Thesis no: BGD-2014-04

Particle Systems Using 3D Vector Fields

with OpenGL Compute Shaders

Johan Anderdahl
Alice Darner

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 in Digital Game Development. The thesis is equivalent to 10 weeks of full-time studies.

Contact Information:
Author(s):
Johan Anderdahl
E-mail: johan.anderdahl@gmail.com
Alice Darner
E-mail: alice.darner@gmail.com

University advisor:
Stefan Petersson
Dept. of Creative Technologies

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

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

Abstract

Context. Particle systems and particle effects are used to simulate a realistic and appealing atmosphere in many virtual environments. However, they do occupy a significant amount of computational resources. The demand for more advanced graphics increases by each generation, likewise does particle systems need to become increasingly more detailed.
Objectives. This thesis proposes a texture-based 3D vector field particle system, computed on the Graphics Processing Unit, and compares it to an equation-based particle system.
Methods. Several tests were conducted comparing different situations and parameters for the methods. All of the tests measured the computational time needed to execute the different methods.
Results. We show that the texture-based method was effective in very specific situations where it was expected to outperform the equation-based. Otherwise, the equation-based particle system is still the most efficient.
Conclusions. Generally the equation-based method is preferred, except for in very specific cases. The texture-based is most efficient to use for static particle systems and when a huge number of forces is applied to a particle system. Texture-based vector fields is hardly useful otherwise.

Keywords: Particle Systems, Vector Fields, GPGPU, Textures.

Acknowledgments

We would like to sincerely thank our supervisor, **Stefan Petersson** for his invaluable support and advice during this project. Without his help this project would never have been possible. We would also thank him for letting us borrow the Nvidia GTX 660 GPU used in this thesis work.

Many people provided feedback and helped us through the project, whom we also would like to give our sincerest gratitude to. This includes but is not exclusive to:

Tim Henriksson, **Daniel Bengtsson**, **Joel Svensson** and **Kim Restad** who gave us advice, technical help and support.
Marie Klevedal, who helped us with mathematical notation and gave us vital support.

Thank you,
Alice Darner
Johan Anderdahl

Contents

Chapter 1

Introduction

1.1 Background

Particles in large quantities often take an important part of dramatic atmospheres around us and in dramatic media and settings, such as films and photography. For example dust, snow, rain and sparkles are common when creating a certain setting. Due to this, the use of particles and particle systems are often beneficial when creating realistic or visually appealing virtual environments. However, the rendering of such systems on the Central Processing Unit (CPU), is usually very ineffective. The Graphical Processing Unit (GPU), has the potential to take its place with its powerful paralleling computational power [1].
One way to use the power of the GPU is to perform the same calculation that a physician would, by using vector fields [2][3]. Vector fields were first introduced by Michael Faraday, and is used in modern physics, meteorology and mathematics. For example, light, magnetic fields and complex weather systems can all be described by vector fields [4]. In this thesis work, we will compare different ways to use the vector fields on the GPU to create complex but fast particle effects.

1.1.1 Related Work

The idea to use vector fields with particle systems and other interactive animation is far from new. Early works includes the research by Hilton-Egbert[3] about how vector fields could be used in order to create an interactive tool, albeit on the CPU. They have several types of forces and includes demitting (see Chapter 2) as a force. Unlike their implementation, this thesis work excludes emitting and demitting from the systems, and some of the forces used in have been generalized. The system used by Hilton-Egbert also use mass and a center of mass on their particles which have been omitted as well from this work.
Established corporations within the game industry, like Epic Games with their Unreal Engine 4 [5][6] and Sucker Punch Productions with their game inFAMOUS Second Son [7], already take use of vector field simulations for their particle systems. inFAMOUS Second Son also has their entire visual effects system on the GPU [7].

Autodesk 3ds Max also has support for vector fields in their application. It can both be used for particle system simulation and as crowd simulation [8].

A path finding solution named flow fields uses vector fields as a way to make agents know which way to go depending on their current position. The game Supreme Commander 2, developed by Gas Powered Games, is a real-time strategy game which uses flow fields for calculating the movements of the units in the game [9].

The use of Vector Fields and particles systems spans further that to games, as Smith shows in his work [2]. Smith implemented a simulation of a sandstorm and the effects on a helicopter using a GPU-based particle system, mostly for use in VR. In his work, Smith is using only 3D textures for storage of the vector field, which is motivated by the usefulness of internal methods for dealing with textures, such as UV-tiling and indexing of textures. His work do not go further into the efficiency of such vector field.

1.2 Hypothesis and Research Questions

- While using particle systems with vector fields, is using a 3D texture faster by computational time than using the vector fields equation?
- When is using a 3D texture more relevant than using the equation?
- Could another type of texture resource type, in essence a 2D texture array, be even more efficient by computational time than a 3D texture?

The hypothesis is that if the number of particles increases to a large number or if there is a lot of forces (see Chapter 2 and Appendix B) applied to the system, a texture-based particle system should become more efficient. If there is only a small number of particles or forces or the area of the particle system needs to be very detailed, the use of on-the-fly equations will probably become more efficient in the long run.

1.3 Purpose

In order to lay a foundation for future complex and aesthetically pleasing particle systems, two systems are implemented using Vector Fields on the GPU. They are then compared by computational time. One of the systems uses a 3D texture for storage of the vector field, while the other computes the vector field each frame. The latter is called an equation based system through this thesis.

1.4 Method

The prototype is implementated using C++ and the OpenGL graphics API. In order to compare the two main implementations, both of them will utilize the GPU, where OpenGLs compute shaders are used. The experimentation is quantitative: multiple versions reflecting different situations are implemented and timed, in order to compare the efficiency of the different kinds of particle systems.

Several tests are executed, comparing the different parameters that can affect the efficiency of a particle system, for example work groups, number of particles and the different forces applied to the particles. Tests are also performed on a GPU of the previous generation, in order to compare it with a more modern one. This comparison is used for a simple makeshift prediction about the future of GPU-based particle systems. Two sub-types of texture-based vector field systems are included in the experiments: 2D texture array and 3D texture, and those are compared with straight-on equations.

Vector Fields

The motion of the system is described with vector fields. In essence, a vector field is an equation that **for any given point within a set space is represented by a vector**.

A simple variation, closely related to a linear equation in an ordinary non-vector system in a 2-dimensional space is:

$x\bar{v} + y\bar{u} - m$

where $\bar{v}$ and $\bar{u}$ are identity vectors and m is a constant. x and y is simply variables which is used to find the certain vector in the vector system. They are any point within the set space.

In this particular example, let m be equal to zero. Then, to find out what vector is at the point $(2, 3)$:

$2\bar{v} + 3\bar{u} = 2(\overline{1, 0}) + 3(\overline{0, 1}) = (\overline{2, 3})$

Vector fields may seem quite trivial when approached with this kind of simplicity, but, just like any other mathematical tool, they become more complex when used for describing more complex equations. See Appendix B for the equations used for the forces in the prototype implementation. See Figure 1.1 for an example of a 3D vector field, used in the prototype.

Three variations of Vector fields are to be compared:

- Texture-based methods: 2D texture array and 3D texture.
- Equation-based vector fields without using texture as a medium.

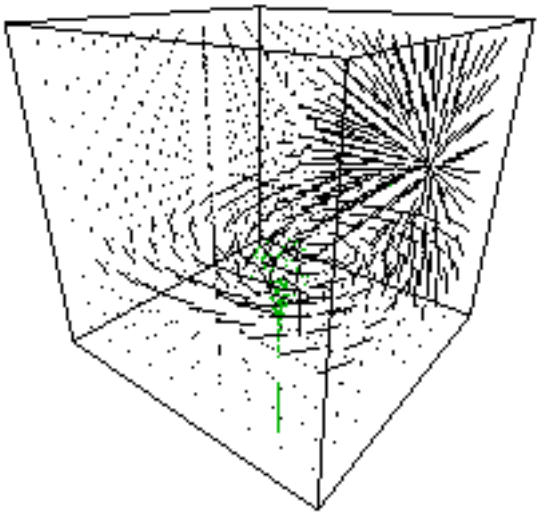


Figure 1.1: Some particles moving through a vector field.

2D Texture Array

2D texture arrays works somewhat differently than a 3D texture. The advantage of a 3D texture is that it is easy to find the correct texel and all of the nearby texels within the texture. 2D texture arrays is in essence multiple 2D Textures that builds up the 3D texture. This makes sampling a more complex task. As the 2D texture arrays are slightly different from 3D textures, the results may be slightly different as well. If that is the case, it is definitely a relevant test to perform. To extend the relevance, both textures have been utilized.

1.5 Delimitations

Measurements

All measurements are conducted with the glQuery command which is included in OpenGL since version 1.5. It was noticed that glQuery does not always give the exact time during the tests. To compensate for the errors of margins, the tests were run over several iterations and taken the average of the computation time; Each test is iterated 100 times before measuring and the next 100 iterations are averaged into the final measured value.

Setup

The comparisons are made on two identical computers, different only by their GPU. For the full set up, see Section 4.1 on page 11.

Emitters

The particle systems are using an exact number of particles for each test. The test omits the use of emitters, even if the source code for one was implemented. There are several reasons for ignoring this code, including:

- The use of an emitter would render inexact and unreliable results for the test.
- The emitters in the implementation use atomic counters. The remaining code, works for almost any GPU that supports opengl 4.4 and compute shaders.
- The use of emitters is not necessary to answer the research questions and to confirm or disprove the hypothesis.

Other limitations

The textures have the ability to use sampling, a collection of data from several pixels close to the sampling point. In the case this thesis work explores, the pixels are exchanged for vectors. While sampling is easy in the case of a 3D texture, it is quite a bit more complicated for a 2D texture array.

The use of sampling is excluded from this thesis work, while it has some relevance to the efficiency of the methods, it is not a necessary part of a vector field-based particle system. The textures are only used to store the vectors that modifies the translation of the particles.

Method Evaluation

There is several advantages and disadvantages for both of the systems. An equation-based particle system has the major advantage that it is boundless opposed to a texture-based particle system. It also does not need to store vectors in a texture meaning that less Video RAM (VRAM) is used. A texture-based system is by its definition bounded by the size of the texture. On the other hand, a texture-based particle system does have an advantage when implementing a non-dynamic particle system. A static vector field can be generated once and stored in a texture without the need to update every frame.

Chapter 2

Particle Systems

2.1 General

The main idea of particle systems in computer science is the approximation of a soft or "fluffy" model [10]. From this perspective, a cloud is just a model, but each particle in it makes up the whole cloud. The "fluffiness" of a particle system or a soft object is not necessarily defined by appearance, but rather by its boundaries: Nothing hinders one from creating a system with boulders rather than smoke, but each boulder is an extension of the soft model, which grows and decreases dynamically depending on the position or the spawn/death rate of the particles.

Usually, a particle system is described with some or all of the following components:

- Particle: the smallest component of the "fluffy" model itself. A separate model which has its own parameters, that varies heavily depending on the implementation. Commonly this includes lifespan, age and velocity. The simplest, earliest forms of particle systems were nothing more than the pixels appearing when the player successfully hit an asteroid in the game Asteroids (see Figure 2.1). From there the definition grew into assigning smaller objects to the particles.

- Emitter: Sometimes called a spawn. This is the point, vector, plane or volume where particles appear after being "born". Emitters are sometimes used together with a demitter/despawn, which "kills" the particles.

- Bounding volume: In extension, this could either be seen as the soft model itself, or as a form of demitter. When a particle hits the edge of its bounding box, it either despawn or collides.

- Force: Describes the motion within the system and is applied to the relevant particles.

The smallest visible components of a particle system is the particles. Depending on the implementation, they can use a wide range of parameters, for example:

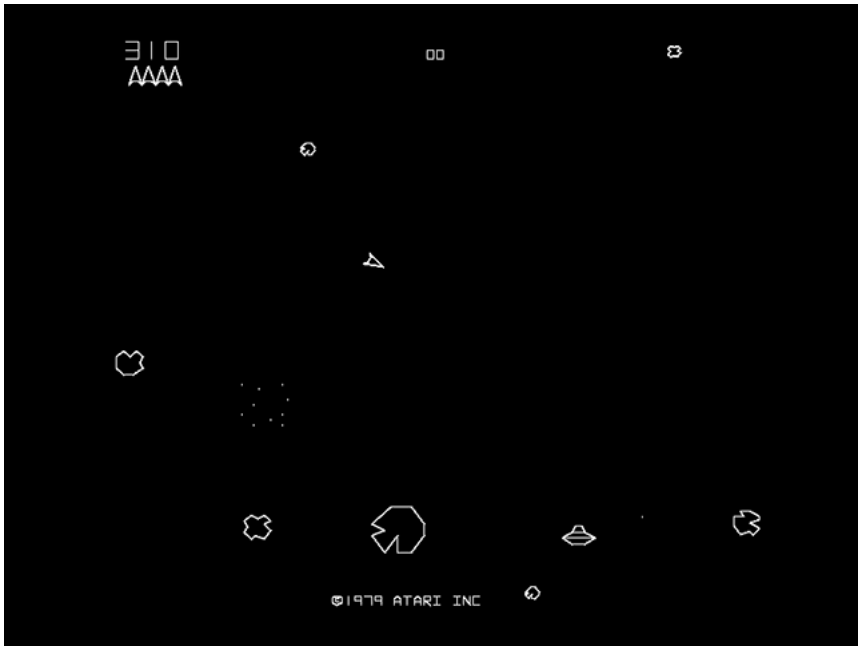


Figure 2.1: Simple Particle System as used in Asteriods (©*Atari INC.*, 1979)

- Age: The time describing how long a certain particle has lived. The age is usually set to 0 when the particle spawns.
- Lifespan: The time it takes for a particle to "die", in essence, disappear from the system.
- Velocity: The current or starting velocity for the particles.

2.2 Optimization versus detail

A generalized formula for particles is:

Number of Particles increases → Smaller scale Particles and a System richer in detail[11] (2.1)

This formula is especially true when creating fluent, complex particle systems such as fire or water. It would make sense to assign as much power as is needed to create systems rich in details.

Optimizations while creating a system of particles are essential. From a game developers perspective, the goal is to make as many features as possible within certain limitations in hardware, software and time. Unfortunately, as particle systems seldom have a key role in games mechanics, the resources allocated for them are even more limited. Even so, aesthetics is a large part of a game experience [12], so the effects of a detailed particle system are not insignificant.

Emitters and demitters control the in- and outflow of particles: Whenever the particle system spawns a particle, it is created at the position of the emitter. The emitter may have a certain particle-per-second parameter to fulfill.

The bounding volume hinders the particle system from growing too large, either in volume or in particle density, depending on how the system handles a collision with the boundary. A bounding volume is not necessary for a particle system, but it could be risky to not use one at all. This is especially true when considering particle systems that contain an emitter, as the uncontrolled flow of new particles could easily make the application run out of memory.

Forces are usually described by one or more equations. This equation can be extended to represent a vector field: A field of vectors over a volume or an area which describes the tendency of an equation (see Section 1.4). Since each and every particles movement then only have to be computed using a position and a vector, it would make sense to use vector fields. The alternative would be to calculate each particle against several forces. The forces may or may not be heavy calculations themselves (see Appendix B).

Chapter 3

GPGPU

3.1 General

GPGPU, or General Purpose computing on Graphics Processing Unit, is a technique where one can use the advantage of the parallelization abilities of the GPU in order to compute large amounts of calculations in a rather short time.

Early the GPU was reserved for rendering pictures, video and graphics, hence the name Graphical Processing Unit. Basically it only rendered pixels to the screen. Back then when rendering 3D graphics, the CPU had to send all the render data to the GPU each frame. It also needed to be in the right order with the furthest away triangle first in the array and the nearest last. Today the GPU has grown beyond only computing graphics and also begin to do general purpose computing.

In order to perform so well for rendering graphics the GPU uses a many core architectual processor with many slower cores, as opposed the the CPU which uses a multi-core processor with a few faster cores. With the high number of cores, and hence threads, the GPU can be a very effective computational parallelizer. Also, as Brookwood (head of Insight64) said: "*GPUs are optimized for taking huge batches of data and performing the same operation over and over very quickly, unlike PC microprocessors, which tend to skip all over the place...*"[13].

A compute shader is a shader made for computing large arrays of data in parallel. It can be used for the same operations as the ordinary shaders do, such as post processing and geometry stages. But a compute shader is not only for graphic calculations. It can do other operations as well. Using the GPU for Artificial Intelligence for crowd simulations is an effective way speed up a program due to the GPUs parallel computational power[14].

A given job for the compute shader is divided into work groups. A work group processes a set of incoming data sequentially. At the dispatch call from the CPU the number of work groups is set for the given job but the size of the groups is set at the initialization of the compute shader. Different sizes can have a drastic effect on the computational time. This is explored in more depth later in chapter 4.

3.2 Compute Shader APIs

There are several implementations of compute shader systems, each of them with their own advantages as well as disadvantages. CUDA, Direct Compute, OpenCL and the implementation used in this thesis work, OpenGLs own compute shader.

- CUDA made by NVIDIA is a GPGPU API that is only functional on NVIDIAs own GPUs. It was first introduced on their GeForce 8000-series and onward. It can use C, C++, Fortran and several other programming languages.

- DirectCompute is an integrated part of the Direct3D API by Microsoft. It works only on Windows Vista and newer Windows releases, and needs a GPU with DirectX 10 support or above[15]. It uses DirectXs shader language HLSL.

- OpenCL is an API made by The Khronos Group. It can be used on many operating systems and on both NVIDIA and AMD graphic cards. Though to use OpenCL on different hardware one has to use the hardware manufacturers own libraries for OpenCL. Other than that OpenCL works on almost any kind of hardware and operating system.

- OpenGL is a graphics API by The Khronos Group. OpenGLs compute shaders were introduced in version 4.3 in OpenGL[16]. Similar to DirectCompute, the compute shaders in OpenGL uses its own shader language, GLSL.

There are several other GPGPU APIs on the market but these are some of the more popular ones.

OpenGL and its own compute shaders were utilized for this thesis project. The project is somewhat aimed toward a real time solution for video games. The use of OpenGL provides an integrated GPGPU system in a graphics API. That way the need of porting the data from one API to another could not hinder the efficiency of this thesis project.

Chapter 4

Proposed Technique

4.1 Overview

This chapter will detail the implementation pipeline and the tests in more depth for the proposed research questions.

Pipeline

The texture-based particle system uses two compute shaders (see Figure 4.1). The first generates the vector field, and the second reads it and then translates the particles of the system accordingly, before being rendered. The rendering shader step is ignored when conducting all of the tests, but still holds true for any practical use. The alternative method, the equation-based, both calculates the vector field and translates the particles in the same step (see Figure 4.2). This causes the equation-based method to use one less shader to perform the computations. This may affect the final result and favor an equation-based system as the shader switch takes time. It is unlikely however, as it is only one shader switch less.

Set up

The following set up were used for all tests:

- Intel(R) Core(TM) i5-4670 CPU @ 3.40Ghz
- 8 GB RAM
- Gigabyte Sniper Motherboard B5
- Windows 7 Pro N 64-bit

The following Graphic Cards were used for the tests, both with an NVIDIA GPU:

- Gigabyte GTX 660 OC 2048MB PCI-Express III [17]
- MSI GTX 760 Twin Frozr IV OC 2048MB PCI-Express III [18]

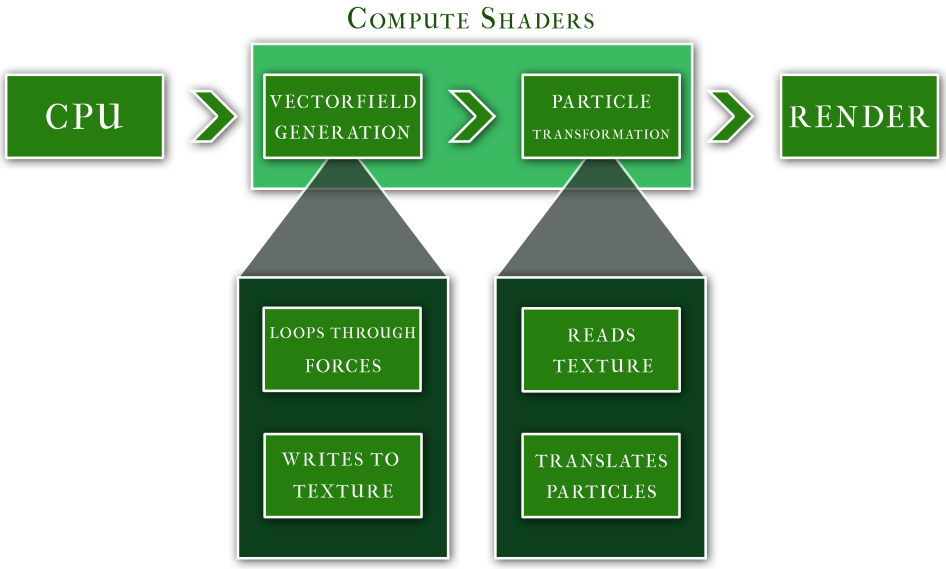


Figure 4.1: The pipeline of the texture-based compute shader stages.

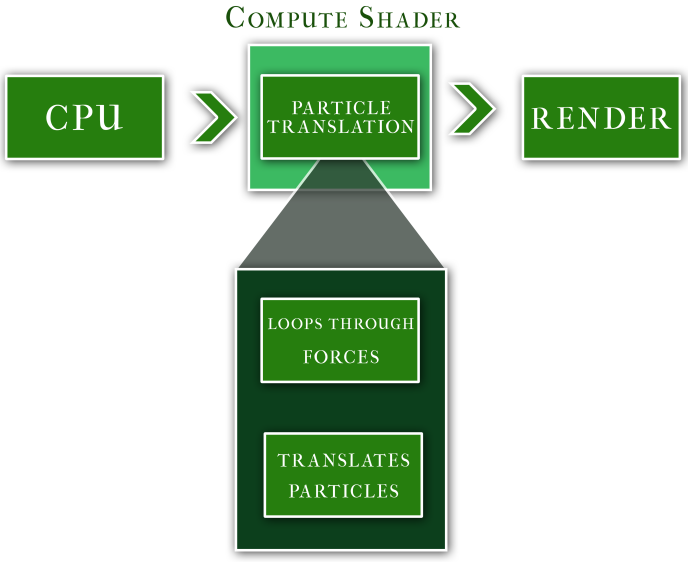


Figure 4.2: The pipeline of the equation-based compute shader stages.

Tests Conducted

The experiment is split into three sections. The first is conducted to define the optimal settings for each method. The set up uses a multitude of settings to find the most optimal for the different methods, which generally is equal for all methods.

This is a list of tests that were executed:

- For all three systems:
  - A Particle System using a large number of particles
    - With a small number of forces
    - With a large number of forces
  - A Particle System using a small number of particles
    - With a small number of forces
    - With a large number of forces
- For both texture-based systems:
  - A Particle System using a high resolution of the vector system
    - With a small number of particles
    - With a large number of particles
  - A Particle System using a low resolution of the vector system
    - With a small number of particles
    - With a large number of particles
  - A Particle System using a large number of workgroups allocated for generating the vector field
  - A Particle System using a small number of workgroups allocated for generating the vector field
- For the equational system:
  - A large number of workgroups allocated for calculating the movement of the particles
  - A small number of workgroups allocated for calculating the movement of the particles

4.2 Implementation

The suggestion is to create a particle system that is completely executed on the GPU, in order to avoid transferring data between the GPU and the CPU. In this solution, multiple compute shaders are used consequently. First, the input forces are used to generate the vector field. The vector field is then saved to a texture; a 3D texture is used for one of the examples and the second uses a 2D texture array. The next compute shader uses this data to move the particles according to the vector field.

In the third case, the particles are moved directly by the equations that describe the vector field.

The implementation do not use texture sampling. Instead of taking an interpolated value from the texture the raw data are taken from the texture. Reading the raw data directly from the memory bypasses the sampling method and gives the non-interpolated pixel. Due to that, the textures are utilized mainly as storage, not for texturing. The vectors could distorted up if they were to be interpolated with adjacent vectors due to to the sampling function.

Why not transform feedback?

Before compute shaders, one would have to set up the render pipeline without a fragmentation stage and use the vertex or geometry shader to send out the new verticies to a new buffer. It requires two buffers, one to read from and one to write to. This is sometimes referred to as a ping-pong buffer technique. One could also use render to texture techniques to store and move the particles around. This also implied that one needed to set up a full screen quad that had to go trough the shaders.

With compute shaders it is far more simple. A compute shader can do all those other techniques and more in a single dispatch call. A compute shader can read and write to the same buffers or textures in a single call. Due to the lesser amount of state changes and calls to the GPU compute shaders are often more beneficial compared to the alternatives.

4.3 Benchmarking & Results

4.3.1 Variable Comparison Tests

To be sure that the results of the tests are somewhat correct, the program is set up to test for aspects that could affect the results.

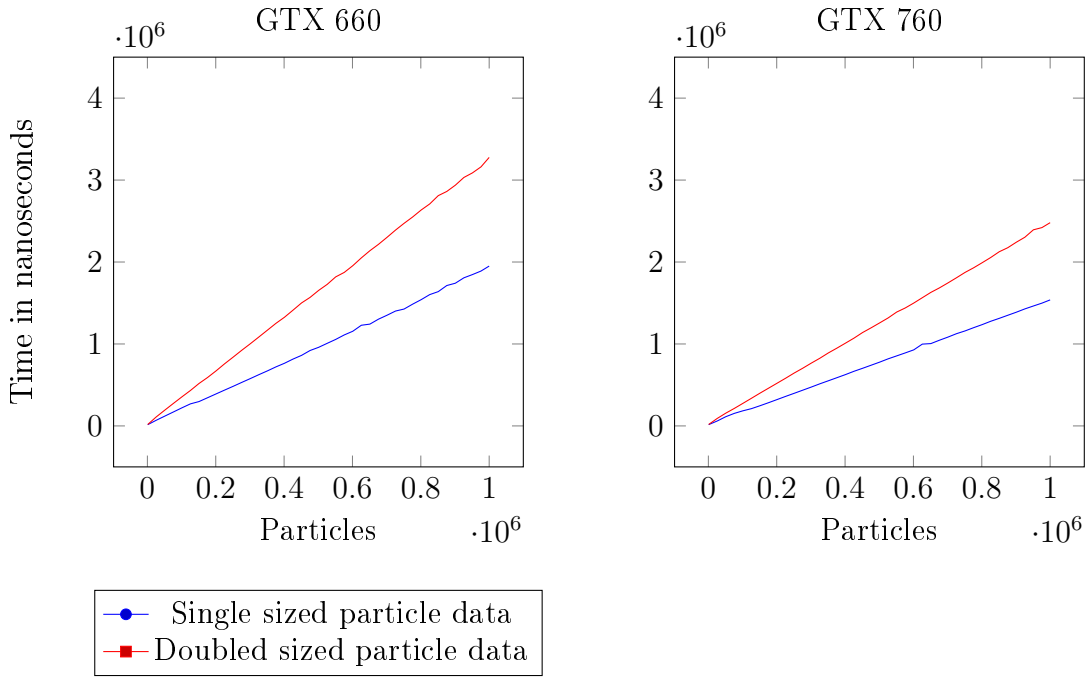


Figure 4.3: Comparison of single sized stucts versus double sized structs.

Number of Particles compared to Size of Data

The questions to be answered are related to the amount of particles and how it would affect the computational time. But that leads to another question: Is the size of the data allocated for each particle affecting the computational time more than the number of particles?

In order to answer this question, an identical compute shader, except using a larger sized data type, was compared to the original compute shader.

The larger set used linearly more time to compute (see Figure 4.3). Though even the extra data in each particle is never used, this could mean that it takes longer to move from one particle to the next due to the distance between the start of the particle and the end inside the array.

Therefore, it can be assumed that it is not directly the number of particles that is affecting the computational time, but rather the size of the data.

For the continuation of this thesis work, number of particles is assuming a larger amount of data.

Size of Resolution and the two texture types

While comparing the two types of textures, one of the aspects that can change the resulting computation time is the resolution of the texture. This is, of course, not applicable on the equation-based solution.

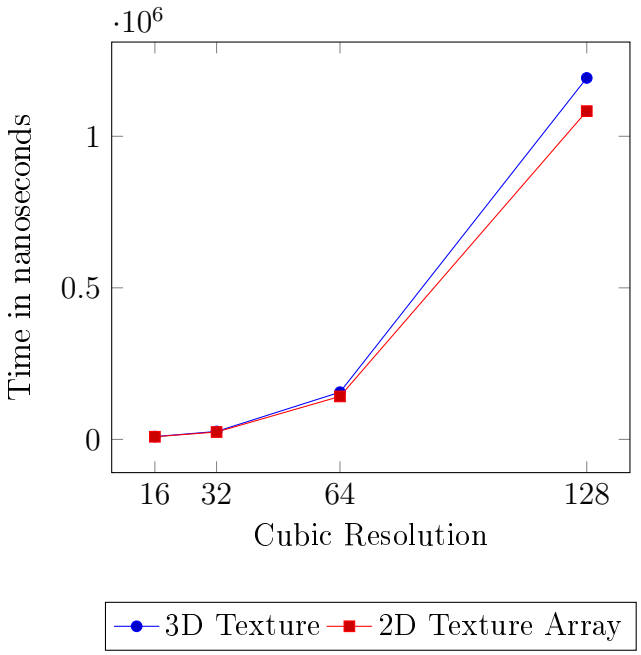


Figure 4.4: Comparison of the efficiency of the textures types dealing with different texture resolutions.

Each "pixel" corresponds to a vector, which describes the direction a single particle should move. These vectors are modified by the forces that are applied to the vector field.

The difference is not significant, but the tendency when reaching a high resolution suggests that the 2D texture array is more efficient than the 3D texture. Note that the measurements depicted in Figure 4.4 are using the length of each size of the cube, i. e. 16 is actually the size of $16^3 = 4096$ vectors represented in the vector field. While the difference may look exponential, one have to take into consideration that each computation is cubicly heavier to compute.

The 2D Texture Array has, as mentioned in Section 1.4, the draw back of not being able to sample in three dimensions. However, this is only something that would be requested for an aesthetic point of view. Functionally, this effect can be ignored. None of the texture types used here are utilizing any sampling methods.

Texture work group size

To verify the tests, the sizes of the work groups were tested as well. This is not related to the research question per se but it may change the outcome of the tests. These work group sizes are tested in the application and hardware and may vary for other implementations but can still be used as guidelines.

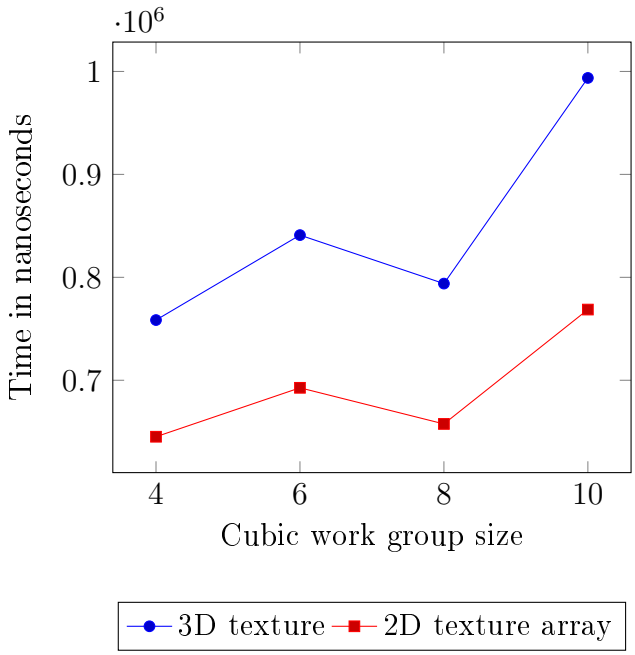


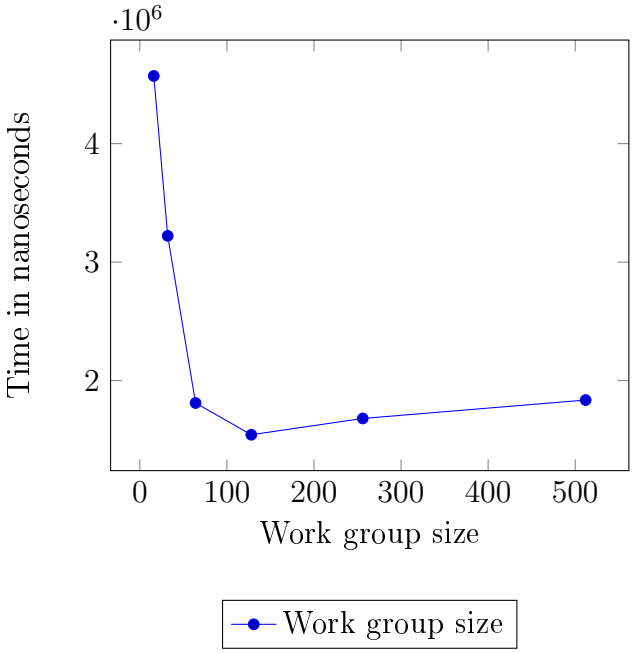
Figure 4.5: Comparison of the efficiency of the textures dealing with different workgroup sizes.

Also here there is a difference between the 3D texture and the 2D texture array. They both have a tendency toward the same curve but the 3D texture has higher values, and therefore more time is needed for the computations (see Figure 4.5).

The actual group size is n^3 due to the 3-dimensional size of the texture, just as in the case of the resolution (see Section 4.3.1).

Particle work group size

Because of the fact that several different compute shaders are utilized, each one needed to be tested. The compute shaders for moving the particles were also tested for different work group sizes. As seen in Figure 4.6 there is a difference in computational time between different work group sizes.



Size	Time
16	$4.57 \cdot 10^6$
32	$3.22 \cdot 10^6$
64	$1.81 \cdot 10^6$
128	$1.54 \cdot 10^6$
256	$1.68 \cdot 10^6$
512	$1.84 \cdot 10^6$

Figure 4.6: Graph depicting the efficency of various workgroup sizes used by the equation-based method

Comparison of different forces

In order to compare the different modifications of the movement that affects the particles, a test was conducted to compare the different forces to each other and how the computational times scales as the number of forces increases.

This test is not necessary for answering the research questions, but instead gives a hint. The implementation of forces can potentially fluctuate between different implementations of a GPU-based particle system. This test gives an idea about how expensive in resources the three different forces used are and how heavy this particular implementation is.

The result (see Figure 4.7) is not very surprising: Vortices require a large amount of computations compared to winds, which is essentially just a single vector with or without modifying noise.

For all other tests conducted, five of each force is used, if nothing else is mentioned.

See Appendix B for explaination of the math used for the forces.

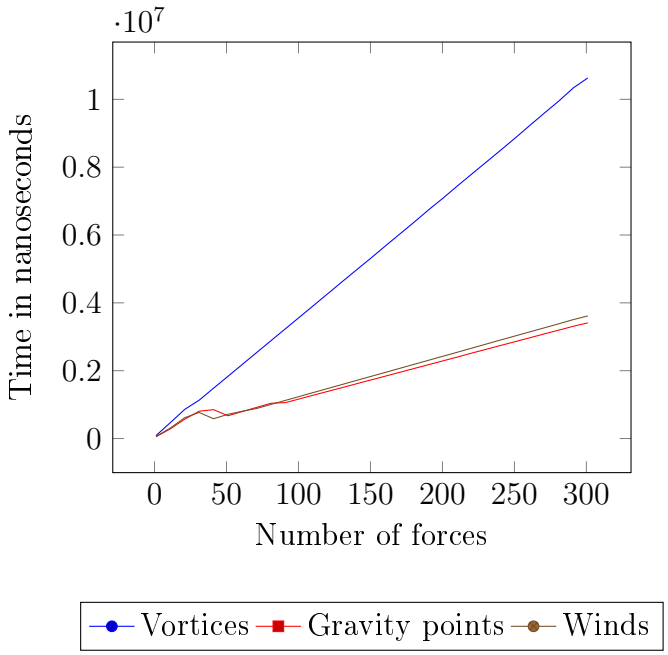


Figure 4.7: Comparison of the computational time of various forces

4.3.2 The Generation Gap

Have the optimal workgroup size changed?

While the average user may not update their GPU to have top-tier computers, the game industry is a branch that is growing each year. For this reason, in order to secure many players and therefore customers, a certain observance to earlier generations is often beneficial. Likewise, a comparison of the current growth in hardware resources can make a crude estimation of what to expect in the future, even if it is almost impossible to predict game-changing developments this way. A certain algorithm may be beneficial for current systems due to its limitations, but as these limitations disappears an older, earlier ineffective, algorithm may bypass the conventional.

As of the GPUs used for the tests, the number of cores used from the GTX 660 have doubled for the GTX 760 [17] [18].

As seen in Figure 4.8 both GPU:s do have identical difference among the set number of workgroups, but the GTX 760 has slightly lower calculation times compared to the GTX 660. An interesting sidenote, the difference puts the 2D Texture array on the GTX 660 on same level as the 3D texture on the GTX 760.

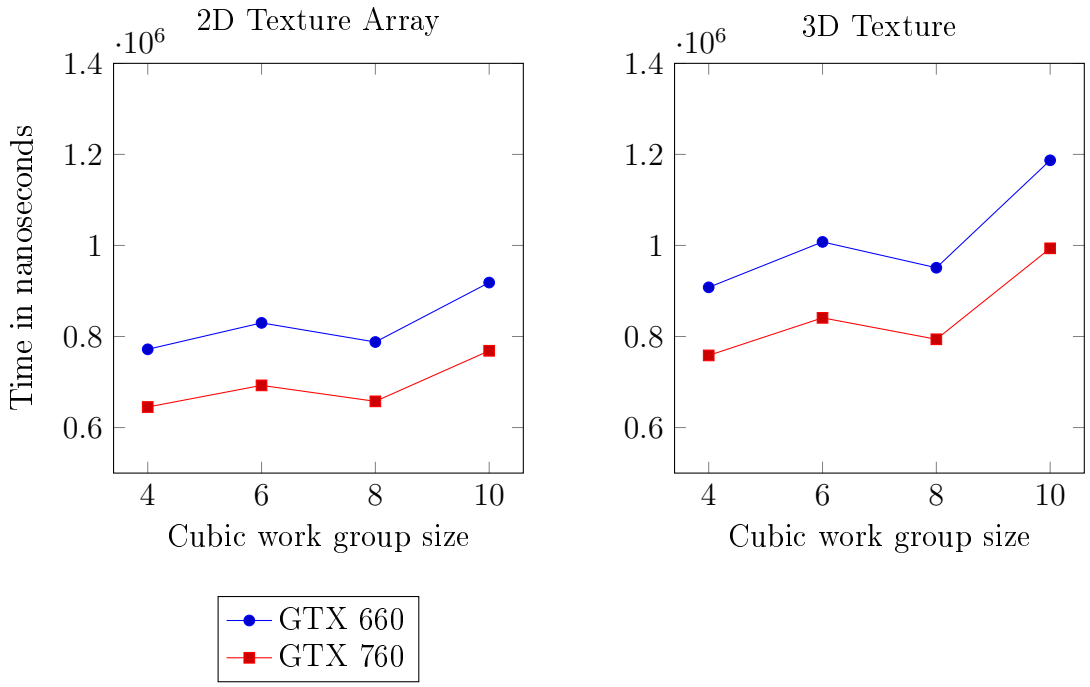
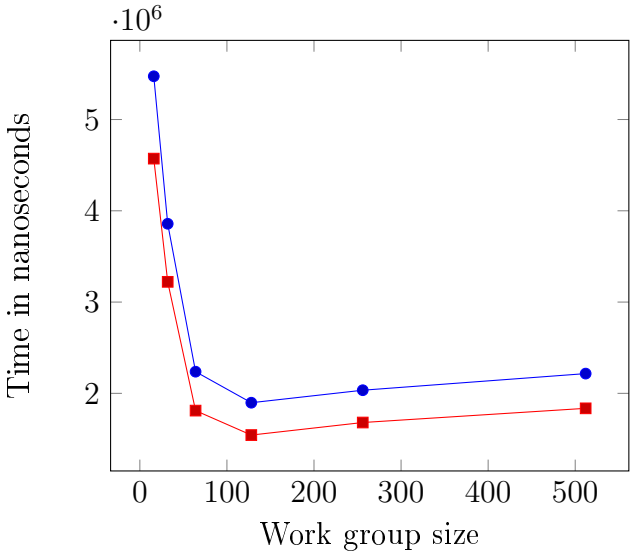


Figure 4.8: Comparison of workgroup sizes and GPUs.



Size	GTX660	GTX760
16	$5.47 \cdot 10^6$	$4.57 \cdot 10^6$
32	$3.86 \cdot 10^6$	$3.22 \cdot 10^6$
64	$2.24 \cdot 10^6$	$1.81 \cdot 10^6$
128	$1.9 \cdot 10^6$	$1.54 \cdot 10^6$
256	$2.03 \cdot 10^6$	$1.68 \cdot 10^6$
512	$2.22 \cdot 10^6$	$1.84 \cdot 10^6$

Figure 4.9: Comparison of particle workgroup sizes and GPUs

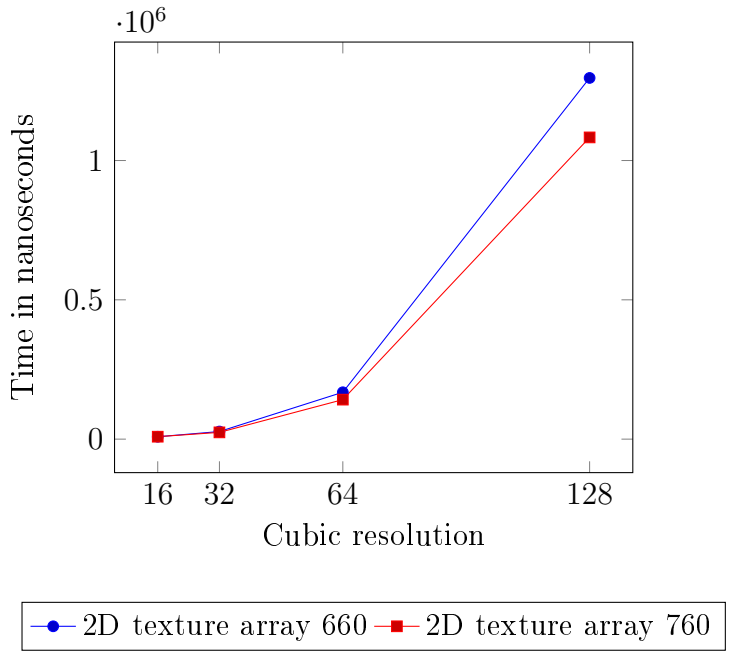


Figure 4.10: Comparison of texture resolutions and GPUs

Particle work group size

The graph shown in Figure 4.9 utilizes the equation-based method and high particle count. The GTX 760 is linearly faster than the GTX 660, which is quite clear in this graph. The optimal number of workgroups is still 128 in this implementation.

Resolution

As in Section 4.3.1 and Figure 4.4, the graph shown in Figure 4.10 appears to be exponential, while the difference between the GPUs are most likely linear. However, the difference is larger than in Figure 4.4. While the other tests, such as in Figure 4.8, suggests that the difference between the methods is about as large as between the GPUS, this test instead has a bit larger performance gap. Still, the resolution is cubic and any difference might be exaggerated; but if the result is not exaggerated, this may suggest that it will be possible to use higher resolutions with less resources in future implementations. However, it is still a vary small difference.

Comparison of different forces

The result in Figure 4.11 shows that indeed, the 760 outperforms the 660. The difference between the two GPU are not that large, and the most clear when there are many vortices included in the vector field.

4.3.3 Main System Test

High and Low Number of Particles - High and Low Texture Resolution

In Figure 4.12 one can clearly see where the different solutions has its advantages and disadvantages. With a high number of particles it takes only half the time for the equation-based system compared to the texture-based system with a high texture resolution. There is also a small difference between the 3D texture and the 2D texture array. Even when the particle amount is low the texture-based system is much slower than the equation-based system. But when a low resolution is applied to the texture with a large amount of particles the texture-based system outperforms the equation-based.

When the system has a very low particle count and low resolution it seems though that it does not matter much what method is implemented. Compared to the other graphs all the plots are floored too the X-axis.

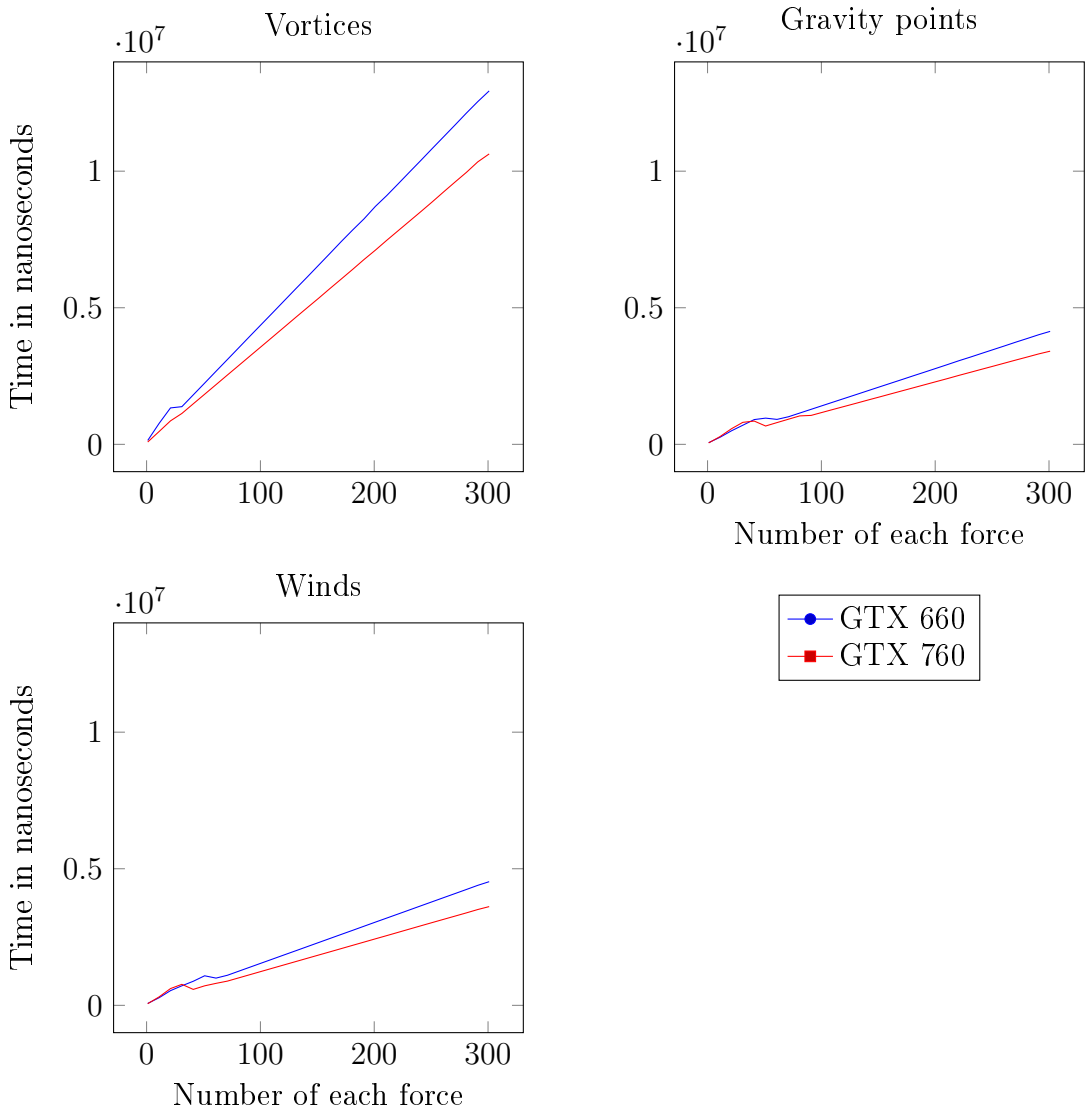


Figure 4.11: Comparison of different forces and GPUs

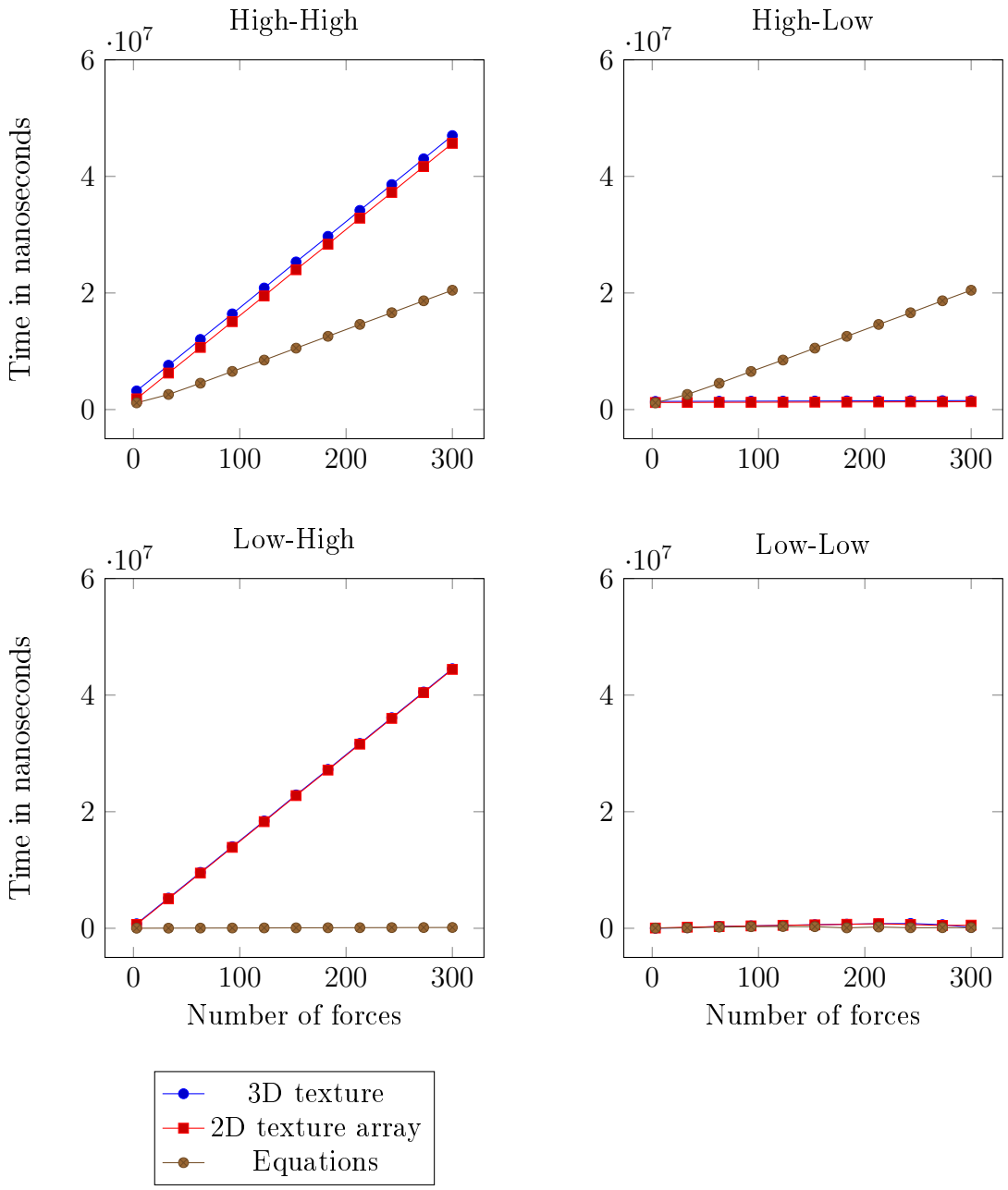


Figure 4.12: Comparison of different cases with high resolution (left) and low resolution (right) to a large number of particles (up) and a small number of particles (down)

Chapter 5
Conclusions and Future Work

5.1 Conclusions

When is a 3D texture more effective by computational time than its equation-based counterpart?

The experiments suggest that indeed, a high number of particles makes a 3D Texture more effective than the equation-based; But this only holds true if the resolution is quite low. Generally, with a modern GPU, an equation-based system is most effective.

Is 2D Texture Array more effective than a 3D Texture?

The 2D Texture array is slightly faster to compute than the 3D texture. The 2D texture array has the major drawback of not being as easy to sample, so a suggestion for future works could include sampling into a test as well.

This was slightly surprising, as the two texture storage types are not very different. One theory is that the textures performs this sampling underneath the API no matter if one explicitly tells it to or not. When the texture is used without the sampling, the pre-made calculations are ignored. The difference between the textures are that the 3D texture is sampled in all three directions. The 2D texture array on the other hand, is only sampled in two directions.

How will the future of vector fields and texture-based particle systems develop?

When comparing the results to those of the previous generation we can see that there still **may** be a future for texture-based systems. The reasoning behind this is based on Figure 4.10 and Figure 4.12: It seems that the biggest difference between the GPUs might be in the computational time for larger resolutions, combined with the fact that equation-based systems only fails to perform when the resolution is low and the number of particles are high.

The texture-based methods are heavily dependent on VRAM, and even if the number of cores may grow, the amount of memory may stagnate. This, in

combination with the other possibilities of equation-based systems and the small growth in efficiency of the resolution makes it unlikely that texture-based systems ever becomes better than equation-based.

When is a 3D texture more relevant?

There is a few, very specific cases when the use of a texture-based system is more relevant than an equation-based. For example, if the resolution of the system does not need to be large, the number of particles are many and if the number of forces also is at a relatively high number, the texture-based system is faster than the equation-based. Also, if the particle system does not need to be dynamic, a texture could be generated beforehand. This would remove the generation of the vector fields each frame from the pipeline. This optimization is something that an equation-based system lacks, and this should be detailed in future work.

5.2 Future Work

As mentioned in the previous section, suggestions for future works include testing a static texture-based particle system for an optimization and an inclusion of a test using sampling of the texture-based system.

This thesis work only concludes NVIDIA brand graphic cards, and it is recognized that running these tests on a GPU developed by other manufactures may give different results, which should be concluded in future works on the subject. As there is several different APIs for implementating Compute Shaders, where OpenGL's Compute Shader is only one, it would prove interesting to make a similar work using for instance OpenCL or Direct Compute.

However, this work takes no reference to the aesthetic point of view when using different types of particle systems, which of course has a great importance when developing particle systems, especially for entertainment or media.

Another suggestion for future works is to make a texture-equation-based hybrid method for creating particle systems. An implementation to test in the future includes a method using Shader Storage Buffer Objects (SSBO) instead of textures, which could possibly be more optimized for compute shaders[19] would also be a relevant work. If our hypothesis about the textures in conclusions is correct, using SSBO would give more control over the use of sampling.

References

[1] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, no. 6, pp. 4–16, Nov. 2012.

[2] M. J. Smith, *Sandstorm: A Dynamic Multi-contextual GPU-based Particle System using Vector Fields for Particle Propagation.* ProQuest, 2008.

[3] T. L. Hilton and P. K. Egbert, "Vector fields: an interactive tool for animation, modeling and simulation with physically based 3D particle systems and soft objects", in *Computer Graphics Forum*, 1994, vol 13, pp 329–338.

[4] M. Levoy, "Light fields and computational imaging", in *IEEE Computer*, vol 39, no. 8, pp 46–55, 2006.

[5] "Unreal Engine | Vector Fields". [Online].
Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/VectorFields/index.html. [Access date: 06-sep-2014].

[6] "Unreal Engine | Vector Field Modules". [Online].
Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/Reference/Modules/VectorField/index.html. [Access date: 06-sep-2014].

[7] "inFAMOUS Second Son: All your questions answered - PlayStation.Blog.Europe". [Online].
Available at: http://blog.eu.playstation.com/2014/03/11/infamous-second-son-questions-answered/. [Access date: 06-sep-2014].

[8] "Help: Vector Field Space Warp". [Online].
Available at: http://help.autodesk.com/view/3DSMAX/2015/ENU/?guid=GUID-523C7F3C-8901-452F-9D9C-19C1222C92E6. [Access date: 10-sep-2014].

[9] "Supreme Commander 2 'Flowfield Pathfinding' Trailer - YouTube". [Online].
Available at:https://www.youtube.com/watch?v=bovlsENv1g4. [Access date: 10-sep-2014].

[10] W. T. Reeves, "Particle systems—a technique for modeling a class of fuzzy objects", in *ACM SIGGRAPH Computer Graphics*, 1983, vol 17, pp 359–375.

[11] "GPU BASED PARTICLE SYSTEMS". [Online]. Available at: http://www.cse.chalmers.se/edu/year/2011/course/TDA361/Advanced%20Computer%20Graphics/AGFXpresentation.pdf. [Access date: 11-sep-2014].

[12] R. Hunicke, M. LeBlanc, and R. Zubek, "MDA: A formal approach to game design and game research", in *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004, pp 04–04.

[13] "FTC Presses On with Intel Probe - Businessweek". [Online]. Available at:http://www.businessweek.com/technology/content/dec2009/tc2009122_478796.htm. [Access date: 11-sep-2014].

[14] E. Passos, M. Joselli, M. Zamith, J. Rocha, A. Montenegro, E. Clua, A. Conci, and B. Feijó, "Supermassive crowd simulation on GPU based on emergent behavior", in *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 2008, pp 81–86.

[15] "Compute Shader Overview (Windows)". [Online]. Available at: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx. [Access date: 09-sep-2014].

[16] M. Segal and K. Akeley, "The OpenGL® Graphics System: A Specification(Version 4.3 (Core Profile) - August 6, 2012)," Aug. 2012.

[17] "GeForce GTX 660 | Specifications | GeForce". [Online]. Available at: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-660/specifications. [Access date: 09-sep-2014].

[18] "GeForce GTX 760 | Specifications | GeForce". [Online]. Available at: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-760/specifications. [Access date: 09-sep-2014].

[19] "Shader Storage Buffer Object - OpenGL.org". [Online]. Available at: http://www.opengl.org/wiki/Shader_Storage_Buffer_Object. [Access date: 13-sep-2014].

Appendices

Appendix A

Standard Variables in Tests

If not otherwise stated in the results these are the used values of the variables used in the tests.

- Lowest resolution for textures:16x16x16.
- Highest resolution for textures: 128x128x128.
- Standard resolution for textures: 64x64x64.
- Lowest number of forces: 5 of each kind equals to 15 forces total.
- Highest number of forces: 100 of each kind equals to 300 forces total.
- Standard number of forces: 5 of each kind equals to 15 forces total.
- Lowest number of particles: 1000.
- Highest number of particles: 1000000.
- Standard number of particles: 1000000
- Standard work group size for vector field compute shader: 4x4x4
- Standard work group size for particle transform and equational vector field compute shader: 128

Some of the tests uses a variable that increases for each final measurement. In such cases, the variable is assigned the lowest value and is then increased until it reaches the highest value stated here. This increase per mesurement varies to fit each test, but all is included in the Benchmarking subsection (see Section 4.3). None of the mesurements are omitted from the graphs.

Appendix B

Equations for Forces

Wind

Wind is described by a linear function at any direction.

$p\hat{d} + n\bar{r}$

Where $\hat{d}$ is the normalized direction of the wind $\bar{d}$, p is the power or strength of the wind and $\bar{r}$ is a randomized vector which gives a slight non-uniformity of the wind. n modifies the power of this conformity.

The power p could without any fault equal a negative number or zero, in which case the wind moves backwards or not at all, respectively. $\bar{r}$ is also ignorable, and in that case n equals zero.

Gravitational Point

Gravity Points are slightly more complex than winds, albeit that their equation may look simple:

$r_{percent}p\hat{d}$

$\hat{d}$ is the normalized direction $\bar{d}$, which is defined as the difference between the gravity points center-point, c, and the current texel in the Vector Field texture, v:

$\bar{d} = \bar{c} - \bar{v}$

$r_{percent}$ is the percentage radius of the gravity point, and is calculated by the length of $\bar{d}$, and a range parameter r:

$r_{percent} = \frac{r-|\bar{d}|}{r}$

p is the power or strength of the gravity point. It is legal to make p a negative number: Then the Gravity Point will repel particles instead of attract them.

So, in reality, the equation describing a Gravity Point is:

$p\frac{r-|\bar{d}|}{r}\widehat{\bar{c} - \bar{v}}$

This Gravity Point ignores mass, and is therefore not related to the Newtonian Gravity Point.

Vortex

The Vortex equation used is undoubtedly the most complex of the three:

$r_{percent}c_{cut}(c_{rotation}s_{power}\overline{s_{vector}} - p_{power}\overline{p_{vector}} + d_{power}\overline{d_{vector}})$

$r_{percent}$ is similar to the same variable of Gravity Points.

$c_{rotation}$ describes if the vortex spins clockwise or counter-clockwise.

c_{cut}

These modifiers gives a large range of possibillites to modify the vortex force:
$s_{power}, p_{power}, d_{power}$ modifers the power of the spin (circular motion), pull (inward) and down (downward motion).
$s_{vector}, p_{vector}, d_{vector}$ modifies the vector by which the vortex is pointed.

In both Spin and Pull parameters, as well as the percentual range, this function is used:

$\overline{t_{tempPos}} = \overline{tx} - \overline{c_{center}}$ Where $\overline{tx}$ is the used texel of the texture, $\overline{c_{center}}$ is the center that is to be used for the center of the vortex.

$\overline{p_{point}} = \overline{d_{dir}} * \frac{\overline{t_{tempPos}} \cdot \overline{d_{dir}}}{\overline{d_{dir}} \cdot \overline{d_{dir}}}$ where $\overline{d_{dir}}$ is the direction of the vortex.

Spin

$\overline{s_{vector}} = (\overline{d_{dir}} \times (\widehat{\overline{t_{tempPos}}} - \overline{p_{point}}))$

Pull

$\overline{p_{vector}} = (\widehat{\overline{t_{tempPos}}} - \overline{p_{point}})$

Down

$\overline{d_{vector}} = \widehat{\overline{d_{dir}}}$

Range

$\overline{d_{distance}} = |\overline{p_{point}}|$
$r_{percent} = \frac{(range - \overline{d_{distance}})}{range}$

Clockwise

The clockwise/counterclockwise function is not at all as interesting mathematically.

The main idea is that a boolean is needed, which is defined by its binary states (True/False), which is to be translated into a mathematical function. If the s_{power} is negative, the vortex will spin counterclockwise, but one need to translate the

False state of the boolean into a negative value, while maintaining that the True state remains positive:

$c_{rotation} = (cw_{in} * 2) - 1$

While the CPU handles a simple if-statement quite well, the same does not go for the GPU, which is less optimized for branching and more for simplier computations a multitude of times each frame (see Chapter 3). This should not affect the end result at all, except all methods being slightly faster.

Random

There is no built in functions on the GPU to emulate pseudo-random numbers in the same manner as on the CPU. However, with a random enough seed, a very simple pseudo-random generator can be implemented on the GPU. This implementation used a very simple pseudo-random function which uses the position of the vector as a seed. It is not at all random, even less than a CPU implementation, but still random enough to fool the eye. This function is included in Appendix C.1 and C.3.

Appendix C

Code

This is the used compute shaders in the prototype implementation.

C.1 Vector field texture compute shader

In this compute shader, a vector is calculated from the forces and then store the vector in the texture.

```
1  #version 440 core
2
3  #ifdef USE_3D_TEXTURE
4  layout (rgba32f) uniform image3D vectorFieldTexture;
5  #else
6  layout (rgba32f) uniform image2DArray vectorFieldTexture;
7  #endif
8
9  #define WORK_GOUP_SIZE_X 1      //Work group size is changed
       during initialization
10 #define WORK_GOUP_SIZE_Y 1      //Work group size is changed
       during initialization
11 #define WORK_GOUP_SIZE_Z 1      //Work group size is changed
       during initialization
12
13 struct VortexStruct
14 {
15         vec4 position;
16         vec4 direction;
17         float range;
18         float height;
19         float spinPower;
20         float pullPower;
21         float downPower;
22         float curve;
23         uint clockwise;
24         int padding;
25 };
26
27 struct GravityPointStruct
28 {
```

```
29          vec4 position;
30          float range;
31          float power;
32
33          float padding;
34          float padding2;
35  };
36
37  struct WindStruct
38  {
39          vec4 direction;
40          float noisePower;
41          float windPower;
42
43          float padding;
44          float padding2;
45  };
46
47  layout (std140, binding = 4) buffer Vort
48  {
49          VortexStruct Vorticies [ ];
50  };
51  layout (std140, binding = 5) buffer Grav
52  {
53          GravityPointStruct GravityPoints [ ];
54  };
55  layout (std140, binding = 6) buffer Win
56  {
57          WindStruct Winds [ ];
58  };
59
60  layout (local_size_x = WORK_GOUP_SIZE_X, local_size_y =
        WORK_GOUP_SIZE_Y, local_size_z = WORK_GOUP_SIZE_Z) in;
61
62
63  uniform vec3 fieldPosition;
64  uniform vec3 fieldSize;
65  uniform vec3 fieldResolution;
66
67  uniform uint numVorticies;
68  uniform uint numGravityPoints;
69  uniform uint numWinds;
70
71
72  float random(vec2 n)
73  {
74          return ((fract(sin(dot(n.xy, vec2(12.9898, 78.233)))*
                43758.5453)) - 0.5)*2;
75  }
76
77  /*
```

```
78 center - The center of the gravitypoint
79 voxelPos - The position of the voxel
80 range - Range of the gravitypoint
81 Power - The strength of the gravitypoint
82 */
83 vec3 Gravity(vec3 center, vec3 voxelPos, float range, float
       power)
84 {
85         vec3 dir = center - voxelPos;
86
87         float distance = length(dir);
88
89         float percent = ((range-distance) / range);
90
91         percent = clamp(percent, 0.0, 1.0);
92
93         dir = normalize(dir);
94
95         return dir * percent  * power;
96 }
97
98
99 /*
100 center - The center of the vortex
101 direction - The direction of the vortex
102 voxelPos - The position of the voxel
103 range - Range of the vortex
104 height - height of the vortex
105 spinPower - How much the vortex spin
106 downPower - How much the vortex pulls downward along its
       direction
107 pullPower - How much the vortex pulls towards the center of
       the vortex
108 curve - The curvature of the vortex
109 clockwise - If true its spins clockwise otherwise
       counterclockwise
110 */
111 vec3 Vortex(vec3 center, vec3 direction, vec3 voxelPos, float
       range, float height, float spinPower, float downPower,
       float pullPower, float curve, uint clockwise)
112 {
113         //Move all the thins to origo
114         vec3 tmpPos = voxelPos - center;
115         vec3 point = (dot(tmpPos, direction)/dot(direction,
              direction) * direction);
116
117         //If the pixel we are testing against is above the
              vortex it shouldn't affect that voxel.
118         bool cut = bool(clamp(dot(point, direction), 0.0, 1.0)
              );
119
```

```
120         vec3 pointVec = tmpPos - point;
121         vec3 pullVec = pointVec;
122 
123         float vort = length(point);
124         float percentVort = ((height - vort)/height);
125         range *= clamp(pow(percentVort, curve), 0.0, 1.0);
126 
127         float dist = length(pointVec);
128         float downDist = length(point);
129 
130         float downPercent = ((height - downDist)/height);
131         float rangePercent = ((range - dist)/range);
132 
133         rangePercent = clamp(rangePercent, 0.0, 1.0);
134         downPercent = clamp(downPercent, 0.0, 1.0);
135 
136         vec3 spinVec = cross(direction, pointVec);
137 
138 
139         vec3 downVec = normalize(direction);
140         normalize(spinVec);
141         normalize(pullVec);
142 
143         float cw = float(clockwise) * 2.0;
144 
145         cw -= 1.0;
146 
147         return (cw * spinVec * spinPower - pullVec * pullPower
                + downVec * downPower) * rangePercent * float(cut)
               ;
148 }
149 
150 
151 /*
152 Direction - The direction of the wind
153 Seed - Randomization seed for noise
154 noiseRange - How much the randomization affects the wind. 0.0
        is no affection, and while any value works, 1.0 should be
        seen as a maximum value
155 Power - The strength of the wind.
156 */
157 vec3 Wind(vec3 direction, vec3 seed, float noiseRange, float
        power)
158 {
159         vec3 dir = normalize(direction);
160         vec3 rand = vec3(random(seed.yz), random( seed.xz),
               random(seed.xy));
161 
162         dir = normalize(dir) * power + normalize(rand) *
               noiseRange;
163 
```

```
164         return dir ;
165 }
166
167
168 void main(void)
169 {
170         ivec3 storePos = ivec3(gl_GlobalInvocationID.xyz);
171
172         vec3 voxelPos = vec3(   float(gl_GlobalInvocationID.x)
                ,         float(gl_GlobalInvocationID.y),float(
                gl_GlobalInvocationID.z));
173
174         voxelPos *= fieldResolution;
175         voxelPos *= fieldSize;
176         voxelPos += fieldPosition;
177
178         vec3 forces = vec3(0.0);
179
180         for(uint i = 0; i < numVorticies; i++)
181         {
182                 forces += Vortex(Vorticies[i].position.xyz,
                        Vorticies[i].direction.xyz, voxelPos,
                        Vorticies[i].range, Vorticies[i].height,
                        Vorticies[i].spinPower, Vorticies[i].
                        pullPower, Vorticies[i].downPower,
                        Vorticies[i].curve, Vorticies[i].clockwise)
                        ;
183         }
184
185         for(uint i = 0; i < numGravityPoints; i++)
186         {
187                 forces += Gravity(GravityPoints[i].position.
                        xyz, voxelPos, GravityPoints[i].range,
                        GravityPoints[i].power);
188         }
189
190         for(uint i = 0; i < numWinds; i++)
191         {
192                 forces += Wind(Winds[i].direction.xyz,
                        voxelPos, Winds[i].noisePower, Winds[i].
                        windPower);
193         }
194
195         vec4 color = vec4(forces, 1.0);
196
197         imageStore(vectorFieldTexture, storePos, color);
198 }
```

C.2 Particle transform compute shader

This compute shader handles reading the vector from the texture and moving the particles along that vector.

```
1  #version 440 core
2  #extension GL_ARB_compute_shader :
                    enable
3  #extension GL_ARB_shader_storage_buffer_object :        enable
4
5  #define WORK_GOUP_SIZE_X 1      //Work group size is changed
       during initialization
6  #define WORK_GOUP_SIZE_Y 1      //Work group size is changed
       during initialization
7  #define WORK_GOUP_SIZE_Z 1      //Work group size is changed
       during initialization
8
9  struct Particle
10 {
11         vec4 position;
12         vec4 velocity;
13
14         float maxVelocity;
15         float minVelocity;
16         float size;
17         float padding;
18
19 };
20
21 layout (std140, binding = 4) buffer Part
22 {
23         Particle particle [ ];
24 };
25
26
27 layout ( local_size_x = WORK_GOUP_SIZE_X, local_size_y =
       WORK_GOUP_SIZE_Y, local_size_z = WORK_GOUP_SIZE_Z) in;
28
29
30 uniform vec3 fieldPosition;
31 uniform vec3 fieldSize;
32 uniform ivec3 fieldResolution;
33 uniform float deltaTime;
34
35 #ifdef USE_3D_TEXTURE
36 layout (rgba32f) uniform image3D vectorFieldTexture;
37 #else
38 layout (rgba32f) uniform image2DArray vectorFieldTexture;
39 #endif
40
41 void main(void)
42 {
```

```
43          uint globalID = gl_GlobalInvocationID.x;
44
45          vec3 position = particle[ globalID ].position.xyz;
46          vec3 velocity = particle[ globalID ].velocity.xyz;
47
48          ivec3 tmpPos = ivec3(((position - fieldPosition) /
               fieldSize) * fieldResolution);
49
50          vec3 dir = imageLoad(vectorFieldTexture, tmpPos).xyz;
51
52          position += velocity * deltaTime;
53          velocity += dir;
54
55
56          float speed = length(velocity);
57          if ( speed > particle[ globalID ].maxVelocity)
58          {
59                  velocity = particle[ globalID ].maxVelocity *
                       velocity / speed;
60          }
61
62          particle[ globalID ].position.xyz = position;
63          particle[ globalID ].velocity.xyz = velocity;
64  }
```

C.3 Equational vector field compute shader

This compute shader first calculates the vector for a particle from the forces and then moves the particle according to that calculated vector.

```
1  #version 440 core
2
3  layout (rgba32f) uniform image3D u_texture;
4  #define WORK_GOUP_SIZE_Y 1      //Work group size is changed
       during initialization
5  #define WORK_GOUP_SIZE_Z 1      //Work group size is changed
       during initialization
6  #define WORK_GOUP_SIZE_X 1      //Work group size is changed
       during initialization
7
8  struct VortexStruct
9  {
10         vec4 position;
11         vec4 direction;
12         float range;
13         float height;
14         float spinPower;
15         float pullPower;
16         float downPower;
17         float curve;
18         uint clockwise;
19         int padding;
20 };
21
22 struct GravityPointStruct
23 {
24         vec4 position;
25         float range;
26         float power;
27
28         float padding;
29         float padding2;
30 };
31
32 struct WindStruct
33 {
34         vec4 direction;
35         float noisePower;
36         float windPower;
37
38         float padding1;
39         float padding2;
40 };
41
42 struct Particle
43 {
44         vec4 position;
```

```
45          vec4 velocity;
46
47          float maxVelocity;
48          float minVelocity;
49          float size;
50          float padding;
51
52  };
53
54  layout (std140, binding = 4) buffer Part
55  {
56          Particle Particles [ ];
57  };
58
59  layout (std140, binding = 5) buffer Vort
60  {
61          VortexStruct Vorticies [ ];
62  };
63
64  layout (std140, binding = 6) buffer Grav
65  {
66          GravityPointStruct GravityPoints [ ];
67  };
68
69  layout (std140, binding = 7) buffer Win
70  {
71          WindStruct Winds [ ];
72  };
73
74
75  layout (local_size_x = WORK_GOUP_SIZE_X, local_size_y =
        WORK_GOUP_SIZE_Y, local_size_z = WORK_GOUP_SIZE_Z) in;
76
77
78  uniform uint numVorticies;
79  uniform uint numGravityPoints;
80  uniform uint numWinds;
81  uniform float deltaTime;
82
83
84
85  float random(vec2 n)
86  {
87          return ((fract(sin(dot(n.xy, vec2(12.9898, 78.233)))*
                43758.5453)) - 0.5)*2;
88  }
89
90  /*
91  center - The center of the gravitypoint
92  particlePos - The position of the particle
93  range - Range of the gravitypoint
```

```
94  Power - The strength of the gravitypoint
95  */
96  vec3 Gravity(vec3 center, vec3 particlePos, float range, float
        power)
97  {
98          vec3 dir = center - particlePos;
99
100         float distance = length(dir);
101
102         float percent = ((range-distance) / range);
103
104         percent = clamp(percent, 0.0, 1.0);
105
106         dir = normalize(dir);
107
108         return dir * percent  * power;
109 }
110
111
112 /*
113 center - The center of the vortex
114 direction - The direction of the vortex
115 particlePos - The position of the particle
116 range - Range of the vortex
117 height - height of the vortex
118 spinPower - How much the vortex spin
119 downPower - How much the vortex pulls downward along its
        direction
120 pullPower - How much the vortex pulls towards the center of
        the vortex
121 curve - The curvature of the vortex
122 clockwise - If true its spins clockwise otherwise
        counterclockwise
123 */
124 vec3 Vortex(vec3 center, vec3 direction, vec3 particlePos,
        float range, float height, float spinPower, float pullPower
        , float downPower, float curve, uint clockwise)
125 {
126         //Move all the things to origo
127         vec3 tmpPos = particlePos - center;
128         vec3 point = (dot(tmpPos, direction)/dot(direction,
            direction) * direction);
129
130         //If the voxel we are testing against is above the
            vortex it shouldn't affect that voxel.
131         bool cut = bool(clamp(dot(point, direction), 0.0, 1.0)
            );
132
133         vec3 pointVec = tmpPos - point;
134         vec3 pullVec = pointVec;
135
```

```
136         float vort = length(point);
137         float percentVort = ((height - vort)/height);
138         range *= clamp(pow(percentVort, curve), 0.0, 1.0);
139 
140         float dist = length(pointVec);
141         float downDist = length(point);
142 
143         float downPercent = ((height - downDist)/height);
144         float rangePercent = ((range - dist)/range);
145 
146         rangePercent = clamp(rangePercent, 0.0, 1.0);
147         downPercent = clamp(downPercent, 0.0, 1.0);
148 
149         vec3 spinVec = cross(direction, pointVec);
150 
151         vec3 downVec = normalize(direction);
152 
153         normalize(spinVec);
154         normalize(pullVec);
155 
156         float cw = float(clockwise) * 2.0;
157 
158         cw -= 1.0;
159 
160         return (cw * spinVec * spinPower - pullVec * pullPower
                + downVec * downPower) * rangePercent * float(cut)
               ;
161 }
162 
163 /*
164 Direction - The direction of the wind
165 Seed - Randomization seed for noise
166 noiseRange - How much the randomization affects the wind. 0.0
        is no affection, and while any value works, 1.0 should be
        seen as a maximum value
167 Power - The strength of the wind.
168 */
169 vec3 Wind(vec3 direction, vec3 seed, float noiseRange, float
        power)
170 {
171         vec3 dir = normalize(direction);
172         vec3 rand = vec3(random(seed.yz), random( seed.xz),
               random(seed.xy));
173 
174         dir = normalize(dir) * power + normalize(rand) *
               noiseRange;
175 
176         return dir;
177 }
178 
179 
```

```
180 void main(void)
181 {
182         uint globalID = gl_GlobalInvocationID.x;
183 
184         vec3 particlePos = Particles[ globalID ].position.xyz;
185         vec3 velocity = Particles[ globalID ].velocity.xyz;
186 
187         vec3 forces = vec3(0.0);
188 
189         for(uint i = 0; i < numVorticies; i++)
190         {
191                 forces += Vortex(Vorticies[i].position.xyz,
                            Vorticies[i].direction.xyz, particlePos,
                            Vorticies[i].range, Vorticies[i].height,
                            Vorticies[i].spinPower, Vorticies[i].
                            pullPower, Vorticies[i].downPower,
                            Vorticies[i].curve, Vorticies[i].clockwise)
                            ;
192         }
193 
194         for(uint i = 0; i < numGravityPoints; i++)
195         {
196                 forces += Gravity(GravityPoints[i].position.
                            xyz, particlePos, GravityPoints[i].range,
                            GravityPoints[i].power);
197         }
198 
199         for(uint i = 0; i < numWinds; i++)
200         {
201                 forces += Wind(Winds[i].direction.xyz,
                            particlePos, Winds[i].noisePower, Winds[i].
                            windPower);
202         }
203 
204         particlePos += velocity * deltaTime;
205         velocity += forces;
206 
207         float speed = length(velocity);
208         if ( speed > Particles[ globalID ].maxVelocity)
209         {
210                 velocity = Particles[ globalID ].maxVelocity *
                             velocity / speed;
211         }
212 
213         Particles[ globalID ].position.xyz = particlePos;
214         Particles[ globalID ].velocity.xyz = velocity;
215 }
```