Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Martin Solheim

# Real-Time Volumetric Smoke Simulation for Games

Masteroppgave i Computer Science
Veileder: Theoharis Theoharis
Juli 2021

NTNU
Kunnskap for en bedre verden

Martin Solheim

# Real-Time Volumetric Smoke Simulation for Games

Masteroppgave i Computer Science
Veileder: Theoharis Theoharis
Juli 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

**Acknowledgements**

I would like to thank my supervisor, Theoharis Theoharis for guidance throughout this process. I would also like to thank Bart Iver van Blokland and Michael Gimle for advice in the early stages of the project.

## Abstract

Physically based smoke simulation has been a major research topic for the past two decades, yet not that many games or real-time applications make use of it. Most of the research is done from the perspective that real-time is anything above 30fps in a simulation context. This is not fast enough to be implemented into games that need to be able to run at more than 60fps by themselves. Any additional computation time can't slow down the program too much, or the simulation is not useful. This thesis takes the perspective of optimizing an implementation for that purpose, with those requirements in mind.

The Vortex Particle Method was chosen as the method for simulating fluid flow. Two different solvers were examined. An implementation based on VorteGrid was made in CUDA using Half-Angle Slice volumetric illumination. Efficient rendering of particle systems was experimented with and examined.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Simulating physical phenomena accurately and efficiently in computer graphics has been a goal in several fields for decades. It is used in scientific areas and engineering to predict how fluid or gas will interact with objects in different circumstances, for computer-animated films such as the works of Pixar or Disney, as well as in video games to create more realistic worlds to explore. These different areas, especially the latter, often have very different requirements in terms of accuracy and performance and need to be developed in different ways. The methods used for weather prediction are not suitable for real-time applications such as games.

Smoke, dust, or particle effects can be an important part of what makes video game scenes immersive [25]. Having an interactive environment can help improve the game flow by allowing the player to interact with the environment itself and not just the objects in it [27].

This thesis will try to evaluate the current state of real-time smoke simulation methods, specifically for video game contexts. It will look at the similarities and differences between the different techniques. One survey was published in 2015 [17] that looked at the state of the art smoke simulation which had a segment on real-time methods. More research has come out since then which will be discussed in the literature review chapter.

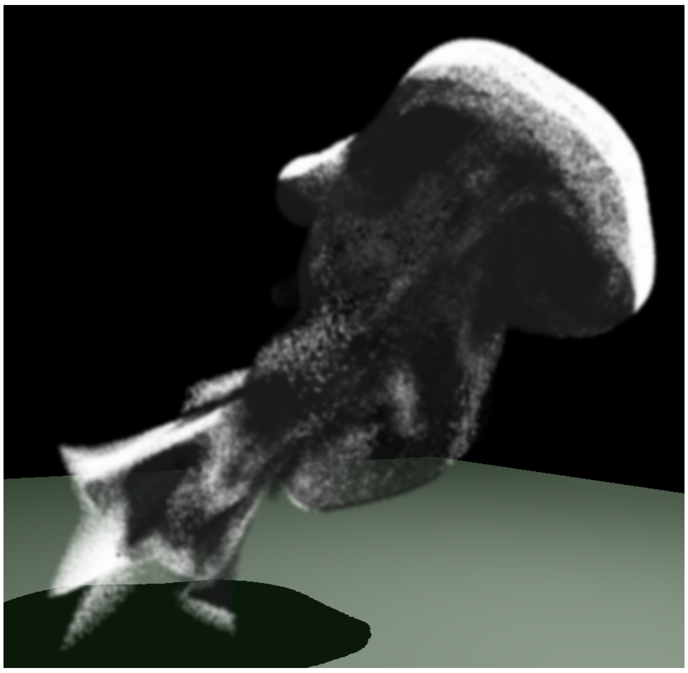


Figure 1.1: Smoke simulation with this implementation.

Smoke effects can be used sparingly or widely in scenes to greatly improve the atmosphere and to complement the visuals in general. It is however implemented very differently in in-game environments today. Computational fluid dynamics, including physically-based volumetric smoke simulation, runs slowly compared to most visual effects. This is due to the large number of particles needed to produce convincing looking smoke in computer graphics [10]. This computational cost is often too big for game developers to prioritize when everything else in a scene needs to be running as well. Fluid simulation is also complicated. Behind the most widely used fluid flow libraries, there are often times multiple developers with a Ph.D. in physics, which raises the floor for those who are able to develop or customize the parameters to better suit their application.

Virtual reality applications and games is an example of a field where a light weight smoke simulation method could be valuable. These kinds of games and applications are steadily entering the mainstream and with it comes their own set of requirements for performance. Rendering the scene for each eye requires a more simple scene to maintain a steady frame rate. VR also has a higher frame rate floor for maintaining the immersion, where lower frame rates can lead to nausea or increased motion sickness. There are emerging technologies to help lessen the computational cost in VR such as foveated render [5], but these techniques are still far from mainstream and require special hardware with eye-tracking.

## 1.2 Requirements

There are several factors that separate offline simulation methods from real-time simulation. Offline methods can be as complex as the writers want to, mimic the fluid dynamics equations precisely, and take every little detail into consideration [17] [25]. The physics simulations used in Pixar movies can be rendered using several hours per frame, while real-time simulation such as video games or simulations used as teaching tools has different requirements both for memory usage and performance. For real-time contexts, especially for game environments, it is only necessary to produce a plausible smoke effect. Most users will not pay attention to the fine detail of the effects by themselves. In the context of games, there will also be several other systems running on the same hardware simultaneously which further pushes the requirements up.

These requirements can be summarized into:

- **R1: Cheap to compute** - The smoke simulation will not have all the hardware for itself and must share with multiple other systems that need to collectively run in real-time. Running below 60 frames per second on high-end hardware would be seen as unoptimized [17]. The simulation must also run smoothly without major spikes in computation time.
- **R2: Stable solver** - The simulation must behave realistically even in weird situations. Games sometimes produce impossible scenarios such as make a character or an object move impossibly fast. The system must behave predictably even if the world around it behaves unnaturally [17].
- **R3: Low memory usage** - Several other systems in the program are running using a finite amount of memory. It is therefore preferable to minimize the memory usage of the simulation [17].
- **R4: Interactive** - Interactiveness is what separates video games from other media. This is different than R1 in that it involves some form of external input to impact the simulation in a meaningful way.

## 1.3 History

In the 1990s and early 2000s, smoke effects were implemented by rendering animated smoke textures on 2D billboards. This method was efficient and produced a reasonably good result for the time.

Around the millennial shift, Jos Stam made significant contributions to the methods of simulating fluid dynamics in real-time with a number of papers [25] [24] [8], and while not real-time for volumetric simulations at the time these papers were a major part of the beginning of real-time computational fluid dynamics. Stam proposed a new way to simulate fluids in real-time in an interactive manner that was based on the Navier Stokes Equations, the same equations used to predict real-world fluids but simplified to prioritize performance and visuals over physical accuracy. The paper specifically for games also featured example code which helped make these techniques more accessible.

Before this, there were many different attempts at cheaper ways of rendering gasses or fluids, such as billboarding animated smoke textures, with varying results. This technique is often used in modern games as well but often layered with additional simple particle effects to simulate a volume. Stam instead used a grid-based particle system based on real fluid dynamics.

In 2008, Robert Bridson released a book called *Fluid Simulation for Computer Graphics*, which stood as a good collection of methods at the time. It got a revision in 2016 with a second edition [3].

In the same year, Simon Green from NVIDIA released *Cuda Particles* [11] which proposed GPU acceleration of the particle systems with parallelization techniques that have been used ever since and made several more methods viable for real-time simulation [6] [22].

NVIDIA has been a major contributor to research into real-time computational fluid dynamics since the late 2000s. They have released a number of libraries such as NVIDIA Flex, which used a constraint-based simulation technique for universal particle systems, as well as their most recent library NVIDIA Flow which uses an adaptive sparse voxel fluid simulation method.

Volumetric smoke effects utilizing GPU acceleration have been used in a handful of games in the 2010s.

An incomplete list of notable titles utilizing GPU accelerated physically based smoke effects:

- Witcher 3(2015) - minor use of smoke effects using PhysX [1]
- Batman: Arkham Knight (2015) - volumetric smoke generated from car using PhysX [2]
- Teardown(still in early access) - volumetric smoke/debris using a custom constraint based method. [3]

[1]https://thewitcher.com/en/witcher3
[2]https://rocksteadyltd.com
[3]https://www.teardowngame.com/

## 1.4 Goals

The goals for this thesis is to:

- G1: Examine the current state of the art smoke simulation methods
- G2: Make an implementation that meets the requirements listed in chapter 1.2

# Chapter 2

# Theoretical background & Related Work

## 2.1 Fluid Dynamics

Most common physically based fluid simulation methods are based on or derived from the Navier Stokes equations. These equations depict the behaviour of fluid flow in relation to forces and their environment.

A lot of theory behind fluid simulation methods is identical as a lot of it is based on the same physical equations that try to emulate real-world fluid flow. A lot of these are based, at least partially, on the Navier Stokes Equations. There are exceptions such as curl noise which is a simpler way of creating turbulence in a velocity grid, which is a good solution for visual effects but this is not physically based.

### 2.1.1 Navier Stokes Equations

The Navier Stokes Equations are part of the fundamental partial differential equations that describe the motion of viscous fluids and gasses. They are named after French physicist Clause-Louis Navier and Irish physicist George Gabriel Stokes.

There are several formulas that describe these forces. For the context of computational fluid dynamics, it was not until the late 90's that they became relevant to real-time simulations when they got popularized by Jos Stam in the paper Stable Fluids [24]. When aiming for performance over accuracy, developers want to simplify this formula as much as possible

These are the fundamental partial differential equations *incompressable Navier-Stokes equations*:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho}\nabla p + v\nabla^2\vec{u} + f \tag{2.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2.2}$$

Equation 2.1 depicts how the fluid flows over time.

$\nabla$ is a vector of spatial partial derivatives. Namely $(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$.

$u$ is the velocity field.

$p$ is the pressure field. - The pressure field describes essentially where the particles should move towards given where they are in the grid. The particles move from high pressure to low pressure.

$\rho$ is the density of the fluid. - The density of the fluid describes how compact it is. Water is about 1000 kg/$m^3$, while air at sea level is about 1.3 kg/$m^3$.

$t$ is time. - Describing the time difference between each time step of the simulation.

$v$ is the kinematic viscosity of the fluid.

$f$ is external forces. These forces can come from multiple different sources, e.g. gravity, wind, an object interacting with the simulation to enact a force upon it. - More about this in 2.1.1

Equation 2.2 simply means that the fluid or gas is incompressible, that is, that it occupies a fixed area of space at any given time. As opposed to compressible fluids e.g. snow, which can grow or shrink in size over time.

Breaking down this formula into separate parts that can be solved by themselves is important for it to make sense intuitively. It is also good for computing, as there are several techniques to achieving the same goals. This next part will try to describe the different stages of the formulas in more detail.

#### Vorticity

Vorticity is defined as the curl of velocity. [3]. For vortex-based fluid simulations, it is used to compute the velocity field indirectly. Vorticity can be defined by this equation:

$$\vec{\omega} = \nabla \times \vec{u} \tag{2.3}$$

where $\omega$ is the vorticity

$\nabla$ is the spatial derivative.

$\vec{u}$ is the velocity.

#### Advection

One fundamental step in fluid simulation is solving the advection equation. Advection simply means to move the fluid through the velocity field. This can be applied to the particles themselves or other fields that needs to be updated relative to the timesteps. There are several algorithms that solve this in different ways, but in general, we want the density of particles to be advected through the velocity field relative to the time step [3]. This can be described with:

$$q^{n+1} = advect(\vec{u}, \Delta t, q^n) \tag{2.4}$$

Where $q^n$ is the current density field at time step $n$, $\vec{u}$ is the velocity field, $\Delta t$ is the difference in time between ticks, and $q^{n+1}$ is the density field in the next time step.

This also means that the overall density of the simulation must remain the same over time. The particles move around but isn't changing [3]. This can be described with:

$$\frac{Dq}{Dt} = 0 \tag{2.5}$$

**Forces**

External forces to the flow can be annotated as $\vec{g}$. Though, it is not always just gravity that is included in in these forces. It is applied to the velocity field such that:

$$\vec{u} \leftarrow \vec{u} + \Delta t \vec{g} \tag{2.6}$$

Where $\vec{g}$ is usually a 3-vector of floats(0.0, -9.81, 0.0)m/$s^2$ while in a passive state with no moving objects and only gravity acting upon it, but additional forces can be added on top such as wind or user-controlled objects to interact with the flow [3]. In methods that take particle temperature into account uses this as a force. Heat is in this case a force that pushes directly up against gravity [3]. This force is usually described as *body forces.* If the fluid or gas collides with another object or gets pushed in some way, this is the part of the equation that gets affected. The additional force is simply added to the force.

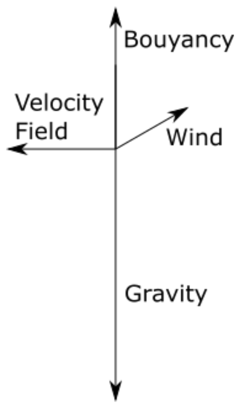


Figure 2.1: An illustration depicting the forces acting upon the simulation

The forces impacting this method are illustrated in Figure 2.1.

Buoyancy is calculated with the mass of each particle and in relation to the density field around it. This depends on what kind of environment the smoke particles are set in.

Gravity is a constant force pointing straight down with (0, -9.81, 0)

The velocity field force is the field that the particles are advected through. This is commonly the strongest force and will change direction and strength over time.

Wind is an optional force that can point in an arbitrary direction and has a customizable strength. It is shifting slightly for each cell in the grid to improve visual quality.

#### Diffuse

Diffusion means calculating the average velocity of the neighboring particles and their impact on the particle at hand. This is where viscosity comes into play. The higher the viscosity, the more impact this average will have on the velocity of the particle, that is how well it sticks together. [3]

Kinematic viscosity describes how *viscous* the fluid is, as in how much resistance there is in the fluid to deforming itself. Fluids such as honey have very high viscosity while water has a relatively low viscosity. This part is often dropped for simulations that don't have high requirements for accuracy. Without this term, the equation is called *Euler Equations*, and is for instance used in the 2010 method by NVIDIA [6].

It is this part of Equation 2.1:

$$v\nabla^2\vec{u} \tag{2.7}$$

The $\nabla^2$ here means the dot product $(\nabla \cdot \nabla)$ not to the power of 2. It is very often written this way, and it is short for $v\nabla \cdot \nabla\vec{u}$

#### Time steps

How big the time steps are is a crucial part to how demanding the method is to calculate. The need for shorter time steps to accurately simulate the desired fluid or gas means that it needs to run more calculations per second than if the method requires longer time steps [3]. In general, the lower the time steps the more accurate a method is, but it is also more expensive to compute.

### 2.1.2 Lagrangian vs Eulerian perspective

There are two main approaches to handling the motion of a continuum, like fluid or gas. These are the Lagrangian perspective and the Eulerian perspective. Lagrangian named after Italian mathematician and astronomer Joseph-Louis Lagrange, and Eulerian named after Swiss mathematician and physicist Leonhard Euler. [3]

#### Lagrangian

The Lagrangian perspective calculates the motion of the particles from the perspective of the individual fluid parcel themselves as they move through space and time [3]. This is the most intuitive way of thinking of particle systems and is the one most commonly implemented in simple particle systems where there is little interaction between the particles. This perspective of fluid flow is simply often referred to as a particle system since the calculations are done using data from the particles themselves. Each particle usually has a position vector and a velocity vector.

One major disadvantage to this method is that for interaction to happen, the neighbors of each particle need to be calculated for each iteration of the simulation. In an Eulerian-based system, this is not necessary.

Smoothed Particle Hydraulics, Vortex Particle Method, and Position Based Dynamics are some examples that are or can be fully Lagrangian [23] [10] [26].

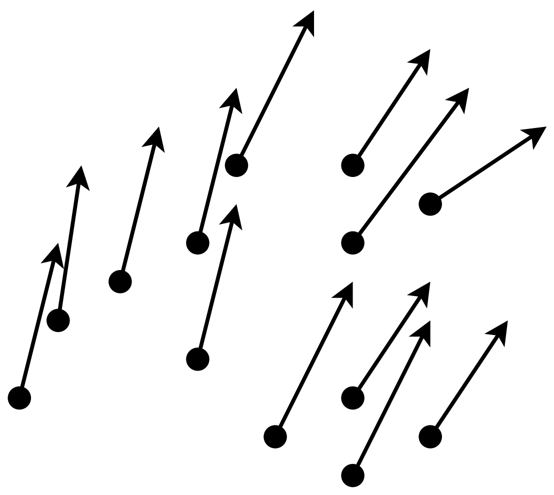


Figure 2.2: Lagrangian perspective illustration

In figure Figure 2.2 the particles themselves have the necessary information to compute the next step in the simulation. The particles illustrated with the black dots represent the position of the particles, while the arrow represents the velocity vector each particle has. While the essential information is on the particles themselves, Lagrangian simulations still often used grids for discretization purposes which makes parallelization simpler to implement.

### Eulerian

The Eulerian perspective calculates the fluid flow based on set locations in a grid that changes as fluid particles flow through it over time. [3]. The grid has a set of points fixed in space in which the velocity changes based on the average motion of the particles passing through that point. This perspective of fluid flow is often called a grid-based system as it calculated the flow based on fixed locations in a grid.

One major disadvantage of this approach is that the system is less easily made sparse. This means that the simulation is inherently bound within a predefined box boundary. For games with large spaces, this is generally less useful. In a lot of cases, scenes in games need gases and fluids to move through large stretches of spaces, making the Eulerian grid approach infeasibly large.

One good example of this method is *Interactive Fluid-Particle Simulation using Translating Eulerian Grids* from 2010 by NVIDIA [6]. It is also used in their newest fluid simulation library, NVIDIA Flow.

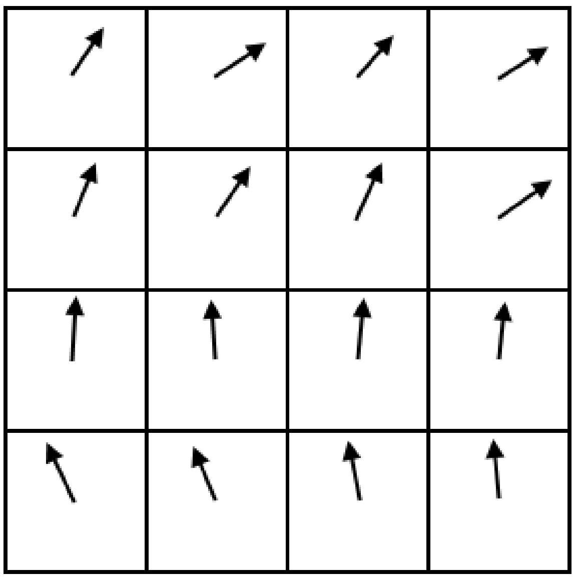


Figure 2.3: Eulerian Grid

Figure 2.3 shows a velocity grid used in Eulerian simulations. Here the grid itself holds the information needed to calculate the next step in the simulation.

### 2.1.3 Discretization

Discretization means to convert a continuous medium into a finite one. One aspect of this is to discretize the forces impacting each particle. This can be done by creating a grid where each cell is assigned a velocity instead of calculating each force individually for each particle. In particle systems, especially those meant for real-time, discretization is necessary. Even in lagrangian simulation methods, a grid is often used. This is however generated each iteration to take into the particles' new boundary positions.

In 3 dimensions, uniform grids are commonly used. Using these in lagrangian methods, each particle is owned by the grid cell that contains it. This discretization makes neighbour search, advection, and diffusion significantly less computationally expensive.

### 2.1.4 Boundary Conditions

The main kinds of boundary conditions in particle and grid systems are *solid walls* and *free surfaces* [3]. A solid wall is the boundary of a solid object that the flow is colliding with. This relates to any object that is in contact with the particles and impacts it in some way like when a ball is falling through a layer of smoke. when a solid wall impacts the simulation in some way, the particles doesn't move relative to the objects normal at that given point. This can be described with Equation 2.8.

$$\vec{u} \cdot \vec{n} = 0 \tag{2.8}$$

where $\vec{u}$ is the velocity

and $\vec{n}$ is the normal of the relevant point of the object colliding with the particles.

*Free surfaces* are the hard edges of a simulation. This is mostly relevant to eulerian simulation methods without sparse grids. Free surfaces are necessary for scenes to encapsulate the simulation to the required space, since the air around the particles also needs to be part of the simulation. To not set boundaries around the particles would result in wasted computation. [3].

### 2.1.5 Sparse Grids

Sparse methods are a property of fluid simulation that is especially attractive to the context of games. It essentially means that the simulation doesn't have hard boundaries that it can not move beyond. It is an inherent property of Lagrangian simulations since there is no grid to restrict it.

An example is if your want smoke to be emitted at one end of the scene and flow all the way to the other side of the scene, if done in a classic Eulerian grid it would need a grid at the size of the whole scene which would use enormous amounts of memory and lots of grid cells would remain empty [3]. There are multiple ways around this dilemma, namely dynamic grids which changes with the simulation such as *Interactive Fluid-Particle Simulation using Translating Eulerian Grids* [6], or a Lagrangian approach where there is no solid boundary and there is only calculated fluid dynamics where interaction is happening such as *Unified Particle Physics* [23].

### 2.1.6 MAC grid technique

The MAC grid method ("Marker And Cell") was originally proposed in 1965 [16]. It has since been adopted by multiple methods for computational fluid and gas simulations. MAC-grids are a way to discretize the flow in a simple way for computation by cutting the flow into cells in a grid that can be accessed by indices $i, j$ in 2D and $i, j, k$ in 3D. They're very useful for both 2D and 3D rendering for enforcing incompressibility. [3].

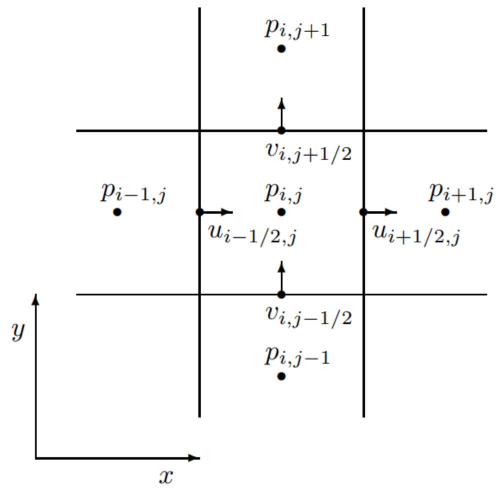


Figure 2.4: An example of a 2D MAC grid. Figure from [3].

Here $p$ represents the pressure field, and $u$ / $v$ represents the velocity of each cell. The pressure field is sampled at the center of each cell, while the velocity is sampled at the faces indicated by

$+-1/2$. This is used to form an average between cells. These kinds of grids are sometimes called *Staggered grids* when they store different amounts of particles in each cell [2].

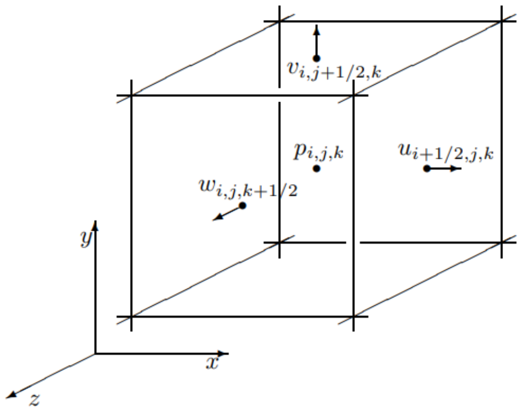


Figure 2.5: An example of a 3D MAC grid. Figure from [3].

It also expands well into 3D examples, such as Figure 2.5

### 2.1.7 Biot–Savart law

The Biot-Savart law is a way to obtain velocity at a given point from the vorticity [10]. It is defined as equation 2.9 below:

$$d\vec{v} = \frac{\vec{\omega} \times \vec{r}}{4\pi r^3} \tag{2.9}$$

where $d\vec{v}$ is the change in velocity at a given point

$\omega$ is the vorticity.

$\vec{r}$ is the position of the given particle relative to the vorticity origin.

It is a useful equation for vortex based simulation methods where the velocity is calculated based on points of vorticity, such as vortex particles.

### 2.1.8 Treecode

Computing the motions of fluids exactly is expensive. Approximations are therefore necessary to be able to run the simulation in real-time [10].

The Barnes–Hut algorithm [1] is an approximation technique widely used in computational astrophysics. These kinds of calculations are similar to the way the Vortex Particle Method works since VPM also calculates the velocity based on multiple individual sources of impact. It works by treating the nearby grid cells as individual particles but further away particles as virtual average particles. As opposed to the exact direct sum which is $O(n^2)$, the Barnes-Hut algorithm can run in $O(n \log n)$, which significantly speeds up the computation time for the velocity grid [10].

**Finite Difference Scheme**

Spatial derivatives are necessary in a handful of operations in fluid simulation. Finite Difference Schemes are a way to approximate spatial derivatives in a fast way with a loss in accuracy. Eq. 2.10 shows a centered difference scheme.

$$\frac{df}{dx} \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{2.10}$$

Velocity fields are discretised into grids to lessen the computational load of having the calculate the velocity at every point within the simulation space. To calculate the velocity at a certain point in between grid cells, interpolation is needed. This is done with spatial derivatives [10].

Finding the partial derivative of all the components of the velocity vector can be done with the Jacobian matrix:

$$\begin{bmatrix} \frac{\delta u}{\delta x} & \frac{\delta u}{\delta y} & \frac{\delta u}{\delta z} \\ \frac{\delta v}{\delta x} & \frac{\delta v}{\delta y} & \frac{\delta v}{\delta z} \\ \frac{\delta w}{\delta x} & \frac{\delta w}{\delta y} & \frac{\delta w}{\delta z} \end{bmatrix} \tag{2.11}$$

**Poisson**

The Poisson equation is a partial differential equation with many utilities in fluid simulation. It is similar to the Laplace equation in that the left hand side of the equation is the same. If the right hand side of the equation is zero, it is a Laplace equation and if it is non-zero, it's a Poisson equation. The Poisson equation can be solved in multiple ways, and this context it is used to obtain velocity from vorticity [10]

It can be defined as eq. 2.12.

$$\nabla^2 f = q \tag{2.12}$$

## 2.2 Literature review

This literature review will focus on real-time methods and tools. The academic field of computational fluid dynamics tend to focus on physical accuracy and visual quality over performance, while the commercial market has produced several tools the last decade specifically for games. It will therefore include a small number of non-academic tools that is worth mentioning.

### 2.2.1 Billboarding

Billboarding is a way to display a texture, often animated, on a 2D plane in a 3D scene. It is done by rendering a quad and having it always face the camera. This can then be used to render animated textures to fake more complex geometry such as explosions or smoke effects. It was frequently used in the 90's in games such as the original Half Life. They used billboards at the same location as the object at hand and rendered an animated texture showing flames and smoke emitting out of the object before disappearing. This technique was not very convincing but it was very cheap to render [25].

This technique is still used today, but often in more stylized games where the goal isn't to render realistic-looking smoke but rather cartoon-looking smoke or clouds, or for low LOD renderings where fine details would be lost anyway. There isn't much research on this method, as it is an old method and is not physically based.

NinjaFluid is also able to produce impressive interactive simulations in 2D with NinjaLive [1]. These simulations can include normal maps for the particles to allow for more complex illumination.

### 2.2.2 Curl Noise

Not all methods for creating plausible particle effects are based on the Navier Stokes equations. Curl-noise based methods are among these that have gotten the most friction. This method is based on Perlin Noise to generate velocity fields. Even though it is not physically based, it can produce impressive results with many parameters for the user to manipulate the flow. They have been shown to be highly relevant for real-time applications [4].

### 2.2.3 Style transfer

The most recent technique, style transfer with neural networks have gotten popular for image manipulation and synthesis. One paper has applied this technique to smoke simulation with interesting results [18]. This is however still far away from real-time performance.

### 2.2.4 Vortex systems

There are also Vortex Particle systems which are fully Lagrangian systems. These got major improvements around 2005 [17], but has gotten less popular since. These systems create velocity fields based on the individual particles' vorticity. It is primarily only used for smoke since free surface boundaries are much harder to deal with [3]. There are several variants of Vortex particle systems.

Over the years of 2009-2016, Dr. Michael J. Gourlay was hired by Intel to make a series of articles on real-time fluid simulation both theory and how to implement it in a real-time context [10]. This work is what the fluid simulation part of this implementation was based on. More about this method in chapter 2.3

### 2.2.5 Baking Simulations

Baking fluid simulations are getting traction with software like FluidNinja for Unreal Engine, or EmberGen by JangeFX [2]. Baking a simulation means that the velocity grid and its changes over time are pre-calculated and can run much faster with more details than interactive simulations. This comes with a cost, however. These simulations are static in they can not interact with a dynamic environment or player interaction with it. This method is promising for fluid systems that are either far away from the player or are not accessible by the players' input.

[1]https://unrealengine.com/marketplace/en-US/product/fluidninja-live

[2]https://jangafx.com/software/embergen/

Baking flow grid has a number of applications and is not exclusively useful for smoke effects. Baked animated grids for fire effects are very useful since these are less likely to be interacted with than e.g. smoke or dust flowing on the ground that the player is walking on.

### 2.2.6 Smoothed-particle hydrodynamics

Smoothed-particle hydrodynamics (SPH) is a Lagrangian method that is among the most used in video games for different kinds of physics simulations. It is however not very suited for smoke simulation due to its need for overlaping particles to be deleted and redistributed when in sparse environments[17]. This method, same as VPM, uses particles to carry the flow.

### 2.2.7 Constraint based

Constraint-based fluid simulation systems are a lesser used method relative to SPH and grid-based methods. NVIDIA has a library for fluid simulation which is called NVIDIA Flex [23]. This library is capable of simulating smoke, but does not directly support it. Constraint-based methods are well suited for real-time applications with a high emphasis on overall physics simulation. With this method every type of object, e.g. rigid bodies, gas particles, water particles, etc, is all in the same particle system. This makes interaction between different types of objects within the simulation cleaner as well as easier to implement since there is only need for one system as opposed to multiple different systems handling the different kinds of particles. Changing the phase of a particle is also trivial in this method. This could be for example when liquid water particles freeze into rigid bodies when frozen to ice.

Dennis Gustafsson, the developer of the game *Teardown* has developed his own constraint-based fluid system which supports an impressive amount of particles in real-time. This is however closed source and not made by a researcher so there are no papers to read. [3]

## 2.3 Vortex Particle Method

The Vortex Particle Method is the method that was chosen to examine in the implementation for this thesis. It is a lesser used method that has certain advantages for smoke and fire simulations over other more widely used lagrangian methods such as SPH.

These advantages include: no numerical dissipation for vorticity, and no divergence for velocity [17]. Since it is a lagrangian method, however, it is harder to calculate boundary conditions with it.

### 2.3.1 Vortex Particles

Vortex particles are particles that have vorticity assigned to them and are advected through the velocity field in the same way as passive particles. These vortex particles are often called *Vortons* while passive particles are often called *Tracers*.

[3]https://www.teardowngame.com/

A highly attractive property of this method is that it stays stable even with low viscosity [10]. This also means that it is easier for an artist to modify the properties of the simulation to the individual game scenes and settings. Other methods often need to apply more viscosity to counteract instability and loses the fine details which is a valued quality for smoke simulation.

The implementation of which this implementation is based on is a series of articles written by Dr. Michael Gourlay, finished in 2016 [10]. This series of articles was written for Intel to utilize their TBB parallelization library. Gourlay wrote this method specifically for game environments and has been optimized for real-time interactive contexts. It is however optimized for CPU. Gourlay argues that physics simulation on the GPU takes away vital computation time from the GPU for rendering, which was correct for the time. Since the start of that project, however, GPUs have increased their potential dramatically. With the trend of GPU performance when parallelizing, it has become a priority issue. NVIDIA has released several physics simulation libraries that utilize parallel GPU performance to great effect. With modern hardware, GPUs has more room for visual effects.

### 2.3.2 Uniform Grid

A uniform grid template is used for multiple different parts of this implementation. These grids have the advantage of being directly translatable which makes interpolation minimal. This implementation contains a grid template that is used for multiple different contexts. The ones used in this implementation are mentioned below.

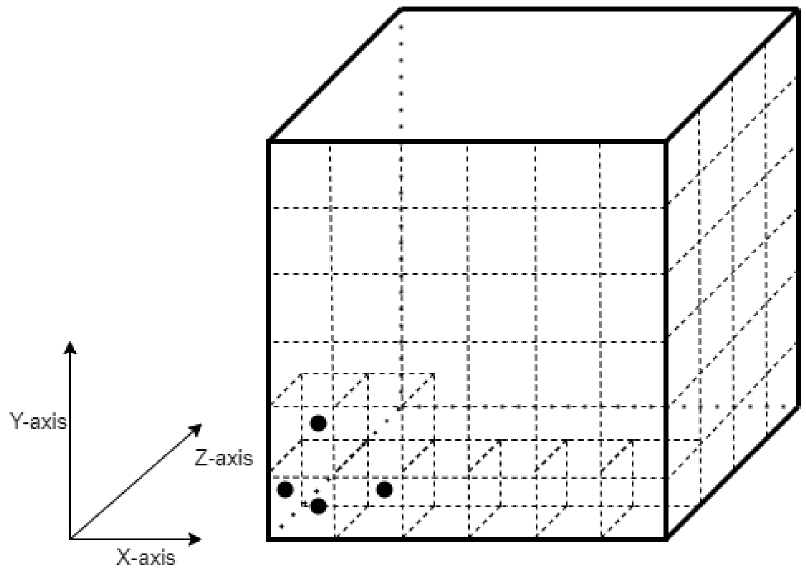


Figure 2.6: Uniform Grid in 3D

In Figure 2.6 we see that the grid cells have a fixed size and distance between each other. This is useful for interpolating which particles belong to each cell. In a grid like this, it is useful to have the grid point closest to the origo index as the "owner" of the cell. From that point, we know which other cells are part of the grid cell.

An example could be that if a grid cell is size 1 and starting index (1,0,0) in each direction, any particle with a position between 1-2 in in the X-axis, and 0-1 in the Y and Z axis belongs to this grid cell. By knowing the grid cell size, interpolation becomes trivial in this grid model.

#### Velocity Grid

The velocity grid is end result of the other grids during computation of the flow field. It is what the particles are advected through.

#### Influence Tree

The influence tree is a nested grid with decreasing resolution each layer. It is used for the octree to average out sections of the flow field that is not being queried for the given particle.

#### Density Grid

The density grid is used to compute the buoyancy of the particles relative to the environment. If the particles are lighter than the surroundings, they will float upwards and if they are heavier they will sink. This applies to the surrounding air around the particles and the mass of the particles.

### 2.3.3 Velocity from vorticity

Other methods usually create the velocity field by interpolating the velocity of the particles to update the velocity grid. From the Navier Stokes equations, vorticity is a product that is extracted from the velocity equation later on. VPM creates the velocity field from the other way around, by assigning vorticity to vortons, advecting them through a field, and calculating the velocity field by working our way backward through the Navier Stokes equations. [10]

The equation for calculating the change in vorticity in a field is given by[10]:

$$\frac{D\vec{\omega}}{Dt} = (\vec{\omega} \cdot \vec{\nabla})\vec{v} + v\nabla^2\vec{\omega} + \frac{\nabla\rho \times \nabla p}{\rho^2} + \vec{\tau} \tag{2.13}$$

where $\omega$ is the vorticity,

$\vec{v}$ is the velocity,

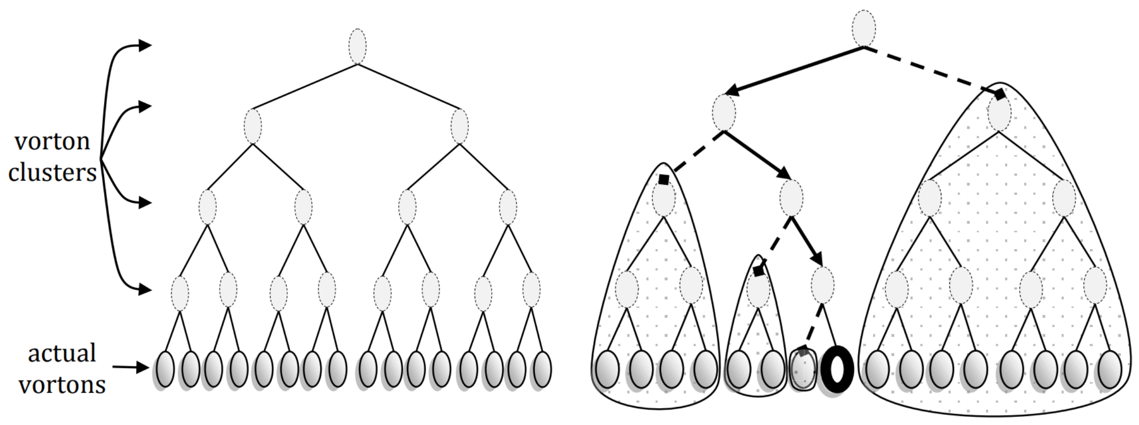
$p$ is the pressure field,

$\rho$ is the density of the fluid,

and $\vec{\tau}$ is the external torque.

### 2.3.4 Octree

Octrees are a way to partition vortons into trees where neighbour vortons are close both physically in the simulation and in memory. These trees can be searched with a query vorton. As seen in figure 2.7b, the vortons at the bottom of the tree are the real vortons, while the clusters further up are virtual super vortons. The vorton which matches the query is taken into the calculation directly while the vortons that does not match can be averaged out to save computation time [10].



(a) Overview of vorton cluster

(b) Search and average out

Figure 2.7: Figure from [10]. Octree of vortons.

Octrees are a good option for real-time fluid simulations that values performance over accuracy. It is used to average out the impact a vorton has on a given particle. This reduces precision but greatly enhances the performance impact.

These clusters are sometimes called ”super vortons”. The averaging to create these are done with equation 2.14, where the magitude of the vortons are taken into account.

$$\vec{x}_{\sum} = \sum_{i=1}^{N} \omega_i \vec{x}_i \tag{2.14}$$

### 2.3.5 Simulation stages

#### Simulation Loop

**Algorithm 1:** Vortex Particle Simulation Loop [10]

```
1 Find Bounding Box
2 Create Influence Tree
3 Compute Velocity Grid
4 Stretch and Tilt Vortons
5 DiffuseVorticityPSE
6 Advect Vortons
7 Advect Tracers
```

#### Find Bounding Box

The bounding box in this context is the surrounding box encapsulating all the particles in the simulation. This needs to be re-calculated each iteration. It is done by simply finding the minimum and maximum x,y, and z position coordinates of the particles and adding a margin of 2 x vorton radius. The grid template which is used to generate the other types of grids is then updated to account for the new, usually increased size.

**Create Influence Tree**

The influence tree is a nested grid with decreasing resolution with each layer. The grid is filled with vortons, where each layer averages out the impact each vorton has on a grid cell.

**Compute Velocity Grid**

The velocity grid is a uniform grid with the same bounding box as the grid for particles and vortons. An important property of uniform grids is that they translate directly without expensive interpolation.

**Stretch and Tilt**

The vortons in the simulation needs to stretch and tilt according to their environment. They need to adapt to the amount of mass which is in a given location. The more mass a location has, the slower a vorton will rotate.

**DiffuseVorticityPSE**

Particle Strength Exchange is a method used to smear the vorticity of the vortons to their neighboring vortons. For this method, we reuse the uniform grid instead of doing a neighbor search for faster computation [10]. This is the diffusion part of the Navier Stokes equations described in Equation 2.7 adapted to the vortex particle method.

The smearing is computed with

$$v\nabla^2\vec{\omega} \tag{2.15}$$

where $v$ is viscosity,

$\nabla^2$ is the spacial partial derivative of the position,

and $\vec{\omega}$ is the vorticity of the vorton at hand.

**Advection**

Advection is done both to the particles and the vortons as the forces impacting both vortons and the particles are the same. The list of particles is looped through where the velocity of each given particle is interpolated with the velocity grid. Since this method uses uniform grids, finding the grid cell that owns the particle simplified significantly. When the position of the particle is known the lookup is as simple as finding the grid point that is closest to the particle relative to the minimum grid point and the 7 other points connecting the cell will be known.

The velocity grid has a velocity value computed for each cell which then advects onto the particles within the cell. This value is then adjusted by the timestep chosen in the configuration.

# Chapter 3

# Technological background

## 3.1 Rendering Particle Systems

This section is based on OpenGL and uses its terminology and functionality to describe the technologies utilised in this implementation. OpenGL is the most used graphics API for developing computer graphics applications in both 2D and 3D [14]. The beginning of this chapter will briefly go over fundamental functionality before diving into the more advanced techniques.

### 3.1.1 Shaders

Shaders in computer graphics are programs on the GPU that take in among other things, vertex data, color data, and the translation matrix. There are several different shader languages, where this implementation uses GLSL shaders. In the rendering pipeline, there are mandatory shaders and optional shaders. The mandatory shaders are the Vertex shader, and the Fragment shader, while the optional include the Geometry shader and the Tesselation shader.

#### Vertex Shader

The first necessary shader in the pipeline is the vertex shader, which processes the vertices that are sent in with a vertex array object. This stage is generally used to calculate position, translation between coordinate spaces, The vertex shader is executed once for every vertex that is to be rendered [14]

#### Fragment Shader

The output of the rastorization is the input of the fragment shader. This is the last programmable stage in the graphics pipeline. This program is executed once for every fragment in the rastorized input image. The input fragments corresponds to the rastorized pixels of the input geometry. It processes each fragment to assign color and depth.

### 3.1.2 Particle Systems

Particle systems are often used in computer graphics to render fuzzy objects like fireworks, or smoke, that are hard to render and animate with well-defined geometry like meshes. They usually consist of a large number of individual points with their own position, size, and other properties so that they can be animated on their own [28]. These points are sometimes rendered as textured points or simple quads with post processing to make them look like the desired particle type.

### 3.1.3 Instancing

Instancing has been supported by OpenGL since version 3.1 [14] and is a way to render multiple instances of a single object.

One of the big bottlenecks to performance is transferring data back and forth between the CPU and the GPU. Smoke simulations tend to render single points as particles and run post-processing like blur or apply a texture to make them look like smoke particles. It can be useful to render meshes as particles in certain contexts, like in a collection of asteroids. When rendering particle systems, thousands or millions of particles are rendered with identical points. This can be significantly wasteful to the bandwidth having to send identical data to the GPU every frame. Instancing is a way to send a mesh to the GPU once, with a baseline position, and then apply an offset buffer with position offsets for every particle. This can make rendering a significant amount of identical meshes greatly more efficient, see figure 3.1.

In OpenGL, this is done by making a new buffer with offset position data, and when making the draw call, you use glDrawElementsInstanced instead of glDrawElements. [14]

```
glDrawElementsInstanced(GLenum prim,        // the kind of primitive to render.
                        GLsizei elCount,    // the number of elements to render.
                        GLenum indType,     // the datatype indices is.
                        const void* indLoc, // the location of the indices.
                        GLsizei instCount); // the amount of instances wanted
```

When instancing, it is also needed to make a call to glVertexAttribDivisor. This tells OpenGL how often to update the the data in the vertex buffer. If it is set to 0, it will update each iteration of the shader call. [14]

```
    glVertexAttribDivisor(2,  // Vertex attribute index
                          1); // How many instances should pass in between
                           ↪  updates of the buffer.
```

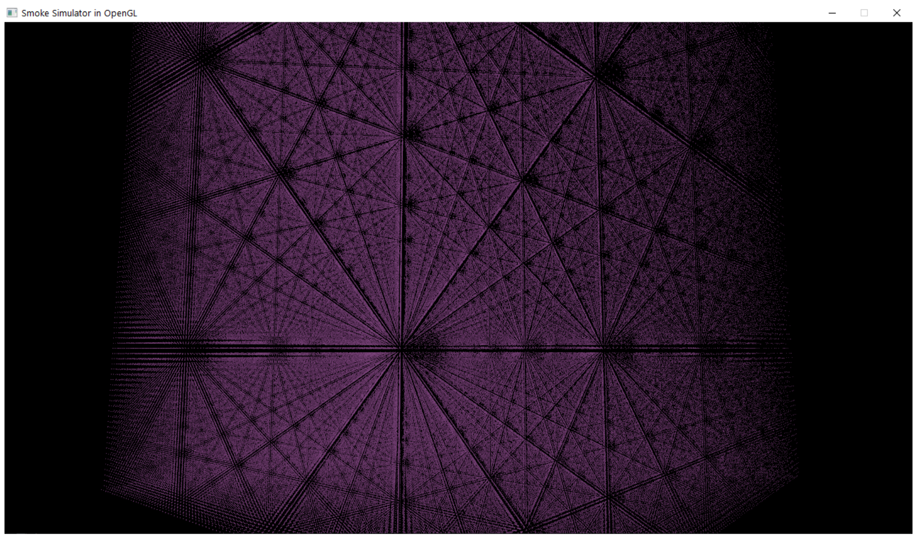


Figure 3.1: One million particles rendered as cubes using instancing in a uniform distanced grid

With instancing, we're able to render a significant amount of meshes without the major overhead. Figure 3.1 shows a million cubes rendered in real-time. One million cubes requires eight million vertices without instancing. This can be reduced down to the original eight needed for the cube plus one million offset vertices which is a significant decrease in bandwidth usage.

### 3.1.4 Persistent mapped buffer

Persistently mapped buffers are a faster way to fill the vertex buffer than the standard fill buffer in OpenGL. It is done by signing a contract with the GPU that this piece of data will only be done certain operations to in this time period [7]. This creates an immutable pointer to the location of the buffer data which can then be written to or read from.

When mapping to or reading from a buffer asynchronously they need fences to lock the buffer. This fence is for the gpu to not read data while it is written to. The buffer is locked when updated using:

```
syncLock = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
```

After the writing to the buffer is done, open the fence using:

```
if(syncLock){
    while(true){
        GLenum waitForReturn = glClientWaitSync( syncLock,
        ↪  GL_SYNC_FLUSH_COMMANDS_BIT, 1 );
        if(waitForReturn == GL_ALREADY_SIGNALED || waitForReturn ==
        ↪  GL_CONDITION_SATISFIED) return;
    }
}
```

### 3.1.5 Framebuffer Objects

In a typical OpenGL rendering setup, the rendering is done to the default framebuffer which is displayed to the screen. Additional framebuffers are called framebuffer objects (FBOs), and can be used to render whole scenes or parts of scenes to an off-screen buffer for additional processing before being applied to the screen at a later time [14].

The FBOs have textures or renderbuffers attached to them which is what they render onto, which can be used later. These textures can be either a color buffer, depth buffer, or a stencil buffer. This can be useful for e.g. post-processing or deferred rendering, to create more complex illumination with less draw calls.

### 3.1.6 Shadow Mapping

Shadow mapping is a technique used in computer graphics to simulate shadows in a scene. It is a typical component in rasterization pipeline rendering. It works by first rendering the scene from the light source's point of view with depth testing enabled to an off-screen framebuffer. This is then used as a texture in the on-screen render from the camera's point of view. This texture contains depth information that the on-screen render can use to determine whether the point is lit by the light or not. It does this by comparing the depth value from the framebuffer and the current depth value. If the point has a higher depth value in the second render than from the shadow pass render, that point is blocked by an object from the light source's point of view and is therefore in shadow.

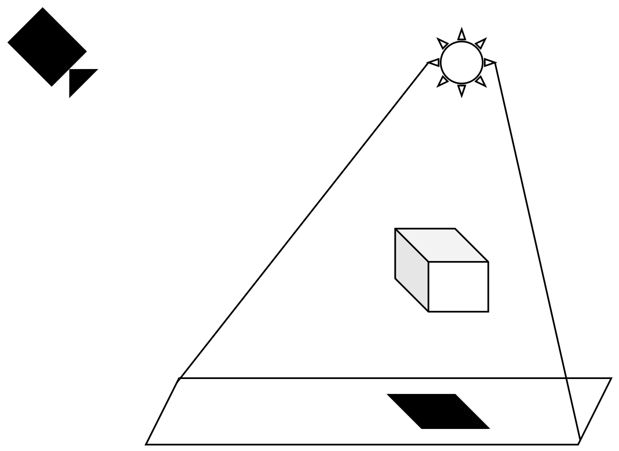


Figure 3.2: Shadow mapping

### 3.1.7 Volume Rendering

There are several ways of rendering volumes such as smoke. The naive way to render a point cloud is to simply render a point for each vertex. Without proper illumination this will simply display a flat cloud of point with no depth. Without shadows, the point cloud will look closer to 2D which defeats the point of doing volumetric simulations.

The illumination of the smoke particles is an essential step to creating a realistic representation of realistic smoke. This implementation is using Half-angle slice rendering, which is a technique from the early 2000's and was popularized by, Real-Time Volume Graphics(2004)[15], and later

NVIDIA in GPU Gems [9]. Simon Green, working for NVIDIA, in 2008 when he implemented it into a smoke simulator, and became part of the CUDA toolkit demos [12].

An important part of illuminating particle systems simulating smoke is the ability to cast shadows on to itself. This is the major effect which highlights the depth of the point cloud. While the illumination model for game environments is an artistic choice, users tend to favor illumination models which include hard shadows to perceive depth within volumes [21].

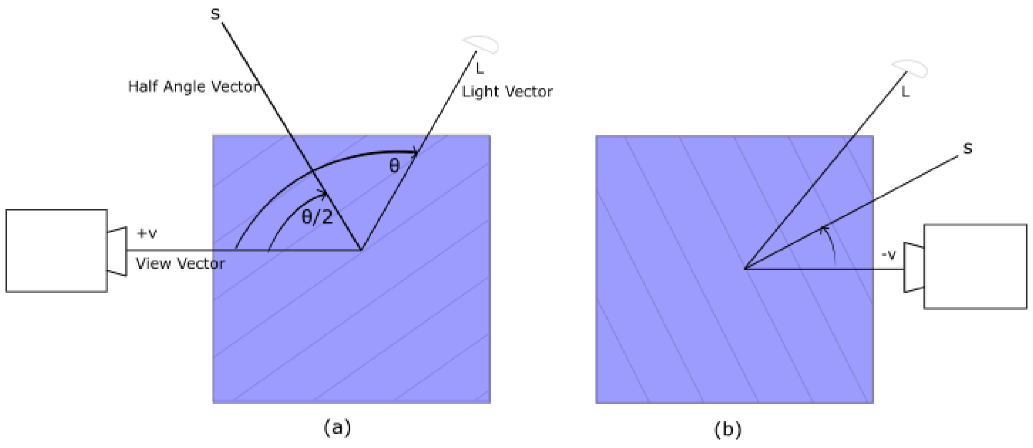


Figure 3.3: Diagram of general consept

Half-angle slice rendering is a volume rendering technique that works by sorting the particles by the axis in between the view vector and the light direction vector. This axis is updated every iteration by simply computing the dot product of the two vectors. By shadowmapping each slice and allowing it to accumulate shadows in the depthbuffer, we can simulate the light going through the smoke. With a single shadowmap pass we would not be able to draw soft shadows within the smoke or how the light goes through it.

When rendering multiple layers to an imagebuffer, composition needs to be correct for whether the new layer is in front or behind the existing layer [15]. The color and alpha blending rendering front-to-back is done with eq. 3.1 and 3.2

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \tag{3.1}$$

$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \tag{3.2}$$

When rendering back-to-front, the blending is done with eq. 3.3

$$C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + C_{src} \tag{3.3}$$

### Post-Processing

Post-processing is calculations done to the image after rasterization. This can include edge detection, blurring, film grain, or any kind of color alteration of the image. It is generally the last stage of rendering, and is typically done in the fragment shader.

For this application blurring is used on the particles to increase their size and to appear more as dust or smoke particles instead of solid dots. The particles are rendered by themselves to an off-screen image framebuffer before blurring is applied in the X and Y-axis of the image. This is then rendered to a quad with the same size as the screen which is applied on top of the other geometry rendered in the scene.

The two-pass Gaussian blur filter as the blur kernel, as seen in Equation 3.4.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{x^2}{2\sigma^2}} \tag{3.4}$$

where $\sigma$ is the standard deviation of the distribution

$x$ is the coordinate of the given pixel in the image.

The blurring is applied to the rastorized image in two passes. This method is called two-pass blurring, and works by first applying the blur kernel in the two axises sequencially in two drawcalls. By doing this as opposed to blurring both axis in a single drawcall, the number of samples necessary is dramatically reduced.

## 3.2 Parallelization for Real-Time Performance

### 3.2.1 CPU vs CUDA

GPU computing power has far succeeded CPU computing power the last 10 years. CPUs generally have between 4 and 12 cores, while modern GPUs can have hundreds or thousands of cores. While at a lower clock speed, the utilization of parallel computing over multiple cores has a significant impact on performance. In terms of parallellization, GPUs are significantly more effect at larger datasets. For smaller datasets and fewer calculations, CPU computation is still more efficient because of the higher clock speed and no need for data transfer between units.

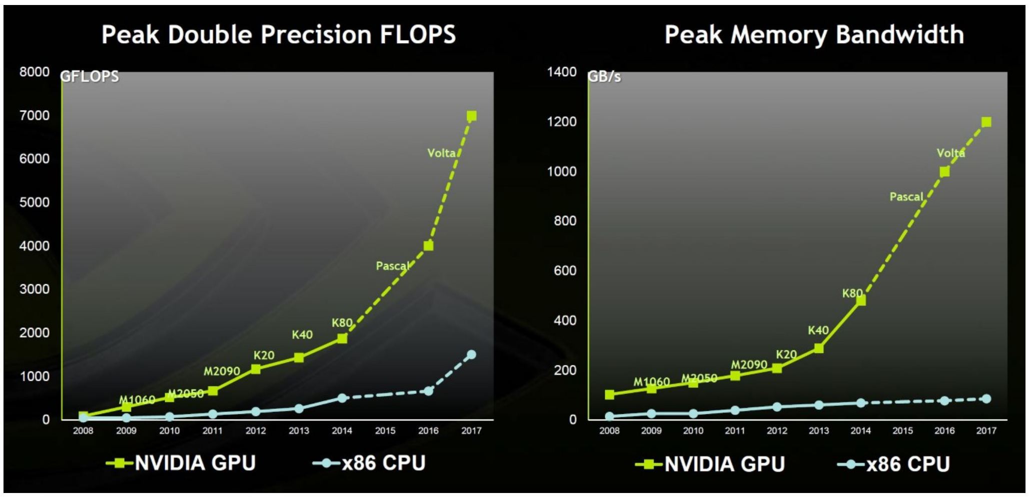


Figure 3.4: Performance for CPU vs GPU in giga flops. Source: NVIDIA

This graph shows the performance increase up until the Volta architecture for NVIDIA GPUs and x86 CPUs.

Performance tests were conducted to determine what kind of memory management was best suited for this application. CPU and CUDA testing was done on an Intel i9 9900 and NVIDIA RTX 2070S.

Unified memory is another approach to memory management in CUDA, where the data is available to both the CPU and GPU. The data is automatically copied between CPU memory and GPU memory and can be accessed at will. This eases the development process and increases productivity, but at a cost of lower efficiency [19]. The common GPU programming model is to initialize the data on the CPU, allocate the needed memory on the GPU, transfer the data to GPU memory, execute parallel computation and finally transfer the data back to CPU memory. These steps can be greatly simplified using unified memory [20], where the developer only needs to allocate the unified memory, fill that memory with the needed data and finally run the parallel computations on the GPU.

$$f(x, y) = cos(x) * sin(y) \tag{3.5}$$

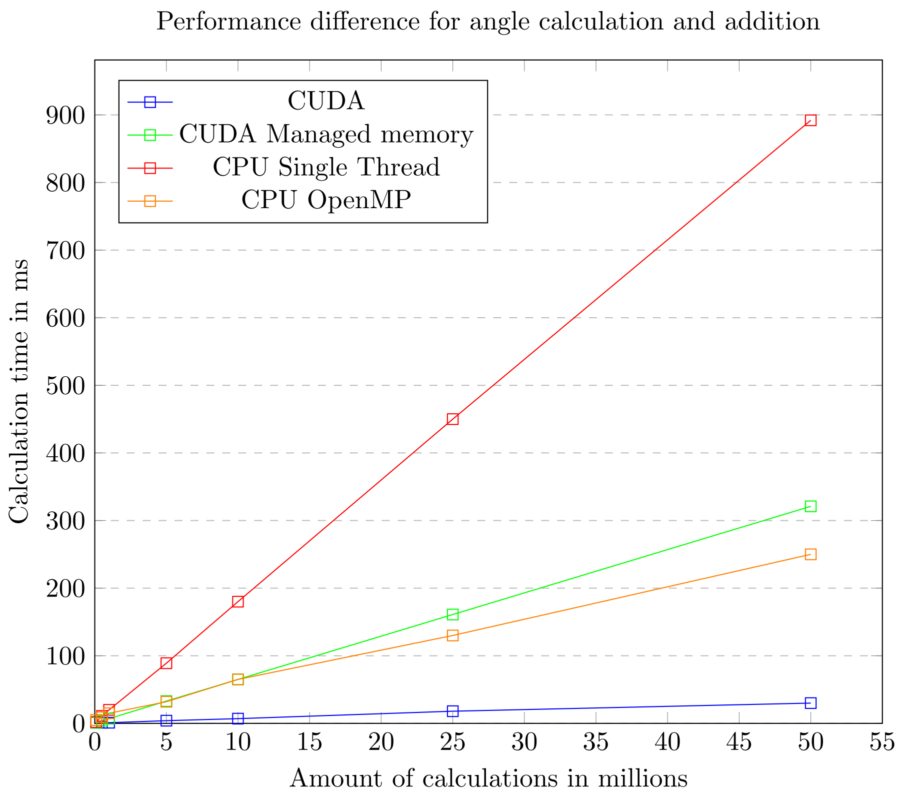


Figure 3.5: Performance difference for angle calculation and addition

These performance largely tests agree with a paper from 2019 benchmarking unified memory in different contexts [19]. For 50 million calculations we see that CUDA with manually allocated memory is 29.7 times faster than CPU single thread. This benchmark includes transfering the data back from the GPU. If the program is designed so that the data can live on the GPU without the need to transfer it back and forth, a higher speed up can be expected.

Unified memory with CUDA is easier to implement in existing CPU programs than standard memory management in CUDA. It is therefore a good tool for prototyping GPU implementations or if only a small amount of data needs to be accessible to both the GPU and CPU. This benchmark does not perfectly reflect how these methods will impact performance in a fluid simulation application but it gives an indication that unified memory isn't well suited for the massive datasets that is required for big scale fluid simulations with millions of particles.

### 3.2.2 Parallelization methods

One of the biggest contributions to making realistic physics simulations is parallelizing on the GPU. NVIDIA has made major contributions to translate existing methods to CUDA with vast improvements in performance [22] [6] [23].

#### OpenMP

OpenMP is a parallelization library for easy parallelization on the CPU. It is meant as a library to lessen the optimization burden from the developer over on the compiler. It can easily be implemented in existing serial projects with compiler flags as long as there are no dependencies within the operations. A significant amount of operations common for fluid simulations are parallelizable by default with grids and operations performed on every particle within each grid cell. This is an easy way of increasing the performance of your code without altering the overall structure. It is however still CPU bound and can only parallelize to the number of cores in the CPU.

After being set up with the compiler flag -fopenmp can be implemented in a C++ project as simply as the following code.

```
#include <omp.h>

#pragma omp parallel for
for(int i = 0; i < MAX_PARTICLES; i++){
    particle[i].velocityUpdate();
    particle[i].positionUpdate();
}
```

#### CUDA

CUDA is a GPGPU (General-Purpose Graphics Processing Unit) API that can be used for graphics computing as well as general purposes. GPUs are optimized for matrix multiplication with their massive amount of threads for parallelization. A CPU typically doesn't have more than 12 cores, a modern typical GPU can have thousands of cores. While these cores are not as powerful as a CPU core, they still outperforms CPUs with their parallelization capabilities.

After a whitepaper from 2008, titled *Particle Simulation using CUDA* [11] was released by Simon Green from NVIDIA, more and more real-time methods started implementing CUDA into their own methods. Almost every fluid dynamics paper and library released by NVIDIA since then have used CUDA, including [6], [23].

### Compute Shader

Another option to run the computations in parallel on the GPU is a compute shader. They have been supported in OpenGL since 4.3. Shader code is meant to run in parallel to begin with and compute shaders are no exception. It is therefore a viable way to run code in parallel in graphics programs or general programs on the GPU.

A compute shader is not part of the render pipeline and is meant to compute arbitrary information. It therefore behaves a bit differently than regular shaders in the pipeline. They also have no user-defined inputs or outputs at all. For input, the compute shader must fetch the data from a sampler. It can then write indirect output to a texture which can then be read from [13].

One advantage compute shaders have over CUDA is that they are platform-independent. CUDA is developed for NVIDIA GPUs, and while there are ways to convert this code to AMD GPUs e.g.

Chapter 4

# Implementation

This implementation is a modified port of VorteGrid by Dr. Michael Gourlay [10]. VorteGrid is a free and open source series of articles with accompanying source code on fluid simulation using the vortex particle method for real-time applications. These articles explore a variety of methods of accomplishing real-time fluid simulation, where this port focused on the smoke simulation using the treecode and poisson solvers. While the original implementation was parallelized using Intels TBB library, this port was first experimented with using OpenMP for prototyping, and later ported to CUDA.

## 4.1 Benchmarking modules sequentially

.

Benchmarking of the different modules in the simulation was done to determine which modules were the most computationally expensive. This testing was done sequentially without parallelization and on the CPU to better understand how big the computational load each module was relative to the simulation as a whole and how big of a priority it was to port them to the GPU. This specific benchmark was run in a mode to stress all modules, including solving boundary conditions with one sphere flowing through the particles.

| Module | Particle Count | Vorton Count | Avg $\mu$s CPU | % of Total |
|---|---|---|---|---|
| FindBoundingBox | 691 200 | 491 | 3238 | 15% |
| CreateInfluenceTree | 691 200 | 491 | 2558 | 11.7% |
| ComputeVelocityGrid | 691 200 | 491 | 1147 | 5.2% |
| StretchAndTiltVortons | 691 200 | 491 | 43 | 0.19% |
| DiffuseVorticityPSE | 691 200 | 491 | 79 | 0.36% |
| AdvectVortons | 691 200 | 491 | 7 | 0.03% |
| AdvectTracers | 691 200 | 491 | 12 020 | 55% |
| SolveBoundaryConditions | 691 200 | 491 | 2588 | 12% |
| GenerateBaroclinicVorticity | 691 200 | 491 | 26 | 0.11% |
| **Total** | | | 21 706 | 100% |

Table 4.1: Performance test of the simulation modules on CPU ran without any parallelization. Intel i9 9900

These benchmarks showed that the advection of the particles was by far the most expensive module in the simulation. This module contains a loop over all the particles and an interpolation method for calculating which cell in the velocity grid that the given particle should be given its velocity from.

The second most expensive module was finding the overarching bounding box of the whole particle set. This module also loops over each particle to reduce the maximum and minimum vertex coordinate among the particles.

The modules related to creating the velocity field was found to be light weight relative to the heaviest ones, only taking up 18% of the total computation time. These modules are the ones that require multiple different grids and the array containing the vortons. The loops in these modules iterate over the 3D grids with index offsets and some rely on the previous iteration, which makes them hard to parallelize. It was therefore chosen to let them stay on the CPU.

## 4.2 Fluid simulation

The following psuedo code shows an overarching view of the simulation loop.

**Algorithm 2:** Simulation loop psuedo code

```
FindBoundingBox particles:
    /* This is done using CUBs reduce functionality */
    minVal ← $min(currentMin, particle.pos)$
    maxVal ← $max(currentMax, particle.pos)$
Create Influence Tree:
    Initialize uniform grid to bounding box
    Calculate base layer of the grid using the vortons
    for i in layers do
        average section of grid using the previous layer
Compute Velocity Grid:
    for i in gridCells do
        Interpolate velocity at gridcells using the influencetree
Stretch and Tilt Vortons:
    Compute the gradient of the velocity field using jacobian matrix
    for i in vortons do
        interpolate $vorton_i$ to jacobianVelocityField
        update $vorton_i$.vorticity
Diffuse Vorticity:
    for each gridCell do
        diffuse vorticity in neighbouring gridcells
Advect Vortons:
    for i in vorton do
        advect $vorton_i$ by interpolating its position to the velocity grid
Advect Particles:
    for i in particle do
        advect $particle_i$ by interpolating its position to the velocity grid
Solve Boundary Condition:
    for i in rigidBodies do
        for i in particles do
            if(distance from particle to surface  particle.radius + surface):
            calculate new speed of particle based on impact with rigid body
```

The overarching modules of the fluid simulation part of the program are illustrated in the flow chart below. Each node represents a function or a set of functions revolving around one specific task in the simulation. These are often one step in the Navier Stokes equations or a necessary step in between them.

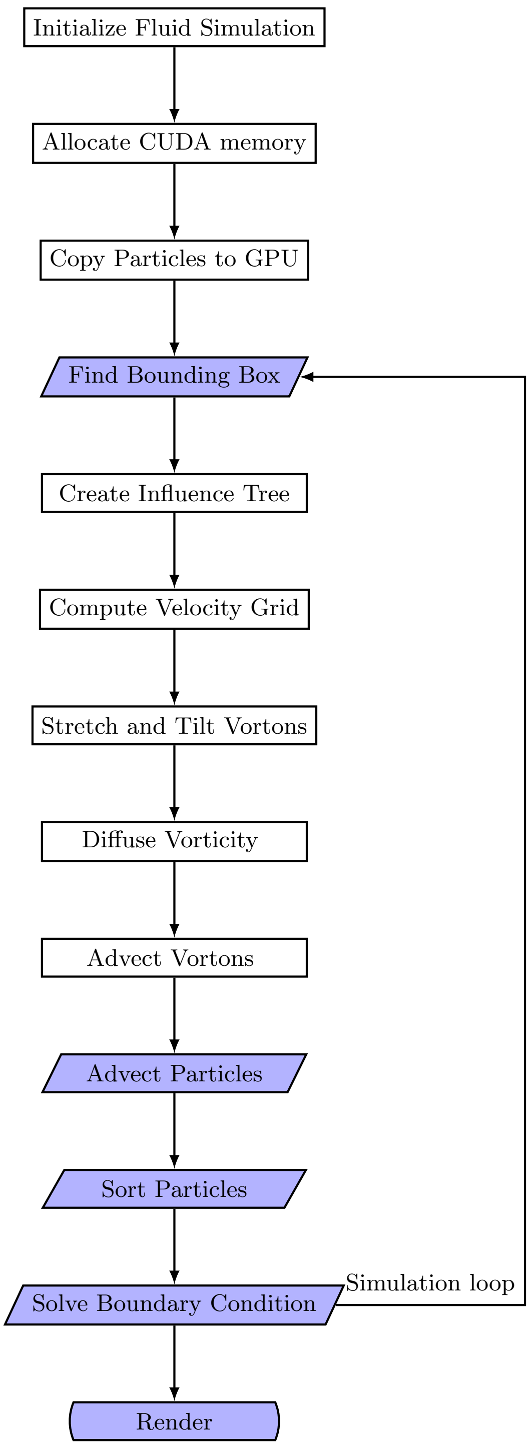


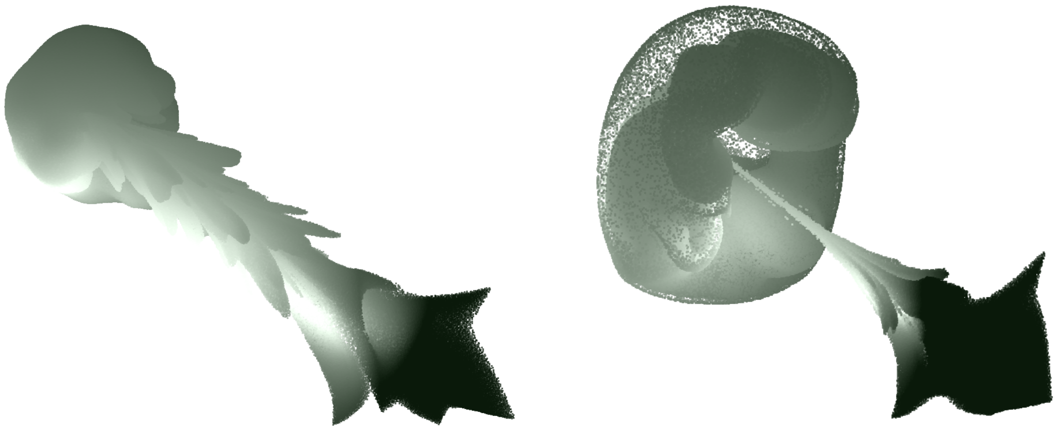
Figure 4.1: Flowchart of CUDA memory management for the particles in the program

In figure 4.1, white nodes are computed on the CPU, while blue nodes are computed on the GPU using CUDA. Find Bounding Box and Sort Particles are computed using CUBs reduce and deviceRadixSort respectively. After Diffuse Vorticity, the velocity grid is finished and is transferred to CUDA. This grid is typically not bigger than 27x27x27, where only one vertex is needed per cell which is not a big load on the bandwidth.

### 4.2.1 Treecode vs. Poisson solver

Gourlays' articles included a variety of solvers. Among these, treecode and Poisson were experimented with both for performance and visual quality. The Poisson solver runs in O($N^{(}3/2)$) while the treecode solver runs in O(N log(N)) [10]. These solvers have different advantages and disadvantages in each favor, and can be used in different scenarios where they are the most suitable.

Comparisons were made to determine the advantages and disadvantages of the different solvers. When comparing the solvers, performance, stability, and visual quality were taken into account.



(a) Treecode

(b) Poisson

Figure 4.2: Initial launch with 1 124 864 particles

Figure 4.2 shows an example where the particles are launched with an initial speed at a diagonal angle with 1 124 864 particles. In figure 4.2b we see that this Poisson implementation tends to work well for mushroom clouds but generally has fewer fine details as most of the particles are inside the plume. In figure 4.2a we see that the treecode implementation tends to have finer details and have more space between the particles along the velocity direction. Both of these have use cases for each visual quality and need to be evaluated to best fit each implementation.

Table 4.2: Overall performance comparison of the solvers in mode 1

| **Solver** | **Particle Count** | **Vorton Count** | **Avg Fps** |
|---|---|---|---|
| Treecode | 175 616 | 105 | 320 |
| Poisson | 175 616 | 105 | 460 |
| Treecode | 1 124 864 | 804 | 105 |
| Poisson | 1 124 864 | 804 | 130 |

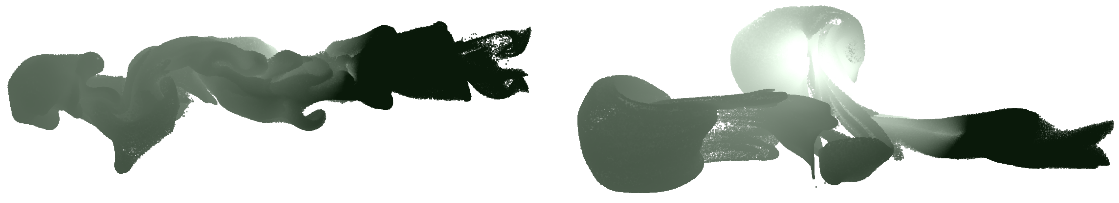

(a) Treecode (b) Poisson

Figure 4.3: Particles colliding with sphere with 1 097 600 particles

Figure 4.3 shows an example where the particles are initially passive with zero velocity and have a sphere shot into them. The velocity field is created by the impact of this collision. In this example, we also see that the treecode method shown in figure 4.3a forms more fine details, while the Poisson method shown in figure 4.3b tends to form mushroom clouds. An interesting note is that the Poisson example in figure 4.3b split into two mushroom clouds instead of one, even with just one sphere colliding with the particles.

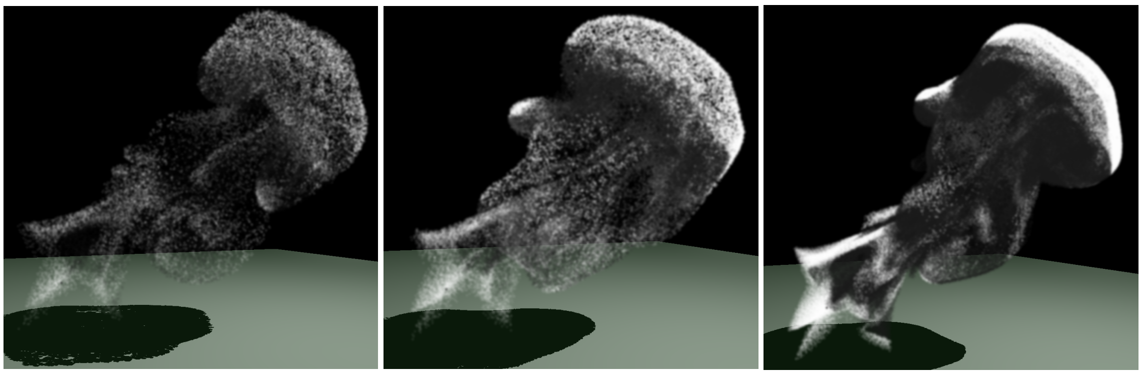
Table 4.3: Overall performance comparison of the solvers in mode 2

| Solver | Particle Count | Vorton Count | Avg Fps |
|---|---|---|---|
| Treecode | 89 856 | 59 | 370 |
| Poisson | 89 856 | 59 | 470 |
| Treecode | 1 097 600 | 491 | 130 |
| Poisson | 1 097 600 | 491 | 155 |

### 4.2.2 Finding the optimal amount of particles

There is a fine balance between simulating enough particles to convincingly portray a continuum of smoke and when more particles do not help the visual quality. The amount of particles heavily impact the performance of a simulation. It is therefore important to approximate a sweet spot in between not simulating enough particles and wasting computation time.

This topic can rely on the context of which the simulation is used in, and there is no answer for all of them. It also relies on rendering factors such as blurring nad the size of particles rendered. If the size is too big, the volume loses detail and if they are too small the volume loses continuity. When including blur in the rendering of the particles, 60 thousand to 70 thousand particles seems to be a suitable number for one smoke source. The NVIDIA 2010 demo used 65 000 which are shown in figure 5.2.



(a) 42 thousand particles (b) 75 thousand particles (c) 343 thousand particles

Figure 4.4: Different number of particles

In figure 4.4a, we see that even with two gaussian blur passes the individual particles are clearly defined. In figure 4.4c, the continuum of particles has less patches and looks more like a thick smoke, but is also more computationally expensive.

### 4.2.3 Memory management

Memory management was handled by initializing the particles on the CPU before allocating them in CUDA. The biggest data sets like the particle array "lives" in CUDA and have all necessary computations done there without transferring them back to the CPU. The lesser data sets, e.g., the min/max coordinates of the grid, which needs to be calculated each iteration, are allocated in unified memory to keep CPU/GPU cross computation easier.

To minimize the data transfer necessary each iteration, the original particle object has been broken up into individual arrays with just the necessary data needed. This includes their position, velocity, angular velocity and age. These arrays are passed to the CUDA kernels only where they are needed.

### 4.2.4 Updating the VBO

VorteGrid, which this implementation is based on was already highly optimized and designed for real-time computation. Benchmarking of the different modules was done, where advection and copying/ moving of data were the most time-consuming operations. With hundreds of thousands or millions of particles all containing 3D vertex data, this amount of data needs to be transferred in the most optimal way. Various different methods were experimented with for updating the VBO; namely, persistently mapped buffers [1], bufferSubData [2], but the method that was found to be the most viable was to update the VBO from CUDA itself. This allows the data to stay on the GPU instead of being transferred through the CPU to fill the OpenGL buffer.

[1]https://www.khronos.org/opengl/wiki/Buffer$_{O}bjectPersistent_{m}apping$
[2]https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBufferSubData.xhtml

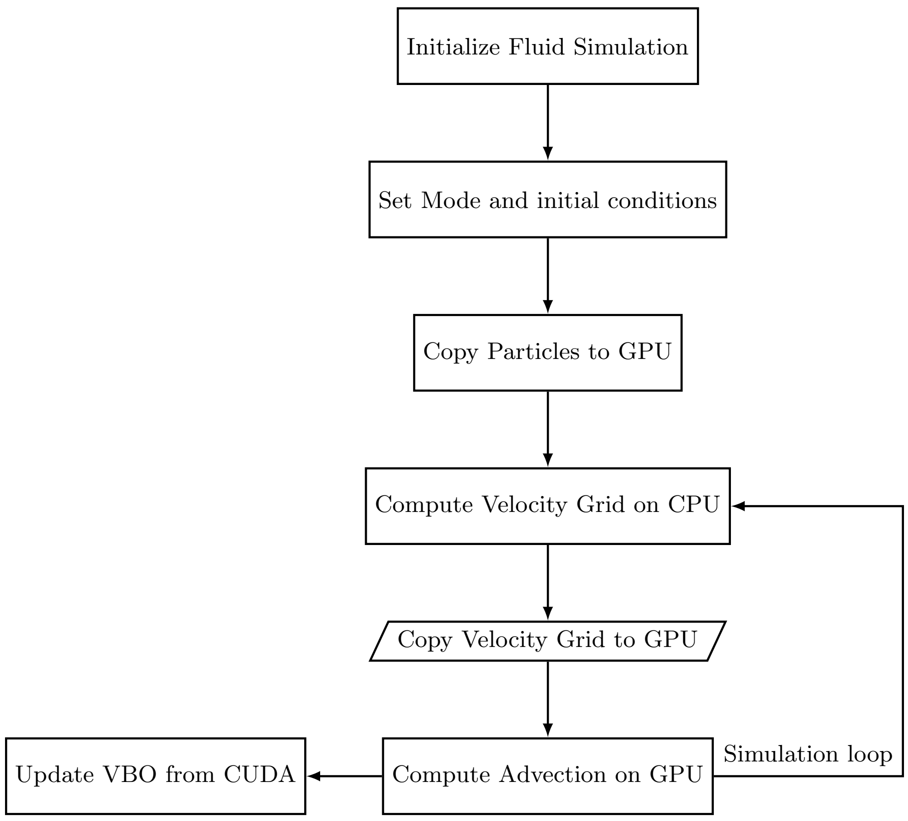

Initialize Fluid Simulation
Set Mode and initial conditions
Copy Particles to GPU
Compute Velocity Grid on CPU
Copy Velocity Grid to GPU
Compute Advection on GPU
Update VBO from CUDA
Simulation loop

Figure 4.5: Flowchart of the GPU memory management in the program

Figure 4.5 depicts the flowchart of how memory management is handled in this implementation. The VBO updating is done by first creating a VBO with OpenGL. This returns a pointer to the memory on the GPU, which can then be registered with CUDA. Before advecting the particles, this pointer is mapped to CUDA to signal that it is ready to be written to. After advection, this pointer is then unmapped which opens up the VBO to OpenGL and makes it available for drawing.

## 4.3 Rendering

The rendering setup for this implementation is based on Half-Angle Slice rendering. Two FBOs are used to enable shadowmapping and post-processing.

### 4.3.1 Rendering process

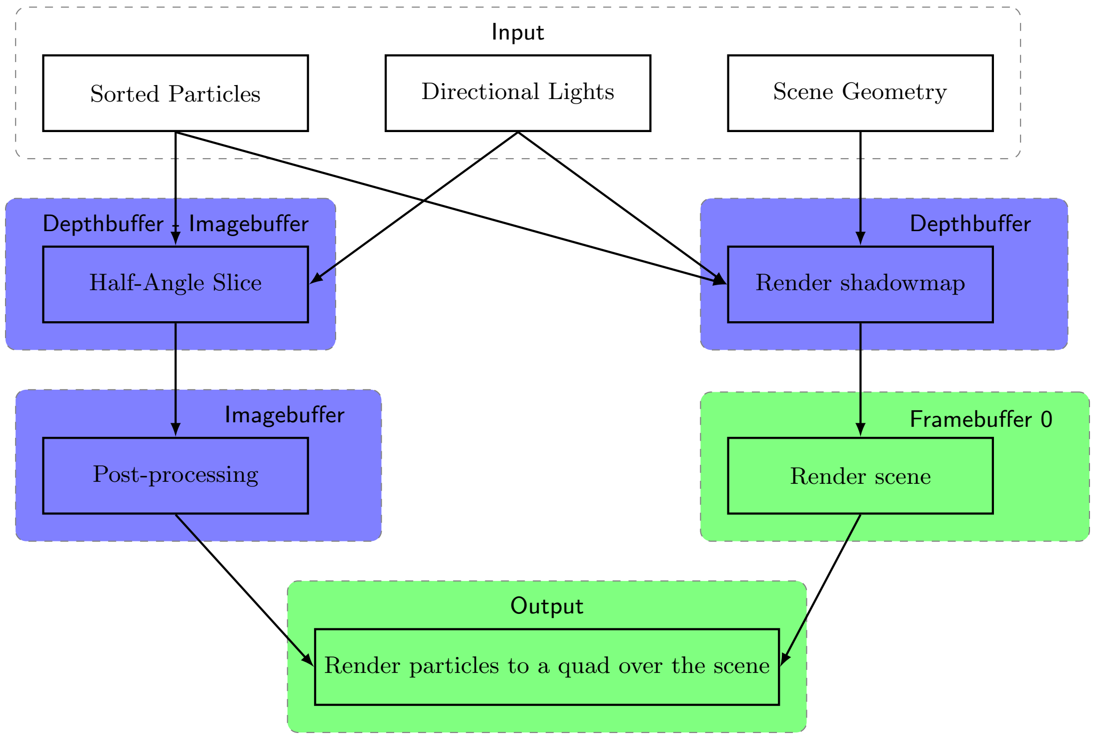
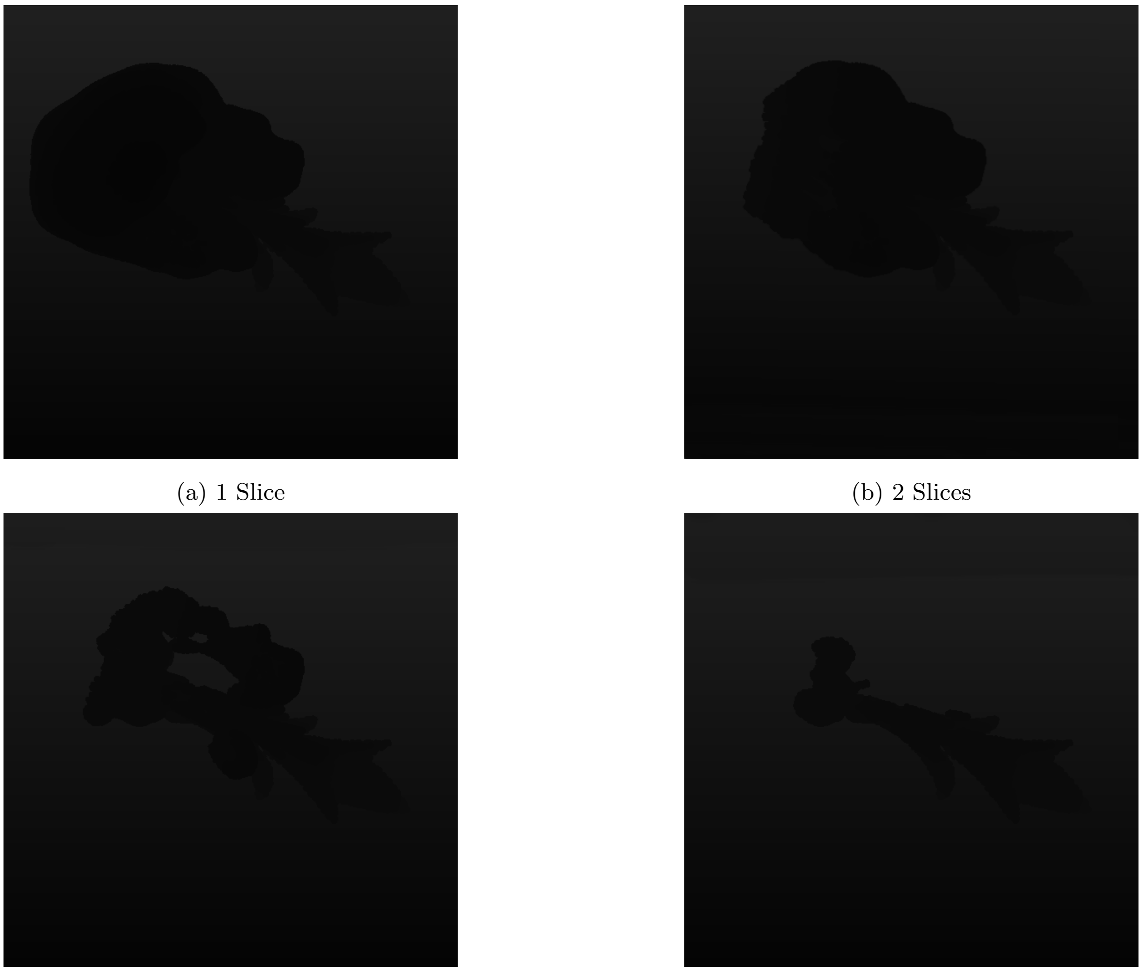


Figure 4.6: Flowchart of the rendering process

Figure 4.6 shows the flowchart of the rendering process. In this diagram, green depicts on-screen rendering, and blue depicts off-screen rendering.

The sorted particles is the output of the smoke simulation. These are already sorted before they are taken as input to the rendering. A directional light is used for shadowmapping. The geometry in the scene outside of the particles is shadowmapped by itself and rendered to the default framebuffer.

### Shadow Mapping


(a) 1 Slice

(b) 2 Slices

(c) 4 Slices

(d) 8 Slices

Figure 4.7: Last slice of shadow map independently rendered

Figure 4.7 depicts the last slice of the shadowmap using a different number of slices. In figure 4.7(d) we only see the last 1/8 of the particle rendered.

This shadowmap is taken in as a texture when rendering to an imagebuffer along with the light's view matrix. With this we can interpolate what parts of the geometry is visible to the light and draw shadows accordingly.

## Rendering Loop

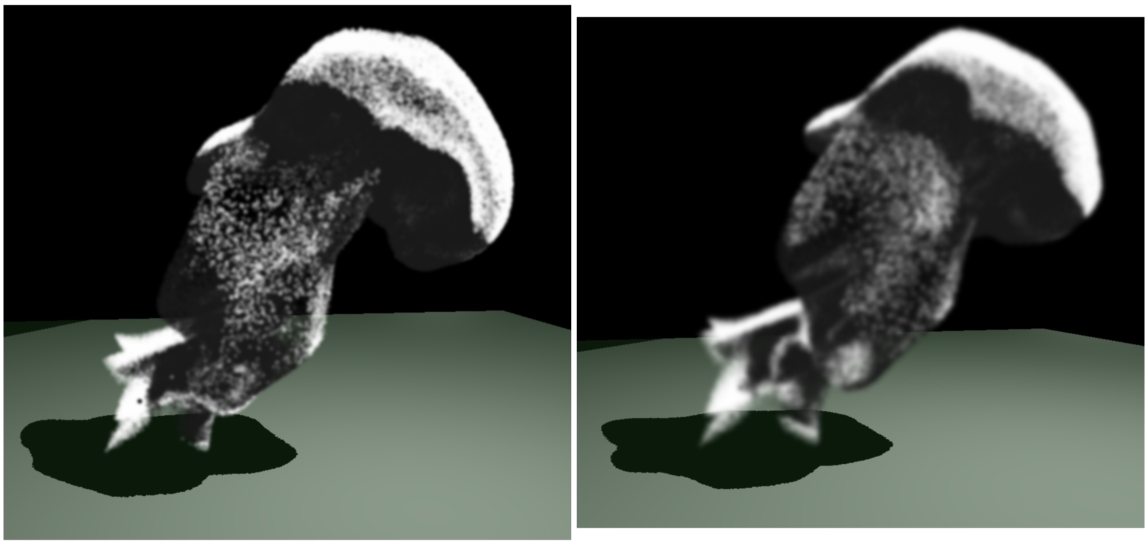
**Algorithm 3:** Rendering loop

```
1  halfAngle ← dot(viewVector, lightVector)
2  Sort particles relative to the half-angle axis
3  batchSize ← particleAmount / numSlices
4  for i in numSlices do
5      Activate depthFBO
6      Activate depthBufferShader
7      render from index (i * batchSize) to (i * batchSize + batchSize)
8      Activate imageFBO
9      Activate particleShader
10     render from index (i * batchSize) to (i * batchSize + batchSize)
```

The particles are rendered as primitives *GL_POINTS* to minimize bandwidth load. With this, each particle only requires one vertex for position. The points are rendered as single quads always facing the camera. These quads have hard edges and are solid colored. To make these quads look more like dust or smoke particles a blur is required to smooth out the edges and make them fade into each other and the background. The gaussian blur algorithm was chosen as the blur function due to its ability to be split into two passes, which significantly reduces texture samples needed.

When rendering the particles they are rendered in slices front to back from the perspective of the light. The blend mode in OpenGL needs to be updated to reflect whether the camera is seeing the light through the smoke such as in Figure 3.3 (a), or that the light is illuminating the particles on the same side as the camera such as in Figure 3.3 (b). These are adjusted to *glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE)* for scenario (a), and *glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)* for scenario (b). Each slice is first rendered to an off-screen depth buffer before rendering the slice with the new shadow buffer. The advantage of this technique is that it is relatively fast when sorting the particles on the GPU as well as its ability to render hard shadows within the particle cloud.

### Post Processing


(a) 1 Gaussian blur passes

(b) 5 Gaussian blur passes

Figure 4.8: Amount of blurring done in post processing

The blurring is done with a 2-pass Gaussian blur with a 5x5 kernel. This is done in an off-screen imagebuffer where they rotate which texture it should render to each draw call.

### Rendering circles

The particles are rendered using *GL_POINT*, which by it self is a simple quad. Circular particles with soft edges are more suitable than squares with hard edges. One way to do this is to load in a circular pre-rendered texture to map onto the quads. A better way to achieve this is to process the quads in the fragment shader before the illumination. This is done by calculating the center of the point, and discarding all fragments outside a given radius of this. This also allows for soft edges by gradually increasing the alpha of the point with how far away a fragment is away from a particle center.

The following code snippet is included in the particles fragment shader.

```
float dist = distance(gl_PointCoord.xy, vec2(0.5, 0.5));
if(dist > 0.5) discard;
float alpha = clamp((1.0 - dist), 0.0, 1.0);
```

This finds the center of the particle using the *gl_pointCoord* glsl variable. This variable contains 2D coordinates of where inside a point primitive the given fragment is. By discarding everything outside of a given radius of the center of the particles, we are left with just the particles themselves in the imagebuffer.

Using this method we also get soft edges for free by getting the alpha value from the distance to the center of the particles.

## 4.4 Debugging

Debugging a graphics program is different than other programs. A lot of the coding is happening on the GPU which has limited debugging capabilites relative to the CPU. Shaders have no print or breakpoint functionality, so external programs are needed to debug them. This section will touch upon the programs used to debug this implementation.

### 4.4.1 RenderDoc

RenderDoc is a debugger for graphics applications. It is able to capture frames and display the current shaders, OpenGL calls, etc. It is a useful tool for debugging parts of graphics programs that are usually hard to debug manually. GPUs often have limited functionality for printing information or displaying debugging values. CUDA has functionality for printing inside of kernel code, while shaders in OpenGL have no such functionality. One technique for manually debugging shaders is to attach contrasting colors to the relevant parts of the scene from the fragment shader or adjust the size to relevant parts of a mesh in the vertex shader to make them stand out. [3]

### 4.4.2 NVIDIA Nsight

NVIDIA Nsight is a debugging tool developed by NVIDIA. It is essential for debugging CUDA code since it can capture all relevant calls to the GPU as well as display performance data for individual calls. A big benefit of using this to debug CUDA programs is its ability to list all CUDA calls used and show detailed error messages not available when running the program normally. [4]

[3]https://renderdoc.org/
[4]https://developer.nvidia.com/nsight-graphics

# Chapter 5

# Results

## 5.1 Performance

### Overall performance

All performance tests are done on an NVIDIA RTX 2070S, Intel i9 9900, and with a resolution of 1280x1024.

Graph showing the overall performance of the implementation relating to particle count.

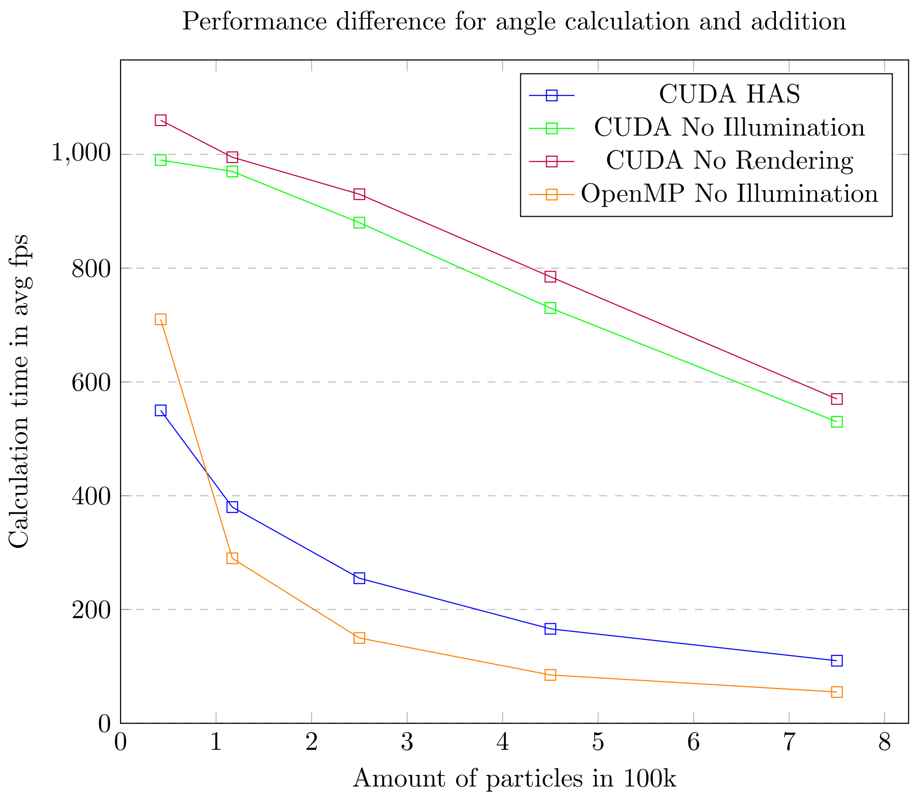


Figure 5.1: Performance of VPM CUDA

Figure 5.1 shows the overall performance in different settings. All of these configurations used the Poisson solver for fluid simulation.

CUDA HAS - a configuration with Half-Angle slice rendering, 4 slices, 1 Gaussian blur iteration.

CUDA No Illumination - volume rendering and sorting turned off, only rendering the particles as simple points.

CUDA No Rendering - is the simulation running without rendering the particles. This configuration is also without sorting the particles since this is only a necessary step for the volume rendering.

OpenMP No Illumination - This is the prototype based in OpenMP. It does not include sorting or any illumination, simply rendering the particles as white points. Note that this implementation is not representative of the original implementation by Gourlay, as that was parallelized with TBB.

**Relative performance**

| Implementation | Particle Count | Slice Count | Avg Fps | Avg ms |
|---|---|---|---|---|
| NVIDIA 2010 | 65 000 | 1 | 425 | 2.35 |
| NVIDIA 2010 | 65 000 | 4 | 390 | 2.56 |
| NVIDIA 2010 | 65 000 | 8 | 370 | 2.70 |
| VPM CUDA Treecode | 74 088 | 1 | 620 | 1.12 |
| VPM CUDA Treecode | 74 088 | 4 | 450 | 2.22 |
| VPM CUDA Treecode | 74 088 | 8 | 275 | 3.63 |
| VPM CUDA Poisson | 74 088 | 1 | 690 | 1.44 |
| VPM CUDA Poisson | 74 088 | 4 | 490 | 2.04 |
| VPM CUDA Poisson | 74 088 | 8 | 370 | 2.70 |

Table 5.1: Overall performance relative to the NVIDIA 2010 demo

VPM CUDA is the implementation for this thesis.

The number of particles in this implementation is based on the number of vortons in each grid cell. This makes it difficult to choose an exact number of total particles. The benefit was therefore given to the contender.

Table 5.1 shows how well this implementation performs relative to the Smoke Particles demo by NVIDIA, which is part of the CUDA Toolkit. VPM CUDA outperforms the NVIDIA demo with lower slice counts but falls short when increasing the slice count. This is largely due to not enough optimizations for the volume rendering.

**CUDA vs CPU modules performance**

Table 5.2: Performance differences of CUDA and CPU (sequential).

| Module | Particle Count | Avg $\mu$s CPU | Avg $\mu$s GPU | Perf. Gain |
|---|---|---|---|---|
| FindBoundingBox | 204 800 | 1019 | 94 | 10.8x |
| AdvectTracers | 204 800 | 4161 | 127 | 32x |
| SolveBoundaryConditions | 204 800 | 912 | 76 | 12x |
| FindBoundingBox | 1 097 600 | 5352 | 130 | 41x |
| AdvectTracers | 1 097 600 | 21196 | 260 | 81x |
| SolveBoundaryConditions | 1 097 600 | 4813 | 120 | 40x |

Table 5.2 depicts how the CUDA implementation scales relative to the sequential CPU prototype. The CUDA kernels will eventually hit a ceiling where the performance gain will stop expanding relative to the sequential implementation.

**Memory usage**

The memory usage of these applications was measured using the Visual Studio profiler. This profiler displays how much RAM is used at any given point.

Table 5.3: Memory use per implementation

| Module | Particle Count | Size in Mb |
|---|---|---|
| NVIDIA2010 | 65 000 | 339 |
| VPM CUDA | 74 088 | 350 |
| VPM CUDA | 175 616 | 380 |

The visual studio profiler measures the memory usage for the whole program and is therefore not representative of the simulation or rendering per se. Some of the memory usage comes from the graphics engine running, which an application needs anyway. As discussed in subsection 4.2.2, 75 000 particles is a suitable amount for a single source of smoke. When increasing the number of particles, the memory usage only increased by 8%, even though the number of particles more than doubled which also requires a bigger grid and more information in the textures used for rendering. Relative to modern games or applications requiring multiple gigabytes of memory, this simulation is within an acceptable memory usage. It is not uncommon for games to have more than 8Gb of memory as the minimum. Cyberpunk 2077 [1] has 12Gb of memory as recommended for high graphics settings. Relative to this 350Mb would be a 3% increase.

**Method for benchmarking**

For benchmarking the standard c++ library Chrono was used to take the time of each module. A start and a stop variable was placed at each side of a function call and was given the time at those moments in time. The time difference was then printed out for each iteration. The average of those

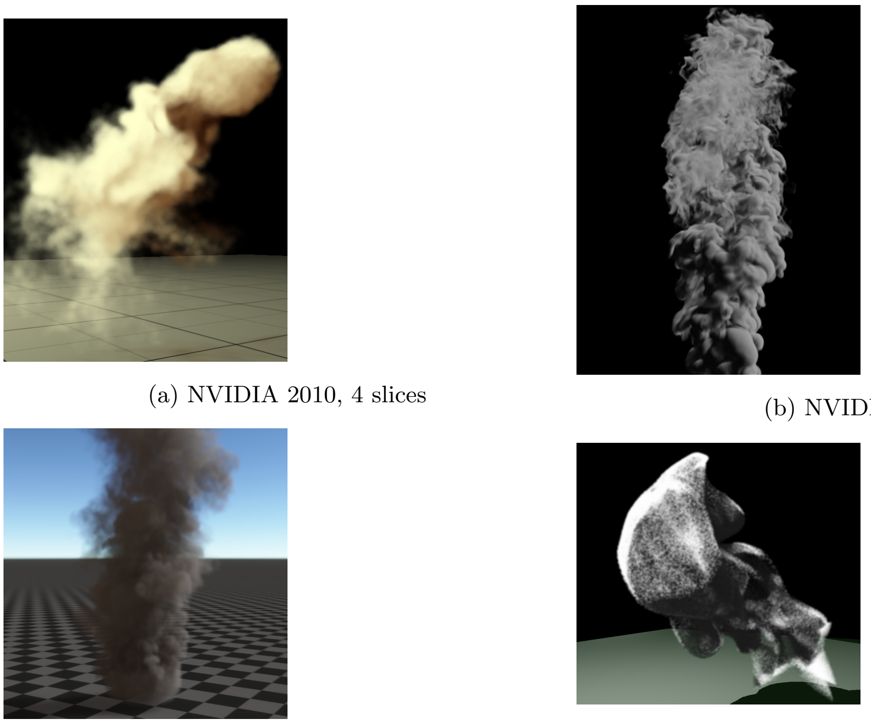
[1] https://www.cyberpunk.net/

was then calculated with a minimum of 20 iterations. For modules where there is a non-constant workload such as the SolveBoundaryConditions module, where the majority of calculations are only active when there is a collision happening, the average time was taken from when the actual particle-sphere interaction was occurring, as that is when the function is under the heaviest load.

CUDA kernels are launched asynchronously, which means that the CPU launches them and moves on until that finished processed data is needed. This means that a genuine benchmark needs the kernel to synchronize before the time difference between start and stop can be captured. For this *cudaDeviceSynchronize*() is used after the kernel launch and before the stop time is captured.

## 5.2 Visual results

Half-angle slice rendering was chosen as the most suitable volume rendering technique in terms of visual quality while still keeping a low computational impact relative to global illumination methods. NVIDIA Flow and EmberGem uses ray marching and ray tracing which has high visual quality, but is more expensive to compute.


(a) NVIDIA 2010, 4 slices

(b) NVIDIA Flow

(c) EmberGem

(d) VPM Cuda treecode 4 slices

Figure 5.2: Comparison between the visual quality of these methods.

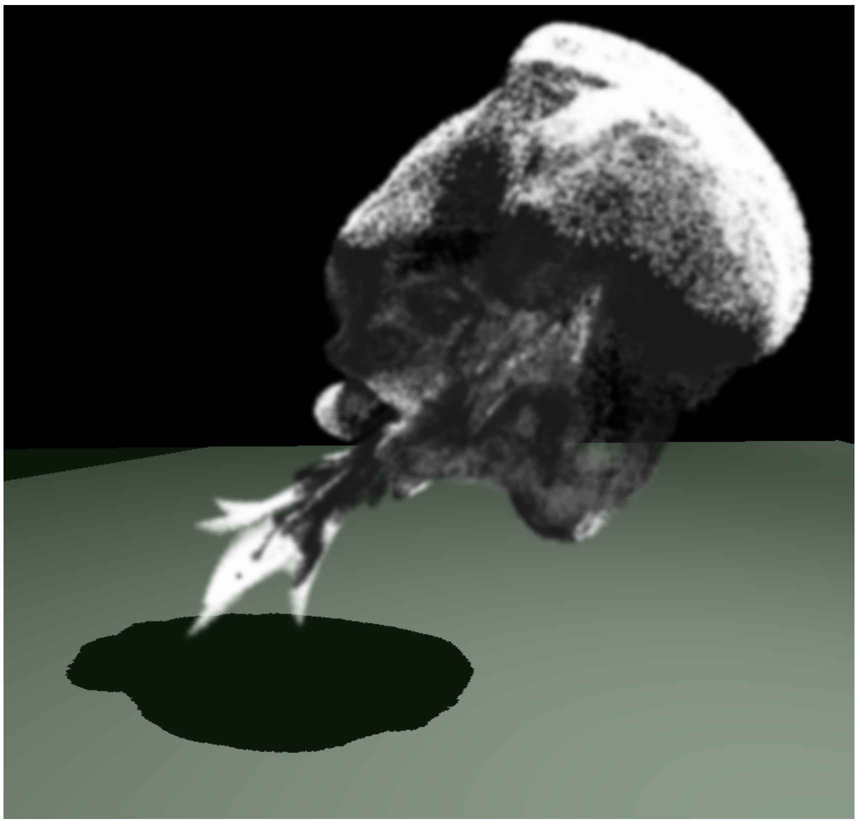


Figure 5.3: 250 000 particles rendered with 8 slices and 1 gaussian blur pass

Figure 5.3 is an example of rendering thick smoke using this implementation.

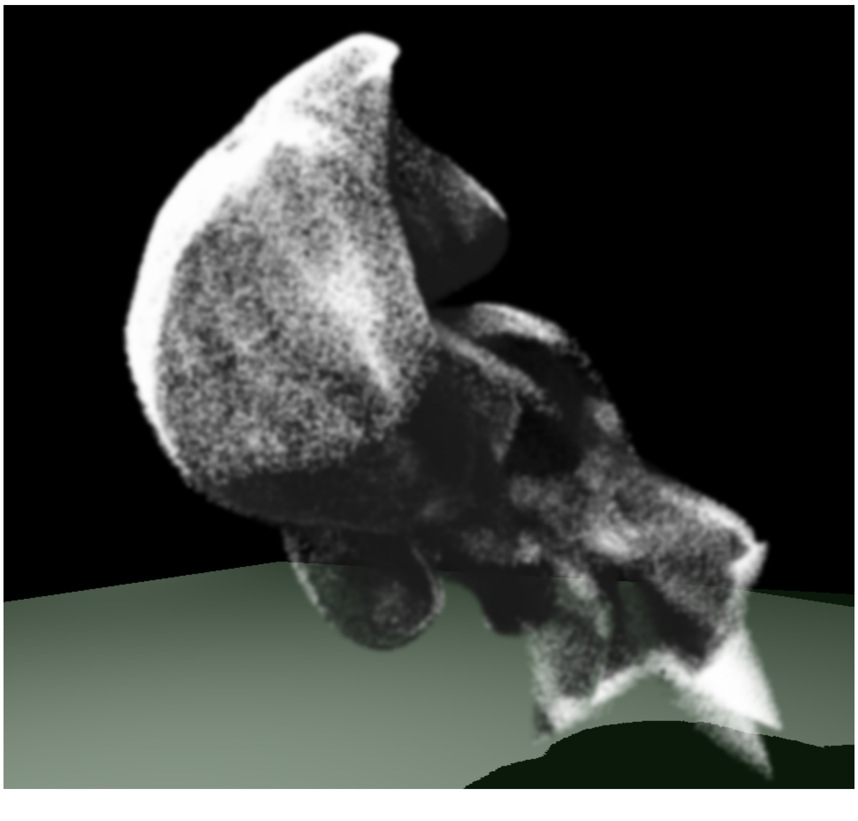


Figure 5.4: 340 000 particles rendered with 4 slices and 1 gaussian blur pass

Figure 5.4 shows rendering thick smoke with more particles but less slices. This is running at 200fps.

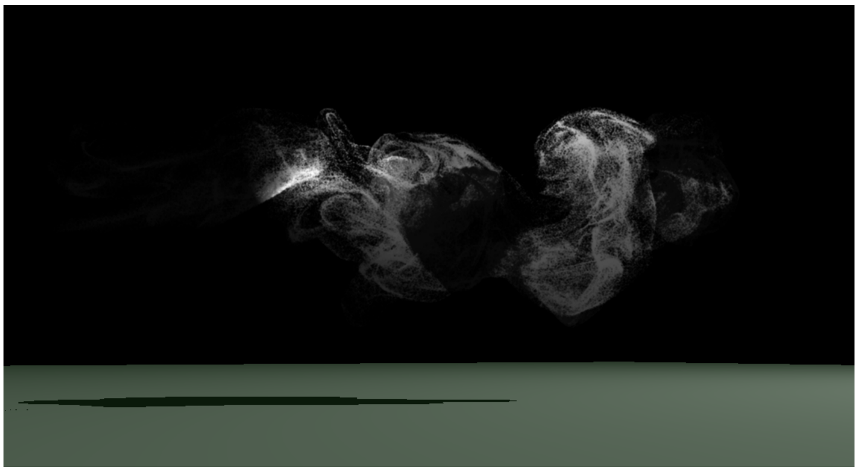


Figure 5.5: Thin smoke with simple shading and 1 gaussian blur pass

Figure 5.5 shows a render of thin smoke, which is possible with decreased particle size.

## 5.3 Comparison to other methods

Finding fitting implementations for comparisons was problematic. Very few modern high-quality demos with performance statistics were available. The methods chosen for comparison were two NVIDIA demos. NVIDIA is one of the leading companies related to real-time computer graphics with a good number of libraries specifically for real-time physics simulation. NVIDIA 2010 is a demo from CUDA toolkit demonstrating a fluid simulation system with CUDA as well as Half-Angle slice rendering. This is not optimal since it is a relatively old demo but it serves well enough for comparisons since it has good customization parameters for an honest comparison and readily available performance statistics.

Very few modern implementations included an extensive profiler or enough parameters for an honest comparison to this implementation. NVIDIAs most recent library for physics simulation, NVIDIA Flow, was available to demo through their Omniverse platform. In this setting, Ray-Tracing could not be turned off and the customization was limited. On top of this, their profiler did not include dedicated simulation time or rendering time, but rather an overall CPU time and GPU time. This made it hard to create a scene that would be an honest comparison to either the simulation or the rendering, and it was therefore only included in the visual results comparison.

In the EmberGem demo, the profiler shows simulation time and rendering time separately, but Ray-Tracing was also the only rendering option. The profiler indicated that the number of particles did not affect the computation time for the simulation which indicates that either advection is not included in the time taking or that the profiler is not extensive enough which meant that it was not a suitable demo for performance comparison either.

The NVIDIA CUDA demo from 2010 was found to be the most suitable for comparison since the profiler and the allowed customization of parameters was extensive enough for an honest comparison and that it uses the same local illumination technique used in this implementation. This is still not an ideal comparison, however, since it is not meant as a demo to showcase a state-of-the-art method, but rather the capabilities of CUDA for high-performance computing.

There are multiple variables that impact the performance of an implementation that isn't always available in the given customizable parameters in the demo. The viscosity and the magnitude of the external forces are two factors that are impacting the simulation in a meaningful way but rarely given their own parameters to the user. This makes a true comparison harder since there is little room to equalize as many as possible parameters before benchmarking. How far away the camera is to the particles impacts the computational load on the shaders which is hard to make exactly equal. The comparisons in this section can therefore not be completely true comparisons.

## 5.4 Discussion

Table 5.1 shows that this method has potential. The Vortex Particle Method can run at an acceptable level for the performance requirements suitable for games. The stability of the simulation method is a good advantage for game designers in that they have more room for customization without fearing that the simulation will explode.

The NVIDIA demo is not a good implementation for comparison due to its age, and it's hard to compare directly with other implementations when there are many factors having a role on performance.

The Vortex Particle Method is still underdeveloped compared to SPH in terms of real-time applications research, especially for game environments. The method has more potential in that it can function well without viscosity to maintain the visual quality and fine details that smoke often has.

Volumetric particle effects are still relatively expensive by nature in that they require a significant amount of particles to convincingly represent a continuous medium. Recognizing when volumetric particle effects have value and when billboarding is sufficient is important to maximize the performance - visual quality balance. Billboarding still has a place in modern computer graphics where the detail added by volume rendering would be lost anyway.

Letting the major data sets stay on the GPU to minimize data transfer is needed for efficient computation. Letting CUDA update the particle VBO is a good way to do this. Persistently mapped buffer is a good alternative if this is not a possibility. Instancing is needed for rendering a significant amount of identical meshes, and can be easily implemented into existing graphics programs using an offset buffer.

Working on fluid simulation while optimizing for performance has shown that it's hard to beat the leaders of the field. Fluid simulation and real-time rendering are complicated fields where the leading actors have decades of experience, with teams including developers with PHDs in physics.

# Chapter 6

# Conclusion and further work

## 6.1 Conclusion

This thesis has explored the field of real-time fluid dynamics from the perspective of games and real-time applications. A CUDA port of VorteGrid with volume rendering has been presented. The rendering is done with multiple shadow map layers to achieve Half-Angle slice volume rendering. Two solvers to the Navier-Stokes equation using the Vortex Particle Method has been evaluated and benchmarked.

Experiments have been done to the various stages of simulating and rendering particle systems. This included data transfer between CPU and GPU, rendering the particles

The implementation performed well enough to meet most of the goals put forward. Relative to the NVIDIA smoke particles demo, it performed better with low slice count. Achieving up to 1000fps for 45 thousand particle and simple rendering.

## 6.2 Meeting the requirements

#### R1: Cheap to compute

This requirement was met well. With the results of this implementation shown in table 5.1, in a scenario where a game is running at 60fps, or 16.6667 ms per frame, in optimal conditions adding this simulation method including volume rendering would add 2.22 ms per frame of computation time, which reduces 60fps to 53fps.

#### R2: Stable solver

This was met well in that VPM is much harder to make explode than other methods.

**R3: Low memory usage**

This was met relatively well. It is hard to determine what is the true memory footprint of this simulation and rendering due to the profiler used only shows total memory usage, which is not representative of how much additional memory this implementation would use if it was put into an existing game.

**R4: Interactive**

This requirement was not met well. As of now, it can only do collision detection with spheres. For this requirement to be met well, it would need to be able to do collision detection with arbitrarily shaped meshes. Lagrangian method are also not good at boundary conditions, as they need to compute collisions for all particles.

## 6.3 Further Work

This subchapter will discuss functionality that would have been implemented if more time was available.

### 6.3.1 Optimizing Half-angle slice rendering

The implementation of Half-angle slice rendering is poorly optimized, and suffers from a high computational load with too many slices. Even though four slices was found sufficient for volume rendering, relative to the NVIDIA demo, it is not optimized enough for a high slice count which would improve visual quality.

### 6.3.2 AST grids

Adaptive Staggered-Tilted Grids are a recent improvement to the uniform grids that most fluid simulations use today. They work by adapting to the stress at individual regions in the simulation to make smaller grids where needed. What is unique about AST grids is that they insert tilted grids in between the uniform grid cells in areas of high stress which enables finer details than uniform grids. This is especially relevant for collision detection [29].

AST grids are complicated to implement but is relatively computational cheap. On average fluid simulations only requires 1-2 percent more computation time to process. This would be ideal for real time smoke simulation in game environments.

### 6.3.3 Arbitrary shape collision

As of now, this implementation only supports collision with spheres. It is possible to combine these to make an approximate surface on objects that have more complex shapes but it is not an elegant solution. A simple solution for a humanoid character - particle collision would be to attach a large sphere around the lower part of the model that pushes away the particles instead of handling all the individual normals of the mesh. This would be computationally cheap but loses visual quality.

### 6.3.4 Additional turbulence

Adding additional turbulence to the velocity field could improve the fine details of the smoke. This can be done with a similar approach as Curl-Noise based methods.

### 6.3.5 CUDA Streams

Asynchronous cuda streams could allow for the CPU to do work while waiting for a kernel to finish. The advection kernel for the particles depend on the earlier modules for it to have an updated velocity grid, but it could run at the same time as the vorton advection function. This function is generally fairly light weight as it has a insignificant amount of particles. With this optimization, only a slight improvement in performance could be expected.

# Bibliography

[1] Joshua Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] C. Braley and A. Sandu. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. 2009.

[3] R. Bridson. *Fluid simulation for computer graphics, Second Edition*. 01 2015.

[4] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Trans. Graph.*, 26(3):46–es, July 2007.

[5] Viviane Clay, Peter König, and Sabine Koenig. Eye tracking in virtual reality. *Journal of Eye Movement Research*, 12, 04 2019.

[6] Jonathan Cohen, Sarah Tariq, and Simon Green. Interactive fluid-particle simulation using translating eulerian grids. pages 15–22, 02 2010.

[7] Cass Everitt and John McDonald. Beyond porting: How modern opengl can radically reduce driver overhead, 2014.

[8] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, 2001.

[9] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[10] Michael J. Gourlay. Interactive fluid simulation for games and movies, 2008.

[11] S Green. Cuda particles. *NVIDIA Whitepaper, November*, 01 2008.

[12] Simon Green. Volumetric particle shadows. 01 2008.

[13] Khronos Group. Khronos wiki.

[14] Khronos Group. Opengl 4.5 reference pages.

[15] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-Time Volume Graphics*. A. K. Peters, Ltd., USA, 2006.

[16] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189, 1965.

[17] Zhanpeng Huang, Guanghong Gong, and Liang Han. Physically-based smoke simulation for computer graphics: a survey. *74*, Multimedia Tools and Applications, 09 2015.

[18] Byungsoo Kim, Vinicius Azevedo, Markus Gross, and Barbara Solenthaler. Transport-based neural style transfer for smoke simulations. 05 2019.

[19] Marcin Knap and Pawel Czarnul. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, 75, 11 2019.

[20] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. pages 1–6, 2014.

[21] Florian Lindemann and Timo Ropinski. About the influence of illumination models on image comprehension in direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1922–1931, December 2011.

[22] Miles Macklin and Matthias Müller. Position based fluids. *ACM Trans. Graph.*, 32(4), July 2013.

[23] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics*, 33:1–12, 07 2014.

[24] Jos Stam. Stable fluids. *ACM SIGGRAPH 99*, 1999, 11 2001.

[25] Jos Stam. Real-time fluid dynamics for games. 05 2003.

[26] Jos Stam. *The art of fluid animation*. 2016.

[27] Penelope Sweetser and Peta Wyeth. Gameflow: A model for evaluating player enjoyment in games. *Comput. Entertain.*, 3(3):3, July 2005.

[28] T. Theoharis, Georgios Papaioannou, Nikos Platis, and Nicholas Patrikalakis. *Graphics and Visualization: Principles Algorithms*. 01 2007.

[29] Yuwei Xiao, Szeyu Chan, Siqi Wang, Bo Zhu, and Xubo Yang. An adaptive staggered-tilted grid for incompressible flow simulation. *ACM Trans. Graph.*, 39(6), November 2020.

# Glossary

**Offline Rendering** Rendering is happening at non-interactive speeds. This typically renders to images that are meant to be grouped together as animation later.

**Real-time rendering** Running the graphics program at an interactive rate. Typically >30 frames per second.

**Tracer** Another name for particle. These are the passive particles that are rendered to the scene.

**VIC** Vorton in Cell is a hybrid between Particle in Cell and the Vorton particles used in Vortex based simulation methods.

**Volumetric Simulation** Volumetric simulation means that the fluid is calculated and rendered in 3 dimensions, as opposed to 2D with billboarding.

**Vorton** Another name for Vortex Particle. These particles exist to create the velocity field from their vorticity and are not rendered.

# Acronyms

| | |
|---|---|
| **AST** | Adaptive Stagered-Tilted Grids. |
| **FBO** | Framebuffer Objects. |
| **FMM** | Fast Monopole Method. |
| **HAS** | Half-angle slice. |
| **LBM** | Lattice Boltzmann Method. |
| **LOD** | Level of detail. |
| **PIC** | Particle in Cell. |
| **PSE** | Particle Strength Exchange. |
| **SPH** | Smoothed-particle hydrodynamics. |
| **VIC** | Vorton in Cell. |
| **VPM** | Vortex Particle Method. |

# Appendix A

# Appendix

## A.1 Sorting the particles

```
__global__ void MakeParticleKeys(int n, glm::vec3 cameraDir, glm::vec3* cTracers,
↪  float* keys, int* keyIndices){
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;

    for(int i = index; i < n; i += stride){
        if(i < n){
            keys[i] = glm::dot(cameraDir, cTracers[i]);
            keyIndices[i] = i;
        }
    }
}


__global__ void SortParticles(int n, Particle* cTracers, glm::vec3* cTracersPos,
↪  glm::vec3* cTracerVel, float* cTracerLife, int* keys, glm::vec3* glPtr){
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;

    for(int i = index; i < n; i += stride){
        if(i < n){
            cTracers[i].pos = cTracersPos[keys[i]];
            cTracers[i].vel = cTracerVel[keys[i]];
            cTracers[i].life = cTracerLife[keys[i]];
            glPtr[i] = cTracersPos[keys[i]];
        }

    }
}
```

```
void VortonSimCUDA::AdvectTracers( const float & timeStep , const unsigned &
↪  uFrame ){
[...]
    MakeParticleKeys<<<gridSize, blockSize>>>(numTracers, cameraDir,
    ↪  cParticlesPos, cKeys, cKeyIndexValue);
    cub::DeviceRadixSort::SortPairs(tempStoragePartSort, tempStorage_size, cKeys,
    ↪  cKeysOut, cKeyIndexValue, cKeyIndexValueOut, numTracers);

    // The first DeviceRadixSort call only functions to determines the size of
    ↪  the temp buffer which then needs to be allocated and used in the second
    ↪  call.
    if(unAllocatedTempBufferSorting){
        cudaMalloc(&tempStoragePartSort, tempStorage_size);
        unAllocatedTempBufferSorting = false;

        // This is only needed while the initial tempStorage is unallocated. The
        ↪  very first call is only used to find the size, not actually sort.
        // The next iterations after this, the sort call above will act as the
        ↪  sorting call.
        cub::DeviceRadixSort::SortPairs(tempStoragePartSort, tempStorage_size,
        ↪  cKeys, cKeysOut, cKeyIndexValue, cKeyIndexValueOut, numTracers);
    }

    SortParticles<<<gridSize, blockSize>>>(numTracers, cTracers, cParticlesPos,
    ↪  cParticlesVel, cParticleLife, cKeyIndexValueOut, cGLPointer);

[...]
}

// This is calculated each iteration and updates the camView variable in the
↪  simulation which MakeParticleKeys creates the sorting keys from. Here,
↪  viewDir is already normalized.
glm::vec3 calcHalfAngle(glm::vec3 viewDir, glm::vec3 lightDir){
    if(glm::dot(viewDir, lightDir) > 0.0f) {
        viewFlipped = false;
        return glm::normalize(viewDir + glm::normalize(lightDir));
    }
    else {
        viewFlipped = true;
        return glm::normalize(-viewDir + glm::normalize(lightDir));
    }
}
```

## A.2 Installation guide & documentation

### A.2.1 Installation

The implementation for this thesis is built on top of the Gloom boilerplate. An NVIDIA GPU is needed with CUDA Toolkit installed. Outside of that, the only necessary downloads are for Gloom itself, which can be easily downloaded with the included git modules. Instructions for this can be found in the README.md file.

This implementation has been developed in Windows 10 with an NVIDIA RTX 2070S and Intel i9 9900.

### A.2.2 Controls

WASD - Move

Mouse - Look around

P - Pause/unpause the simulation

O - Turn on/off shadows for the particles

1 - Number of slices for Half Angle Slice Rendering divided by 2

2 - Number of slices for Half Angle Slice Rendering multiplied by 2