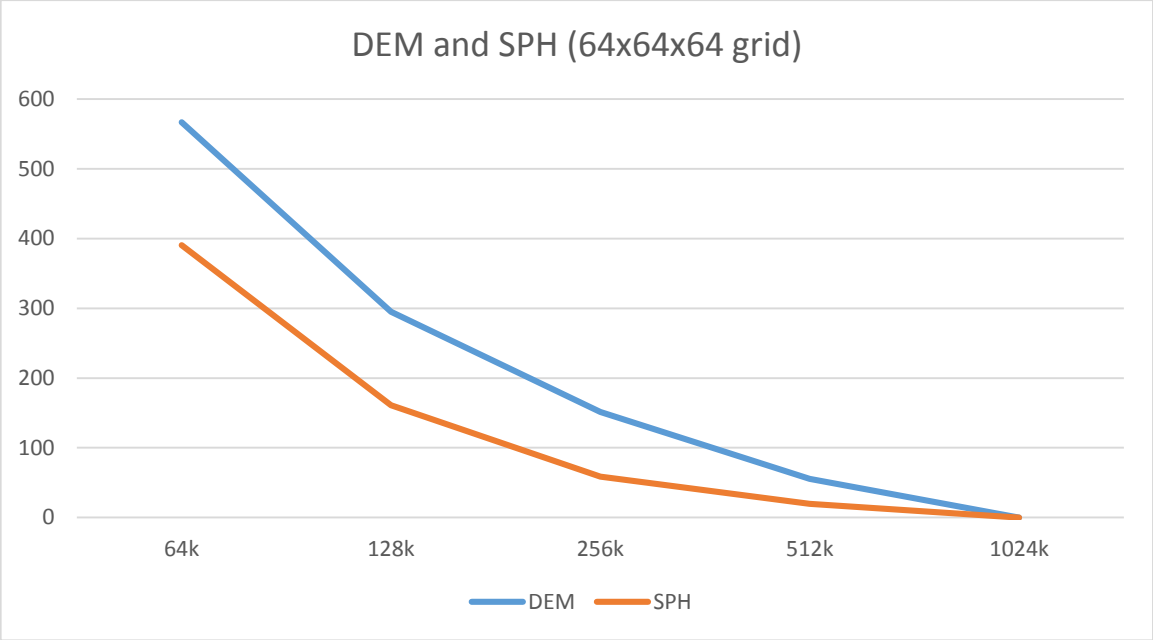

Figure 13. Line chart displaying the average frame rate results for a given particle amount

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
DEM	64k	435	688	566.667	1.7647
DEM	128k	245	392	295.033	3.3894
DEM	256k	139	186	151.133	6.6167
DEM	512k	45	84	55.433	18.0397
DEM	1024k	0	0	0	0

Table 6. Results from using DEM

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
SPH	64k	339	580	390.567	2.5604
SPH	128k	136	312	160.900	6.2150
SPH	256k	24	153	58.700	17.0358
SPH	512k	1	62	19.733	50.6757
SPH	1024k	0	0	0	0

Table 7. Results from using SPH

6.1.2 128^3-dimensional grid (2,097,152 cells)

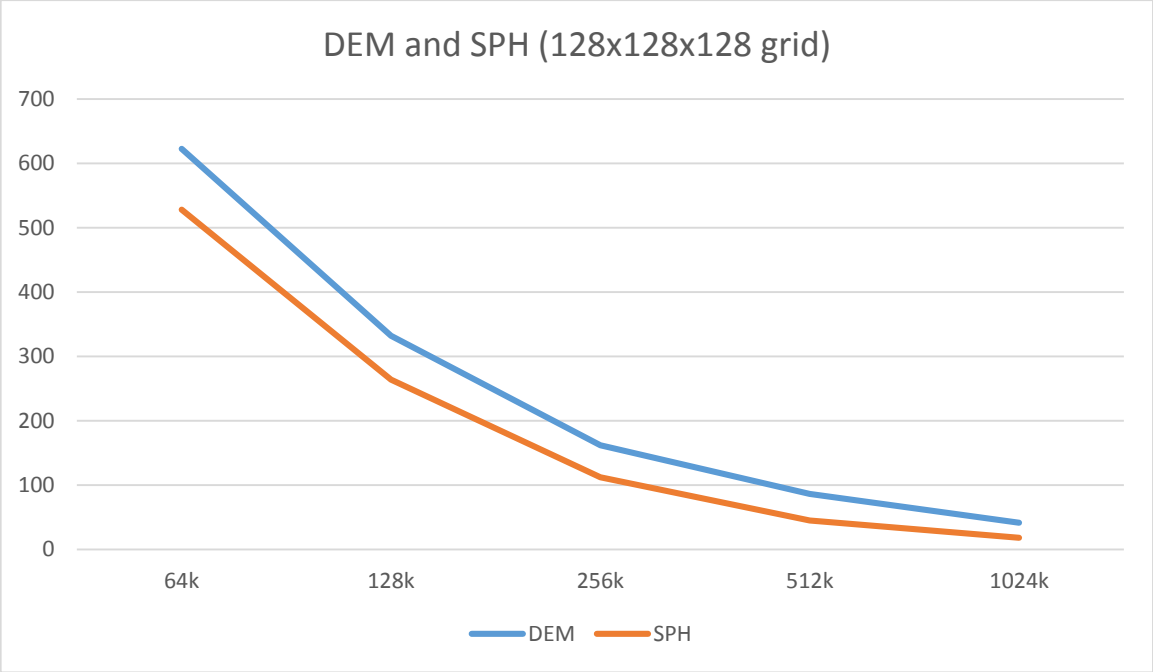

Figure 14. Line chart displaying the average frame rate results for a given particle amount

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
DEM	64k	428	719	622.900	1.6054
DEM	128k	241	376	332.300	3.0093
DEM	256k	140	179	162.000	6.1728
DEM	512k	80	99	86.300	11.5875
DEM	1024k	40	44	41.867	23.8853

Table 8. Results from using DEM

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
SPH	64k	326	625	528.333	1.8927
SPH	128k	208	292	263.800	3.7907
SPH	256k	101	149	112.100	8.9206
SPH	512k	29	84	44.933	22.2552
SPH	1024k	10	39	18.100	55.2486

Table 9. Results from using SPH

6.1.3 256^3-dimensional grid (16,777,216 cells)

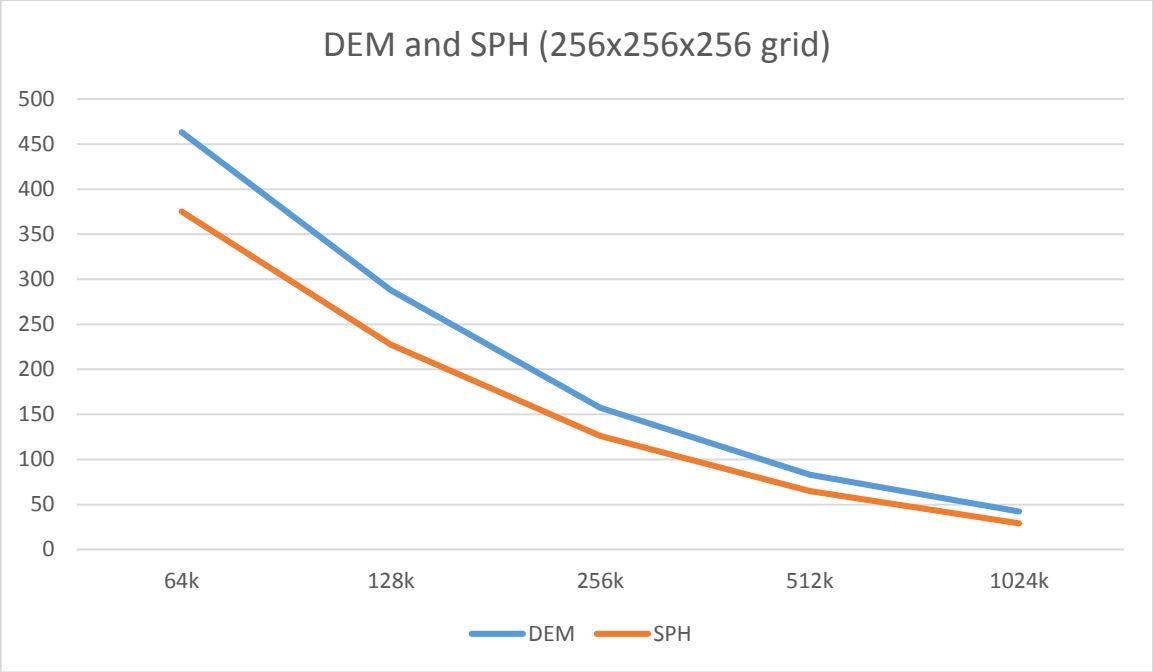

Figure 15. Line chart displaying the average frame rate results for a given particle amount

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
DEM	64k	285	564	463.167	2.1590
DEM	128k	185	344	287.900	3.4734
DEM	256k	140	179	162.000	6.3627
DEM	512k	80	99	86.300	12.0433
DEM	1024k	37	45	42.233	23.6779

Table 10. Results from using DEM

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
SPH	64k	207	481	375.267	2.6648
SPH	128k	121	279	227.367	4.3982
SPH	256k	70	149	125.933	7.9407
SPH	512k	44	75	64.767	15.4400
SPH	1024k	27	32	29.067	34.4037

Table 11. Results from using SPH

6.2 Excluded grid and fluid physics calculations

This time we have excluded the grid building process and the fluid physics calculations and instead we only calculate each particles velocity and position based on gravity.

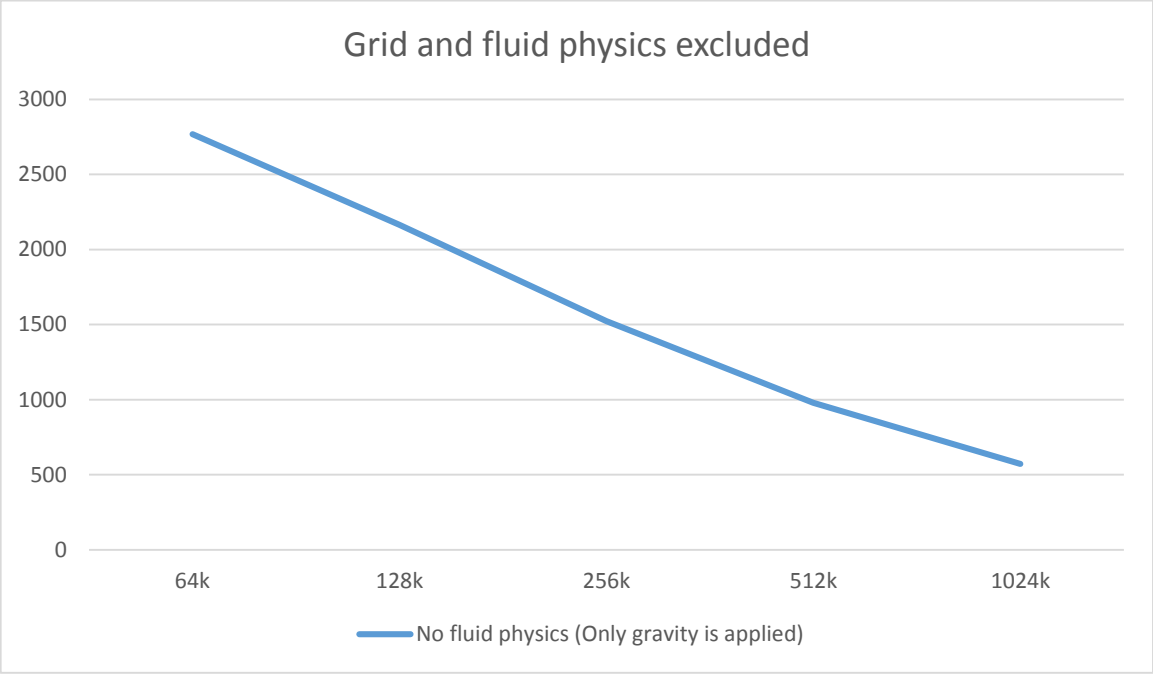

Figure 16. Line chart displaying the average frame rate results for a given particle amount

Method	Particles	Min frame rate	Max frame rate	Avg. frame rate	Avg. frame time (ms)
None	64k	1435	3032	2769.367	0.3611
None	128k	1053	2379	2164.767	0.4619
None	256k	778	1670	1523.467	0.6564
None	512k	568	1052	979.600	1.0208
None	1024k	399	604	571.733	1.7490

Table 12. Results from only applying gravity

7. Discussion

When studying the presented graphs and tables in the previous chapter, you may notice that there is an optimal grid size for a specific amount of particles. For example, 64k particles performs worse when simulated in a 128^3 grid than it does in a 64^3 grid, it performs even worse when simulated in a 256^3 grid compared to in a 128^3 grid. This observation applies to both of the methods. The reason for this is because the potential of a performance gain stops at when the grid is the size of 64^3, the particles would not theoretically be able to fill up the grid over the size of 64^3, it would not even be able to fill up a grid when over the size of 32^3, assuming the grid has a size that is the power of 2. The smallest grid we simulated was at a size of 64^3, this is not beneficial for when our particles is in the number of 64k, it may even have been a performance decrease compared to having a grid in the size of 32^3. The reason for this minimum size is because we have focused on simulating a large number of particles.

One might wonder why there is such a thing as an optimal grid size for a particular amount of particles. The particles iterates through every neighboring particle, meaning, they iterate through every neighboring cell, including the current cell it is located in. For each cell we then have to iterate through each particle located in this cell, this could potentially be a lot of iterations for every particle. The best case scenario would be, that the particles are evenly spread out in every cell, every cell contains the same amount of particles as every other cell. This would lead to an even amount of iterations for every cell. We basically want to balance the amount of neighbors there potentially could be for each particle, while utilizing as many cells as possible for the particles in our grid. The grid is very static in this sense, there is no way to predict the behavior of the fluid-particle system, potentially leading to a few cells that are being overwhelmed with particles while the rest of the cells contains only a few particles to none. This could also possibly explain the fluctuating frame rate, the minimum and maximum frame rate numbers are found to be very distant from each other.

A general observation of our performance results is that using SPH is slower than using DEM. This result is logical and was expected, as SPH requires iterating over neighboring particles twice instead of once as when using DEM. However, SPH yielded a more realistic fluid simulation, as opposed to when using DEM. This decrease in performance may or may not be worth the realism of the simulation, depending on the situation.

There are also situations when the amount of particles are simply too many to fit in a grid with a specific grid size, and lower. By studying our results, we see that these situations have

occurred when the grid was created with the grid size of 64^3. When simulating an extremely large number of particles, in this case, 1024k particles, the particles would simply not fit into our grid. The calculations performed would only cause numerical explosions rendering the system unstable and ultimately crashing.

If we study the results from when we completely excluded the fluid simulation and grid building process, we witness some extreme performance results. The performance results are almost in a linear decline by increasing the number of particles, making it a very consistent system performance-wise, as opposed to when including the fluid simulation and grid building process, which requires measurements to ultimately find which grid size is optimal for a specific amount of particles. The linear decline is a logical outcome for this system, as for each experiment, the amount of particles is being doubled, and there is no underlying grid or neighboring particles affecting the performance of the system. The non-linear decline of performance of the fluid-particle system could be explained by using our motivation for the optimal grid size explained earlier in this chapter. The performance decline is actually less severe than the decline of the particle system using no fluid physics. As the particle amount increases, every cell in the grid has a higher potential of being utilized by the fluid-particle system, leading to a more effectively utilized grid.

8. Conclusion

When developing fluid-particle systems you have to be wary of the many performance pitfalls. The ultimate goal is to balance performance with fluid simulation realism. If using a grid-based solution, the size of the grid would have to be adjusted to fit the amount of particles being simulated for best performance results.

If we take a look at our first research question, "How scalable are fluid-particle systems of today?", we are able to conclude that when using a grid-based solution using either DEM or SPH to calculate the fluid physics, even when particles are in great numbers, such as in the number of just over a million, we receive acceptable performance results given the frame rate, in our case.

By looking at the performance differences between the particle system with only gravity applied and the other two using the two different fluid physics calculation methods we may also answer our second research question. The performance differences are enormous, as we may have expected. The simple particle system with only gravity affecting it does not require any grid building, sorting, neighbor finding, collision detection and response.

Ultimately, one has to ask oneself, "Do I strive for fluid realism or do I strive for quantity and performance?" Based on the outcome of this question you might find that developing a fluid-particle system using a more simple method than using SPH, or even DEM, is more than enough for you. You might have found that the system is visually pleasing and the performance fits your application well. Nevertheless, there are no obstacles for games to use physically based fluid-particle systems.

8.1 Future work

As the particle system presented in this thesis is only rendered using a simple method of just rendering each particle as a sphere, future work could include finding different fluid rendering methods and comparing the performance results and their visual results, and also measure the performance impacts these methods have on the fluid-particle system, if any significant, compared to as only rendering them as spheres.

Future work could also include finding the most optimal data structure to store the particles in, thus finding the most optimal way to perform particle neighbor finding. This thesis uses a relatively simple method of storing particles and finding their neighbors, it is very likely for there to exist much more optimal ways to do this. If one could do this, and maybe even together

with an even faster sorting algorithm, the number of particles being active in a simulation would be even greater than what we have reached in the experiments of this thesis.

9. References

1. Keenan Crane, Ignacio Llamas and Sarah Tariq. 2007. *Real-Time Simulation and Rendering of 3D Fluids*. Retrieved April 25, 2014 from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html

2. Peter Kipfer and Rüdiger Westermann. 2006. *Realistic and Interactive Simulation of Rivers*. GI '06 Proceedings of Graphics Interface 2006, 41-48.

3. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone and James C. Phillips. 2008. *GPU Computing*. Proceedings of the IEEE, (Volume 96, Issue 5), 879-899.

4. Miles Macklin and Matthias Müller. 2013. *Position Based Fluids*. ACM Transactions on Graphics (TOG), (Volume 32, Issue 4) (July 2013), Proceedings of ACM SIGGRAPH 2013.

5. Erwin Coumans. 2013. *GPU Rigid Body Simulation*. Game Developers Conference 2013.

6. Tianyun Ni. 2009. *Direct Compute – Bring GPU Computing to the Mainstream*. GPU Technology Conference 2009.

7. Creative Computing. 1981. *The origin of Spacewar*. Retreived June 24, 2014 from http://www.wheels.org/spacewar/creative/SpacewarOrigin.html

8. Pyarelal Knowles. 2009. *GPGPU Based Particle System Simulation*. RMIT University.

9. Trona M. Roy. 1995. *Physically-Based Fluid Modeling using Smoothed Particle Hydrodynamics*. University of Illinois at Chicago. Retrieved June 29, 2014 from http://www.plunk.org/~trina/thesis/html/thesis_toc.html

10. Microsoft. 2014. *Compute Shader Overview*. Retrieved June 29, 2014 from http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx

11. Lutz Latta. 2007. *Everything about Particle Effects*. Game Developers Conference 2007.

12. Andrew Burnes. 2013. *Hawken: Advanced PhysX Effects Coming Soon To F2P Shooter*. Retrieved July 2, 2014 from http://www.geforce.com/whats-new/articles/hawken-physx

13. Valve Corporation. 2011. *Portal 2*. Retrieved July 2, 2014 from http://store.steampowered.com/app/620/

14. Direct to video. 2009. *A thoroughly modern particle system*. Retrieved July 2, 2014 from http://directtovideo.wordpress.com/2009/10/06/a-thoroughly-modern-particle-system/

15. D. Breen, M. Lin. 2003. *Particle-Based Fluid Simulation for Interactive Applications*. SIGGRAPH Symposium on Computer Animation (2003).

16. Harada Takahiro. 2007. *Real-Time Rigid Body Simulation on GPUs*. Addison Wesley.

17. Matthias Müller. 2003. *Particle-Based Fluid Simulation for Interactive Applications*. Eurographics/SIGGRAPH Symposium on Computer Animation (2003).

18. Simon Green. 2013. *Particle Simulation using CUDA*. NVIDIA Corporation.

19. Christer Ericson. 2005. *Real-Time Collision Detection*. Morgan Kaufmann.

20. Hans Werner Lang. 2009. *Bitonic sorting network for n not a power of 2*. FH Flensburg. Retrieved August 17, 2014 from http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm

21. J. J. Monaghan. 1992. *Smoothed Particle Hydrodynamics*. Annual Review of Astronomy and Astrophysics. Vol. 30, p. 543-574.

22. B.K Mishra. 2003. *A review of computer simulation of tumbling mills by the discrete element method: Part I—contact mechanics.* International Journal of Mineral Processing. Vol. 71, Issues 1-4, p. 73-93.

10. Appendices

10.1 Compute shader for the SPH forces calculations

```
float CalculatePressure(float density)
{
	// Implements this equation:
	// Pressure = B * ((rho / rho_0)^y  - 1)
	return g_fPressureStiffness * max(pow(density / g_fRestDensity, 3) - 1, 0);
}

float3 CalculateGradPressure(float r, float P_pressure, float N_pressure,
	float N_density, float3 diff)
{
	const float h = g_fSmoothlen;
	float avg_pressure = 0.5f * (N_pressure + P_pressure);
	// Implements this equation:
	// W_spkiey(r, h) = 15 / (pi * h^6) * (h - r)^3
	// GRAD( W_spikey(r, h) ) = -45 / (pi * h^6) * (h - r)^2
	// g_fGradPressureCoef = fParticleMass * -45.0f / (PI * fSmoothlen^6)
	return g_fGradPressureCoef * avg_pressure / N_density * (h - r) * (h - r) / r
		* (diff);
}

float3 CalculateLapVelocity(float r, float3 P_velocity, float3 N_velocity,
	float N_density)
{
	const float h = g_fSmoothlen;
	float3 vel_diff = (N_velocity - P_velocity);
		// Implements this equation:
		// W_viscosity(r, h) = 15 / (2 * pi * h^3) *
		// (-r^3 / (2 * h^3) + r^2 / h^2 + h / (2 * r) - 1)
		// LAPLACIAN( W_viscosity(r, h) ) = 45 / (pi * h^6) * (h - r)
		// g_fLapViscosityCoef = fParticleMass * fViscosity * 45.0f /
		// (PI * fSmoothlen^6)
		return g_fLapViscosityCoef / N_density * (h - r) * vel_diff;
}
```

```
[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
        uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
        const unsigned int ID = DTid.x;

        // Get current particle properties
        float3 position = Particles[ID].position;
        float3 velocity = Particles[ID].velocity;
        float density = ParticlesDensity[ID].density;
        float pressure = CalculatePressure(density);

        const float h_sq = g_fSmoothlen * g_fSmoothlen;

        float3 acceleration = float3(0.0f, 0.0f, 0.0f);

        // Calculate acceleration based on neighbors from the 8 adjacent cells
        // and current cell
        int3 gridPos = CalcGridPos(position, originPosW, cellSize);
        uint gridHash = CalcGridHash(gridPos, gridSize);

        // Iterate through every neighboring cell (including the current cell)
        for (int z = -1; z <= 1; ++z)
        {
                for (int y = -1; y <= 1; ++y)
                {
                        for (int x = -1; x <= 1; ++x)
                        {
                                int3 neighborGridPos = gridPos + int3(x, y, z);
                                uint neighborGridHash = CalcGridHash(
                                neighborGridPos,
                                gridSize);

                                uint2 start_end = GridIndices[neighborGridHash];

                                for (unsigned int i = start_end.x; i < start_end.y; ++i)
                                {
                                        float3 neighborPos = Particles[i].position;
                                        float3 diff = neighborPos - position;
                                        float r_sq = dot(diff, diff);

                                        if (r_sq < h_sq && ID != i)
                                        {
                                                float3 neighborVel = Particles[i].velocity;
                                                float neighborDensity =
                                                        ParticlesDensity[i].density;
                                                float neighborPressure =
                                                        CalculatePressure(neighborDensity);
                                                float r = sqrt(r_sq);

                                                // Pressure term
                                                acceleration += CalculateGradPressure(
                                                        r,
                                                        pressure,
                                                        neighborPressure,
                                                        neighborDensity,
                                                        diff);

                                                acceleration += CalculateLapVelocity(
                                                        r,
                                                        velocity,
                                                        neighborVel,
                                                        neighborDensity);
                                        }
                                }
                        }
                }
        }

        // Update forces with the calculated ones
        ParticlesForces[ID].acceleration = acceleration / density;
}
```

10.2 Compute shader for the SPH density calculations

```
float CalculateDensity(float r_sq)
{
	const float h_sq = g_fSmoothlen * g_fSmoothlen;
	// Implements this equation:
	// W_poly6(r, h) = 315 / (64 * pi * h^9) * (h^2 - r^2)^3
	// g_fDensityCoef = fParticleMass * 315.0f / (64.0f * PI * fSmoothlen^9)
	return g_fDensityCoef * (h_sq - r_sq) * (h_sq - r_sq) * (h_sq - r_sq);
}

[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
	uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
	const unsigned int ID = DTid.x;
	const float h_sq = g_fSmoothlen * g_fSmoothlen;
	float3 position = Particles[ID].position;

	float density = 0.0f;

	// Calculate the density based on neighbors from the 8 adjacent cells
	// and current cell
	int3 gridPos = CalcGridPos(position, originPosW, cellSize);
	uint gridHash = CalcGridHash(gridPos, gridSize);

	// Iterate through every neighboring cell (including the current cell)
	for (int z = -1; z <= 1; ++z)
	{
		for (int y = -1; y <= 1; ++y)
		{
			for (int x = -1; x <= 1; ++x)
			{
				int3 neighborGridPos = gridPos + int3(x, y, z);
				uint neighborGridHash = CalcGridHash(
				neighborGridPos,
				gridSize);

				uint2 start_end = GridIndices[neighborGridHash];

				for (unsigned int i = start_end.x; i < start_end.y; ++i)
				{
					float3 neighborPos = Particles[i].position;

					float3 diff = neighborPos - position;
					float r_sq = dot(diff, diff);

					if (r_sq < h_sq)
					{
						density += CalculateDensity(r_sq);
					}
				}
			}
		}
	}

	// Update density with the calculated one
	ParticlesDensity[ID].density = density;
}
```

10.3 Compute shader for the DEM forces calculations

```
[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
        uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
        const unsigned int ID = DTid.x;

        float3 position = Particles[ID].position;
        float3 velocity = Particles[ID].velocity;

        int3 gridPos = CalcGridPos(position, originPosW, cellSize);

        float3 acceleration = float3(0.0f, 0.0f, 0.0f);

        for (int z = -1; z <= 1; z++)
        {
                for (int y = -1; y <= 1; y++)
                {
                        for (int x = -1; x <= 1; x++)
                        {
                                int3 neighbourGridPos = gridPos + int3(x, y, z);
                                uint neighborGridHash = CalcGridHash(
                                neighbourGridPos,
                                gridSize);

                                uint2 start_end = GridIndices[neighborGridHash];

                                for (unsigned int i = start_end.x; i < start_end.y; ++i)
                                {
                                        if (i != ID)
                                        {
                                                Particle curParticle = Particles[i];
                                                float3 relPos = curParticle.position - position;
                                                float dist = length(relPos);
                                                float collideDist = g_fSphereRadius
                                                        + g_fSphereRadius;

                                                float3 force = float3(0.0f, 0.0f, 0.0f);

                                                if (dist < collideDist)
                                                {
                                                        float3 norm = relPos / dist;

                                                        // Relative velocity
                                                        float3 relVel = curParticle.velocity
                                                                - velocity;
                                                        // Relative tangential velocity
                                                        float3 tanVel = relVel
                                                                - (dot(relVel, norm) * norm);
                                                        // Spring force
                                                        force = -g_fParamsSpring
                                                                * (collideDist - dist) * norm;
                                                        // Dashpot (damping) force
                                                        force += g_fParamsDamping * relVel;
                                                        // Tangential shear force
                                                        force += g_fParamsShear * tanVel;
                                                        // Attraction
                                                        force += g_fParamsAttraction * relPos;

                                                        acceleration += force;
                                                }
                                        }
                                }
                        }
                }
        }

        ParticlesForces[ID].acceleration = acceleration;
}
```

10.4 Compute shader for the grid building

```
[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
	uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
	const unsigned int ID = DTid.x;

	float3 position = Particles[ID].position;

	// Calculate which grid cell this particle belongs to
	int3 gridPos = CalcGridPos(position, originPosW, cellSize);
	// Grid hash means cell index in this case
	uint gridHash = CalcGridHash(gridPos, gridSize);

	GridKeyValuePair[ID] = uint2(gridHash, ID);
}
```

10.5 Compute shader for the grid index building

```
[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
	uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
	// The grid key value pair is now a sorted list consisting of
	// (grid hash, particle id)
	// Example: 0:(0,1), 1:(0,4), 2:(1,6), 3:(1,3), 4:(2,7), 5:(3,9)
	const unsigned int G_ID = DTid.x; // Grid ID (Key value pair) to operate on
	unsigned int G_ID_PREV = (G_ID == 0) ? g_iNumElements : G_ID; G_ID_PREV--;
	unsigned int G_ID_NEXT = G_ID + 1;
	if (G_ID_NEXT == g_iNumElements) { G_ID_NEXT = 0; }

	uint cell = GridKeyValueRO[G_ID].x;
	uint cell_prev = GridKeyValueRO[G_ID_PREV].x;
	uint cell_next = GridKeyValueRO[G_ID_NEXT].x;

	if (cell != cell_prev)
	{
		// The cell starts at this index in our grid
		GridIndicesRW[cell].x = G_ID;
	}

	if (cell != cell_next)
	{
		// The cell ends at this index in our grid
		GridIndicesRW[cell].y = G_ID + 1;
	}
}
```

10.6 Compute shader for the grid index clearing

```
[numthreads(BLOCKSIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID,
	uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex)
{
	GridIndicesRW[DTid.x] = uint2(0, 0);
}
```

10.7 Compute shader for the Bitonic mergesort algorithm

```
[numthreads(BITONIC_BLOCK_SIZE, 1, 1)]
void main(uint3 Gid : SV_GroupID,
	uint3 DTid : SV_DispatchThreadID,
	uint3 GTid : SV_GroupThreadID,
	uint GI : SV_GroupIndex)
{
	// Load shared data
	shared_data[GI] = Data[DTid.x];
	GroupMemoryBarrierWithGroupSync();

	// Sort the shared data
	for (unsigned int j = g_iLevel >> 1; j > 0; j >>= 1)
	{
		uint2 result = ((shared_data[GI & ~j].x <= shared_data[GI | j].x)
			== (bool)(g_iLevelMask & DTid.x)) ?
			shared_data[GI ^ j] : shared_data[GI];
		GroupMemoryBarrierWithGroupSync();
		shared_data[GI] = result;
		GroupMemoryBarrierWithGroupSync();
	}

	// Store shared data
	Data[DTid.x] = shared_data[GI];
}
```