Targeted Smoke Simulation Combining Control and Turbulent Flow

by

Jamie Madill

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science
in

Computer Science
Carleton University
Ottawa, Ontario, Canada
January 2013

Acknowledgments

A number of people contributed to the development of this work. A profound thanks goes to my supervisor David Mould for his help both in developing the methods and in his reading and revising the written report. The assistance, commentary and humor made the job easier, and importantly, a fun ordeal.

Several people helped to complete this thesis in different ways. Michael Gourlay's insightful series of articles on vortex simulation were essential in getting the code off the ground. Additional thanks to Mykola Konyk for his help with technical aspects of C++ and OpenGL. A special thanks to my parents for providing me with support, security, and free meals.

Finally I would like to acknowledge financial support provided by the Computer Science department of Carleton University in the form of teaching assistanceships, and financial support from NSERC and the Ontario Graduate Scholarship program, without whom this work would not be possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The field of fluid simulation, even restricted to its applications in computer graphics, is a vast and complicated subject. It has applications in films and animation, video games, design, engineering, GIS, and many other areas. Fluid phenomena include smoke, water, dust clouds, explosions, fire, and even the motion of nebulae and gas giants. Traditionally we consider fluid simulation to be hard to control and complex, however in many cases fluid systems are highly ordered, and even simple to compute. The task of an implementation of a fluid system in the context of computer games and animation, is to control and shape the physics of the simulation to achieve a specific desired result. Thus control is a coupled problem to simulation; it is essential that animators tasked with rigging a particular explosion, or shaping a massive tidal wave, can effectively use the simulator to get what they want.

The key problem we are addressing is targeted control of particle-based smoke simulation for effect production. A particle for our purposes is a small point in space with some associated attributes, such as velocity. Our problem, then, is to move a large collection of these points, possibly numbering in the millions, towards a target shape, which we represent by a triangular mesh. For instance, a cloud of smoke may take the shape of a ghostly visage that appears in front of a character. This problem can be difficult because physically based smoke simulation is founded on the solution

of numerical equations, which govern how the fluid reacts to the current state. They do not inherently allow for specific control such as this shape matching problem. Thus, we must augment this physically based system with a control framework, to direct the simulation towards a specific end result.

When designing a control framework for a fluid simulation, there are a few important criteria. Without a high-level control system, the artist may have to manipulate physical parameters and inject forces in many places to achieve a particular gust of wind or shape. It becomes important, then, that a control system be intuitive and easy to use; this can mean a higher level control abstraction than just a myriad of physical parameters. Additionally, given that fluid systems are usually complex, and potentially expensive to compute, it is important that both the simulation and the control scheme the animators use are as computationally efficient as possible. Of course, a third important measure is that the artist can achieve the effect he desires; the control scheme must be effective. A final important criteria is that the control system be gentle; when there is too much control applied to a fluid system, it loses the appearance of being a fluid, and the realism and beauty that accompanies fluid motion. Thus even if control is not entirely physically based, it must be subtle enough that it appears to be realistic.

Instead of guiding the motion of the particles with artist-tuned forces, a useful abstraction in designing a control system is to provide a target that the particles travel towards. For example, a target density function, or mesh, can represent a person made out of water or smoke. There are many examples of creatures, characters and objects made up of fluid found in animation and movies, such as the the rivergod of Narnia [70] or the galley made of smoke Gandalf blows through a smoke ring [16]. In these cases, where the animator designs a scene around a shape, it is advantageous to represent the control framework for the simulation based on the target shape itself; instead of manipulating forces or parameters, an animator places a target mesh, animates it as

though it were a character, and the control system will determine how to route the fluid towards this destination.

Fluid simulation is traditionally split between two different viewpoints, called Eulerian and Lagrangian, and each has a different representation of the fluid simulation elements. In an Eulerian approach, the simulation elements are fixed in space; while their values change over time, their locations do not. This often means fluid elements are represented at the cell centers or faces of a fixed voxel grid. In a Lagrangian frame, the simulation elements move; for example, we might form a cloud of smoke or magical dust from tens of millions of particles. Particle-based approaches have the advantage of being inherently adaptive; areas with more particles have more detail, while empty areas can be ignored.

1.1 Overview

In our approach to targeted fluid control, we attempt to show that by splitting the fluid flow in two, between a control flow field moves the smoke directly towards the target, and a turbulent flow field adds the illusion of fluid motion, we can design an effective and easy to use target-based fluid control system. Our proposed method is divided into the following steps:

1. Particle initialization
    - Emit smoke marker particles
    - Initialize control particles inside any emitted smoke
    - Initialize target particles inside the target shape, for new control particles
2. Compute the bulk control flow field (Chapter 3)
    - Generate attraction force on the control particles

- Compute control flow field from the control particles

3. Compute the turbulent vortex flow field (Chapter 4)
    - Spawn vortex particles inside the smoke
    - Evaluate vortex velocity field
    - Update vortex vorticity vectors
4. Particle update
    - Particle advection (Section 4.6)
    - Smoke particle redistribution (Section 4.7)

1.2 Contributions

The specific contributions of this work are:

- A unique point-based shape matching algorithm based on point correspondence
- An efficient and parallel direct evaluation method for finite support vortex particles
- A method to combine vortex particles with a shape matching flow for targeted smoke control

We present the work in the following order: we first introduce the problem, and our approach, in this chapter. Next, we will discuss prior work related to fluid simulation, fluid control, fluid rendering, and also signed distance fields, a useful data structure when dealing with closed boundaries such as triangular meshes. In Chapter 3, we present our target-based particle control algorithm, which generates the bulk flow field driving the smoke to the destination. In Chapter 4, we describe our implementation

of Lagrangian vortex particle simulation, and how we use it to generate our high-frequency turbulence field. In Chapter 5, we discuss some results of this method, and then we conclude in Chapter 6 with a brief discussion and ideas for future work.

Chapter 2

Previous Work

2.1 Fluid Simulation

Fluids in nature are a complex phenomena that has been studied often, and simulated with various methods. There are many ways to describe fluid, for instance, a fluid can have the form of a gas (e.g. air) as in Figure 1b, or a liquid, such as water in Figure 1a. Smoke is often represented in simulation as a *density function* in space, while water is usually characterized by a sharp *interface* between liquid and air. These two phases require very different methods for simulation and rendering. For our purposes, we are interested in simulating incompressible, slightly viscous smoke; we will describe the properties of compressibility and viscidity below.

There are other properties of fluids that are relevant to simulation. We say a fluid is *inviscid* if it is not affected by viscosity. Viscosity is the resistance of a fluid to being deformed; highly viscous fluids such as thick syrup or tar appear somewhat elastic and seem to keep a bit of their shape as they are stirred or otherwise perturbed. Fluids can also be said to be *compressible* or *incompressible*; incompressible fluids have the property that small parcels of fluid keep their density as they travel through the flow. In graphics we are usually interested in incompressible fluids, as this means fluids will not appear or disappear throughout the simulation. Fire and explosions are the

(a) Fluid in a liquid state. Image by Mark Shaiken. Creative Commons.

(b) Fluid as a gas. Image by Dave Mellors. Creative Commons.

Figure 1: Fluid as a gas or liquid

result of compressible flow; however they can be approximated with incompressible fluid simulation.

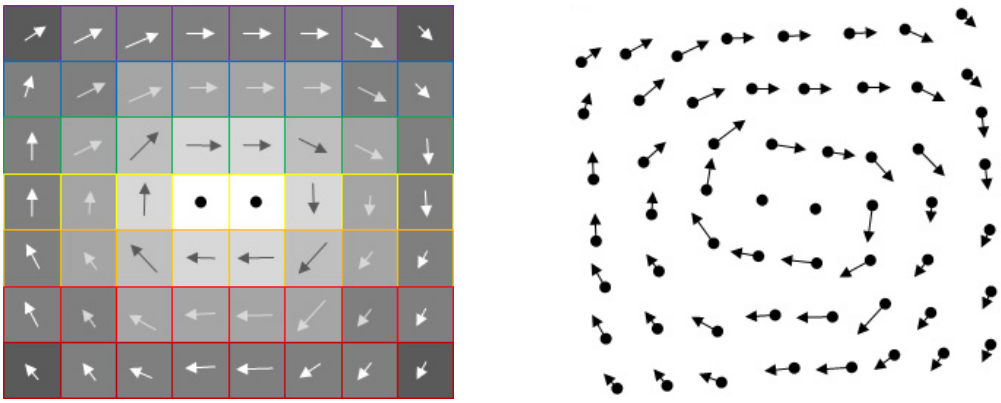
When trying to determine how the fluid behaves in its in environment, we must use complex systems of equations. In the field of computer graphics and animation, however, the most relevant are the incompressible Navier-Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \bigtriangledown)\mathbf{u} - \frac{1}{\rho} \bigtriangledown p + \nu \bigtriangledown^2 \mathbf{u} + \mathbf{f} \tag{1}$$

and

$$\bigtriangledown \cdot \mathbf{u} = 0, \tag{2}$$

a set of partial differential equations that hold throughout the fluid. How we might solve these equations depends on how we store the fluid elements. Simulations that store data at fixed locations in space, such at the cell centers or edges of a grid, are typically called Eulerian simulations (see Figure 2a). Methods that allow the simulation elements to move in space, for example, by tracking the movements of point particles, are Lagrangian methods (see Figure 2b).



(a) Eulerian view: uniform grid **(b)** Lagrangian view: point cloud

Figure 2: Eulerian vs. Lagrangian simulation. Eulerian simulations are characterized by fixed simulation elements, while in the Lagrangian view the fluid elements move with the simulation. Images © Michael Gourlay. Used with permission.

Since our work is focused primarily on the control aspect of fluid simulation, we first briefly discuss key works in the areas of fluid simulation, before describing more exhaustively important works in fluid control. An interested reader can refer to the excellent treatment of the subject of fluid simulation, with emphasis on the Eulerian viewpoint, and the connection between the Lagrangian and Eulerian viewpoints, in the books by Chorin and Marsden [11] and Bridson [7]. Bridson's book takes on the subject from a modern, graphics-focused, perspective. For further information on the field of vortex simulation, the reader can refer to the excellent book by Cottet and Koumoutsakos [12].

2.1.1 Eulerian Simulation

Eulerian fluid simulation is characterized by fixing the simulation elements in space. In this case, fluid elements may be represented either at the cell centers, or staggered in centers of cell faces [8] in a volumetric grid. The simulator's principal job is to solve the incompressible Navier-Stokes equations, listed in equation (1). These equations govern how fluid quantities, like velocity or density, change and evolve over time [11],

and can be derived directly from Newton's equation of motion [7].

In equation (1), ρ represents the fluid density, ν is the kinematic viscosity and $\mathbf{f}$ is the external force, such as gravity. By setting the viscosity to zero, the fluid is considered inviscid; this is usually the case when dealing with smoke or water, and simplifies the solution somewhat. The second equation (2) is called the incompressibility constraint, and specifies that the fluid mass should be conserved throughout the domain of simulation.

A numerical solver for the Navier-Stokes equations makes the velocity field of the fluid incompressible, i.e., enforces the incompressibility constraint, by solving for a scalar pressure field, and then subtracting the gradient of pressure, as in equation (3):

$$\mathbf{u}_{n+1} = \mathbf{u}_n - \triangle t \frac{1}{\rho} \bigtriangledown p, \tag{3}$$

where $\mathbf{u}_i$ represents the flow field at time step i and p is a scalar pressure field.

Equation (3) has the form of a scalar Poisson equation; the discretized equations form a sparse linear system which is then typically solved via an efficient and stable iterative method. For a full explanation on these subjects the interested reader can refer to the book by Bridson [7].

Early work in fluid simulation by Foster and Metaxas is characterized by small grids in 2D and 3D [19, 20]. Foster and Metaxas' seminal publications were the first in the field of computer graphics to describe solutions to the incompressible Navier-Stokes equations with a specific focus on rendering and animation; they also required strict conditions on the size of the simulation time-step to ensure stability. The first work to achieve unconditionally stable Eulerian simulation was that of Stam; he coined his method semi-Lagrangian advection [61]. Stam's work forms an important basis for future fluid simulation in computer graphics. Instead of advecting at a cell

forward in time, which could cause excess velocity to be created due to numerical errors, then leading to instability, Stam used ideas from the Lagrangian viewpoint. By treating fluid at a cell center as the end result of a particle's motion, he steps backward in time to the particle's origin and advects that velocity to the new cell center; this avoids generating excess velocity.

Subsequent work by Fedkiw et al. introduced the notion of vorticity confinement to counter numerical diffusion [17]. The curl, $\bigtriangledown \times \mathbf{u}$, of a vector field $\mathbf{u}$, where $\bigtriangledown$ is the del operator, is a vector operator that described the rotation of the vector field. The curl of a velocity field is called the vorticity, ω; in fluids, this represents the small scale structure, where we can metaphorically think of each piece of vorticity as a paddle wheel, spinning the flow in a certain direction. Vorticity confinement works by finding areas where these paddle wheels are damped out, and adding back the vorticity.

Numerical diffusion can also be very problematic in liquid simulation. In this case, diffusion causes the simulation to gradually lose volume over time; the fluid can appear to "leak" out. The particle level set method, first introduced by Foster et al. [18, 24], provides counter-measures against this leaking, by tracking the surface of the fluid using particles, tying them to the interface between fluid and air, and then extracting an isosurface, to form the full 3D surface mesh.

Nguyen et al. describe a physically based simulation of a premixed oxidized flame [41], modeling the reaction front of a blue flame using the particle level set technique. This reaction front is the interface between two fluids, with different properties; conserving mass across the flame front, as fuel ignites, and turns to hot gaseous products, provides realistic premixed flame behaviour.

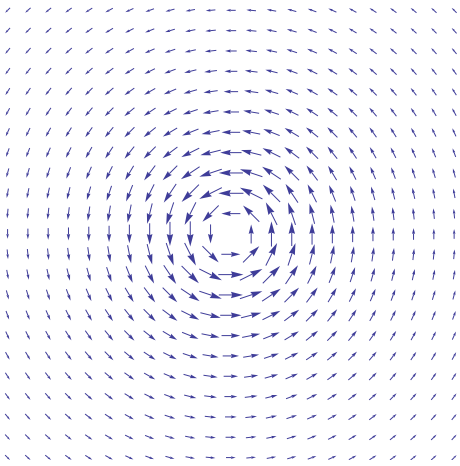


Figure 3: A point vortex vector field in 2D, a simple vortical structure.

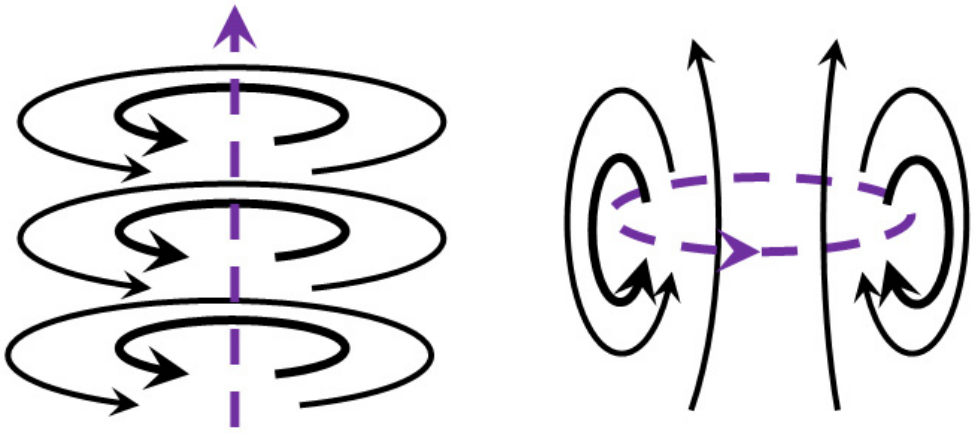
2.1.2 Vortex Simulation

Lagrangian approaches are characterized by simulation elements moving through the domain; there are several ways to model fluids from this perspective. Smoothed Particle Hydrodynamics (commonly referred to as SPH) solves the Navier-Stokes equations by interpolating values in space, given a smoothing kernel [6, 31, 34]. For our work, we do not use this approach, but instead consider the spinning vortical flow of turbulent fluid motion.

Vortices occur in nature and are familiar to most people; you see them by watching tornadoes on the news, water flow down a sink, or milk in coffee. Smoke rings are a common self-propagating vortical structure, and a familiar pattern with characters fond of smoking pipes. Some simple vortical structures are shown in figures 3 and 4. By taking the curl of the Navier-Stokes equations, we can formulate the vorticity transport equation:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \bigtriangledown)\omega = (\bigtriangledown \mathbf{u}) \cdot \omega + v \bigtriangledown^2 \omega + \frac{1}{\rho} \bigtriangledown \times \mathbf{f} \tag{4}$$

These equations are the foundation of vortex methods [12]. We can solve these equations to determine the velocity of the flow field, and how vorticity of the system



(a) A line vortex **(b)** A ring vortex

Figure 4: Other simple vortical structures. Images © Michael Gourlay. Used with permission.

changes with the flow field.

The left hand side of the equation represents the advection in a Eulerian description. The first term on the right hand side is the vorticity stretching term; this term describes how vortices not only affect each others positions, but also affect the spin of neighbouring elements. Often, this is implemented as allowing the vorticity field to spin the nearby vortex particles along with the fluid flow. The second term is a diffusion term due to viscosity. The last term is an external force's effect on the vorticity field.

In our work, vorticity is represented as a point, around which there is a spin, though it can also be a line or other higher dimensional shape, such as a sheet. Computing a velocity field from these equations, and advecting these particles though the simulation domain, produces characteristic rolling and billowing smoke motion. If instead of using particles, we chose to solve these equations on a grid, it would involve solving a vector-valued Poisson problem [52].

The velocity for a single vortex particle is given by the Biot-Savart law:

$$\mathbf{u}_v(x) = \frac{1}{4\pi} \sum_{i=1}^{N} \frac{(x - x_i) \times \omega}{(x - x_i)^3} \tag{5}$$

where x is the position in space and x_i is the position of vortex i, and ω is the vorticity vector of the vortex. The formula is said to have *infinite support*, that is, it is non-zero, extending to infinity; it also has a singularity at $x = x_i$. Various methods exist to *mollify* this kernel, that is, to remove the singularity, and additionally to limit to the support to a compact region. Examples of this mollification, and examples of finite support kernels, can be found in the literature [3, 42, 52]. Although most implementations choose to use finite support kernels for performance, Speck [60] proposes a very fast and accurate method of multipole expansion to compute velocity for infinite support kernels, where groups of distant vortices are treated as a single vortex, when computing velocity at a point.

Vortex simulation uses primitives that spin the fluid elements, where all elements together produce a characteristic rolling turbulent fluid flow [12]. Vortex particles are the simplest such primitive, and are often used for gaseous fluid media such as smoke [21]. It is also possible to use vortex particles to treat the viscous diffusion term, and handle boundary conditions, such as flow near solid walls [54]. Handling no-slip and no-through boundary conditions, as in Park et al. [42], produces an effect known as vortex shedding, where new vortices form a turbulent wake behind a solid object. Selle et al. [52] also demonstrate that vortex particles can be added to a Eulerian fluid simulation to counter the effects of numerical diffusion. Combining vortex particles with a low resolution simulation [79] adds turbulence to a Eulerian simulation. Similarly, Pfaff et al. [44] use a novel type of anisotropic turbulence particle to add high-frequency detail to a coarse fluid simulation. The two-scale method of Pfaff et al. produces attractive results at near-interactive rates, by avoiding precise numerical solutions required by detailed fluid solvers in favour of this new class

of particle. Yoon et al. [79] propose using a precomputation step that computes an artificial boundary layer, where high Reynolds stress can lead to the formation of vortices; this is then used to model how vortices are formed near object boundaries, in order to add detailed turbulence to a static scene.

The point vortex is only one way of representing vortical structure. Gates proposes a line vortex primitive [22]. Angelidis et al. propose methods for using a series of connected line segments [2] to form vortex filaments, and subsequently a Fourier series, to represent a continuous curve [3]. Because vortex filaments can represent higher-order structure, they can allow for a more detailed simulation phenomena, using fewer primitives. This comes at the cost of more complex implementation of the simulation. For instance, since vortex filaments can stretch over time, Angelidis et al. limit the radius of a filament, preventing excessive stretching and shearing. Improving on this, Weimann et al. propose a method to dynamically reconnect vortex filaments throughout the course of the simulation; allowing more natural turbulent motion [74]. Recent work by Pfaff et al. introduces the idea of a vortex sheet [45]; by solving the vortex sheet equations coupled with a traditional Eulerian solver, they simulate sharp interfaces between fluid and air.

2.1.3 Eulerian vs. Lagrangian Approaches

There are distinct tradeoffs when working in either the Eulerian or Lagrangian frame. Eulerian approaches traditionally require solving the pressure equation; this has the advantage of always generating a flow field that conserves mass, regardless of the initial state. This can mean arbitrary force fields and velocity affectors are smoothed out during this pressure solve, allowing for a great deal of flexibility; the control framework may superimpose any set of flow fields, for example, some with high divergence, and the pressure projection step will shape the combined flow to a divergent-free velocity field. However, Lagrangian frames have advantages of their own. First, the

pressure solve can potentially be a very expensive operation when dealing with large grids, because this involves solving a large linear system; Lagrangian approaches do not have this expensive step. Often we solve the linear system via an iterative method, which may require dozens of iterations to converge. Parallelizing this step is essential and requires some expertise and technical knowledge. Second, when dealing with Lagrangian particles, the domain of computation becomes implicitly adaptive; we only do computation in the regions where there are more particles, and we can increase detail in these areas by using more particles.

2.2 Fluid Control

2.2.1 Introduction

Without an effective method to manipulate a simulation, an animator is left at the mercy of a myriad of physical parameters and forces. Control is an extremely important aspect of animation; effect production often calls for a certain shape or shaped fluid motion [70, 75]. In general, any good control scheme must be expressive, intuitive, robust, and computationally efficient. Fluid control is a frequently studied area of research, and many different approaches and methods have been explored since fluid simulation has become more popular, as reported in the survey paper by Limtrakul et al. [35].

Using broad strokes, one class of control algorithm uses primitives, such as points, lines or curves, to generate velocity, or force fields, which influence and direct the fluid flow. An example is found in early work by Gates [22], where he combines flow fields from several divergence-free primitives. Line vortices, sources, sinks, and spline curves, each manipulate the fluid flow. Source and sink primitives attract or repel the flow. Vortex line primitives give a spin around an axis. Uniform primitives act

as wind fields, and divergence free directional flow primitives act as a guide along a spline curve through space. Since individually each of these primitives are divergence free, the principal of superposition dictates that when they are combined they form a mass-conserving flow field.

In early work by Foster and Metaxas [20], they modify the pressure equation directly. For instance, by modifying the pressure of a fluid grid cell based on its location relative to a central axis, they give the effect of a splashing, outwards driven water fountain. They also propose further tunable parameters such as physically based surface tension control, and show how to simulate interactions with moving boundaries. One criticism is that parameter control, such as pressure and surface tension, affect the simulation in a somewhat indirect manner; the artist or animator may not know exactly the kind of result he would get from changing the pressure equations directly, and would prefer a more direct approach.

2.2.2 Target Control

Instead of influencing the motion of the fluid with directed primitives, specifying a direct target for the fluid can provide a more high-level instrument of control. A target can take several forms, such as a density field, a set of particles representing a density function, or a mesh. The simulation then attempts to guide the fluid towards this target. In some cases, the target shape moves as the simulation takes place, such as a walking person or galloping horse; each position of the target shape at a certain time we call a keyframe.

Keyframe control of smoke simulations

The seminal publication, "Keyframe control of smoke animation", by Treuille et al. [69], and later improved by McNamara et al. by using the adjoint method [38] uses artist supplied keyframes of a target density function to directly specify the fluid

destination. Using general nonlinear optimization, they find the optimal amount of wind and vortex forces that will move the source fluid density to the target, while simultaneously minimizing the amount of total control needed. Although they do obtain good results, they must find the derivatives of every control parameter with respect to every simulation element. Thus, their simulation times are extremely long. Additionally they report their method as highly unstable in certain situations. The adjoint method improves on the simulation time greatly, by instead solving the dual of the problem. Although this is several orders of magnitude faster, this method traverses the entire solution space forwards and in reverse, from the last state to the first, and must store every state along this path, causing huge memory requirements.

Target driven smoke animation

In the work by Fattal and Lischinski [16] and Shi and Yu [56,57], the control framework is also given as a series of density keyframes, usually representing an object; this could be a shape such as a galloping horse, or sailing ship. Fattal and Lischinski add two terms to the simulation; a *driving force*, $\mathbf{F}$, shown in in equation (6), and a *gathering* term, $\mathbf{G}$, shown in in equation (8). The driving force drives the smoke towards the target, along the gradient of the target density. The gathering term adjusts the source density of the smoke cloud, ρ, to match that of the density of the target, ρ^*.

$$\frac{\partial \mathbf{u}}{\partial t} = \nu_f \mathbf{F}(\rho, \rho^*) \tag{6}$$

$$\mathbf{F}(\rho, \rho^*) = \tilde{\rho} \frac{\bigtriangledown \tilde{\rho^*}}{\tilde{\rho^*}} \tag{7}$$

$$\frac{\partial \rho}{\partial t} = \nu_g \mathbf{G}(\rho, \rho^*) \tag{8}$$

$$\mathbf{G}(\rho, \rho^*) = \bigtriangledown \cdot [\rho \tilde{\rho^*} \bigtriangledown (\rho - \rho^*)] \tag{9}$$

In this case, $\tilde{\rho}$ represents a smoothed density field ρ, and ν_f and ν_g are scalar strength parameters. Fattal and Lischinsky's control framework is very efficient, adding very little overhead to the simulation time. Since the driving force is based on the target gradient, however, the density functions ρ^* and $\tilde{\rho^*}$ must be non-uniform throughout the domain of computation; otherwise, the gradient would be zero in some regions, which would case the resulting force field to be zero in these areas. This could stop smoke outside the target shape from moving towards the target. In some cases, this restriction might require large smoothing kernels. Additionally once the smoke reaches the target, it stops moving, and loses the characteristic rolling turbulent fluid motion.

Controllable smoke animation with guiding objects

Instead of considering a target density field directly, Shi and Yu [56, 57], instead convert the density field to a level set, and apply control to match the source boundary to the target boundary. Using the variational shape interpolation proposed by Turk and O'Brian [71], they generate a morph sequence of *guide shapes* from the initial smoke density level set, to the target density level set. Using these guide shapes, they generate velocity constraints to force the smoke to these intermediate targets. Since they only apply velocity constraints along the boundary, their method allows smoke to roll and flow naturally, even after reaching the target. Example velocity constraints are illustrated in Figure 5. To enforce these constraints, they propose a novel partly compressible fluid solver, which is reported to be quite efficient and gives results similar to a standard incompressible solver.

They also propose applying a similar method to free surface liquids [57]. Since

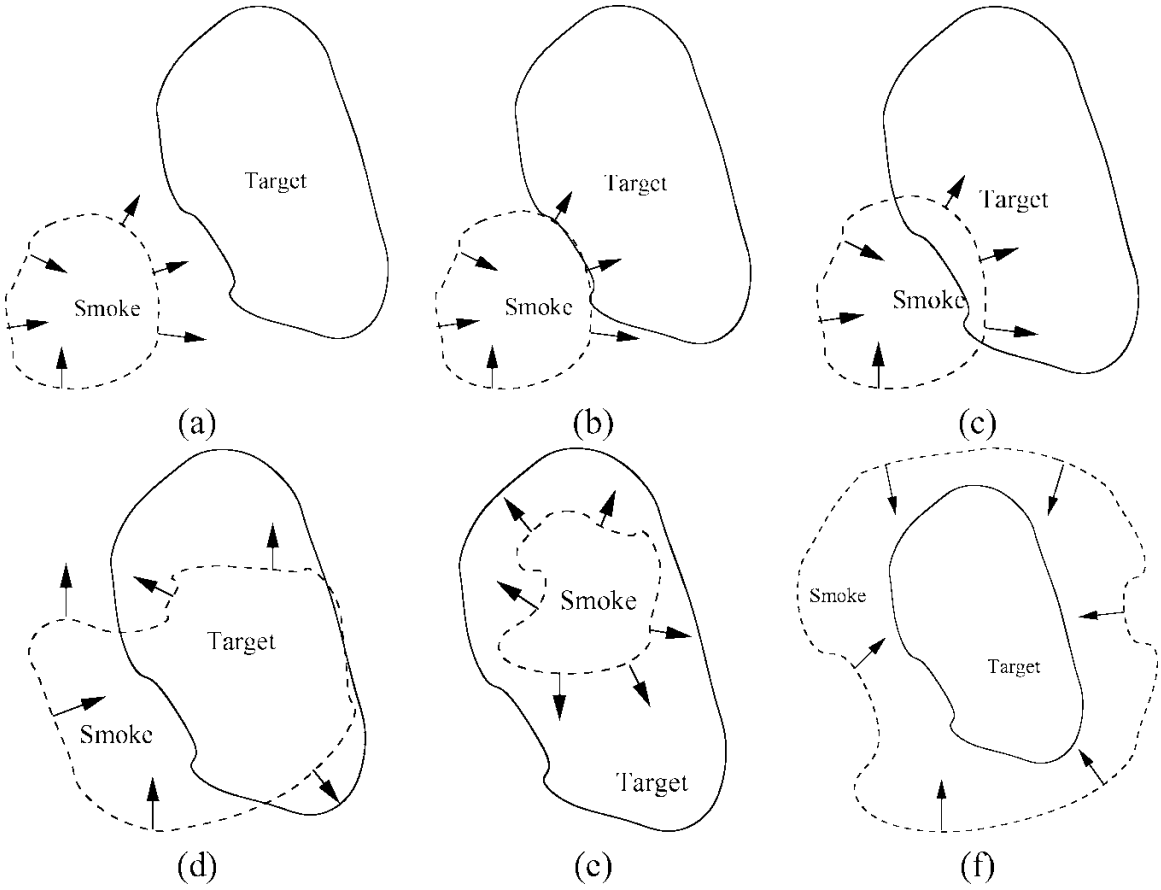


Figure 5: The velocity constraints applied by Shi and Yu [57] to match the smoke density to the target. In (a) and (b), the smoke density is entirely outside the target. In (c) and (d), it partially overlaps. In (e), the smoke is entirely contained, and in (f), it encloses the target. Image © ACM, used with permission.

liquids such as water are dense enough to be much more strongly affected by gravity than smoke, when animated, they usually are influenced by this external gravitational force; this produces the characteristic incompressible flowing liquid behaviour we expect. Thus, Shi and Yu propose an alternative to gravity, that considers the skeleton of the target mesh to be the area the fluid "falls towards"; using a potential field similar to a work by Hong [25], this field acts as a sort of gravity, and pulls the liquid towards the skeleton of the shape.

Prior to their work with guide shapes, Shi and Yu [58] propose using a series of forces to steer fluid to match a target density. In their system, short range forces match the density gradient with the target density gradient, while long range forces generate a coarse attraction, using electric fields. Their early work also allows the smoke to continue to move naturally after convergence, along the direction perpendicular to the target gradient.

Controlling fluid animation with geometric potential

Instead of using a level set or density field, Hong [25] uses a target control given from a potential field. This potential field, generated during preprocessing, is given for each keyframe, and can be created automatically, or augmented via user input. This method is very efficient, and provides a very little computational overhead. However, choice of the potential field determines the quality of the result. Without user interaction, their method misses thin features. Additionally, if the target changes shape often, the need to recompute the potential field each time step could cause the performance of the algorithm to suffer.

2.2.3 Particle Control

Particles are a popular method of achieving volumetric effects, and can store arbitrary properties, such as color, size, or other attributes. They can be a flexible way for an

animator to apply varying types of control through a volume; certain regions may be filled with a particle with one type of properties, and another with a set of particles with a different set of properties. This is also be applied directly to fluid simulation, and provides very effective and efficient control.

Directable photorealistic liquids

Rasmussen et al., [48] propose a framework where the artist is given the ability to place control particles, governing the velocity, level set (where liquid is injected or erased), and divergence of the simulation. Additionally these particles can be erasers or emitters that add or remove liquid. Individual particles are given weights, and a spatial kernel over which they apply their effects. Further augmenting the control scheme, constraints can either be "soft" control, where control values are blended with current values of the simulation, or "hard" control, where the animator can force the simulation to take a fixed value. These soft and hard controls can influence velocity and other fluid parameters. They often use very fine-grained control, with many thousands of control particles, all tuned by the animator to achieve a specific effect (see Figure 6). Their control framework is scalable; they report their method as being several orders of magnitude faster than the method of McNamara et al. [38]. However, the control scheme requires the animator to be proficient in manipulating large clouds of particles. It also does not provide a large scale control abstraction such as a target destination.

Detail-preserving fluid control

Unlike Rasmussen et al. who provide many classes of artist control, Thurey et al. [68] propose a simple control framework with a single class of control particle. Thurey et al.'s framework is designed for particle fluids; in their case they use the Lattice Boltzmann method [62]. Alternately, we may use a single class of control particle

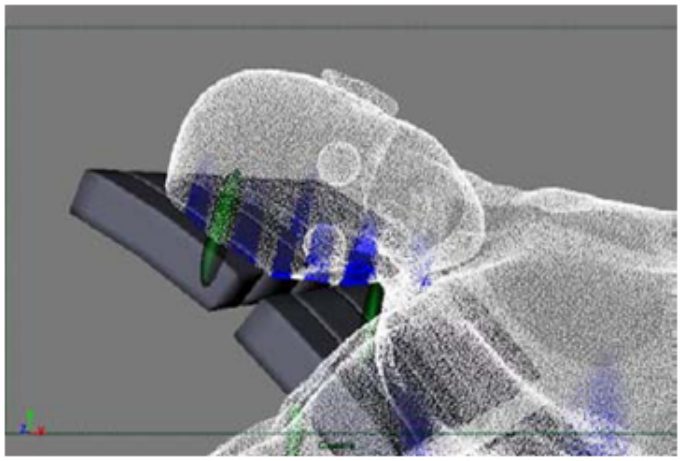


Figure 6: The particle control method of Rasmussen et al. [48]. White particles apply hard velocity control, green soft velocity, and blue are CSG liquid erasers.

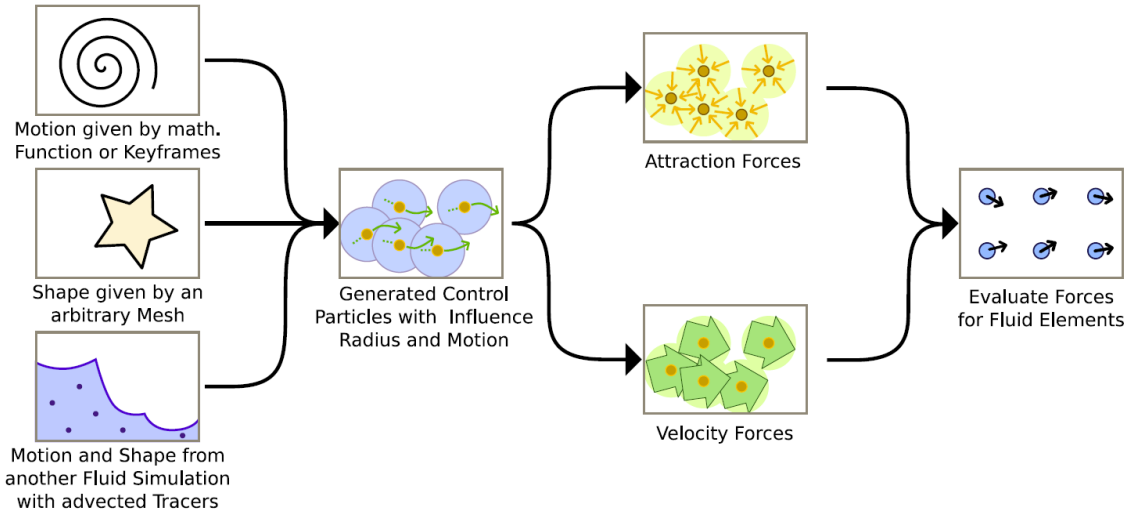


Figure 7: The fluid control framework of Thurey et al. [68]. Their system uses a single class of control particle, which apply attraction and velocity forces. Image © Elsevier, reprinted with permission.

control, using SPH for fluid effects [64]. The control particles of Thurey et al. act as both locally based magnets, that attract nearby fluid particles, and wind forces, that transport fluid particles along the moving path of the control particle.

In a separate computational step, Thurey et al. isolate turbulent detail by using a low pass filter, and apply the control velocity only to the low frequency flow. See Figure 7 for a depiction of their control system. Turbulent flow is then added back, preserving detail. By capturing the output of a standard forwards fluid simulation, then they cleverly use this as a control flow, and make fluid flow up to the form of a target shape. One potential issue with their method is that fluid particles must be within the control radius of the target particles, or they are not affected.

Particle control can be shaped based on a correspondence between source and target particles [67]. In this case, the motion of the particles, which may not even be that of a fluid, is driven towards a target density much as in the target based control methods. This is a distinct difference from the method of Thurey et al. and leads to a more abstracted control system.

2.2.4 Precomputation

An entirely different approach to fluid control is to run a standard fluid simulation off-line, to produce a database of results. These results are then used to generate an on-line controlled simulation. For example, in the work of Mihalef et al. [39], they focus their scope on the animation of breaking waves, using an artist controlled framework, with a database back end. By generating a library of 2D slices of breaking waves, they then select a series of these slices and then composite them, forming a 3D animation adequately resembling the initial wave shape.

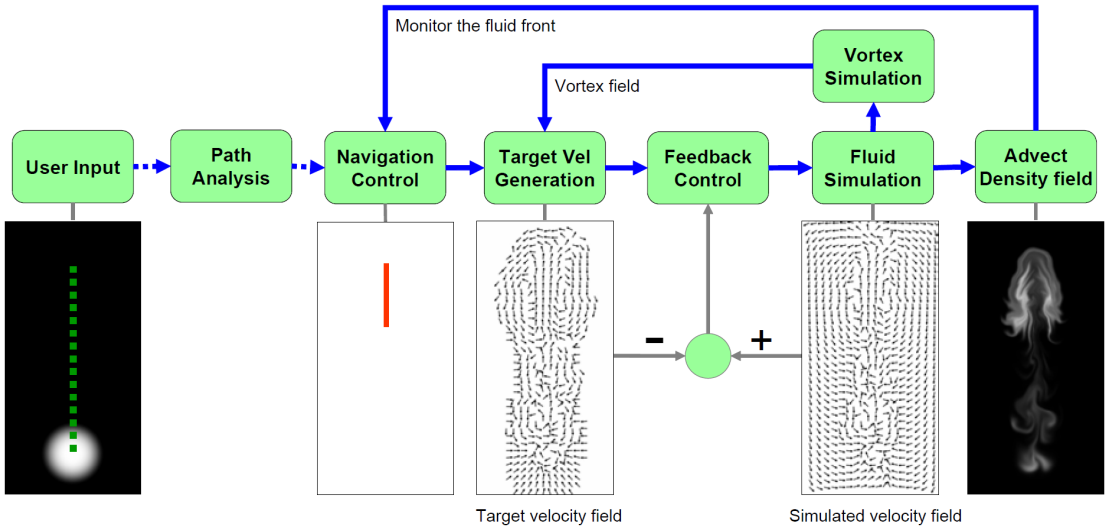


Figure 8: Control framework for path-based control in Kim et al. [32]. Image © authors, used with permission.

2.2.5 Path Control

Spline curves can also serve as a high-level control primitive. Artists can use splines to control the flow of fluid along the path defined by a spline, instead of towards a target shape. Splines have the distinct advantage of dictating how fluid moves while it is in transit; in contrast, target control specifies the end condition rather than the intermediate steps. Gates proposes a spline primitive, consisting of a series of directional flow primitives, to generate a divergence free velocity field [22]. Kim et al. [32] augmented the path control spline by other fluid techniques: namely vortex particles, curling Rankine vortices, and additionally the ability to handle path self-intersections. Their control framework is illustrated in Figure 8. Their control system is computationally efficient, adding little overhead, and is able to handle complex paths, with attractive results.

Angelidis et al., in their work on vortex filaments, propose a controlled method of adjusting the properties of a vortex ring, to guide the motion of the filaments along a curve [3]. In general, path control is a useful control abstraction; although it does not

guarantee the fluid will arrive at a particular point, it provides a means of guiding how the fluid travels.

2.3 Rendering

Fluid rendering is a complementary problem to fluid simulation; statistics on fluid density or velocity are meaningless to an film audience until they are expressed in a visual representation. Additionally, fluids with different physical properties require specialized methods for visualization. For free surface fluids, we generally first convert the volume representation to a mesh, then render the mesh in a traditional manner. Smoke is called a participating media; small particles scatter, absorb, or even emit light. Similar to simulation, Lagrangian smoke rendering uses a particle based representation, and Eulerian approaches use fixed spatial structures, such as grids or octrees.

2.3.1 Rendering Liquids

Although we do not deal with free-surface liquids, we will discuss rendering applications, as they are an important topic in fluid simulation. In this case, the fluid is usually converted to an implicit function, using particle potential, or the free surface level set. An isosurface extraction algorithm then converts this field to a triangle mesh. Once converted to a mesh, a ray tracer, or forward rasterization pipeline, converts this to a two-dimensional image suitable for displaying on a screen.

Marching Cubes [37] is the canonical isosurface extraction algorithm. This method first labels each voxel with an index, based on the value of the isosurface function at the voxel corners; a zero bit indicates a corner is below this reference value, and a one means is is equal to or above. The index fetches a row in a look up table, giving a list of the triangles to be emitted at that voxel; these faces are then stitched together

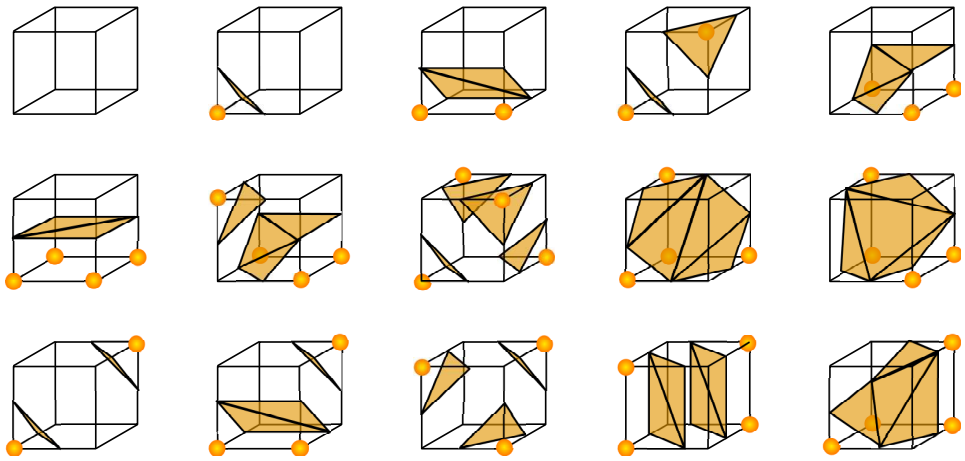


Figure 9: A few of the cases in the Marching Cubes look up table. The algorithm quickly queries the look up table to determine which triangles to emit. Image from Wikimedia Commons.

to reduce the number of elements. Figure 9 illustrates a few of the entries in the Marching Cubes look-up table. Importantly, some of the cases in this table will be symmetric; thus it is ambiguous which set of triangles to emit. Modern alternatives to Marching Cubes, such as Marching Tetrahedra [13], dual methods such as Dual Contouring [29], and other meshing techniques [76] do not have this degeneracy.

2.3.2 Rendering Smoke

Meshing methods do not work for rendering smoke, since it is a participating media; light beams interact with the smoke particles, giving a translucent, cloudy appearance. General techniques for smoke rendering are categorized as *volume rendering*, and this is a large and complex subject, so we will only cover a few relevant techniques. Interested readers can refer to the comprehensive resource, *Physically Based Rendering*, by Pharr and Humphreys [47].

Lagrangian Point Rendering

In our work, we deal with the Lagrangian frame, so we are interested in a Lagrangian point rendering method. We will briefly discuss some various approaches to rendering point clouds; our chosen algorithm is somewhat novel, and we describe it in Chapter 5. The ideal rendering method is efficient, both in computational time and memory, temporally coherent, provides high quality renders, and preferably will work well on graphics hardware.

When we represent smoke as a point cloud, we commonly use a three step process for rendering; a first pass computes the opacity of a particle, a second sorts the particles relative to the camera, and a final pass composites the particles to an image via a blending equation. The simple alpha blend equation, or *over* operator, is: $C_o = C_a\alpha_a + C_b\alpha_b(1 - \alpha_a)$. Here C_o is the output color value, C_a is the point color, C_b the current frame buffer color and α_c is the alpha value, or transparency, of color c.

Because the alpha blend equation requires a spatial sort of the particle data, new techniques for order-independent shading promise to increase performance, by obviating the need for a full sort. A new family of GPU point rendering methods use a structure called the A-buffer [78], or accumulation buffer. These methods render the scene twice, first storing each rendered fragment to a per-pixel linked list in GPU memory, and then, in a second pass, sorting the fragments before compositing them to the screen using standard alpha-blending equation. Alternately, Salvi et al. propose that the fragments be converted to an adaptive transmittance function [49], to speed up the second pass. This technique is similar to a deep shadow map [36]. A-buffer methods provides order-independent processing of individual points, avoiding a sort of all particle in host memory, and also provide fully pixel-perfect results for primitives that overlap, or intersect each other.

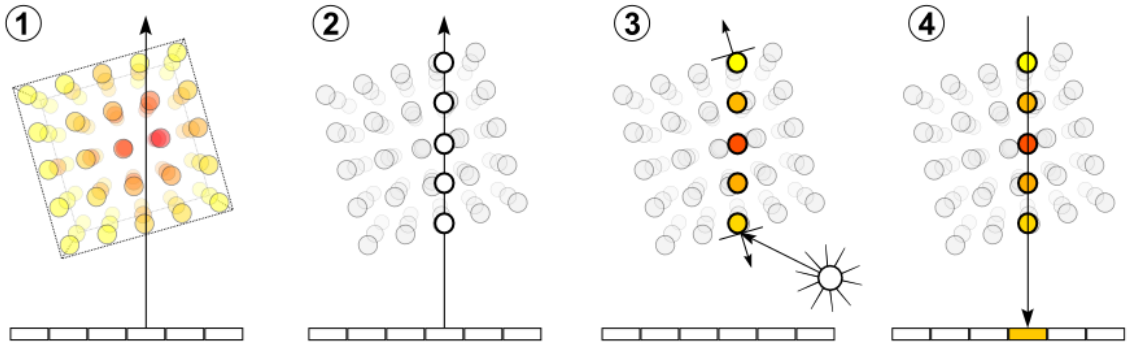


Figure 10: An illustration of the steps in volume ray casting. (1) Ray Casting (2) Sampling (3) Shading (4) Compositing. Image from Wikimedia commons.

Eulerian Voxel Rendering

When we represent our smoke state by a voxel grid, octree, or other fixed spatial structure, there are a number of rendering techniques available. In some cases, even when representing smoke as particles, we evaluate a potential field, to simplify the rendering step. Ray marching, or volume ray casting, shoots a ray, originating from the camera, through the scene; this ray is sampled in intervals, the samples are shaded, and then composited in with a blending equation. This process is illustrated in Figure 10. Sharshach proposed a method to efficient cast rays through volumes on the GPU [51]. Ikits et al. describe a related method, where they divide the volume into view-aligned slices, and then composite the slices, in reverse order from the camera [27]. Monette proposes using an adaptive structure, the density octree [40], to improve the performance of ray marching by grouping uniform regions of density in large cells. In general, for best results, Eulerian methods must have sufficient grid resolution in the areas of high-frequency data.

2.4 Signed Distance Fields

When dealing with closed shapes, such as triangular meshes, we often need a way to efficiently find the distance to the surface, from a point in space. Similarly, we might wish to find the closest point on the surface, to a point in space. The signed distance field is a data structure that compactly encodes the closest point information, as a scalar field, indicating the distance to the nearest point on the surface. The 'sign' of the distance value indicates if a point is inside or outside the shape; being inside the shape means the point is enclosed by the triangle faces with all of the normals pointing outwards. A positive sign indicates the point is inside the shape, and outside of the shape, the sign of the distance is negative (or vice-versa). Figure 11 illustrates a simple signed distance field. The gradient of the signed distance field gives an approximate direction to the closest point on the surface. It is often very convenient to store the signed distance information in a uniform grid. In this case, we can easily interpolate distance values anywhere in the grid using bilinear or, in 3D, trilinear interpolation, with an $O(1)$ computation cost. For better results, we can even use a higher-order interpolation technique, such as cubic interpolation.

Signed distance fields are an extremely useful tool in computer graphics, and in simulation. For our purposes, whenever we deal with a shape boundary, we prefer to always work with a signed distance field, instead of a triangular mesh, since it is so much more efficient to query. Although we do not implement boundary conditions in our simulation, they are very helpful when querying if a point is inside or outside of a mesh; this is a common task in a particle system's collision detection component. Additionally, although we do not implement a GPU implementation of our simulation, signed distance fields are a data structure that is much more GPU-friendly than a structure such as a KD-tree, due to the similarity to a traditional texture. Thus they are a good option no matter the architecture in question.

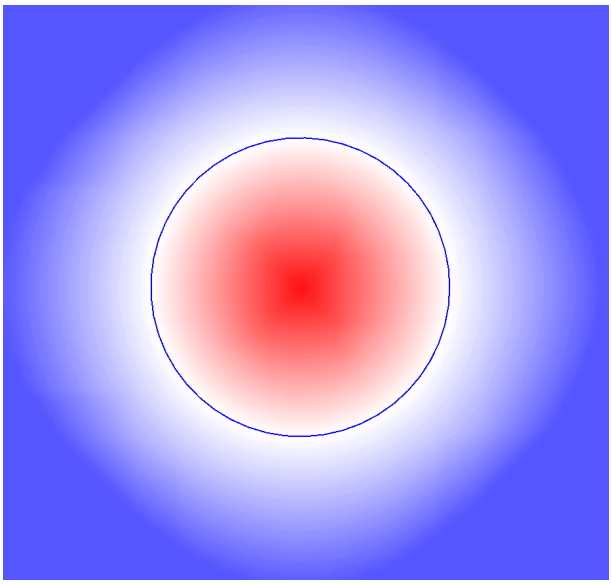


Figure 11: A simple signed distance field generated from a circle. Red values indicate a negative sign (inside the shape) and blue values indicate a positive sign (outside the shape).

Signed Distance Fields of Triangular Meshes

Computing the signed distance field of a triangular mesh is a frequently studied problem, and there are many available algorithms [4, 5, 10, 28]. Recent algorithms exploit GPU parallelism to generate SDFs extremely quickly, using consumer graphics hardware [59, 63]. Computing a SDF from a mesh starts by initializing a region of distances, or thin shell, near the boundary; a second pass propagates this information throughout the volume, using a distance transform. For a more complete review of signed distance fields and their applications, see the summary by Jones et al. [28].

One simple way of finding correct unsigned distances near the boundary of a triangle mesh, is to first compute a bounding box around each mesh triangle face, and then compute distances to every nearby grid cell. This is optimized with oriented bounding boxes, and specialized efficient distance calculations, as proposed by Erleben and Dohlmann [15]. The angle-weighted pseudonormal, proposed by Brentzen and Aans [4], provides an extremely elegant method to compute the sign of an unsigned

distance, when dealing with triangular meshes. At each vertex, the pseudonormal is the weighted sum of the normals of the incident faces, each face weighted by the individual corner angle. To compute the sign of the distance, they find an interpolated pseudonormal using linear interpolation, at the closest point to the voxel, and test its orientation against the distance vector with the dot product.

Distance Transform

Once we have our narrow band of correct distance values, we propagate distance values to the rest of the field, using a distance transform (DT). There are many varieties of DT; we can classify them according to how they compute distance values. *Chamfer* DTs compute new distances of a voxel from the distances of its neighbors by adding values from a distance template [1]. *Vector* DTs, where each processed voxel stores a vector to its nearest surface point and the vector at an unprocessed voxel is computed from the vectors at its neighbors by means of a vector template, tend to have increased accuracy [50]. Eikonal solvers estimate a voxel's distance from that of its neighbours by a first or second order estimator [81]. By classifying DTs according how they traverse through the volume, we see two categories. *Sweeping* schemes propagate distance information from one corner, to another, in row-by-row or column-by-column order, using several passes [65]. *Wavefront* schemes, such as the Fast Marching Method, propagate distances outward from the surface [53].

Vector DTs perform the best in terms of accuracy, according to Jones et al. [28]. In general sweeping schemes are simpler to implement than wavefront schemes, although they may require several passes for improving accuracy. We opt to use a simple sweeping scheme, which is extremely fast, more accurate close to the mesh, and less accurate farther away. We describe our implementation in Section 3.3.2.

2.5 Summary

We have briefly covered the topics of fluid simulation, including the Eulerian and Lagrangian perspectives, and mentioned some of the key prior work in the area of fluid control. Our work is a method of particle-based target control, using Lagrangian vortex particles; a method that has not been addressed exactly in any previous work, although it shares many similarities with other methods of target and particle control. We mentioned that we use a Lagrangian point rendering algorithm for our results, and we discussed a few related smoke rendering algorithms. Finally we discussed applications of signed distance fields, a useful data structure when dealing with boundaries. The next section will introduce the key control algorithms, related to particle control and target control, that we use in our framework.

Chapter 3

Particle Shape Matching

3.1 Overview

The basic components of our system are twofold; we have a base bulk flow that moves the smoke particles towards the target, and a detailed turbulent flow, giving a fluid motion. The bulk flow, which we describe in this chapter, provides a coarse motion driving the smoke towards the position of the target shape, as placed by the animator. This provides our basic control velocity. The detail flow, which we describe in Chapter 4, adds a high-frequency layer of turbulent motion. This gives the smoke the appearance of being a fluid. The combined result of both flow fields added together gives a controllable, directed, and turbulent smoke cloud.

Smoke is a complex phenomena involving the movement of tiny particles through air; although gases such as air are more compressible than liquids such as water, in graphics, we simulate them as incompressible fluids. Additionally, though pressure forces cause gases to expand, in our work we do not explicitly simulate pressure forces, instead we choose to use the equations of vortex flow. Smoke rendering is also an important part of realistic smoke animation; tiny smoke particles scatter light as the light rays travel through smoke volume. We discuss the details of our fluid simulation using vortex particles in Chapter 4, and the smoke rendering in Section 5.1.4.

We are primarily interested in the effect of incompressible smoke simulation, with a degree of viscosity added for artist effect. We also are interested in a highly turbulent smoke simulation, as compared to previous methods which use very slowly moving smoke, such as from the butt of a cigarette. In contrast the turbulent energy of our simulations is much stronger. The fluid equations of the vortex particle method are quite capable of simulating inviscid and viscid smoke, with varying degrees of high and low energy; the numerical basis for our fluid simulation is the same as previous methods. However, we use a marker particle redistribution scheme, which we discuss in Section 4.7, instead of explicit boundary conditions, to constrain our smoke to the target shape; higher energy smoke obscures the effect of this non-physical redistribution, thus we generally show results with more turbulent energy than previous methods.

3.2 Bulk Flow

In this chapter we describe how our base flow system moves the fluid to the target. Since high frequency details would be lost in the turbulence, we only care about the large scale flow behaviour. Our basic approach consists of the following steps: first, we sample a target shape, generating a set of target particles. Then, we place the same number of control particles inside the smoke cloud. A correspondence between the target and control particles provides the basis for an attraction force, which then steers the control particle velocity. We then interpolate this control velocity into a bulk flow field; this bulk flow field is then added together with the detailed turbulent flow described in Chapter 4. We then use the combined velocity field to advect the fluid marker particles and advance the simulation.

The following sections describe the control particle initialization, target particle

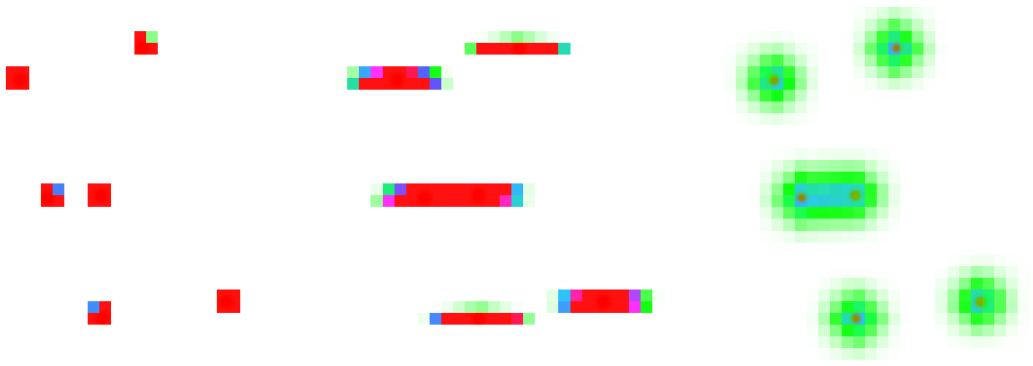
generation, the attraction force, how we optimize this attraction, velocity interpolation, and also how we deal with the target particles when the meshes move, animate or deform.

3.3 Target Particles

3.3.1 Control Particles

The first step in our algorithm is to generate a set of control particles, inside the smoke cloud. The control particles provide the driving velocity that moves the marker particles towards the target shape. We apply an attraction force, described in Section 3.4.1, to these particles; this is integrated over time to give a velocity for each control particle. From the velocities, we generate a velocity field, and then use this field to move the smoke marker particles towards the target. Note that the control particles do not move with the velocity field, but with their individually integrated velocities.

Our first step, then, is to determine how to place our control particles. We wish to place them evenly over the area where there are marker particles, since we wish to move these marker particles towards the target. Thus, we compute a field, which we call marker particle density field, representing the distribution of the M marker particles over space. Here, we use the word density to represent the amount of marker particle energy at a point in space. By using a separable tent blur with kernel radius r_m, we can compute this field very efficiently. If this is the first frame of our simulation, and we have no control particles, we initialize an empty control density field, P_C; otherwise, we evaluate the control density similarly to the marker particle density, with radius r_c. Good values of r_m and r_c are typically quite small, and though they depend on the size of the simulation, the values of r_m or r_c might be as small as 5% of the width of the bounding box of the particles. This way the density fields



(a) Point evaluation **(b)** After blur pass in x **(c)** After blur pass in y

(d) Density intensity. Higher values correspond to red, and low to white.

Figure 12: A separable tent blur in two dimensions, distributing particle density over space. Note that the total density remains the same at every step of the evaluation.

cover the domain, without being excessively blurred from using a large radius. Figure 12 shows a simple example of a separable tent blur in two dimensions.

Next, we sample the space of the marker particles uniformly, placing a control particle where the ratio of control density to marker density is below a given threshold. This number can vary quite a bit; if you prefer a dense packing, use a large ratio, or a small one for only allowing a few. Then, if we have placed a new particle, we update the density field, and repeat this process C_{init} times. This dart throwing approach places control particles where control particle density is low, eventually saturating the cloud over several frames of animation. We will have N control particles, where $N \ll M$.

3.3.2 Particle Initialization

The next step is to generate target particles inside the target shape. Since we will be repeatedly testing if a point in space is inside or outside the mesh, we first convert the target shape to a signed distance field representation, using an approach detailed

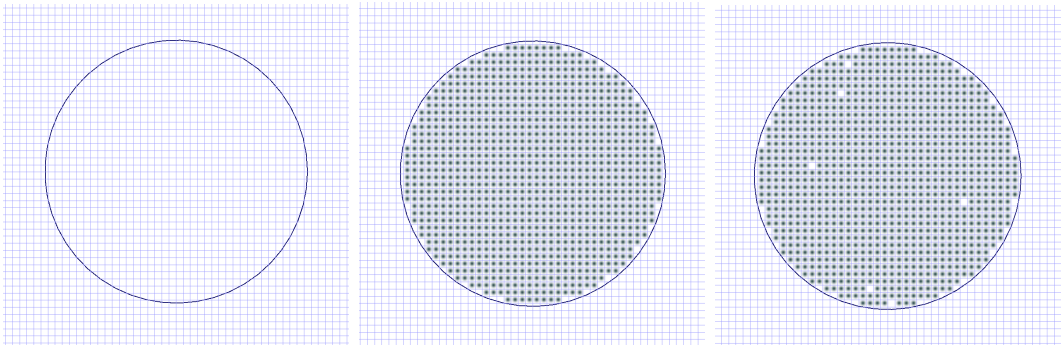
in Section 2.4, described by Swoboda [65]. We find an axis-aligned bounding box around mesh faces, and using the optimized triangle-point distance test, described by Erleben and Dohlmann [15], we find exact signed distances within a narrow band near the mesh. Then, using a fast and simple sweeping technique, we propagate signed distance values through the field using the expression

$$d_{i+1}(x) = \begin{cases} d_i(x) & \text{if } |d_i(x)| < |d_i(x_{prev}) + c| \\ d_i(x_{prev}) + c & \text{otherwise} \end{cases}. \tag{10}$$

In equation (10), $d_i(x)$ is the distance value at cell x after i iterations of the sweeping algorithm, c is the uniform (axis aligned) cell width, and x_{prev} is the last cell visited along the sweep. Note, this method is not perfectly accurate, and accuracy decreases as the distance increases from the initial narrow band. For our purposes, we only care about precise results close to the mesh, so this is entirely acceptable. We perform sweeps along the positive and negative directions of the x, y, and in 3D, z axes, and do not perform multiple passes, since we have found one pass gives sufficient accuracy.

Once we compute this SDF, we use it to generate N target particles, the same number as control particles. Stratified sampling of a grid, aligned with the SDF, gives a particle at every cell center, for every cell center inside the boundary. We use equations (11) and (12) to produce an approximate sampling of a mesh $\mathbf{T}$; a second pass adds or deletes random particles, until we have the exact required sample size of N points. If we have an undersampling, and need to generate more points to produce exactly N samples, we use uniform random sampling; we generate points randomly in a box around the mesh, and reject points that are outside. In the equation

$$\textsc{GridSize}(\mathbf{T}) = [\textsc{Volume}(\mathbf{T}) - A * \textsc{SurfaceSize}(\mathbf{T})]^{1/d}, \tag{11}$$



(a) The sampling grid **(b)** Stratified sampling **(c)** After point removal

Figure 13: Exact stratified sampling of a simple circle target shape, producing N points.

where

$$\textsc{SurfaceSize}(\mathbf{T}) = D_{min} * \textsc{Radius}(\mathbf{T}) * \textsc{SurfaceArea}(\mathbf{T}), \tag{12}$$

$\textsc{Radius}(\mathbf{M})$ is approximated by $\textsc{Volume}(\textsc{BoundingBox}(\mathbf{M}))^{1/d}$. In all cases d is the dimension; in 2D Volume is the interior area, and SurfaceArea is the boundary size. A here is an estimated over- or under-sampling parameter, close to 1, and D_{min} is a parameter that specifies the minimum sampling distance to the boundary. In our tests this equation provides a count of samples reasonably close to N, in 2D and 3D, for a variety of input meshes. Figure 13 illustrates an example of our sampling method for a simple test case. We have found for small scenes, we can effectively initialize the control particles once, when the marker particles are emitted. For more complex scenes, we might adopt other strategies, such as adding or deleting control particles, or redistributing them if they stray too far from the region of interest.

Alternately, instead of creating target particles from meshes, we could extend our system to use a custom cloud of target particles, provided by an artist or animator. Technically this would necessitate a few changes in our procedures; it would require

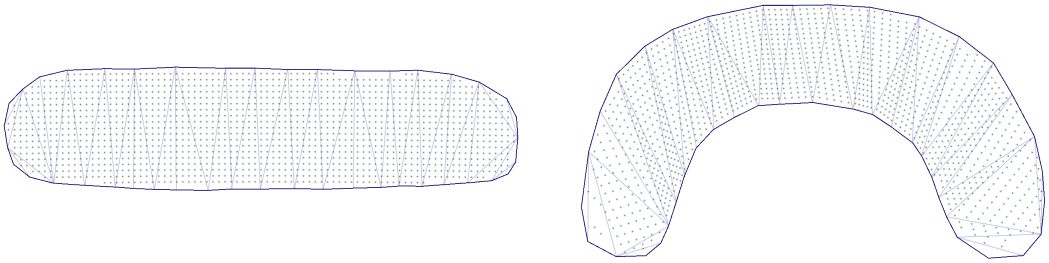
generating the signed distance field from the density field of these particles, for use in the attraction force calculation. The other main change would be either requiring a method to generate an exact number of control particles, as target particles, or a change in our method that would allow us to handle a differing count of control and target particles. The advantages of this approach would be to allow target shapes that are represented purely by particles, which may be convenient for animators who have existing tools specifically optimized for particle clouds, or who prefer to use this method of control.

Now that we have described how we generate the control and target particles, we will describe how these particles move when the target shape is also moving, undergoing skeletal deformation, or other types of transformation without large deformations.

3.3.3 Moving Target Meshes

In many instances, the artist, for a particular effect, will want to move or animate the control mesh during the course of the simulation. For instance, a person made of smoke may jump out of view, or pirouette and vanish. In our framework, this means that our target particles, sampled from a triangular mesh, must move as the mesh changes position, orientation, or undergoes other transformations.

To move the points with the changing mesh, we use linear interpolation, over the Delaunay triangulation, or tetrahedralization in 3D (which we will, for simplicity, refer to as a triangulation). The triangulation is computed once, when the mesh is initialized. After the target particles are initially sampled, we consider the mesh element that contains each target particle, in the mesh's rest configuration. Note that a mesh element is a triangle in 2D, and tetrahedron in 3D. So, we store, for every target particle, the corresponding coordinate for this point inside this element, which is its barycentric coordinate, and the element's identifier. For efficiency, we accelerate this spatial lookup of mesh elements using a uniform grid data structure; when we



(a) Rest particle configuration and interpolation mesh

(b) Interpolated positions of the target particles

Figure 14: A simple scene showing the interpolation of the Delaunay mesh for moving the target particles as the target mesh transforms.

have a mesh element e, we restrict our region of interest to this element's axis-aligned bounding box, and end up only searching a local region of space.

The barycentric coordinate of a point p is a vector of scalar weights (3 elements in 2D, and 4 is 3D), where each weight corresponds to a vertex of the mesh element containing p. The weighted linear combination of the element's vertices, at the mesh rest position, will equal exactly p. Then, when the vertices of the mesh change position, the linear combination of the new vertex positions, with the old barycentric weights, will produce a new interpolated position of p. This method has the advantages of being efficient, and simple to implement. The formula for computing the barycentric coordinate of a 3D point p in mesh element e is shown in equations (13) and (14), and the reverse in (15).

If our point is $p = (x, y, z)$, our mesh element, a tetrahedron in 3D, is $e = (A, B, C, D)$ and our barycentric coordinate is $b = (b_1, b_2, b_3, b_4)$,

$$b' = \begin{bmatrix} A - D \\ B - D \\ C - D \end{bmatrix}^{-1} (p - D), \tag{13}$$

$$b = (b'_1, b'_2, b'_3, 1 - b'_1 - b'_2 - b'_3), \tag{14}$$

$$p = (b_1 * A, b_2 * B, b_3 * C, b_4 * D). \tag{15}$$

In some cases simple linear interpolation may not be general enough; for instance, non-uniform scaling, causing some triangles to become very thin, while others large, will skew the distribution of target points so that some regions have smaller concentrations of target particles. Dramatic deformations may cause the Delaunay triangulation to interpolate points to positions outside the target shape. Other, more accurate, interpolation algorithms, such as mean value coordinates [30], a global method where every mesh point influences every interpolated point, produce reasonable interpolation even when the mesh deforms, or changes shape dramatically. For our purposes, we assume the mesh only undergoes small local deformations as it is animated, and we choose the good performance of linear interpolation. An example result from our interpolation method is shown in Figure 14.

The uniform grid data structure, also called a spatial hashtable, is a uniformly divided grid of cells embedded in a 3D (or 2D) box, each cell in this grid being bucket that contains a linked list. The box containing the grid, in our case, is the axis-aligned bounding box of the cloud of target particles. To find the points at a region of space, we simply enumerate the points in the enclosed cells, and test them against the boundaries of this region. The nice property of a uniform grid is a $O(1)$ access time for each particular cell, and, if we sort the points, the particles will be arranged linearly in memory, which allows very simple traversal. The lack of any random memory accesses in this format is a very good thing for speed.

3.4 Bulk Flow Field

Now that we have our two clouds of N particles, the control cloud and the target cloud, we want to derive a flow field that drives the smoke towards its targets. We want to design a flow field such that the motion is broad and uniform, so later we can add turbulent detail; we also wish to generate this field from a correspondence between source and target particles. We choose to implement this field by using a per-particle attraction force to drive the control particles towards their targets, and then interpolating the velocities of the control particles through the region of the marker particles. We next discuss the attraction force, followed by a discussion of velocity interpolation and extrapolation.

3.4.1 Attraction Force

To drive this flow field, we exert an attraction force F_a, on the control particles, directed towards the target particles. We do this by matching each control particle uniquely with a target particle, such that a energy function expressed by this configuration of sources and targets is globally minimal. Our energy metric must be based on distance between particles; however, note that, although we prefer short paths, since that will mean a faster simulation, we also prefer paths of uniform length. Distant particles should arrive at the target at the same time, or around the same time as nearby particles. Thus, we choose our energy metric to be *squared* Euclidean distance, instead of distance. This will prefer short paths, but will also prefer somewhat uniform path lengths; longer paths are penalized significantly more than shorter ones.

Having N target and control particles, we initially assign each control particle to a unique random target. Then, at every time step, we perform a fixed number of optimization steps, C_{opt}. The optimization algorithm algorithm is shown in listing 1. This procedure is to randomly test two control particles a and b, to see if exchanging

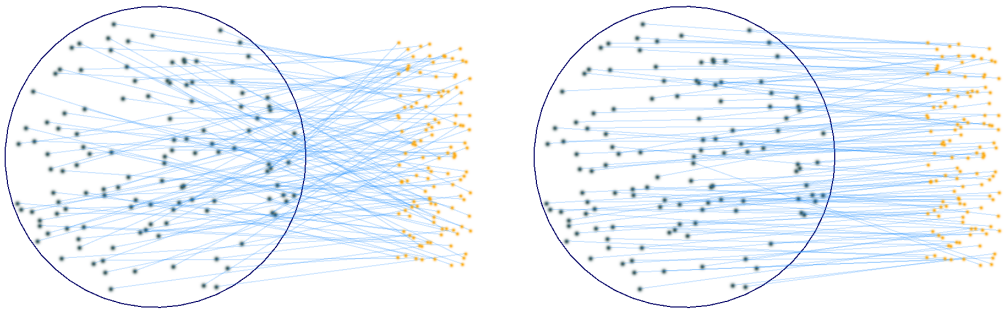
their targets produces an improvement in global energy. If so, we exchange their targets, and then continue to the next trial. Since we perform a fixed number of steps per frame, the solution improves gradually, keeping coherence, and allowing us to provide a fixed run-time budget for the algorithm. We have found this method gives good results, although it is not guaranteed to ever converge to the optimal solution. Examples of an initial and optimal configuration are shown in Figure 15. The solution converged to the optimal configuration after several thousand iterations.

Figure 16 shows a sample of our the performance of our algorithm in a simple 2D scene with 10,000 particles. In this scene, a 5x5 box is filled with flow particles, and encloses a 2x2 target box. After 80,000 iterations, the algorithm is still incrementally improving the solution; however the rate of improvement has slowed down quite a bit since the first 10-20 thousand iterations. Since in most of our results we use only a few thousand control particles, we arrive at a near-optimal solution quite quickly with fewer iterations than in this example.

Algorithm 1 Incremental target optimization

for 1 to C_{opt} **do**
  $i \leftarrow \textsc{Random}(1, N)$
  $j \leftarrow \textsc{Random}(1, N)$
  $a \leftarrow \textsc{Position}(i)$
  $b \leftarrow \textsc{Position}(j)$
  $t_a \leftarrow \textsc{TargetPosition}(i)$
  $t_b \leftarrow \textsc{TargetPosition}(j)$
  $d_{old} \leftarrow |a - t_a|^2 + |b - t_b|^2$
  $d_{new} \leftarrow |a - t_b|^2 + |b - t_a|^2$
  if $d_{new} < d_{old}$ **then**
    $\textsc{Swap}(\textsc{Position}(i), \textsc{Position}(j))$
    $\textsc{Swap}(\textsc{TargetPosition}(i), \textsc{TargetPosition}(j))$
  end if
end for

Our matching optimization can be framed as an instance of the assignment problem, a special case of the transportation problem. Here, the task is finding a maximum



(a) Initial random matching

(b) Optimal energy configuration after several iterations

Figure 15: Effects of the optimization on a distribution of control and target particles. Figure (b) shows the optimal solution after several thousand iterations.

weight matching in a weighted bipartite graph. A set of nodes V represents the control and target points, and a set of edges E connect each control point to every target point. In total this graph will have $2N$ nodes and N^2 edges. The Hungarian algorithm solves this problem exactly, with a time complexity that varies depending on the implementation; however the simplest implementation has a runtime of $O(N^4)$ which would lead to one trillion iterations to find the solution of a problem with 1000 control points. Other less trivial algorithms have runtime complexity of $O(N^3)$, however this is still prohibitively expensive; moreover, since we do not need an optimal solution, these exact algorithms are not as useful for our approach.

Our point matching problem can also be framed as a general instance of the transportation problem, a special case of linear programming. In this problem statement, we have a set of nodes which provide supply, called 'sources', and nodes which consume the supply, called 'sinks'. The sources provide supply over edges, each with a cost, and satisfy the required demand of the sinks. In our case, the problem is very simplified; we have exactly N sources and sinks, each with supply and demand 1, we permit exactly one edge used at every node, and our fully connected graph has edges with cost equal to the squared Euclidean distance. An exact, optimal solution to

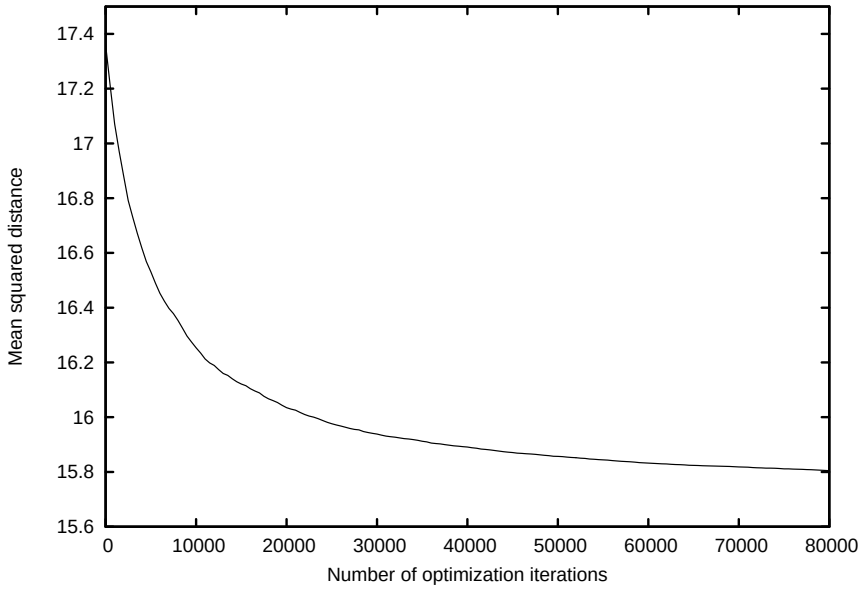


Figure 16: Example convergence of the optimization algorithm with 10,000 particles

this problem can be very expensive to compute, hence we prefer our stochastic minimization algorithm. Interested readers can refer to the paper by Zhang et al. [80], where they treat a similar problem, and frame their solution as an instance of the transportation problem, although differently from our problem. In our method, we match target particles one-to-one with control particles, while in their method, they group particles in clusters, which have different weight. They also allow for different numbers of source and destination clusters.

Now that we have established a reasonable correspondence between our sets of particle, we use this to compute an attraction weight w_a. This attraction weight describes how much a particle wants to move to its target; we then use this weight to compute attraction force, which affects the particle velocity. Our attraction weight is $w_a = (\frac{|x-x_t|}{E(x_t)} - A_\phi)A_\rho$. In this equation, x is the position of the control particle, x_t is the position of the target particle for this control particle, $E(x_t)$ is the feature size of the mesh at x_t, which we describe below, and A_ϕ and A_ρ are scaling parameters. The force, F_a, is computed as $F_a = \frac{x-x_t}{|x-x_t|} * \textsc{Saturate}(w_a) * S_a$ where S_a is a global attraction

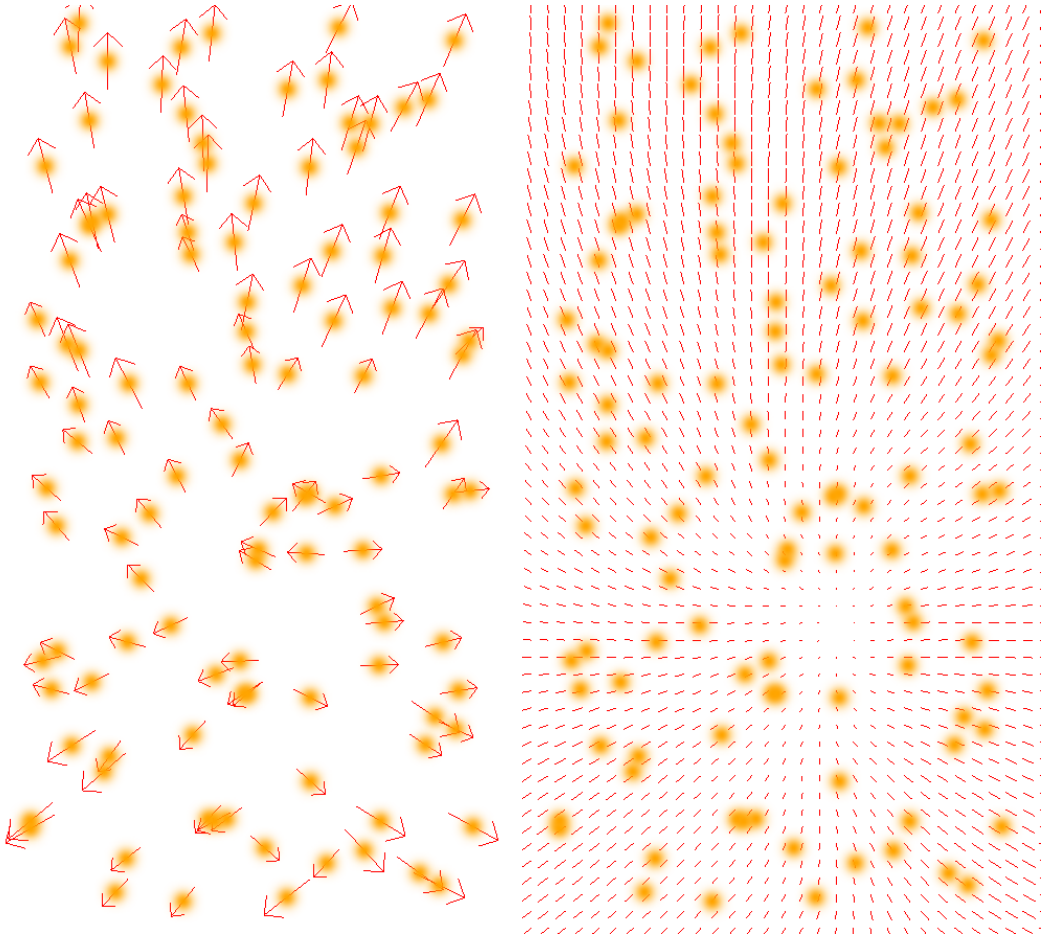
strength parameter, and SATURATE clamps w_a to $[0, 1]$. This linear function gives an increasing weight with distance, allowing the attraction force to gradually ramp down as particles approach their targets.

Feature size of a mesh is a method of measuring mesh size; it is defined as the distance from a point inside the mesh, to the medial axis, as explained by Persson [43]. The medial axis visually resembles the skeleton of the mesh; it is a tree of line segments running through the center of the mesh. We estimate feature size by marching a ray along the reverse gradient of the signed distance field, until we reach a point where the direction of the gradient is reversed; this will occur on the other side of the medial axis. Using feature size conveniently gives us the ability to scale the attraction force according to the size of the mesh, which usually indicates the size of the scene itself.

We apply a simple damping step after we have integrated the attraction force over time to compute a new velocity. If the velocity of the particle is v_f after integrating the attraction force, the damped velocity is $v_d = v_f - v_f * d$, where d is a scalar damping strength parameter, usually quite close to zero. This damping effect is crucial as it stops velocity from accumulating frame to frame and allows the points to settle slowly when we stop the attraction force. We solve for the new control particle velocity via Euler integration, and advect control particles using a second-order Euler integration.

3.4.2 Velocity Interpolation and Extrapolation

Given our cloud of moving control points, each with a unique velocity, we want to use these sparsely distributed velocities to transport a cloud of marker particles representing our smoke volume. In a sense, we must take as input this sparse velocity set, and transform it to a vector field that covers the domain of the smoke. This process is called interpolation, for determining velocities between velocity samples, and extrapolation, for determining velocities outside the region of the samples. This interpolated and extrapolated velocity then governs the flow of the smoke marker



(a) Control particle velocities **(b)** Interpolated velocity field

Figure 17: Velocities from the control particles are interpolated over an axis-aligned grid to form a velocity field, that we then use to advect the marker particles during the particle update.

particles. There is a large repertoire of techniques for interpolating and extrapolating sparse data, each with various properties; for example, linear interpolation over a Delaunay mesh gives exact first-order accuracy (the meaning of exact explained below). The higher-order natural neighbour technique gives second order accuracy, by computing an informative Voronoi tessellation, and using this to find a weighted average of nearby points. For a comparison of several methods, with advantages and disadvantages, see the study by Yang et al. [77].

We use a variant of Shepard's Method, or inverse distance weighting [55], to

interpolate velocity between control points, and extrapolate it over the nearby area. In the original Shepard's method, when we have a collection of N samples u_i, each interpolated control velocity $\mathbf{u}_c(\mathbf{x})$ at a point $\mathbf{x}$ is expressed as:

$$\mathbf{u}_c(\mathbf{x}) = \sum_{i=1}^{N} \frac{w_i(\mathbf{x}) u_i}{\sum_{j=1}^{N} w_j(\mathbf{x})}, \tag{16}$$

where

$$w_i(\mathbf{x}) = \frac{1}{d(\mathbf{x}, \mathbf{x_i})^p}. \tag{17}$$

Here d is a distance metric, such as Euclidean distance, and p is a positive real number. We use a slightly different distance weighting

$$w_i(\mathbf{x}) = \begin{cases} (r_v^2 - d(\mathbf{x}, \mathbf{x_i})^2)^3 & \text{if } d(\mathbf{x}, \mathbf{x_i}) < r_v \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

which has a clamped radius r_v. This distance weighting, conveniently, is not just defined between velocity samples, but over the entire radius r_v. That is, it does not just interpolate the velocity samples, but extrapolates velocity outside the convex hull of the data points, out to a distance of r_v. In practice this function gives good results, and is very simple to implement. In our tests this produces similar results to inverse distance weighting, where the influence of a point decreases with distance. An example of our interpolation is illustrated in Figure 17. We use a temporary grid to accelerate the bulk flow field sampling step: we compute the interpolated and extrapolated velocities at the cell centers of this temporary grid. Later, during the particle update step, we sample velocities from the grid to advect the smoke particles with the bulk flow velocity. This temporary grid provides a great speed improvement at a small cost in accuracy; since we are mostly interested in low-frequency effects,

the trade-off is acceptable.

Other more sophisticated choices for interpolating velocity from sparse data include methods which are inherently divergence-free, a very attractive property for fluid simulation. Hong and Yoon describe a divergence-free method of interpolating velocity that gives fluid-like results without using the Navier-Stokes equations at all [26], using moving least squares. By using diffusive derivatives and moving divergence constraints, they achieve fluid-like interpolation, solving a moving least squares problem at every grid cell. The drawback, however, is they report their method as extremely computationally expensive. Vennell and Beatson propose using divergence-free radial basis functions to interpolate sparse velocity information [72], while being both accurate and implicitly divergence-free, for ocean flow data sets. A radial basis function, or RBF, is a real-valued function whose value depends only on the distance from the origin, or some other point, called a center. Linear combinations of RBFs are used to approximate other, more complex functions.

We chose our method of distance weighting for speed, simplicity, and ease of extrapolation; however, more complex approaches may by necessary for other applications. Note, as in evident in Figure 17, the interpolation is not exact. That is, the interpolated velocity at control particle position p does not equal the original control velocity. In effect, we have a smoothed velocity field, averaged over nearby particles; this produces coarse, and smooth overall motion. Since we are creating a bulk flow velocity, this is adequate; however, some applications may require an exact method.

3.5 Summary

We have presented here an overview of our method of generating control and target particles, and how these control particles are moved through the domain. The control particles are moved towards their targets with an attraction force, generated from the

correspondence with target particles, and then the resulting velocity is distributed over space to transport the smoke marker particles. This is the process by which we compute our bulk flow field, and in the next section, we will describe how we implement our turbulent flow field.

Chapter 4

Turbulent Vortex Flow

In the previous chapter, we described how we generate the base flow field that drives the smoke towards the target shape. This bulk flow efficiently matches the source density to the target density; however, on its own, it does not have any turbulent motion, and does not appear fluid-like. That is, it lacks fluid behaviour, and moves directly towards the target, in a somewhat inert and uninteresting manner. To add the missing characteristic rolling, turbulent fluid motion, we generate a turbulent flow field, using the vortex particle method. We then combine the base flow field with the turbulent flow field, to get a final combined result. This base and detail approach will drive the smoke towards the target shape, while simultaneously having the fluid-like rolling motion characteristic of a smoke cloud. Since the turbulent flow field does not dissipate immediately, the smoke will not come to a complete stop after reaching the target; it continues to move on its own until the user stops the vortex simulation, through an artist-controlled dissipation effect. This prevents the simulation from coming to a full stop too quickly after reaching the target, which could have an unnatural, static appearance.

4.1 Vortex Flow

As we mentioned in Chapter 2, vortices often occur in nature. We see them in tornadoes, when water empties down the drain of a sink, or in smoke rings. From a mathematical perspective, the equations for vortex flow are derived directly from the Navier-Stokes equations. The curl of a vector field V is defined as $\bigtriangledown \times V$, where $\bigtriangledown$ is the *del* operator, a vector operator which in 2D is $\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$ and in 3D is $\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$. The curl of a vector field can be thought of as a way of representing the flow of the field, where at each point in the vector field, there is a little spinning paddle wheel, which is the curl at that point. By taking the curl of the Navier Stokes equations, we obtain equation (19):

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \bigtriangledown)\omega = (\bigtriangledown \mathbf{u}) \cdot \omega + v \bigtriangledown^2 \omega + \frac{1}{\rho} \bigtriangledown \times \mathbf{f}. \tag{19}$$

The left hand side of this equation describes the velocity given by the vorticity of the system. In this section we describe how to determine the flow field from a single vortex, which we represent as a vortex particle; this is the same thing as solving the left-hand side. The right hand side of the equation contains three terms, and all of these terms describe how the vorticity of the system changes over time. The first term, which we describe in Section 4.4, describes how the spin of individual vortices is itself rotated, stretched and sheared by other vortices. The second term, which we will describe in Section 4.3, is the diffusion term, which gradually reduces the energy in the system. The last term described the effect of external forces on the system. This, however, we do not treat specially; vortex particles can move according to any flow field, or any external force. Applying external forces to the vortex particles directly is sufficient to handle the action of the external force term.

These equations can be solved on a grid; however, instead of solving a scalar Poisson equation for pressure, as with a traditional Eulerian fluid solver, we end up with a

vector-valued Poisson equation. Solving a vector Poisson equation can be a complex task, and is easily side-stepped by using the Lagrangian vortex particle method. Particle based methods also have a important secondary advantage of avoiding grid based numerical dissipation, a common drawback of Eulerian techniques, which we mention briefly in Section 2.1.1. Although there are methods, such as vorticity confinement, to combat numerical dissipation, purely Langragian vortex methods have precisely zero dissipation, and do not require a precise linear solver to reduce error. For the full derivation of the solution to computing the velocity from the vorticity, refer to the textbook by Cottet and Koumoutsakos [12]. We are most interested in the result that the velocity contribution, $\mathrm{v}(x)$ of a vortex particle at a point x, is described by the Biot-Savart kernel, which is listed in equation (20):

$$\mathbf{u}_v(x) = \frac{1}{4\pi} \sum_{i=1}^{N_v} \frac{(x - x_i) \times \omega}{|x - x_i|^3}. \tag{20}$$

Here N_v is the number of vortex particles. This kernel gives the exact analytic solution to computing velocity from vorticity, which is exactly what we want. However, it has two serious problems we must solve. First, the magnitude of $\mathbf{u}_v(x)$ approaches infinity as x approaches x_i, and it has a singularity at $x = x_i$. This will produce large velocities at some central points. Second, it has *infinite support*; that is, the magnitude of the kernel is greater than zero at every point in space. This means computing the velocity for a large cloud of vortices is quite expensive, as we need to visit every point in space for each vortex.

One technique for solving the issue of the singularity is to *mollify* the Biot-Savart kernel; 'Mollification' refers to using a smooth function to approximate a non-smooth one. In our case, the Biot-Savart kernel is not smooth when the point in space x we are computing velocity at is equal to the position of the vortex x_i; at this point there is a singularity. By using a linear law inside a certain fixed radius, σ, we can solve

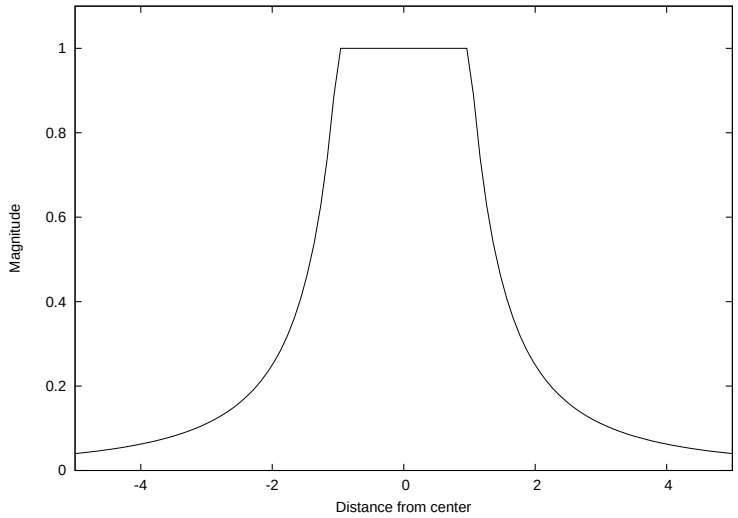


Figure 18: Mollified Biot-Savart energy kernel of a vortex with radius $R = 1$.

this issue, where

$$\mathbf{u}_v(x) = \frac{1}{4\pi} \sum_{i=1}^{N_v} \frac{(x - x_i) \times \omega}{\max(|x - x_i|^3, \sigma^3)}. \tag{21}$$

Thus, if $|x - x_i| < \sigma$, the magnitude of the kernel approaches zero, as x approaches x_i [23]. We depict the energy of this kernel in Figure 18. Additionally we may choose other kernels to replace the Biot-Savart kernel, that do not have a singularity. The second issue, the infinite radius, is a performance problem, and various approaches exist. Usually they involve a small degree of approximation. For instance, we may use a cut-off value; if the magnitude of the velocity beyond this radius is below a specified minimum, we clamp this magnitude to zero [46]. This means we do not have to visit every point in space for every vortex. For our work, we use the method of Angelidis et al. [52], where instead of using the "ground-truth" Biot-Savart kernel, we express the velocity-from-vorticity equation as

$$\mathbf{u}_v(x) = \sum_{i=1}^{N_v} [(x - x_i) \times \omega] \, \xi(|x - x_i|^2). \tag{22}$$

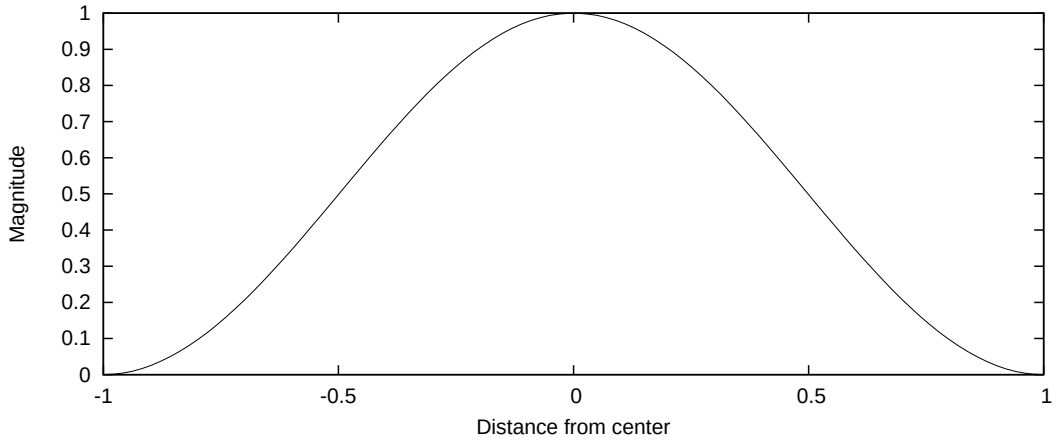


Figure 19: Compact energy kernel of a vortex.

Here, ξ defines the strength of a vortex, given a squared distance to a point in space. We define the compact energy kernel ξ of a vortex, as in Angelidis et al. [52], and shown in Figure 19, as

$$\xi(x^2) = \begin{cases} \left(4 - \frac{20}{x^2+4}\right)^2 & \text{if } x^2 < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

This expression requires no square root computation, which is is good for performance. Additionally, it is smooth throughout the domain, and is zero everywhere where $x^2 \geq 1$. Using this kernel as a function for vortex strength, we also store a vortex radius, σ. This radius determines the spatial range of the vortex, and the vorticity value, ω determines both the direction and magnitude of rotation. Thus the expression for vortex vorticity in our simulation is $\xi(x^2)\sigma$. The task then becomes, given these expressions, how to compute the accumulated velocity at every point in our domain of interest, for every active vortex particle.

4.2 Velocity from Vorticity

In Section 4.1 we described how we compute the velocity from a single vortex, using equations (22) and (23). The vortex simulation engine's job is to use these equations to produce a velocity field that the smoke particles travel along. This vortex particle method will produce natural, turbulent fluid motion. So, for every point in space where there is smoke, we must compute the net vortex velocity at this point, given a collection of vortex particles, each with a radius and vorticity vector. Although there are many different methods to computing velocity from vorticity, they generally can be divided in two groups: exact methods and approximate methods.

Exact methods require that, for every smoke particle, we compute the precise vortex velocity at this point; then the smoke moves with that correct velocity. The simplest method, brute force, belongs to this category of approaches. We compute the total velocity contribution of every vortex particle, at every smoke particle's position, using the equations above, and use this velocity to advect the smoke particles. Intuition may suggest that this technique may be very slow, and this is correct; this algorithm quickly becomes infeasible with thousands of vortices, and hundreds of thousands, or millions, of smoke particles. A more efficient alternative is take advantage of the fact that all of our vortices have a finite radius. Because they have zero influence beyond their area of effect, a spatial search of nearby vortices can reduce the computational cost significantly; we can ignore all vortices whose area of influence does not reach the point at which we are computing velocity. To implement this, we might use a data structure such as the uniform grid described in Section 3.3.3, or a binary tree structure, such as a KD-tree [73]. Note, however, although this will greatly improve runtimes from brute force, each vortex may affect thousands of smoke particles. Furthermore, it does not reduce the worst case computation time. For example, in the degenerate instance where all points and vortices all have the

same position, spatial acceleration fares no better than brute-force. Thus we might look for a faster alternative to spatial search.

A good alternative to exact methods is to sacrifice a bit of precision, for a hopefully large gain in performance. The family of algorithms known as *multipole methods* [60], offer large gains in speed for relatively small losses in accuracy. Furthermore, they do not require that the vortex kernel is finite; they prefer vortices to have infinite support. To understand how multipole methods work, consider a smoke particle, far away from a cluster of distant vortices. The effect of each vortex on this point is very small, and in fact all these vortices have very similar velocity contribution, because the displacement vector from each vortex to this point will be very similar. The key idea in the multipole method is to treat distant groups of vortices as a single vortex; hence we only compute velocity with nearby vortices exactly, while distant vortices are summed to average vortices, and then the single average vortex contributes velocity to the distant smoke particle. These velocities are stored in a tree data structure, which has fast lookup for these average vortices. This approach offers compelling advantages in terms of accuracy and performance; however, since we are interested in maximizing performance, and the multipole method, by nature, is specialized for vortices with infinite support, we still might find a method that is even faster.

Although we are using Lagrangian vortex particles, with a point-based approach, we can still integrate some elements of the Eulerian frame, to make our velocity computation much faster. By placing a bounding box around the dynamic region of our simulation, we can subdivide that region into a grid of cells, or a uniform grid. Then, we can compute the exact vortex velocity at every cell center, directly. Given this uniform grid, computing an approximate vortex velocity anywhere in the domain is simply a matter of linear interpolation from the nearest cells. Note, we must make sure to use a grid size small enough that the features of the vortices are preserved. We are not attempting to have perfect physical accuracy, however, if the grid size is

too small the structure of the vortices will be lost due to aliasing.

We do this direct velocity from vorticity computation using a scanline "rasterization" approach, inspired by pixel shaders. For every vortex, we find a bounding box of grid cells from its area of influence, given by its radius and magnitude; then we traverse each cell in order, adding the new vortex velocity to each cell center we traverse. We use SIMD vector instructions for this computation, which gives the key performance advantage of this method. These instructions peform four velocity computations in a single instruction, and are optimized for speed. Although we use linear interpolation for points in between grid cells, and this introduces a source of error, we are interested in interactive simulation rates, and the general appearance of a fluid rather than strict physical accuracy. For our method, the performance advantages are compelling. We now have our velocity field; however, the other terms in equation (19), which describe how the vorticity of the particles change over time, require separate solutions.

4.3 Particle Strength Exchange

The second term of the right hand side of equation (19) is $v \bigtriangledown^2 \omega$, which is a diffusion term that describes how the spinning vorticity slows down over time. Since the vortex particle method has no viscous forces at all, it is useful to add in a controlled amount of viscosity for artistic choice. In our Lagrangian vortex particle framework, we implement this diffusion effect using a technique known as *particle strength exchange* [12,42], or PSE. Each vortex gradually gives a a little bit of its own vorticity value to nearby vortex particles, and in turn takes a little bit of the vorticity of its neighbours. Over time, vortices with opposite spin will exchange their vorticity, bringing the net result closer to zero. This will slow down the rate of spinning, and simulate the process of diffusion.

We implement PSE using a uniform grid data structure. We arrange the vortex particles into this spatial grid table, using the same method as we describe in Section 3.3.3. Then, finding neighbouring pairs of vortices is simply a query into this uniform grid of indexed vortices. If we have a pair of vortices with vorticity vectors ω_i and ω_j, and kernel radii σ_i and σ_j, we then compute the new vorticity value for vortex i, ω_i', as

$$\omega_i' = \omega_i + \upsilon * d_{ij} * (\sigma_j^3 \omega_j - \sigma_i^3 \omega_i), \tag{24}$$

where

$$d_{ij} = \textsc{Saturate}\big(1 - \frac{|p_i - p_j|}{d_{max}}\big). \tag{25}$$

The equations for ω_j' are similar. Here, SATURATE is a function which clamps the input value to $[0, 1]$, and d_{max} is a parameter which governs the maximum distance at which vortex particle can affect another. The parameter υ is a global diffusion strength, governing how much diffusion is present in our simulation. Note that in equation (24), the difference in total energy between the two vortices determines how much energy is transferred from one to the other. The strength of the exchange decreases linearly with the distance between the particles; nearby particles exchange much more energy than distant ones. The next section will describe the other change in vorticity during the course of the simulation; it describes how the direction of rotation of the vortex particles is affected by the system.

4.4 Vortex Spinning

The first term on the right side of equation (19), $(\bigtriangledown \mathbf{u}) \cdot \omega$, is called the stress term. It describes the stretching and tilting of vortices, due to the velocity field. This term

only applies in 3D, and in our system, we implement it by changing the orientation of a vortex particle's vorticity vector at every timestep. By expanding this term in three dimensions, where the 3D velocity vector $\mathbf{u} = (u, v, w)$), we get

$$\omega_x \frac{\partial \mathbf{u}}{\partial x} + \omega_y \frac{\partial \mathbf{u}}{\partial y} + \omega_z \frac{\partial \mathbf{u}}{\partial z} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = J(\mathbf{u})\omega. \tag{26}$$

Note that this involves spatial derivatives, specifically the gradients of velocity. The gradient of a vector field in 3D is a 3 by 3 matrix, called a Jacobian matrix, which we annotate in equation (26) as $J(\mathbf{u})$. In our implementation, which is similar to that of Gourlay [23], we compute the Jacobian of the velocity field, and then compute the new tilted vorticity vector by multiplying a bilinearly interpolated Jacobian matrix with the old vorticity vector. We also scale the result of this term by a strength parameter, S_j, before applying it to the vortex particles. Note that as Gourlay observes, this implementation of the tilting and stretching term will produce errors in vorticity, as a result of round-off and truncation due to linear interpolation. These errors can have the effect of energy being gained or lost from the system, and ultimately cause instability. We constrain the energy of each vortex by normalizing the new vorticity vectors, after the application of the stress term. Thus, each vortex will have the same vorticity magnitude as it did before the tilting term. This normalization has the effect of only changing the orientation of the vortices, not changing their magnitudes; it will stop the simulation from becoming visibly unstable from this step.

4.5 Spawning Vortex Particles

We have described how vortex particles produce velocity, are affected by diffusion via particle strength exchange, and spin with the flow. In this section, we describe how and where we spawn vortex particles in the simulation. To place vortices, we use a simple dart-throwing scheme, similar in nature to how we place control particles, as described in Section 3.3.1. We initialize a uniform grid, storing the current turbulent energy of the system, for use in quickly determining how much vorticity is in a particular region of space. We also store marker particle density in a separate uniform grid, as we described in Section 3.3.1. We only wish to place turbulence in areas where there are marker particles, since vortex particles by themselves are invisible. Thus, after computing the bounding box of the marker particles, and inflating it by a fixed amount to cover the borders, we randomly throw darts into this volume (or area in 2D); when the marker particle density is above a given threshold (which prevents turbulence being created in uninteresting regions), and turbulent energy is below a certain threshold (preventing regions of excessive turbulence), we place a new vortex particle at that dart's position. The particle is given a random vorticity magnitude, in a random direction sampled from the unit sphere, and a random radius. Radius and vorticity magnitude both sample from normal distributions. In this manner we repeatedly spawn vortices, until our domain has a sufficient ambient vortex energy.

One key consideration is that the finite radius particles have sufficient distribution to cover the entire domain of simulation. Because we do not simulate infinite support, there will likely be regions of space that are completely unaffected by any vortex particles. However, our domain of interest is the region of our smoke marker particles. The randomized rejection approach described above, given enough trials per frame, is enough to ensure there is a sufficient amount of turbulent energy throughout the smoke volume.

4.6 Particle Advection

Now that we have described our methods of vortex creation and simulation, in this section we describe how we move the control particles, the marker particles, and the vortex particles themselves with the bulk flow and turbulent flow fields. Since we intend to simulate hundreds of thousands to millions of marker particles, we wish to do this with a preference for performance, without sacrificing numerical accuracy. We first add the interpolated control velocity field, described in Section 3.4.2, with the turbulent vortex velocity field described in Section 4.2. This single velocity field is stored in a single uniform grid structure, and can be quickly sampled for an approximation of flow velocity, using bilinear interpolation in 2D, or trilinear interpolation in 3D. Depending on the choice of parameters, the control flow field and turbulent flow field may or may not be the same size and grid coarseness; thus, we combine them into a new grid by sampling velocities from both fields.

We perform particle advection in two steps. In a first pass, we advect the marker and vortex particles with the combined flow field described above. Since control particles are affected by attraction force, as we describe in Section 3.4.1, and are the source for the bulk flow field, we move them separately, in a second pass, according to integrated velocities from the attraction force. The second pass we describe below in Section 4.6.2. We also do not always apply vortex motion to the control particles, instead leaving that as a choice for the animator; advecting the particles with the vortices causes a slower convergence to the target shape, but also allows for more fluid motion.

4.6.1 Marker and Vortex Particles

Given the previously described merged flow field, which combines our individual bulk and turbulent flow velocity fields, we now wish to advect the marker and vortex particles. Marker particles and vortex particles move purely according to the combined flow field. To find the new position of a particle given a velocity, the problem statement takes the form of an ordinary differential equation. However, in our case, we can make a simplification; because we have a velocity field, we can simply step the position of the particles through this field, instead of recalculating the entire state vector at every step. In this simplified case, if the position of a particle i is x_i, we have $x_{n+1} = f(\Delta t, x_n)$, where f is a function that computes a new position from the previous position for a given time step value, Δt. The simplest solution to this advection problem is the Euler method. If $\mathbf{u}(x)$ computes the velocity of the field at x, the Euler method solution is $x_{n+1} = x_n + \mathbf{u}(x_n)\Delta t$.

Although the Euler method is fast, it is only a first-order method, and thus produces a great deal of error. This error grows as it accumulates over multiple frames, and in our simulation, causes the rapidly rotating points near the centers of vortices to over-shoot their targets, and travel gradually outwards. This leaves 'holes' located in near the center of the vortex particles, where the error is larger. We instead use a second-order Adams-Bashforth scheme, described by Park and Kim [42], where $x_{n+1} = x_n + \Delta t \left(\frac{3}{2}\mathbf{u}(x_n) - \frac{1}{2}\mathbf{u}(x_{n-1})\right)$. This higher-order scheme uses the previous frame's velocity to estimate a more accurate particle trajectory. It has the downside of requiring additional storage, but has the convenient property of only requiring a single velocity sample per particle, per frame, which makes it nearly as fast as the Euler method, and significantly faster than methods which require multiple samples of the simulation. We have found this scheme effectively addresses the problem of vortex 'holes', producing a desirable mix of accuracy and performance.

4.6.2 Control Particles

Control particles have two main considerations to distinguish them from the marker and vortex particles. Firstly, they are affected by the attraction force, or other external forces, which then are integrated over time to produce a velocity, independently of the bulk and turbulent velocity fields described in previous sections. Secondly, we may or may not wish the control particles to move with the turbulent fluid flow field. Choosing not to move the control particles with the vortices can have the effect of increasing control, or alternately, choosing to move them with the vortices can increase the appearance of being a flowing fluid. If we decide to leave the control particles unaffected by the flow field, they retain their trajectories quite easily, and assume the target shape very quickly. When we move them with the fluid particles, the net effect produces a simulation with more overall turbulent motion and appearance of a fluid, but the particles converge more slowly towards the target shape. It is also possible to blend the velocity of the bulk flow field with a strength parameter between zero and one to advect them partially with the flow, giving a trade-off between these two effects.

We integrate the attraction force, and any other acting forces, over time, using the Euler method described above. This produces a velocity, which we then store along with each particle. In the case where we choose to move the control particles with the turbulent flow field, we sample the vortex velocity field at the positions of the control particles, and add this sampled velocity together with the previously integrated velocity. We then use the Adams-Bashforth scheme, above, to integrate velocity over time, finding the new positions for the control particles. In this manner we strive for good performance and accuracy when advecting the control particles, while maintaining their trajectories given by the attraction force.

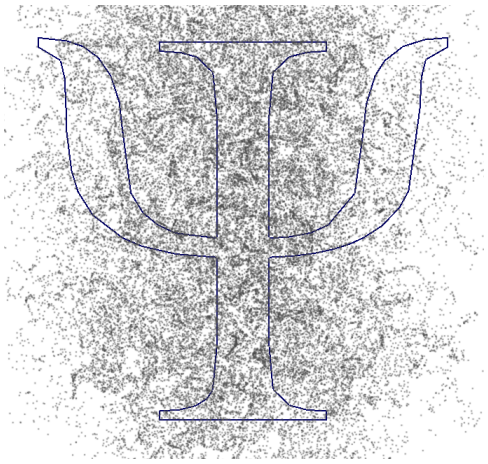


Figure 20: The result of disabling the marker particle redistribution.

4.7 Marker Particle Redistribution

When we advect the marker particles along with the combined flow field, they often end up outside the radius of the control particles. This happens because the chaotic movement from the vortex flow field may often trend away from the region of influence of the control particles. This has the undesirable effect of leaving marker particles outside the area of control, and obscuring the shape boundary behind a cloud of smoke. Because we generate the vorticity from random turbulence vectors, with their strengths and vorticity vectors sampled from a uniform distribution, the next flow field will spread the smoke through space. This spreading, a similar effect to diffusion, produces a cloud of smoke enclosing the object, and the details of the mesh become less apparent. Figure 20 shows an example of this effect.

Since we are more concerned with producing a pleasing effect than physical accuracy, we have a certain amount of leeway in choosing how to solve this issue. We use a simple randomized incremental scheme to find marker particles outside of the control area, and randomly place them using rejection sampling. The algorithm is listed in listing 2. Note that for sampling control density, we use the control particle density field described in Section 3.3.1. Note we must ensure that the value for ϵ is

small, otherwise the algorithm will not terminate.

Algorithm 2 Incremental randomized marker particle redistribution

```
ControlRegion ← ComputeBoundingBox(ControlPoints)
for 1 to C_redist do
    i ← Random(1, NumMarkerParticles)
    p ← MarkerParticlePosition(i)
    if ControlDensityAt(p) < ε then
        repeat
            p' ← RandomPointIn(ControlRegion)
        until ControlDensityAt(p') < ε
        MarkerParticlePosition(i) ← p'
    end if
end for
```

Although it is not guaranteed to fix every marker particle each frame, the incremental nature of the algorithm is such that, if C_{redist} is sufficiently large, gradually the particles move back inside the volume. If the number of redistributed particles is chosen to be fairly low, compared to the overall number of particles, this effect is not noticeable, and appears to be smoke gradually dissipating into the atmosphere. It also has the advantage of being simple to implement, and computationally inexpensive. This method will prevent a cloud of shapeless marker particles from being left outside the region of influence of the control points.

4.8 Summary

In this section, we presented our methods for vortex particle creation and simulation, and particle advection. We describe how we compute the velocity for a single vortex, and extended this to efficiently computing vortex velocity over the entire domain. We also discussed how we spawn vortex particles, how we keep the marker particles within the area of interest, and how we move particles in the simulation along with the flow fields and forces. In the next chapter we will present some results from

our simulations, describe the performance of our implementation, and discuss the advantages of our approach.

Chapter 5

Results and Discussion

Now that we have discussed the details of our method, namely our base flow field and our turbulent flow field, we will present and discuss some results from our simulations. We also will first discuss the technical details of our implementation, with particular attention to how we implement optimizations with parallel programming and rendering. We will then compare our results with the results of other implementations of target-based fluid control, and then continue to discuss some of the advantages and open issues remaining from our work.

5.1 Implementation

5.1.1 Parallel Implementation

One of the main criteria we selected when designing our proposed solutions is to prefer methods which are fast, and preferably easily parallelizable, over more accurate, slower, and complex techniques. By selecting simpler algorithms, for example by avoiding recursive functions usually involved in spatial structures such as octrees and KD-trees, and preferring uniform grids and large particle clouds, we often can make use of the parallel processing ability of modern CPUs, to reduce our computation

time. We also exploit the GPU for rendering. This often leads us to a point where we run our simulations at interactive rates, even with reasonably detailed and interesting simulations.

There are three main parallel architectures that we will discuss, each of which has advantages and disadvantages for implementing parallel algorithms. These methods are threading, SIMD vector instructions, and GPU programming.

5.1.2 Threading, SIMD and GPU Programming

Multiple hardware threads and cores, available on most, if not all, modern CPUs, allow simultaneous execution of different code in different mini-programs, or 'threads', while using a shared memory space. This parallel programming architecture is the most versatile, as all of these threads operate independently, and are capable of processing different instructions at different times. The main limitation imposed by this architecture on the algorithm designer is to ensure multiple threads do not write to the same memory location at the same time. This can cause a conflict known as a race condition. Using 'thread local storage', or TLS, we overcome this limitation. TLS provides a local data buffer for each thread, which, at the end of the routine, we combine in a *reduction* operation to form our final result. For example, when we compute the velocity grid from a collection of vortices, each thread stores a separate, identically sized, uniform velocity grid. Each thread individually does not need to worry about other threads, as they can not possibly conflict during a write. The grids are then reduced, via summation over all local grids, to form a final superimposed velocity field.

We also exploit a common feature of modern CPUs called *single instruction, multiple-data*, or SIMD, vector instructions. The most ubiquitous set of such instructions is the Intel SSE instruction set, which is available in several versions, some of which are almost universally available on modern desktop CPUs. In this architecture,

we store a packed vector of numbers in memory, most commonly groups of four floating point, or integer, numbers, forming a 128-bit vector data type. The programmer then instructs the program to execute instructions that perform four computations at once, on working sets of 128-bit values. For instance, using the SSE 'ADDPS' instruction, we can sum four floating point values in a single instruction. The 'RSQRTPS' instruction is a notable special case, as it computes four reciprocal square roots in one instruction, quite a bit faster than using a standard library computation for square root. For our threading implementation, we use Intel's Threading Building Blocks library, a high-level set of routines that facilitate cross-platform, powerful threading primitives.

There are two significant limitations imposed by the SIMD architecture on the algorithm designer. It is more efficient, though not strictly necessary, to load and store data to and from these 128-bit registers in aligned 128-bit memory accesses. That is, adding four values randomly scattered through RAM will require the CPU to load these values single-threaded, and then the programmer to insert them into a single 128-bit packed vector. Avoiding this gather operation will dramatically improve performance due to how the CPU stores memory in cache; it is much faster to load sequential data than randomly accessing it. The second important limitation is that "single instruction, multiple data", by definition, means a single instruction has the same operation on all elements, of the vector. It is impossible to, in a single SSE instruction, add two of the elements in a vector, while computing the difference of the other two. This restriction makes implementing divergent code branches much more difficult than using hardware threads; the designer must visit all paths of the code, for every thread, unless all or none of the threads will access that branch. Nevertheless, if these restrictions do not degrade the speed of the implementation too much, SSE code can provide speed increases ranging from 2x to 20x, depending on the instructions in question. We discuss more details of our SIMD implementation in

Section 5.1.3.

A third and final important parallel architecture we will discuss, although one we do not exploit for our simulation, is the GPU programming architecture. Consumer GPUs, widely available on desktop computers, offer incredible amounts of raw floating point operations per second. The restrictions of GPU programming are similar to SIMD; however, GPUs are extremely complicated machines, and are becoming ever more general purpose with subsequent hardware revisions. They also allow more flexibility than SIMD design, such as allowing divergent code branches, or random memory access. However, algorithm performance on GPUs can vary widely between hardware vendors; sometimes the penalties for using techniques such as branching differs widely. We did not exploit the potential of using GPUs to compute our simulation data; however, we implement our rendering algorithm using the GPU, as we describe in Section 5.1.4. Moreover, since the SIMD architecture is strictly less flexible than the GPU architecture, our simulation algorithms that use SIMD code are capable of running on the GPU. We believe exploiting the processing power of consumer GPUs for our work is a promising direction of research.

5.1.3 Array of Structures vs Structure of Arrays

We must carefully design our algorithms to fit the constraints of each particular parallel programming architecture, though they share some common ground. As previously mentioned, multi-core and multi-thread implementations rely on data exclusivity, or the ability to write to memory guaranteed to be exclusively accessed by that thread, and to not read from memory currently being modified. SIMD instructions require both aligned and linear memory access for best performance. Furthermore, they enforce data parallelism, with strict instruction symmetry; a single instruction executes for all threads. Now, since we require linear, aligned access for our data structure, when loading 2D or 3D vector data, if we store vectors as component-wise structures,

(x, y) or (x, y, z) in 3D, we have a few problems. First, our data is not aligned in 3D, we have three components for a four element register, which means we have a complex task if we want to access aligned memory. We also will often be left with the task of adding elements horizontally (within the same vector), such as adding the x and y components of the vector. These problems can quickly complicate the optimization process.

Instead of storing a single array of packed structures with (x, y, z) data per element, called the 'structure-of-arrays' layout, we choose the 'array-of-structures' layout. The array-of-structures layout stores data in a structure of two or three large linear single-component arrays; it stores two large float arrays for 2D data, and three large arrays for 3D data. We now perform our mathematical routines on the 2D or 3D vectors by loading four elements for every vector component, where we have complete independence between the different x, y and z values. Without the previously alluded overhead of reorganizing our data in memory to fit SIMD operations, this allows us very good performance when accessing memory. It also allows a degree of simplicity when designing our algorithms.

5.1.4 Rendering Implementation

Smoke rendering, as we discuss in Section 2.3.2, is a broad topic, with many algorithms to choose from. Most offer various advantages and disadvantages in terms of performance, accuracy and ease of implementation. We chose for our animations a method proposed by Swoboda which he calls "stochastic" shadow mapping [66]. Note that his method is not the same as "stochastic transparency", proposed by Enderton et al. [14]. Swoboda's method has the main advantages of being relatively cheap to compute, simplicity of implementation, and attractive results.

Stochastic shadow mapping is based on a similar idea to traditional shadow mapping. In traditional shadow mapping, for each light, we first render the scene to a 2D

buffer called the shadow map; we transform the scene objects into light space, and store the smallest depth value at each pixel position. Then, in a second pass, in a fragment shader, we sample the shadow map to determine if the scene is in shadow. We again transform the position of the object (pixel) in question to light space, and test this against the shadow map value; if the depth is smaller, it is lit, and if it is larger, then it is behind a lit surface, and is in shadow.

Stochastic shadow mapping uses a similar basis for its computation. Using a large shadow map, in our results 4096x4096 pixels, we write the smoke particles as single points. The x and y coordinates of the point in light space become texture coordinates into the 2D shadow map, and we store the z depth of the point to the light. Note that many particles may collide; we will discuss this problem shortly. Assuming no collisions, we can perform a traditional shadow computation as follows: in a second pass, for every smoke particle, we compute the light space coordinates of the particle and and sample the shadow map with a large (say 20x20) kernel. At each sample, we test the stored particle depth versus the current particle's depth; the fraction of particles in front of this particle determines if the particle is in shadow or not. Instead of looking up a single shadow value at the particle for opacity, the large kernel averages the many surrounding shadow values to give a smoother, softer result. The fraction of occluded samples determines the final opacity value for the particle. In this fashion, particles behind many nearby particles are shadowed, while particles near the light are visible and rendered at maximum brightness.

As we mentioned, when drawing the smoke particles as points in the shadow map, there may be collisions. If we only draw certain particles to the map, we will be left with noisy opacity values. Thus we introduce a stochastic nature to the algorithm, to ensure full coverage of the smoke particle data set. We insert particles into the shadow map with a small, random offset. This helps compensate for the fact that some particles may almost always be occluded by nearby particles and ensures that

every few frames, each particle will visit the shadow map at least once.

Now we have a different problem: sometimes very rapid changes in how particles occlude each other can result in flickering, as particles come out in front or behind other particles. We solve this flickering using the same method as Swoboda; we use a temporal smoothing step to blend each particle's final shadow amount value with that of its shadow value from the previous frame. This blending converges to a stable solution after a few frames, and because the topology of the cloud does not change dramatically from frame-to-frame, the approach offers reasonably alias-free renderings with good temporal coherence.

The stochastic shadow mapping approach is simple to implement, very fast, and only requires one texture write and k texture lookups per smoke particle, where the total kernel size is k pixels. It has one disadvantage compared to some other techniques we discuss in Section 2.3.2; proper rendering requires fully ordered transparency for the alpha-blend step. This means that unlike the order-independent methods we described before, we must sort our particles back-to-front relative to the camera before rendering. For this we use a threaded sort, implemented using Intel's TBB library. An interesting alternative to this CPU sort is a GPU-only incremental even-odd merge sort [33], which, due to its incremental nature, can be amortized over several frames, further reducing computation time.

The downside to the stochastic rendering process is a large degree of temporal aliasing. Additionally, since particle density varies spatially, some areas may be under-sampled and have a spotty appearance. Previous work in smoke simulation frequently uses Eulerian simulation, which has a smoother appearance, despite having less detail in some areas. Using volumetric methods, as we discussed in Section 2.3.2, would provide a more alias-free shadowing, at a cost of performance. Increasing the particle count helps the under-sampling issue; additionally using an anisotropic particle kernel

can provide a smoother appearance, as Angelidis et al. [3] propose in their particle-based method.

Now that we have discussed some parallel optimizations in our method, and the implementation of our point renderer, we will present some results from our simulation framework.

5.2 Results

In general for a good fluid simulation there should be rolling, turbulent motion with a high degree of detail. In our case we also wish the smoke to take the shape of a target mesh: a good simulation should make the mesh features visible while still allowing the smoke to move in a turbulent manner. Although previous methods show gentle rolling smoke, we believe our method is best suited to more violent turbulence with a great deal of vortex energy; thus a large degree of turbulent motion is another feature we are looking for in our simulations.

Figure 21 shows a sequence in which the target mesh takes the form of the Greek letter psi, then shifts to the to the letter omega, causing the resulting flow to 'morph' between the two targets. This experiment is similar to an example of Fattal et al. [16]; we show their results in Figure 22. Our result has the advantage of having continuous turbulent motion even after the smoke has reached the destination shape. In contrast, the smoke in Fattal et al. stops exactly upon reaching the target. On the other hand, their simulation matches the shape border more precisely, and offers somewhat more detailed turbulent flow. Their detail comes from the expensive pressure solving step, performed on a large grid; in comparison the simulation and rendering times of our results are much shorter. For reference, we show a visualization of the paths of the control points from the morph sequence in Figure 23.

For the letter morph scene, our simulator takes on average under 500 ms to simulate and render the scene per frame. This scene has 2 million marker marker particles, 10,000 control and target points, and roughly 10,000 vortices. Because we use the GPU for interactive rendering, we render the scene simultaneously as we simulate the next frame. Furthermore, the total render time is always dominated by the simulation time. We do not report on a split between simulation and rendering times, as disabling rendering does not reduce the rum time budget for these scenes. Detailed timing numbers for the letter morph simulation are listed in Table 1. These timings are generated on a machine with an 4 core 8 thread core i7 Intel processor, with a nVidia GeForce 570 GTX graphics card. Fattal et al. report their simulation to take about 10.6 seconds per time step on a Pentium IV, with several time steps per frame, for a total time of 35 minutes of computation time per second of animation. They also do not list render times. Although it is very difficult to compare performance on machines from different generations, the decrease in run time in our simulation is likely not due to just the difference in computational power.

In Figure 24, we again present a comparative result, to the results of Shi and Yu [56]. Their results are shown in Figure 25. For this test, we again used 2 million marker particles, 10,000 control particles, with approximately 10,000 vortices. This scene, which uses the mesh of a standing horse as a target shape, again shows less accurate matching of the target shape boundary; however, since they also have interior motion inside the shape, they show generally less precise boundaries than Fattal et al.. They also use a different rendering method, which has the appearance of lighter more transparent smoke. They do not report the profile of the computer they use for their simulation; however, they report simulation times of 4 hours and 15 minutes to generate a 1250 frame animation sequence, giving an average simulation time of 12.24 seconds per frame. In contrast, our result takes around 500 milliseconds per frame. Again they do not report render times, and our result is more than an order

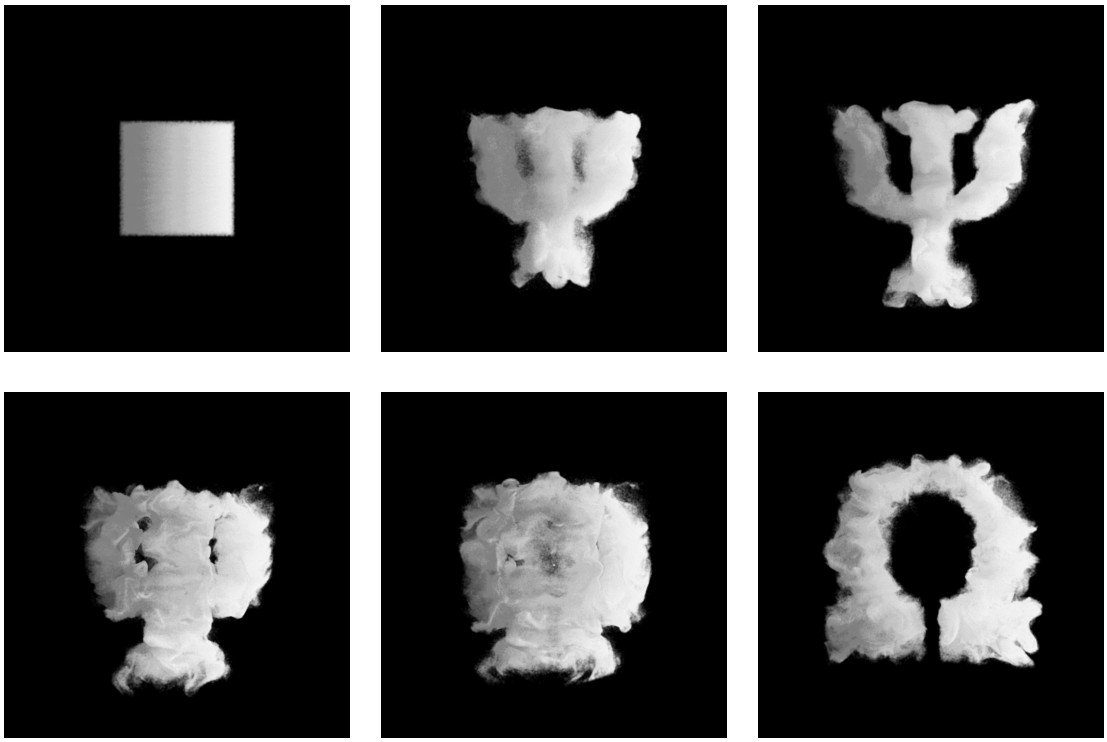


Figure 21: Morph sequence

of magnitude faster.

We also present the result from matching an animated target mesh in Figure 26. Skeletal animation does not cause our simulation any significant performance overhead, with this scene taking again between 120-170 ms per frame with 400,000 marker particles, 1000 control points and 10,000 vortices. This character has the appearance of a ghostly smoke creature; although features are more turbulent than sharp, the motion of the smoke has an interesting, lively appearance.

5.3 Parameter Choices

Now that we have presented some of our results, we will make a note of a few important steps in how we generated them. Specifically, we will address common problems

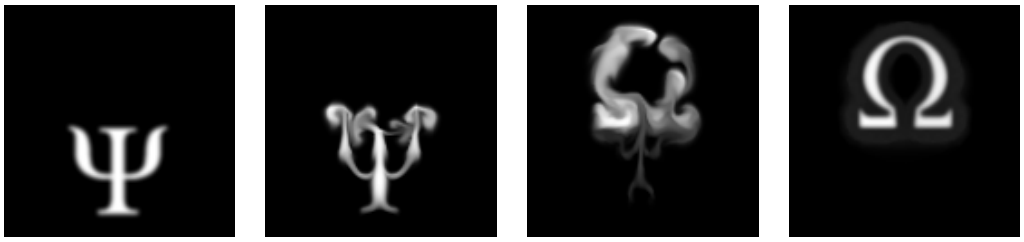


Figure 22: Morph sequence of Fattal et al. [16]. Image © ACM, used with permission.

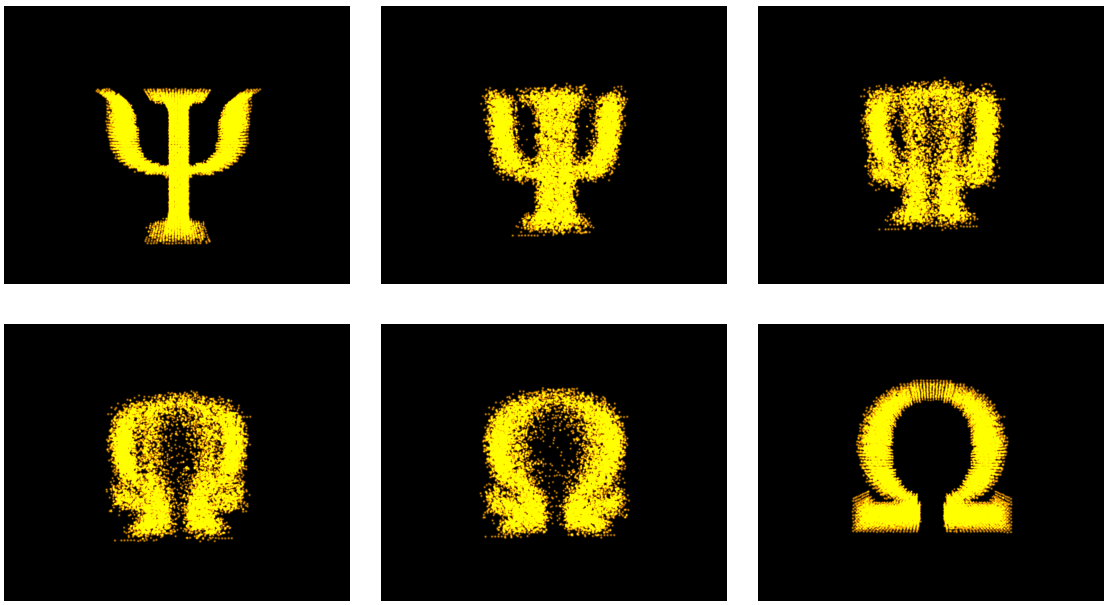


Figure 23: Morph sequence, control points only

Module	Time (ms)	Time (%)	Parallelism
Evaluate Marker Density	102	21%	Multi-thread, SIMD
Evaluate Control Particles	82.6	17%	Multi-thread, SIMD
Advect Marker Particles	117	24%	Multi-thread, SIMD
Redistribute Marker Particles	36.4	7%	Multi-thread
Depth Sort	127	26%	Multi-thread
All Other Operations	23	5%	Various
Total Time	488	100%	N/A

Table 1: Timing results for the letter morph sequence

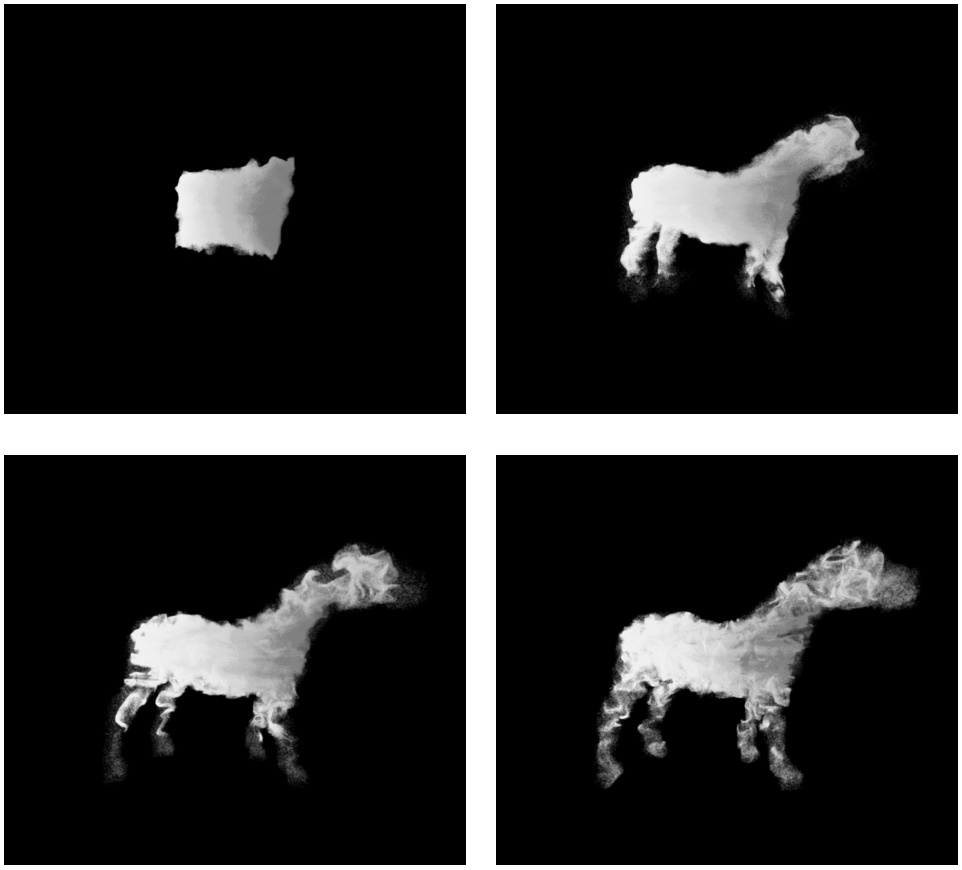


Figure 24: A sample animation matching the shape of a horse

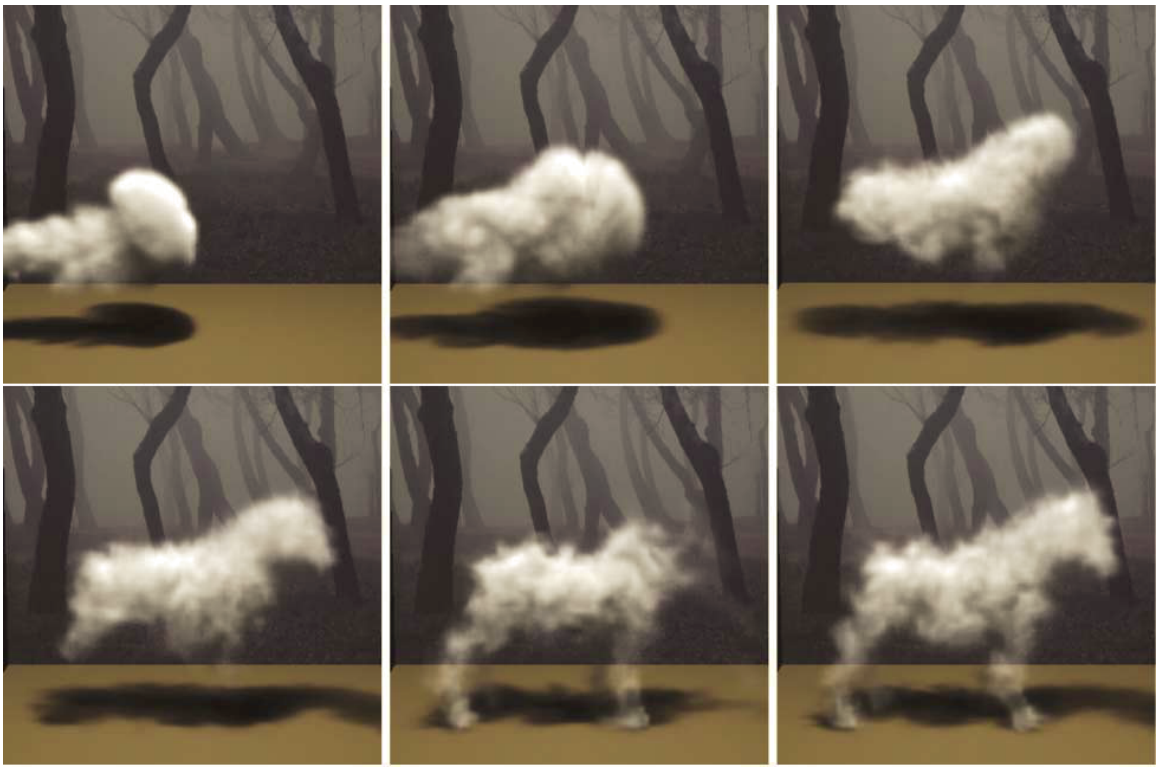


Figure 25: Horse shape matching animation from Shi and Yu [56]. Image © ACM, used with permission.

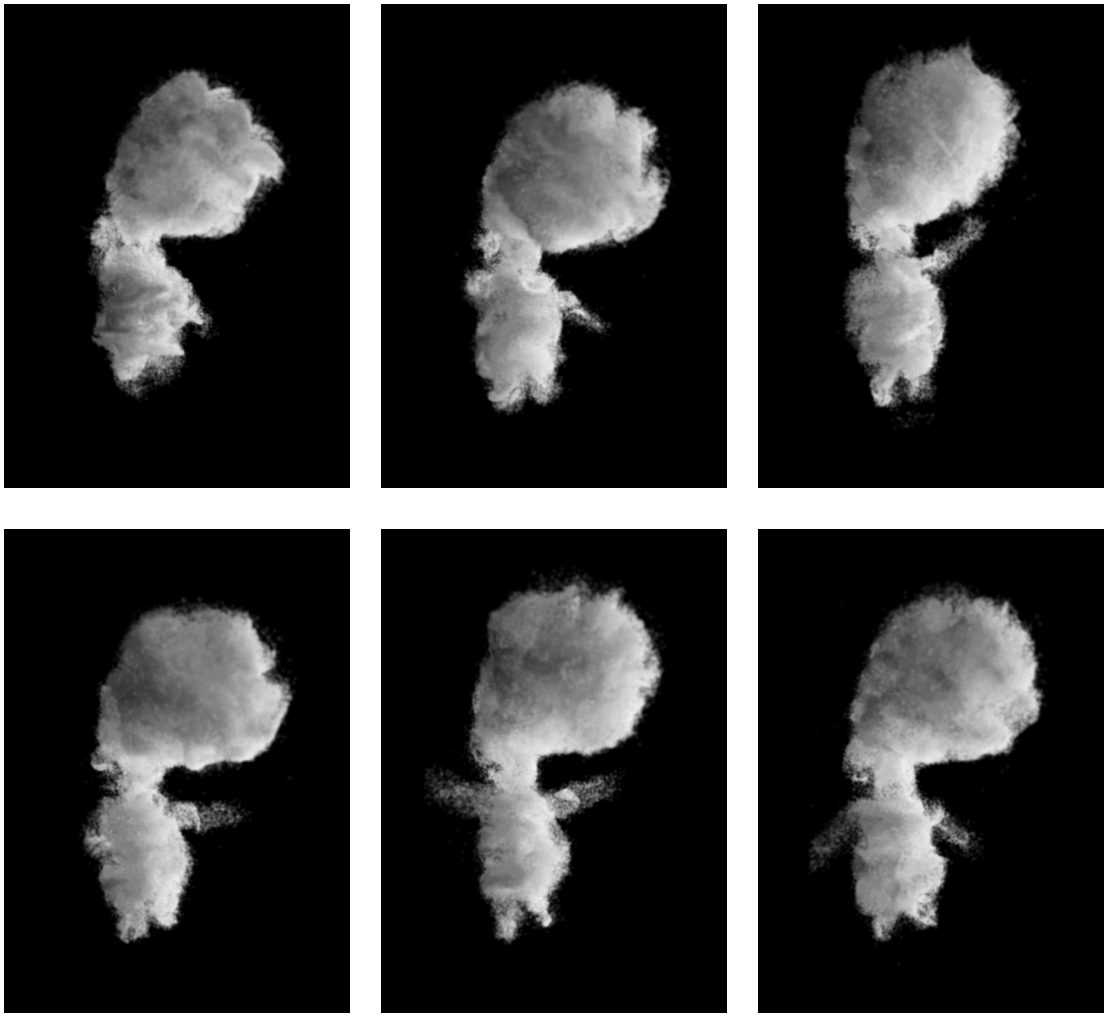


Figure 26: A smoke creature formed from an animated target mesh

with several settings of the simulation, and how to tune these parameters appropriately. Although we have many parameters, which we list in Table Table 2, only a few 'critical' parameters have a great effect on the simulation. Many could be determined automatically from the size of the scene, or have little effect on the simulation result beyond an initial sensible setting.

Some parameters, such as the number of optimization iterations, are relatively insensitive beyond a certain minimum, many simulation variables must have constrained values to achieve a good looking simulation. The simplest such parameter is the size of the dynamically generated grids: the control velocity grid, the vortex velocity grid, the combined flow grid, the marker particle potential grid, the control particle potential grid, the vortex energy grid, and the signed distance field. If there are too few cells, the resulting information such as the vortex velocity, or the values of the signed distance field, will not be able to represent thin, high-frequency features. Figure 27 shows an example of this problem, where, because the signed distance field has an overly coarse resolution, the thin features of the mesh are lost; the sampling process does not place target particles in these thin areas, and some parts of the mesh are under-sampled. As a corollary, there must be a sufficient number of target particle to resolve all the mesh features; there must be an adequate threshold to allow for generating sufficient control particles in the source particle cloud. We use a similar set of parameters as listed in Table 2 for our other scenes; in general we find these parameters work well for many scenes of the same general size.

Another simple issue is that of excessive control velocity force, where S, the global attraction strength parameter, is set too high. Excessive force can overpower the turbulent detail flow; if too high, the points will rush to their targets without any turbulent motion, because the bulk flow velocity will proportionally be larger than the turbulent velocity. For instance, in the smoke creature scene, we use an attraction scaling strength of 30, with a damping constant of 0.3. When the artist adjusts

parameters to shape the behaviour of a scene, the attraction strength parameters are very instrumental. Additionally the number of redistributions, C_{redist}, strongly affects the appearance of the scene.

In addition to control parameters, control particles and marker particles in our simulation are associated with a radius. This radius must be carefully chosen when evaluating marker particle potential, which we described in Section 3.3.1, and for the vortex velocity interpolation, which we described in Section 3.4.2. Setting a very narrow size for the control potential allows for sharper detail in the shape matching; this will constrain the particle redistribution as discussed in Section 4.7, and force the marker particles to an area closer to the mesh. Conversely, setting a larger radius when evaluating control velocity ensures that the bulk flow is smoother. We use a flow particle velocity radius of 0.1 in our smoke creature scene, while we use a potential radius of 0.01; the much smaller potential radius ensures tighter redistribution. Note also this bulk flow is not divergence-free, which can lead to undesirable visual artifacts, though this does not occur in practice in our tests. This leads us to our next section, where we will discuss some of the drawbacks and outstanding problems with our approach.

5.4 Outstanding Issues and Future Work

Although we do not set as one of our objects a purely physically accurate simulation, and are not completely restricted by the constraints of physical realism, there are non-physical aspects to our work that can cause visual artifacts, or noticeable and undesirable behaviour. For instance, our bulk flow field is not divergence-free; as we mentioned in Chapter 2, fluids such as water typically do not exhibit divergence. This can mean that in some simulations, at some times, the divergence in the bulk flow causes points to spread out then later merge back together, giving an uneven

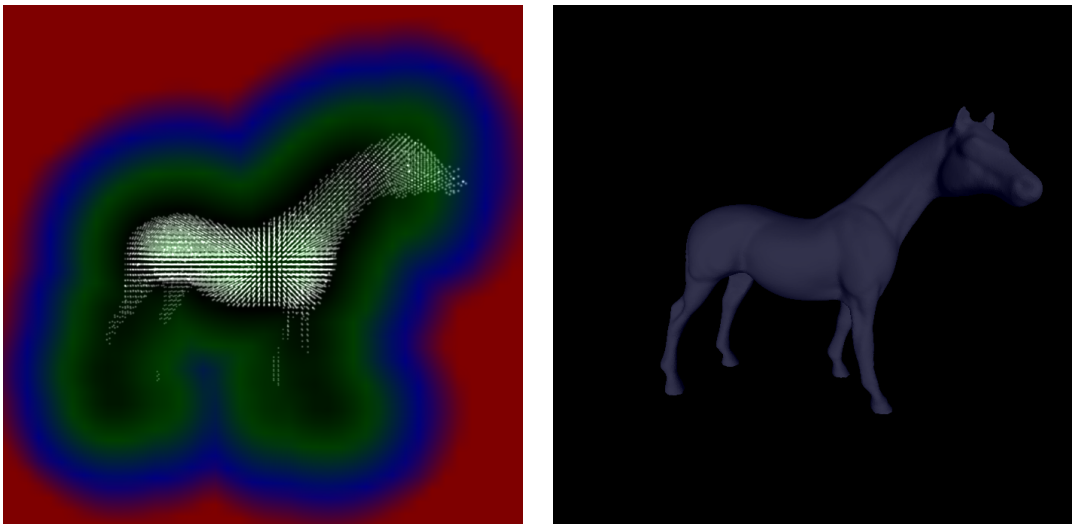
Symbol	Parameter	Value
Critical parameters		
M	Number of marker points	400000
N	Number of control/target points	10000
r_m	Marker particle radius	0.01
r_c	Control particle radius	0.01
S_a	Global attraction strength	30
d	Attraction velocity damping	0.3
	Mean vortex radius	0.5
	Mean vortex magnitude	4
C_{redist}	Marker particle redistribution iterations	400000
Other parameters		
C_{init}	Control particle initialization darts	10000
	Minimum number of SDF cells	30000
	SDF narrowband width	0.1
	Minimum cells in other fields	20000
A	Mesh sampling parameter	0.6
D_{min}	Minimum mesh sampling distance	0.0001
	Mesh point location grid cells	13824
C_{opt}	Optimization iterations per frame	10000
A_ϕ	Attraction strength offset	0.05
A_ρ	Attraction strength scaling	0.5
r_v	Velocity interpolation radius	0.1
	Domain bounding box scale	1.2
	Vortex radius variance	0.01
	Vortex magnitude variance	0.2
	Vortex initialization darts per frame	100
	Vortex initialization minimum energy	300
	Vortex initialization maximum energy	500
	Vortex initialization minimum particle density	0.1
υ	Vortex viscosity (PSE)	0
S_j	Vortex spin strength	0
	Render resolution	1200 by 800
	Shadow map size	4096
	Shadow kernel size	20
	Shadow map draw pixel offset	0
	Shadowing temporal blend weight	0.8
	Particle render size	0.15
	Individual particle density	0.1

Table 2: Parameter values for the smoke creature scene

appearance. Although this effect is undesirable, it is often unnoticeable in our tests; furthermore, since the movement of the control particles is independent of any other particle except its target, and they always arrive at the target exactly, ensuring the target particles are spread out guarantees the net motion of the smoke will be spread out. Note the uniform sampling and stratified sampling both provide a reasonably spread out distribution of target particles. Thus, over time, the marker particles will arrive at a spread out, accurate destination inside the mesh. One possible solution to the problem of intermediate divergence, is to apply the kind of divergence-free pressure projection we described in Chapter 2, which we find in a traditional fluid solver. However, because our simulation discards grids between subsequent frames, the projected velocity field does not iteratively improve the solution over subsequent frames, and the velocity field may require many iterations to converge at a divergence-free solution.

Additionally as we mentioned in Section 5.2, compared to other methods, our simulations do not match the target boundaries as precisely. Because we do not explicitly deal with any sort of boundary conditions, the shape edges are often chaotic and turbulent. For future work, we might investigate the advantages of implementing simple boundary conditions to contain the smoke to the target shape. For instance, using mirrored vortices, paired with vortices inside the boundary, cancels out 'through'-velocity from the vortex flow field [9]. Additionally, panel methods, which involve placing sources, or vortices, along the boundary, and then solving for how strong each source or vortex primitive should be to cancel the through-velocity, is a frequently used method [42]. This form of boundary condition solution might allow for more precise matching of fine features.

Finally, the divide between base flow and turbulent flow sometimes causes an appearance of smoke motion without large-scale turbulence. The base flow is entirely turbulence-free, thus when the simulation drives motion more from the base flow than



(a) Missing details after SDF sampling **(b)** Original model with details

Figure 27: Insufficient grid resolution in the signed distance field

the turbulent flow, it does not appear to be fluid-like. It may be interesting in future work to look in to other directions to approach bulk flow, such as using a hybrid method, by generating a bulk flow field using a Eulerian approach similar to Fattal et al. [16], or by replacing the bulk flow with some method of directed flow from a purely Lagrangian vortex particle frame; that is, to direct vortex particles using a modified vortex kernel or a carefully placed arrangement of vortices.

Generally our results can be characterized as ghostly and turbulent. Performance-wise, we simulate and render our results far faster than previous methods, sometimes bordering on interactivity. In this chapter, we have presented some of our results, discussed the details of our implementation for parallel code and rendering, and mentioned a few limitations we have found in our work. We also discussed ideas for our future work. In the next chapter we will conclude, as well as elaborating on future directions for this work.

Chapter 6

Conclusion

Over the course of this discussion, we presented our novel two-layer approach to shape-targeted fluid control. Fluid control is a difficult, but important problem; fluid systems by their nature are chaotic and governed by complex equations. Fluid control means giving artists and animator effective, efficient, and expressive means to direct a fluid simulation. There are many different options to choose from when designing a system for fluid control. Many use artist controlled primitives, or clouds of control particles, or even methods of precomputation, to speed up a controlled solution. Some, like us, use a target mesh to guide the fluid towards a target density.

In this work, we have presented a style of target control using a Lagrangian particle-based approach, specifically tailored for smoke simulation; using control and target particles, and vortex particles for turbulence, we separate flow between a shape matching bulk flow, and a turbulence layer. The vortex particle method, a method of Lagrangian fluid simulation which represents turbulent motion as a set of spinning particles called vortices, represents each fluid particle much like a rotating paddle wheel. Each vortex particle spins the fluid around a point in space. This is how we implement our turbulence model; we drive the smoke towards the target mesh using a set of control particles which steer towards target particles, the former placed inside

the smoke density, and the latter placed automatically inside the mesh. We compute the bulk flow field from the integrated velocity of a set of control particles; we drive the acting attraction force on the control particles via a correspondence between control and target particles. Our contributions are: a system of fluid target control derived from a separation of bulk flow and detail flow, that accurately matches a target shape; a novel way of determining a driving force from a correspondence between source and target particles, which is simple to implement, fast, and accurate; and a SIMD-optimized extremely efficient implementation of velocity-from-vorticity, which we described in Section 4.2, for use with compact vortex kernels.

Our efficient, and complete, algorithm for targeted shape matching and fluid control has very low simulation times compared to previous methods. However, in the future, we believe exploiting the massive parallelism of consumer GPUs could provide an interesting future direction of research. This could lead to interactive simulation times, even for counts of marker particles above one million. Furthermore, although our particle redistribution technique adequately enforces a rough shape boundary, experimenting with explicit boundary conditions for our vortex simulation could give more definition to small mesh features, such as the ridges of eyebrows, other facial features or smaller details. We would also like to investigate methods of divergence-free directed flow control, to enforce a divergence-free condition on the bulk flow field; this would provide for a fix to some artifacts that appear when the target attraction is strong. Our simple and effective methods of simulating smoke gives us an near-interactive simulation for smoke shape matching. We have attempted with this work to give artists and animators a fast control scheme with a high level of abstraction, when designing visual effects with a fluid simulation system. This is one type of application of fluid control; ultimately there are many possible such applications and this work is a small step towards complete artist control over fluid effects and other natural phenomena.

List of References

[1] M Akmal Butt and P Maragos. Optimum design of chamfer distance transforms. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 7(10):1477–84, January 1998.

[2] Alexis Angelidis and Fabrice Neyret. Simulation of smoke based on vortex filament primitives. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, number July, pages 87–96, New York, New York, USA, 2005. ACM.

[3] Alexis Angelidis, Fabrice Neyret, Karan Singh, and Derek Nowrouzezahrai. A controllable, fast and stable basis for vortex based smoke simulation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 25–32. Eurographics Association, 2006.

[4] J. Andreas Bærentzen and Henrik Aanæs. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005.

[5] J.A. Bærentzen and Henrik Aanæs. Generating signed distance fields from triangle meshes. Technical Report Imm, Technical University of Denmark, Kongens Lyngby, Denmark, 2002.

[6] Markus Becker and Matthias Teschner. Weakly compressible SPH for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217. Eurographics Association, 2007.

[7] Robert Bridson. *Fluid Simulation for Computer Graphics*. AK Peters, 3 edition, 2008.

[8] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: SIGGRAPH 2007 course notes. In *ACM SIGGRAPH 2007 courses*, pages 1–81. ACM, 2007.

[9] Tyson Brochu. Vortex Methods for Smoke Simulation. Technical report, 2005.

[10] B Chang, D Cha, and I Ihm. Computing Local Signed Distance Fields for Large Polygonal Models. *Computer Graphics Forum*, 27(3):799–806, 2008.

[11] Alexandre J. Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer, 1993.

[12] Georges-Henri Cottet and Petros D. Koumoutsakos. *Vortex Methods: Theory and Practice*. Cambridge University Press, 2000.

[13] Akio Doi and Akio Koide. An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells. *IEICE TRANSACTIONS on Information and Systems*, E74-D(1):214–224, January 1991.

[14] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. *IEEE transactions on visualization and computer graphics*, 17(8):1036–47, August 2011.

[15] Kenny Erleben and Henrik Dohlmann. Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra. In H. Nguyen, editor, *GPU Gems 3*, chapter 34, pages 741–763. AddisonWesley, Upper Saddle River, 2008.

[16] Raanan Fattal and Dani Lischinski. Target-driven smoke animation. *ACM Transactions on Graphics*, 23(3):441–448, August 2004.

[17] 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. ACM, 2001.

[18] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pages 23–30, New York, New York, USA, 2001. ACM Press.

[19] Nick Foster and Dimitri Metaxas. Realistic Animation of Liquids. *Graphical Models and Image Processing*, 58(5):471–483, September 1996.

[20] Nick Foster and Dimitris Metaxas. Controlling fluid animation. In *Proceedings of the 1997 Conference on Computer Graphics International*, pages 178–188. IEEE, 1997.

[21] Manuel Noronha Gamito. Two-dimensional simulation of gaseous phenomena using vortex particles. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15. Springer-Verlag, 1995.

[22] William Franklin Gates. *Interactive Flow Field Modeling for the Design and Control of Fluid Motion in Computer Animation*. PhD thesis, The University of California at Davis, 1994.

[23] Michael J Gourlay. Fluid Simulation for Video Games. `http://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1`, 2012.

[24] Simone E. Hieber and Petros D. Koumoutsakos. A Lagrangian particle level set method. *Journal of Computational Physics*, 210(1):342–367, November 2005.

[25] Jeong-mo Hong and Chang-hun Kim. Controlling fluid animation with geometric potential. *Computer Animation and Virtual Worlds*, 15(34):147–157, July 2004.

[26] Jeong-Mo Hong, Jong-Chul Yoon, and Chang-Hun Kim. Divergence-constrained moving least squares for fluid simulation. *Computer Animation and Virtual Worlds*, 19(3-4):469–477, 2008.

[27] M Ikits, J Kniss, A Lefohn, and C Hansen. Volume rendering techniques. In *GPU Gems: Programming techniques, tips, and tricks for real-time graphics*. 2004.

[28] M.W. Jones and Others. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[29] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Transactions on Graphics*, 21(3):339–346, July 2002.

[30] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Transactions on Graphics*, 24(3):561–566, 2005.

[31] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Technical report, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 2006.

[32] Yootai Kim, Raghu Machiraju, and David Thompson. Path-based control of smoke simulations. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 33–42. Eurographics Association, 2006.

[33] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS '04, pages 123–131, New York, New York, USA, 2004. ACM Press.

[34] Toon Lenaerts. *Unified Particle Simulations and Interactions in Computer Animation*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[35] Saithip Limtrakul, Wisut Hantanong, Pizzanu Kanongchaiyos, and Tomoyuki Nishita. Reviews on Physically Based Controllable Fluid Animation. *Engineering Journal*, 14(2):41–52, April 2010.

[36] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, pages 385–392, New York, New York, USA, 2000. ACM Press.

[37] WE Lorensen and HE Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Seminal graphics*, volume 21, pages 347–353. ACM, New York, NY, USA, 1998.

[38] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Transactions on Graphics*, 23(3):449–456, 2004.

[39] Viorel Mihalef, Dimitris Metaxas, and Mark Sussman. Animation and control of breaking waves. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 315–324, New York, New York, USA, 2004. ACM Press.

[40] Richard Elliot Monette. *Particle Based Participating Media Rendering Using Density Octrees*. PhD thesis, Carleton University, 2011.

[41] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. *ACM Transactions on Graphics*, 21(3):721–728, July 2002.

[42] Sang Il Park and Myoung Jun Kim. Vortex fluid for gaseous phenomena. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, number July, pages 261–270, New York, New York, USA, 2005. ACM Press.

[43] Per-Olof Persson. *Mesh generation for implicit geometries*. PhD thesis, Massachusetts Institute Of Technology, 2004.

[44] Tobias Pfaff, Nils Thuerey, Jonathan Cohen, Sarah Tariq, and Markus Gross. Scalable fluid simulation using anisotropic turbulence particles. *ACM Trans. Graph.*, 29(6):174:1–174:8, December 2010.

[45] Tobias Pfaff, Nils Thuerey, and Markus Gross. Lagrangian vortex sheets for animating fluids. *ACM Transactions on Graphics (TOG)*, 31(4):112:1–112:8, 2012.

[46] Tobias Pfaff, Nils Thuerey, Andrew Selle, and Markus Gross. Synthetic turbulence using artificial boundary layers. *ACM Transactions on Graphics*, 28(5):121:1–121:10, December 2009.

[47] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., July 2004.

[48] Nick Rasmussen, Douglas Enright, Duc Nguyen, S. Marino, N. Sumner, Willi Geiger, Samir Hoon, and Ronald Fedkiw. Directable photorealistic liquids. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 193–202, New York, New York, USA, 2004. ACM Press.

[49] Marco Salvi and Jefferson Montgomery. Adaptive Transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 119–126, 2011.

[50] Richard Satherley and M.W. Jones. Hybrid distance field computation. In *Proceedings of the 2001 Eurographics conference on Volume Graphics*, pages 195–209, 2001.

[51] Henning Scharsach. Advanced GPU Raycasting. In *Proceedings of CESCG 2005*, pages 69–76, 2005.

[52] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Transactions on Graphics*, 24(3):910–914, July 2005.

[53] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[54] S. Shankar. *A new mesh-free vortex method*. PhD thesis, Florida State University, 1996.

[55] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524, New York, New York, USA, January 1968. ACM Press.

[56] Lin Shi and Yizhou Yu. Controllable smoke animation with guiding objects. *ACM Transactions on Graphics*, 24(1):140–164, January 2005.

[57] Lin Shi and Yizhou Yu. Taming liquids for rapidly changing targets. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, number July, pages 229–236, New York, New York, USA, 2005. ACM Press.

[58] Yu Shi, Lin; Yizho. Object modeling and animation with smoke. Technical report, University of Illinois at Urbana-Champaign, 2002.

[59] C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, pages 83–90. Ieee, 2003.

[60] Robert Speck. *Generalized Algebraic kernels and multipole expansions for massively parallel Vortex particle methods*. Forschungszentrum Jülich, 2011.

[61] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[62] Sauro Succi. *The Lattice Boltzmann equation for fluid dynamics and beyond*. 2002.

[63] Avneesh Sud, Miguel a. Otaduy, and Dinesh Manocha. DiFi: Fast 3D Distance Field Computation Using Graphics Hardware. *Computer Graphics Forum*, 23(3):557–566, September 2004.

[64] A. Sumleeon and P. Kanongchaiyos. Keyframe Control of Fluid Simulation Using Skeletal Particles. In *The International Conference of Computational Methods*, pages 4–6. Citeseer, 2007.

[65] Matt Swoboda. Advanced Procedural Rendering in DirectX 11, 2011.

[66] Matt Swoboda. numb res. `http://directtovideo.wordpress.com/2011/05/03/numb-res/`, 2011.

[67] MR Syamsuddin and Jinwook Kim. Controllable simulation of particle system. In *Proceedings of the 7th international conference on Advances in visual computing - Volume Part II*, pages 715–724, 2011.

[68] N Thürey, R. Keiser, M. Pauly, and U. Rüde. Detail-preserving fluid control. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 7–12. Eurographics Association, November 2006.

[69] Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. Keyframe control of smoke simulations. *ACM Transactions on Graphics*, 22(3):716—-723, July 2003.

[70] Stephan Trojansky. Raging waters: the rivergod of Narnia. In *ACM SIGGRAPH 2008 Talks*, pages 74:1–74:1, 2008.

[71] Greg Turk and James F. O'Brien. Shape transformation using variational implicit functions. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, pages 335–342, New York, New York, USA, 1999. ACM Press.

[72] Ross Vennell and Rick Beatson. A divergence-free spatial interpolator for large sparse velocity data sets. *Journal of Geophysical Research*, 114(C10):1–14, October 2009.

[73] Ingo Wald and Vlastimil Havran. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69. Ieee, September 2006.

[74] Steffen Weißmann and Ulrich Pinkall. Filament-based smoke with vortex shedding and variational reconnection. *ACM Transactions on Graphics (TOG)*, 29(4):115:1–115:12, 2010.

[75] Mark Wiebe and Ben Houston. The tar monster: Creating a character with fluid simulation. In *ACM SIGGRAPH 2004 Sketches*, page 64, 2004.

[76] Brent Warren Williams. *Fluid surface reconstruction from particles*. PhD thesis, The University of Waterloo, 2008.

[77] CS Yang, SP Kao, FB Lee, and PS Hung. Twelve different interpolation methods: A case study of Surfer 8.0. In *Proceedings of the XXth ISPRS Congress*, pages 778–785, 2004.

[78] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. In *Computer Graphics Forum*, pages 1297–1304, August 2010.

[79] Jong-Chul Yoon, H.R. Kam, J.M. Hong, S.J. Kang, and Chang-hun Kim. Procedural synthesis using vortex particle method for fluid simulation. *Computer Graphics Forum*, 28(7):1853–1859, 2009.

[80] Guijuan Zhang, Dengming Zhu, Xianjie Qiu, and Zhaoqi Wang. Skeleton-based control of fluid animation. *The Visual Computer*, 27(3):199–210, September 2010.

[81] Hongkai Zhao. A fast sweeping method for Eikonal equations. *Mathematics of Computation*, 74(250):603–628, May 2004.