Example-Based Fluid Simulation

by

Ming Chang

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of

Master of Computer Science

in

Ottawa-Carleton Institute for Computer Science

School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
September 2011

© Ming Chang, Ottawa, Canada, 2011

Abstract

We present a novel method for example-based simulation of fluid flow. We reconstruct fluid animation from physically based fluid simulation examples. Our framework shows how to decompose a given series of fluid motion example data into small units and then recompose them. We capture the properties of local fluid behavior by dicing the fluid motion example data into sequences of fragments, which have smaller volume and shorter length. We build a database out of these fragments, and propose a matching strategy to generate new fluid animation. To achieve highly efficient database query, we project our fragments onto lower dimensional subspace using Principal Component Analysis (PCA) approach, and construct our data structure as a kd-tree by treating each fragment as a point in this subspace. Our method has been implemented in synthesizing both two-dimensional (2D) and three-dimensional (3D) fluid's velocity fields.

Acknowledgements

It would not have been possible to finish this thesis without the support and guidance of many people around me. First and foremost, my utmost gratitude and thanks go to my thesis supervisor, Prof. WonSook Lee, her kind concern and sincere advices have led me to accomplish my thesis as well as to undertake my research in the two-year Master's studies. My research colleagues in the lab also deserve my special thanks for their valuable input and unselfish suggestions throughout my researching and thesis writing. In addition, I would like to thank all the professors and staffs I have contacted during the past two years, for the knowledge they have shared and the help they have devoted. I would like to acknowledge the publisher ACM for granting permissions to reuse their figures in this thesis. Last but not least, I am also grateful to my beloved parents and friends who are always there to support me and encourage me in one way or another so I can finally complete my Master's studies.

Table of Contents

Abstract **2**

Acknowledgements **3**

Table of Contents **4**

List of Tables **6**

List of Figures **7**

1 Chapter: Introduction **11**

1.1 Motivation 11

1.2 Problem statement 13

1.3 Proposed solution 14

1.4 System overview and document organization 15

2 Chapter: Literature Review **18**

2.1 Physically based fluid modeling 18

2.1.1 Navier-Stokes equations 18

2.1.2 Fluid modeling in early stage 22

2.1.3 Eulerian method 23

2.1.4 Lagrangian method 27

2.1.5 Comparisons of two methods and hybrid methods 32

2.1.6 Model reduction 35

2.2 Data-driven motion synthesis 40

2.3 PCA based pattern matching 43

3 Chapter: Fluid Example Database Construction **46**

3.1 Grid based fluid simulation setup 46

3.2 Example data collection 48

3.2.1 Fluid solver 48
3.3 Segmentation process 52
4 Chapter: Fluid Generation 55
4.1 Initial state 56
4.2 Processing order 58
4.3 Fragment pattern matching schemes 62
4.3.1 Full matching 62
4.3.2 Partial matching 64
4.4 Boundary condition 67
5 Chapter: Database Query Acceleration 69
5.1 Dimension reduction using PCA 69
5.2 Nearest neighbor search using kd-tree 74
5.3 Results 87
6 Chapter: Conclusions 95
6.1 Contribution 96
6.2 Discussion 97
6.3 Future Research 102
References 107

List of Tables

Table 1 Computational times for preprocessing and fluid generation with different settings. The “Reduced data dimensions” column shows the number of dimensions we reduced to when using the PCA approach. All times are in seconds. The time before slash shows the time for preprocessing, while the one behind slash shows the time for fluid generation..... 92

List of Figures

Figure 1 High level transaction flow diagram of the system 16

Figure 2 Two-dimensional and three-dimensional MAC grid (reprinted from [17]). 24

Figure 3 The velocity field is defined at the center of each cell (reprinted from [2]). 47

Figure 4 Four steps need to be iterated over each time span (reprinted from [2]). 50

Figure 5 Fluid example database construction. One fragment from prior frame, one from successive frame. Put all the pairs into a sequential data structure for fast and random access. 53

Figure 6 Fluid generation. For each fragment of the current frame, we look for a most similar one in the first fragments of the pairs in database, and use the second fragment in that pair as the next frame's corresponding fragment. 55

Figure 7 The left side is the initial state, a frame within which only one single cell has nonzero-velocity vector; imagine a situation where an external force acts on the cell to a direction. The right side is its corresponding divergence-free state calculated by the fluid solver from the initial state. Here velocity vector is indicated by line segment. 57

Figure 8 Multivalued function problem in the initial frame and its successive frame. Two adjacent fragments showing this problem are highlighted with red rectangles. Line segment indicates the velocity vector. 58

Figure 9 Sampled substeps from the initial state to the next frame. The serial number is displayed in the upper right corner of each image. Line segment corresponds to the velocity vector, and is colored red when being updated. Grid cells are highlighted by different colors or patterns for different meanings. Yellow interior: fragment selected to

be processed; Green interior: center cell of the fragment selected to be processed; Brown boundary: cells being updated; Blue boundary: cells updated..... 61

Figure 10 A 2-dimensional kd-tree example: on the left the way the plane is subdivided, and on the right the corresponding binary tree. 76

Figure 11 Validation of the synthesized fluid flow in 2D. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 100^{th}, and 600^{th} frames. The initial states of both sides were set identical. 88

Figure 12 Another validation of the synthesized fluid flow in 2D with a different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 100^{th}, and 600^{th} frames. The initial states of both sides were set identical. 89

Figure 13 Validation of the synthesized fluid flow in 3D. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 30^{th}, and 100^{th} frames. The initial states of both sides were set identical. 90

Figure 14 Another validation of the synthesized fluid flow in 3D with a different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 30^{th}, and 100^{th} frames. The initial states of both sides were set identical. 91

Figure 15 Quality degradation in 2D system when fragment data is reduced by PCA. On the left the frames of velocity field were generated with fragment data reduced to 1 dimension; on the right the frames were synthesized with fragment data reduced to 2 dimensions. The top two frames were generated from the initial state shown in Figure 11; the bottom two frames share the initial state shown in Figure 12. All of them are the 10^{th} frames. 93

Figure 16 Quality degradation in 3D system when fragment data is reduced by PCA. On the left the frames of velocity field were generated with fragment data reduced to 2 dimensions; on the right the frames were synthesized with fragment data reduced to 3 dimensions. The top two frames were generated from the initial state shown in Figure 13; the bottom two frames share the initial state shown in Figure 14. All of them are the 100^{th} frames. 94

Figure 17 Errors occurred with different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 50^{th}, and 100^{th} frames. The initial states of both sides were set identical, but they are different from the one we applied to collect the example data, shown in Figure 7, with the nonzero-velocity cell translated along *x*-axis and its magnitude reduced slightly. 99

Figure 18 Errors become smaller when more fluid motion example cases are included into database. On the left the frames were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 50^{th}, and 100^{th} frames. The initial states of both sides were set identical to the one

in Figure 17. The newly included fluid example has a more similar initial state, with the nonzero-velocity cell only slightly translated. 100

Figure 19 Frames generated with different dimension reduction ratios. All frames were generated from the same initial state and database as in Figure 17, and they are all the 100^{th} frames. The number displayed in upper right corner of the frame indicates the dimensions PCA reduced to. For comparison, the frame generated without dimension reduction is provided at the lower right spot, identical to the bottom left image in Figure 17. 101

Figure 20 A limitation of the current method when the initial state has multiple nonzero-velocity cells (forces) of the same value. On the left the frames of velocity field were synthesized by the current method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states and 2^{nd} frames. The initial states of both sides were set identical, and there are two nonzero-velocity cells with the same vector value in the initial state. 104

Figure 21 Sampled substeps from the initial state to the next frame for the example shown in Figure 19. The serial number is displayed in the upper right corner of each image. The grid cells and the line segments (velocity vectors) are highlighted by different colors or patterns under the same convention as in Figure 9. 105

1 Chapter: Introduction

1.1 Motivation

Animating fluid such as water, fire, and smoke is increasingly important in movies, video games, and virtual environment. Meanwhile, in computer graphics (CG) fluid simulation remains one of the most challenging problems, due to its computational complexity and the complicated fluid behavior.

The complexity of fluid behavior lies in the complex interplay of various phenomena such as convection, diffusion, turbulence and surface tension [7]. For these phenomena, it is impossible for artists to animate frame by frame, and it is also hard for people to find a simple model that could depict their behaviors. Therefore, physically based methods have been mainly used to generate realistic fluid animation.

Although the famous Navier-Stokes equations, a set of partial differential equations that govern fluid flow and hold throughout the fluid, are proposed more than one hundred years ago, the general closed form solutions remain undiscovered [16]. With the development of computer technology, a new research subject as a branch of fluid mechanics, Computational Fluid Dynamics (CFD), was established to approximate the Navier-Stokes solutions using numerical methods and algorithms. Most scientists in fluid simulation use fluid mechanics as a standard mathematical framework. In computer graphics society, Foster and Metaxas [8] began the current thrust in fluid simulation by discretizing the Navier-Stokes equations and using a finite difference method to solve them. Since then computer graphics researchers have been focused on techniques for

solving these equations to obtain physically accurate simulations. Several techniques have been developed to provide accurate and realistic simulations of real-world fluid phenomena, however, at a great computational cost.

In computer graphics, although many ideas and algorithms of fluid animation are derived from CFD literature, fluid animation has a different purpose compared with CFD. CFD mainly focuses on predicting the actual fluid flow through accurate simulations, while in computer graphics the primary purpose is to obtain visually plausible effects, and physical accuracy is secondary or irrelevant in some cases. In fact, many approximations of fluid motion have been developed, such as stable fluid [2], random turbulence [6], height field approximations for liquid surfaces [5], and particle-based simulations [7, 15].

On the other hand, in other areas in computer graphics, example based approach has been popular, to name a few, modeling human [9, 10], surface deformation [12] and texture synthesis [11]. Widely used marker-based optical motion capture system is an example to achieve natural looking human movement without going through physically based approach. The underlined bone and muscle structure are ignored and only the outside skin motion has been captured and the results show more natural animation. One drawback of this method is that raw motion capture data is difficult to re-use, and editing techniques are reliable only for small changes to a motion. In order to overcome these deficiencies, some kinds of motion synthesis methods have been presented. Quite a number of these motion synthesis methods build their data structure as a motion graph [3, 4] based on existing motion capture data, convert the problem into a graph search, and finally generate new animation from the existing data.

Inspired by the example based modeling and animation, specially the motion synthesis idea, considering the specialty of fluid motion from computer graphics' point of view, we present a novel example based method for fluid simulation.

1.2 Problem statement

From the observation of how fluid behaves in real life, we know that, in a grid-based setup, the evolution of a given cell's velocity is related to the neighboring cells in the fluid space. We question that if we have a database with enough information of the cell's velocity transition related to the surrounding cells, are we able to retrieve the velocity of the cell for the next frame without calculating the heavy Navier-Stokes equation?

This thesis addresses the problem of developing a system that can synthesize fluid flow given only the initial fluid frame, by first decomposing a series of fluid motion example data into small units and then recomposing them. In order to solve this problem, we focus on three main issues:

- Given the fluid motion example data, to reuse the example data and synthesize the fluid behavior similar to the original fluid motion, how to organize and build a database out of the input data.
- According to that database built in the previous step, given the initial fluid frame, how to generate the following fluid flow that looks similar to the input fluid motion example.
- Considering the specialties of fluid motion, the size of the database we built would be relatively large. Consequently, the new fluid flow's generation, which depends on querying the database, would be considerably expensive.

How to accelerate our database query procedure to make the whole system work efficiently.

1.3 Proposed solution

In this thesis, we propose a solution to the problem stated in the previous section. Our solution can be summarized as below:

- In the preprocessing stage, we collect the fluid motion example data, namely sequence of velocity field, calculated by a fluid solver. In order to capture the properties of local fluid behavior, we divide the collected fluid simulation example data into fragment pairs, which are actually simulations of smaller size and shorter length, with the first fragment standing for previous status and the second fragment standing for the successive status. Then we store the fragment pairs sequentially in our database so that every element can be accessed by its index number in constant running time.
- After the database is constructed, given an initial or prior fluid frame of the velocity field, we can generate the following frame, fragment by fragment, according to the database. To make this happen, we overcome the multivalued function problem by choosing a proper fragment processing order, and develop two different pattern matching schemes, “full matching” and “partial matching”, to handle different kinds of situations when locating a query fragment pair among existing elements in the database for the following frame’s information. Then we repeat this procedure until the desired number of frames is achieved.

- As “full matching” is needed for most of the time when querying the database and it consumes dominant amount of computational time, we make use of two techniques, PCA and kd-tree, to accelerate this procedure. As “full matching” concerns fixed-dimension data comparison (as opposed to the dynamic-dimension data comparison for “partial matching”), we first utilize PCA approach to project the elements of our database onto a lower dimensional subspace. Then we build a kd-tree of this lower dimensionality, and treat element’s corresponding lower dimensional data as its position, and insert the database index number of each element into the kd-tree at its corresponding position. So when we need to do “full matching” of a fragment in the database, instead of iterating and comparing every elements in the database, we can just project this fragment onto this same subspace, and find the target element’s database index number from the kd-tree.

1.4 System overview and document organization

A high level transaction flow diagram of our system is shown in Figure 1. As it displays, our system contains two main stages, namely preprocessing stage and fluid generation stage. In the preprocessing stage, our database is constructed from the collected fluid motion example data by segmenting the data into fragment pairs, and then with the PCA approach performed on these fragment pairs, the kd-tree data structure can be built for the sake of accelerating database query. In the fluid generation stage, after the initial state is specified, the database and the acceleration data structure kd-tree are used for pattern matching, which generates the new fluid frame as its result, and then this procedure is

repeated until the desired number of frames is achieved. Detailed description of these stages and their explanations occur in the following chapters.

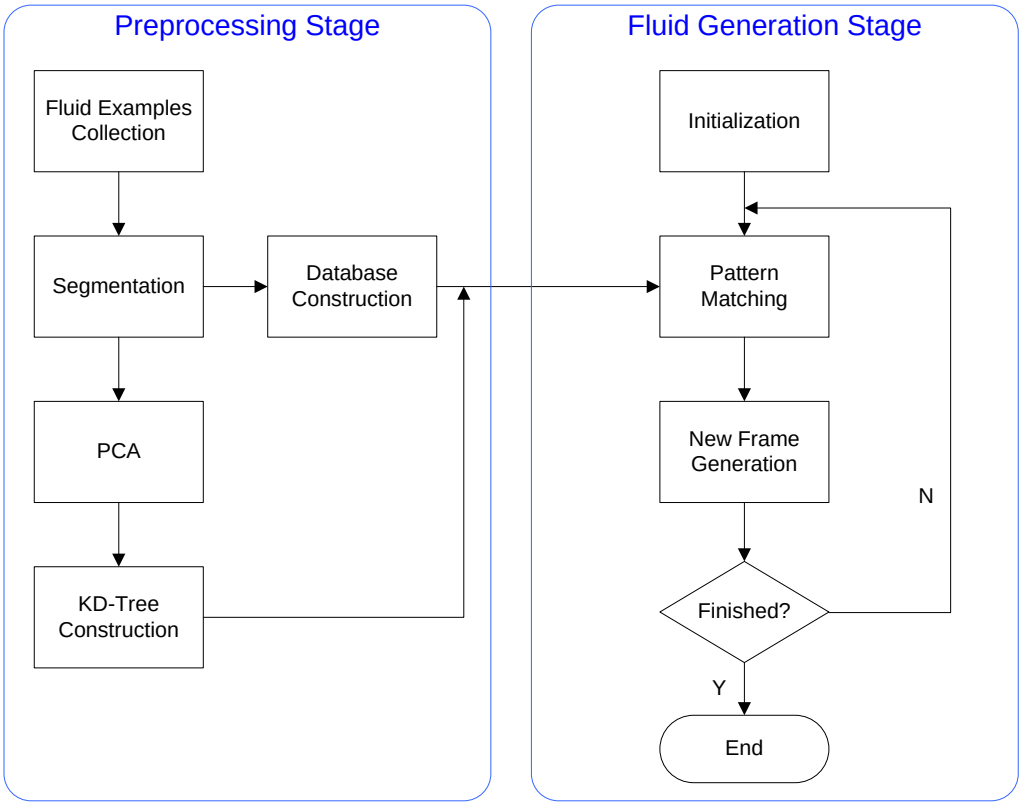

Figure 1 **High level transaction flow diagram of the system**

The remainder of this thesis is organized as follows. In Chapter 2, we provide a literature review on fluid modeling as well as some other topics related to our work. In Chapter 3 we introduce the database construction procedure in the preprocessing stage, which contains the description of grid-based fluid simulation setup, example data collection and the segmentation process. In Chapter 4, we present the fluid generation stage, including the fragment processing order and different pattern matching schemes. We explain how we accelerate our database query, namely the use of PCA and kd-tree

data structure, and show our results in Chapter 5. At the end, we conclude in Chapter 6 with a discussion of the scalability of our approach to general condition and potential future research.

2 Chapter: Literature Review

In this chapter, we describe the literature that our solution involves, including physically based fluid modeling, motion synthesis and PCA based pattern matching method.

2.1 Physically based fluid modeling

In this section, we present a literature review on the development of physically based fluid modeling for computer animation. First, we introduce the well known Navier-Stokes equations, which are the essential model from which most fluid simulation methods are derived. Then we discuss the classification of the methods employed in this field, namely the Eulerian method and the Lagrangian method. At last, we describe a special research branch, model reduction, which has a few attributes related to our solution.

2.1.1 Navier-Stokes equations

Computational Fluid Dynamics (CFD) has a relatively long history. In the early 19th century, Claude Navier and George Stokes formulated the famous Navier-Stokes Equations that describe the dynamics of fluids [7]. Besides the Navier-Stokes equation which describes conservation of momentum, fluid flow is also governed by another two fundamental equations of fluid dynamics, continuity and energy equations. The continuity and energy equations describe mass and energy conservation respectively. The

three equations together mathematically state the fundamental physical principles governing the fluid motion.

Since fluids' volume usually does not change very much and the change ordinarily has so small effect on fluids' movement at a macroscopic level that it is practically irrelevant for animation, most fluid flow of interest in animation can be treated incompressible [17]. For the case of incompressible fluids, the flow can be described by the momentum conservation and the mass conservation equation without solving the energy equation. Therefore, the equations are usually written as:

$$\frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla \vec{u} + \vec{g} \qquad (2.1\text{ a})$$

$$\nabla \cdot \vec{u} = 0 \qquad (2.1\text{ b})$$

In Equation 2.1, the symbols are defined in the following:

- $\vec{u}$ and $\vec{g}$ are vector quantities. $\vec{u}$ corresponds to the fluid velocity, and $\vec{g}$ is the acceleration due to the external force, such as gravity.
- p and t stand for the pressure and time, respectively.
- ν ($\geqslant 0$) is called kinematic viscosity coefficient. It measures how viscous the fluid is, how much the fluid resists deforming while it flows. Fluids such as molasses have high viscosity, and fluids like alcohol have low viscosity.
- ρ corresponds to the density of the fluid.
- $\cdot$ denotes a dot product between vectors.

- ∇ and $\nabla \cdot$ are the gradient and divergence operators, respectively. $\nabla \cdot \nabla$, the divergence of the gradient, is the Laplacian operator, sometimes also written as ∇^2 or Δ. These differential operators are fundamental to vector calculus. Sometimes it can be helpful to think of ∇ as a symbolic vector, e.g., in three dimensions:

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \tag{2.2}$$

Both equations in Equation 2.1 are three (two, if in two dimensions) in one wrapped up as vector equations. They are called incompressible Navier-Stokes equations. The first differential equation 2.1a is named the Navier-Stokes equation. It becomes the Euler equation when viscosity *v* equals to zero, and such an ideal fluid with no viscosity is called "inviscid". The equation is also called "momentum equation", because it is actually derived from the Newton's Second Law of Motion:

$$\vec{F} = m\vec{a} \tag{2.3}$$

"A body of mass *m* subject to a net force $\vec{F}$ undergoes an acceleration $\vec{a}$ that has the same direction as the force and a magnitude that is directly proportional to the force and inversely proportional to the mass." from the Wikipedia article "Newton's laws of motion". Momentum equation tells us how the fluid accelerates due to the forces acting on it.

The mass conservation equation 2.1b is named the "incompressibility condition". It is often called the continuity equation as well. This equation is derived from the obvious theory that the volume of an incompressible fluid stays constant. For apparent reasons a vector-field that satisfies the incompressibility condition is called "divergence-free". More detailed derivation of Equation 2.1 can be found in any textbook about fluid dynamics, such as Anderson's work [18]. Readers interested in mathematics may wish to consult the book by Chorin and Marsden [19].

Actually, Equation 2.1 is the velocity-pressure formulation of the Navier-Stokes equations since the dependent variables of the equations are velocity and pressure. The initial condition and boundary conditions must be provided to define a flow. The term $\vec{u} \cdot \nabla \vec{u}$ in Equation 2.1a represents "advection" (sometimes "convection" or "transport"), meaning each velocity component is carried by the velocity field. The term $\nu \nabla \cdot \nabla \vec{u}$ represents "diffusion", which is the force due to viscosity, intuitively a force that tries to minimize differences in velocity between nearby bits of fluid.

The Navier-Stokes equations are the partial differential equations (PDE) that are extremely difficult to solve for the following reasons:

- The equations are nonlinear due to the advection term, which actually involves an acceleration associated with the change in velocity over position.
- Inside the vector equations, the associated scalar equations are intricately coupled to each other.
- The dependent variable pressure field is implicitly constrained by the continuity equation 2.1b. (divergence-gradient coupling)

In fact, solving the equations is so difficult that it still remains as one of the Clay institute's "Millennium Problems". "Although these equations were written down in the 19th Century, our understanding of them remains minimal. The challenge is to make substantial progress toward a mathematical theory which will unlock the secrets hidden in the Navier-Stokes equations." from Clay Mathematics Institute website.

2.1.2 Fluid modeling in early stage

In early stage, due to the limited computational ability, fluid animation was mainly based on parametric modeling method [21]. Peachey [20] presented a model suitable for the animation of waves approaching and breaking on a sloping beach. Waveforms were made consist of a numerically integrated phase function and a wave profile. The phase function is used to simulate the wave refraction effects and the change of speed and wavelength in shallow water. The wave profile changes according to wave steepness and water depth. Particle systems were used to model the spray produced by wave breaking and collisions with obstacles. In order to overcome the issue that basic height field could not simulate the effect of overhanging waves, Fournier and Reeves [22] used Gerstner model where particles on water surface go through a circular or elliptical motion as a wave passes by, and modeled ocean surface as a parametric surface, using depth of ocean floor and slope of beach to control sine function wave profiles. In both of these two methods, since the particles used for modeling water only move nearby their initial positions, neither of them could produce the real effect of fluid flow, or deal with the influence of boundaries on water surface. Tessendorf [23] synthesized a patch of ocean waves from a Fast Fourier Transform (FFT) prescription. The patch can be tiled over a larger domain, and was made

up of many sine waves whose parameters come from empirically based models of the ocean. This method can depict the ocean surface well under fair-weather conditions.

However, when using these models, people feel difficult to control, and it is impossible to simulate some complicated effects with more luxuriant details. Thus, many researchers turned to physically based methods. Physically based fluid modeling methods can be basically divided into two camps: Eulerian method and Lagrangian method.

2.1.3 Eulerian method

The name, Eulerian, is named after the Swiss mathematician Euler. The Eulerian viewpoint of the fluid field is a way of looking at fluid motion that focuses on fixed locations in the fluid space through which the fluid flows as time passes. In the Eulerian specification of the fluid field, the fluid quantities, such as velocity, pressure, density, temperature, etc., are depicted as a function of fixed position $\vec{x}$ and time *t*. For instance, the fluid velocity can be described as a vector field $\vec{u}(\vec{x},t)$, and the pressure inside the fluid can be described as a scalar field $p(\vec{x},t)$. Thus, the whole region of fluid is modeled as fields of fluid quantities. To visualize the Eulerian viewpoint, imagine when you are doing a weather report, you are stuck on the ground, measuring the pressure, temperature and humidity etc. of the air that is flowing past.

In the Eulerian approach, Navier-Stokes equations are usually discretized using a fixed grid, which does not change in space even if the fluid flows through it. Then the changes of fluid quantities are computed at each of the fixed nodes in the grid, thereby, the whole fields of fluid quantities are obtained.

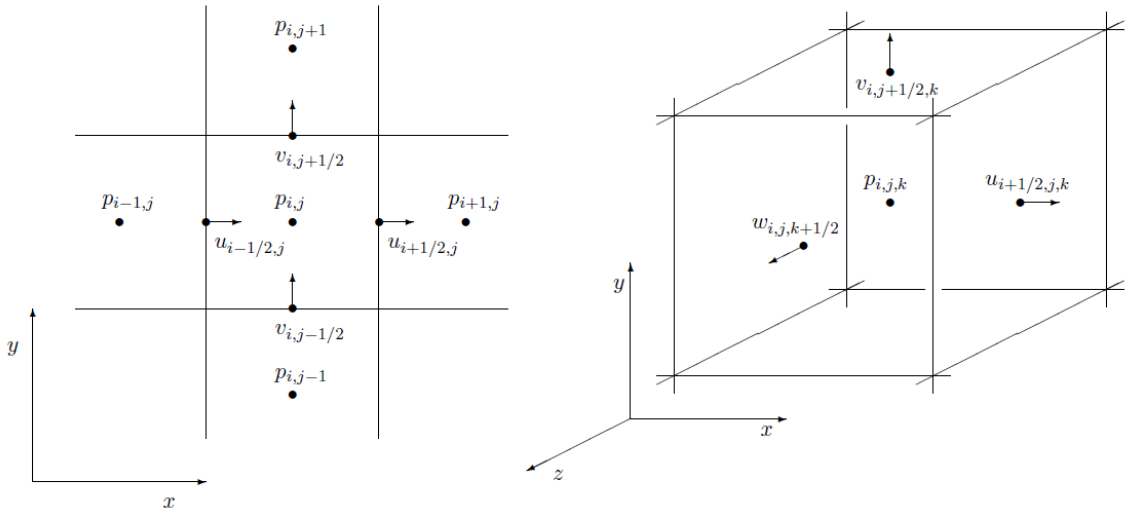

Figure 2 Two-dimensional and three-dimensional MAC grid (reprinted from [17]).

Usually, there are two different ways to make use of the grid to do the discretization. One is called Marker and Cell (MAC) grid, first introduced by Harlow and Welch [24]. The MAC grid is a staggered grid, which means different variables are stored at different locations. See Figure 2. In two dimensions, for example, in grid cell *(i, j)*, the pressure is sampled at the center, indicated by $p_{i,j}$. The velocity is divided into two Cartesian components, *u* and *v*. At the centers of the vertical cell faces, the horizontal *u* component is sampled, while the vertical *v* component is sampled at the centers of the horizontal cell faces. For instance, the horizontal velocity component between cells *(i, j)* and *(i+1, j)* is indicated by $u_{i+1/2,j}$, the vertical velocity component between cells *(i, j)* and *(i, j +1)* is indicated by $v_{i,j+1/2}$ in the figure. Therefore, for grid cell *(i, j)*, the normal component of the velocity at the center of each of its faces is sampled. This fact helps to estimate the amount of fluid flowing into and out of the cell, which makes the conservation condition easy to deal with. In three dimensions, the MAC grid is set up similarly. Scalar quantities, such as pressures, are sampled at the center of each grid cell.

Vector quantities, such as velocities, are split into three components, with normal component sampled at the center of each cell face. Through MAC grid, unbiased and second order accurate central difference can be achieved.

The other way to utilize the grid structure is that different variables are stored at the same locations. For instance, in Stam's work [2], all kinds of values of the discretized fields, both scalar quantities and vector quantities, are defined at the center of the grid cells. Simplicity is the advantage of this treatment. Since interpolations are significantly reduced, it is more straightforward to implement. Different variables could be handled consistently.

In the early stages of physically based fluid animation, in lieu of simulating the sophisticated Navier-Stokes equations of fluid flow directly, several assumptions were made for the sake of simplification. In order to solve the height field which could approximate the water surface, Kass and Miller [5] simplified the shallow water equations [26, 27] into a version of wave equation. Then with the use of an implicit integration scheme, they achieved a rapid and stable solution by solving a simple tridiagonal linear system. Similarly using height field to represent the water surface, to approach even closer to the physical law of nature, Chen et al. [28] solved the 2D Navier-Stokes equations discretized on a MAC grid to obtain the velocity field and pressure field for the 2D fluid surface, and then used the Bernoulli equation and the pressure of 2D fluid surface to calculate the height field. They also simulated the effects of movable object floating in the fluid and changeable boundaries. The approximation, using height field as fluid surface, reduces the whole simulation into 2D, and avoids the cost of

calculation in 3D, but many interesting fluid phenomena, such as overhanging waves, sprays and splashes, could not be captured satisfyingly.

Truly making use of the full three-dimensional Navier-Stokes equations to animate fluid behavior started from Foster and Metaxas [8]. Discretized on a MAC grid, the incompressible Navier-Stokes equations were solved with an explicit finite difference approximation. But because of the CFL condition (Courant–Friedrichs–Lewy Condition) [29], their solution is stable only when the time step is sufficiently small. Years later, Stam [2] achieved an unconditionally stable model, by introducing the semi-Lagrangian method for the advection term and applying the implicit methods for the diffusion and pressure terms. From then on, using Navier-Stokes equations to simulate truly 3D fluid has become increasingly popularized.

After that, Osher and Fedkiw [30] proposed the level set method, an Eulerian approach, which is proved to be suitable to represent liquid surfaces. This approach introduced another scalar field, $\phi(\vec{x})$. $\phi(\vec{x})$ is defined by the signed perpendicular distance from position $\vec{x}$ to the liquid surface. Therefore, the fluid surface could be generated from the implicit function $\phi(\vec{x}) = 0$. The sign of $\phi(\vec{x})$ is set negative inside the liquid and positive outside. Furthermore, $\phi(\vec{x})$ is convected by the velocity field:

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0 \tag{2.4}$$

Thus, $\phi(\vec{x})$ can be evolved as well. Many stunning results were produced by combining the Eulerian method with the level set based surface tracking algorithms. Since the level

set evolution (Equation 2.4) suffers from server volume loss, Foster and Fedkiw [31] coupled the level set with particles to alleviate this problem. Enright et al. [32] further improved this method with a "thickened" front tracking approach, called particle level set method (PLS). They placed particles at both sides of the liquid interface. With particles the level set values are revised for each frame such that the total volume of fluid and the detailed surface features are preserved. Mihalef et al. [33] proposed a marker level set method (MLS), which seeds the marker particles only along the interface, yielding more efficient and accurate results. Moreover, the surface markers allow non-diffusively surface texture advection, which is a rare capability in the realm of Eulerian method.

Recently, Chentanez and Müller [72] presented a new Eulerian method, which allows real-time fluid simulations of large scale. To accelerate the simulation, they used a hybrid grid representation composed of regular cubic cells on top of a layer of tall cells. This layout allows representing water above an arbitrary terrain without consuming an excessive amount of memory and compute power, while focusing resources on the area near the surface. They introduced a fast, parallel, multigrid Poisson solver for the tall cell grid. To reduce the computational time even further, they also modified the level set method and velocity advection scheme to allow both larger time steps and an efficient GPU implementation.

2.1.4 Lagrangian method

The name, Lagrangian, is named after the famous French mathematician Lagrange. The Lagrangian viewpoint of the flow field is a way of looking at fluid motion where the

observer follows an individual fluid particle as it moves through space and time. Plotting the position of an individual particle through time gives the path line of this particle. In the Lagrangian specification, fluid is treated as a particles system, and all fluid particles are labeled by some quantity that is time-independent for each fluid particle. Often, that quantity is chosen to be the positions of the particles at some initial time t_0. Now, for the sake of clarity, we label each particle by a number, and the quantities of a certain particle can be expressed with that label, such as $\vec{u}_i$ is the velocity of particle *i*. To visualize the Lagrangian viewpoint, imagine when you are doing a weather report, you are sitting in a balloon floating along with the wind, measuring the pressure, temperature and humidity etc. of the air that is flowing alongside you.

Instead of a stationary grid, in the Lagrangian approach, particles are used, and the incompressible Navier-Stokes equations become:

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + v\nabla\cdot\nabla\vec{u} + \vec{g} \tag{2.5}$$

Compared with Equation 2.1, the equation above is substantially simplified. First, because the amount of particles is constant and each particle has a changeless mass, mass conservation is inherently guaranteed and Equation 2.1b is omitted entirely. Second, two terms $\partial\vec{u}/\partial t$ and $\vec{u}\cdot\nabla\vec{u}$ in Equation 2.1a are replaced by the material derivative (also known as substantial derivative) $D\vec{u}/Dt$. Since the material derivative is a derivative taken along a path moving with the velocity and the particle moves with the fluid, the material derivative of the velocity field is simply the acceleration of the particle [7]. That

is how fast $\vec{u}$ is changing for the particle. The first term, $\partial \vec{u}/\partial t$, only indicates how fast $\vec{u}$ is changing at the fixed points in space. The second term, $\vec{u} \cdot \nabla \vec{u}$, is correcting for how much of the change is due to differences in the fluid moving past [17].

It is worth noting that on the right hand side of Equation 2.5 the three terms, $-\nabla p/\rho$, $\nu \nabla \cdot \nabla \vec{u}$ and $\vec{g}$ are related to three forces, pressure, viscosity and external force, respectively. If we sum these forces together into a net force, the equation will degenerate into a modified form of the Newton's Second Law of Motion:

$$\vec{a}_i = \frac{d\vec{u}_i}{dt} = \frac{\vec{f}_i}{\rho_i} \tag{2.6}$$

where $\vec{a}_i$ and $\vec{u}_i$ are the acceleration and velocity of particle *i*, $\vec{f}_i$ and ρ_i are the force density field and the density field evaluated at the location of particle *i*, respectively. Thus, essentially the Lagrangian approach is about analyzing the forces acting upon separate particles and with the use of integration methods calculating the positions and other quantities of these particles for the next time step, thereby evolving the status of the whole fluid.

Reeves [34] introduced the particle system as a method for modeling fuzzy objects such as fire, cloud and water. Because of particle system's simplicity and flexibility, it was used to simulate fluid flow [35]. Also, for the effect of splash [36, 49], foam [37] and bubble [50, 51], it is easy to think of using particle system to simulate.

In the Lagrangian approach, fluid is discretized irregularly by the particle system, and quantities such as velocity, mass and position, are carried by the moving particles. In

order to tackle Equation 2.5 and calculate the spatial derivatives in it, such as gradient operator ∇ and Laplacian operator $\nabla \cdot \nabla$, field quantities of the fluid space need to be well defined under such an irregular discretization. Monaghan [38] developed Smoothed Particle Hydrodynamics (SPH) to address this issue for the simulation of astrophysical problems, but the method is general enough to be used in any kind of fluid simulation. SPH is an interpolation method for particle systems. With the help of smoothing kernel functions, SPH interpolates from the neighboring particles, such that field quantities only defined at discrete particle locations can be evaluated at arbitrary positions in space. According to SPH, a scalar quantity A at location $\vec{r}$ is assessed as [7]:

$$A(\vec{r}) = \sum_i m_i \frac{A_i}{\rho_i} W(\vec{r} - \vec{r}_i, h) \tag{2.7}$$

where i iterates over all particles, m_i corresponds to the mass of particle i, ρ_i is its density, $\vec{r}_i$ the position and A_i the field quantity at $\vec{r}_i$. Typically, the smoothing kernel $W(\vec{r}, h)$ is a normalized (i.e. $\int W(\vec{r}) d\vec{r} = 1$) and radial symmetric function with finite support, with h as the radius of support. The mass m_i is constant, while density ρ_i evolves. ρ_i can be evaluated at every time step by the Equation 2.7 itself with the quantity A replaced by the density ρ and then reduced by the same denominator. It is worth noting that with the SPH approach, spatial derivatives of field quantities only affect the smoothing kernel. The gradient and Laplacian of quantity A are simply:

$$\nabla A(\vec{r}) = \sum_i m_i \frac{A_i}{\rho_i} \nabla W(\vec{r} - \vec{r}_i, h) \tag{2.8a}$$

$$\nabla \cdot \nabla A(\vec{r}) = \sum_i m_i \frac{A_i}{\rho_i} \nabla \cdot \nabla W(\vec{r} - \vec{r}_i, h) \tag{2.8b}$$

SPH was first introduced to the graphics community by Stam and Fiume [39] to depict fire and other gaseous phenomena. In [7, 40], the SPH method was used to simulate fluid with free surfaces. They modeled surface tension forces, and applied point splatting technique or marching cubes algorithm to render the surfaces.

The SPH method is flexible, but it can only solve compressible fluid flow. Later, another gridless particle method called the Moving-Particle Semi-Implicit (MPS) was developed for the incompressible fluids [13]. In the MPS method, the motion of each particle is calculated through interactions with neighboring particles. Premože et al. [15] adopted the MPS and MPS-MAFL (meshless advection using flow-directional local grid) methods to simulate mixing fluids with different physical properties, with the Navier-Stokes equations discretized using moving particles and their interactions.

The Lattice Boltzmann method (LBM) was introduced into the computer graphics community by Li et al. [42]. LBM is another kind of Lagrangian approach. Built on a mesoscopic picture, LBM models the fluid consisting of a group of particles that propagate and collide with each other over a discrete lattice mesh. Instead of tracking individual particles, LBM solves for the particle distribution functions, which are then transformed into the macroscopic quantities such as velocity and pressure. It is proved that in the incompressible limit this simplified propagation–collision dynamics can lead

to the incompressible Navier–Stokes equations [43, 44]. Because of its particulate nature and local dynamics (collisions in the lattice only requires accessing neighbors), LBM has some advantages over other conventional methods, including convenience of implementation, efficiency and parallel scalability [45], and capability of dealing with complex boundaries [46, 47, 48].

More recently, Solenthaler and Gross [73] proposed a two-scale method for particle-based fluid simulation. The method uses a low- and a high-resolution simulation that run simultaneously. While in the coarse simulation the fluid is represented by large particles, in the high-resolution simulation small particles are used to capture complex flows and fine-scaled surface details. The coupling of the different simulation levels is done by including feedback forces and defining appropriate boundary conditions. They also showed that this model could be easily integrated into a standard SPH simulation to improve the computational efficiency. By allocating computational resources to regions where a higher resolution is desirable, this two-scale method preserves similar surface details compared to the single-resolution simulation, while improving the efficiency linearly to the achieved reduction rate of the particle number.

2.1.5 Comparisons of two methods and hybrid methods

Both the Lagrangian approach and Eulerian approach have merits and demerits. For the Lagrangian approach, the concept and implementation are straightforward, mass conservation is guaranteed inherently, without computing throughout the whole space grid the computational overhead is reduced, the detailed fluid surface features are well preserved, and user control can be added easily. But the Lagrangian approach has

difficulties when volume conservation needs to be strictly guaranteed. Furthermore, there is no straightforward way to construct the actual fluid surface for rendering. The changing topology of the free surface requires complicated algorithms to extract a smooth polygonal (or parametric) description, and the computational cost often expands with the particle amount. The Eulerian approach, in contrast, has the advantages of smooth fluid surfaces and large time steps, but suffers from expensive computational cost for global pressure correction, aliasing boundary discretization, and poor scalability.

Therefore, many researchers integrated the grid based Eulerian approach with the particle based Lagrangian approach together to extract their advantages and overcome the weaknesses of both. For instance, Stam [2] used both Lagrangian and implicit Eulerian methods to solve the Navier-Stokes equations, which was later named as the Semi-Lagrangian scheme (SLS) and widely used in fluid animation. Lentine et al. [71] modified this semi-Lagrangian method to make it fully conserve momentum. Selle et al. [52] seeded the Lagrangian vorticity carrying particles into the traditional Eulerian grid based fluid to create highly turbulent effects in smoke, water and explosion simulations. Kang et al. [53] presented a hybrid method for simulating chemically reactive fluids, such as catalysis, erosion, fire, and explosions. They employed the Lagrangian particles to bring about a chemical reaction that affected the overall flow, and used the Eulerian method to model the fluid to be visualized in the rendering stage. Because, with a limited grid resolution, grid based methods have difficulties for simulating small-scale phenomena such as spray and foam, Kim et al. [49] created turbulent splashing water by extending the Eulerian particle level set method (PLS) with an advanced particle simulation system. The massless marker particles that escape from the main body of

water, despite the level set correction, are transformed into water particles to produce subcell-level detail. They estimated the volume loss quantitatively and distributed it to the water particles. Losasso et al. [54] presented a two-way coupled simulation framework. They used the Eulerian particle level set method (PLS) as well to model dense fluid volumes, and simulated diffuse regions, such as sprays, with a smoothed particle hydrodynamics method (SPH). Hong et al. [51] proposed a hybrid approach for simulating bubbly water. To obtain subgrid visual details, they incorporated a new bubble model based on the Lagrangian SPH into an Eulerian grid based simulation that handles background flows of large bodies of water and air. Thürey et al. [59] presented a multiscale approach to simulate flows driven by surface tension based on triangle meshes. Their approach consists of two simulation layers. One layer is an Eulerian method that simulates surface tension forces, and the other layer is a Lagrangian finite element method for simulating subgrid scale wave details on the fluid surface.

Very recently, Nugjgar et al. [14] presented another hybrid model to create an endless animation of water stream. They introduced a Markov-type velocity field (MTVF), which is composed of transition probabilities of the velocity vectors between the sectors of the underlying grid node. Like our system, their method also consists two main phases. In the preprocessing stage, they used a fully physically based simulation, the Lagrangian method Moving-Particle Semi-Implicit (MPS), to construct the MTVF, by collecting the stochastic information of the moving particles' motion behavior over each grid node, which is an Eulerian setting. In the second stage, the endless animation is created. At each animation step a temporary velocity field is created on the basis of the MTVF and a random process. Then, the tracer particles can be advected through the

temporary velocity field by assigning the velocities of the grid points to them. Like our system, this physical-stochastic method also avoided solving the complex equations of fluid dynamics for every simulation step. Unlike our system, their method only animates water streams in an endless scenario, and focuses on the surface of the water. Our method works in a different scenario, models the inside flow rather than the surface, and furthermore our method decomposes the fluid domain into smaller subdomains.

2.1.6 Model reduction

Besides the Eulerian and Lagrangian viewpoints, a lot of specific techniques are adopted to improve different aspects of fluid animation, such as detailed features, computational acceleration and fluid control. Model reduction is one of these techniques that focus on computational acceleration. Since some model reduction approaches use Principal Component Analysis (PCA) which we employed in our own solution, and some methods decompose the whole fluid domain into subdomains as we do, we would like to provide a literature review of this technique here.

When the simulation of dynamical system includes more details, the dimensionality of such system may increase to unmanageable levels and require unacceptable amount of computational resources. The purpose of model reduction is to produce a low dimensional system which requires far less storage and much lower evaluation time but still has the same response characteristics as the original system. Due to the limited length of this thesis, we mainly review model reduction in the realm of fluid modeling. For more general and detailed explanation, readers could consult the survey provided by Antoulas et al. [55].

Model reduction has a relatively long history in the applied mathematics literature, as well as in computational fluid dynamics (CFD) [56]. In computer graphics literature, model reduction is increasingly important. Although model reduction had been used to overcome a wide range of problems, ranging from global illumination to elastostatics and dynamics, it had not been applied to fluid simulation until recent years.

Treuille et al. [57] first introduced the model reduction approach into the fluid animation community. First they produced a set of high resolution fluid simulations representative of the expected user input with an accurate off-line solver. They distilled these velocity fields into a small basis of size proportional to the system's principle modes of variation. Then with techniques drawn from model reduction, namely the Galerkin projection, the incompressible Navier-Stokes equations were projected onto this low dimensional basis. After these precomputations, the velocity field could be simulated very quickly through the subspace. Since the simulation runtime was polynomial in the number of basis states rather than the number of simulation voxels, they achieved large, real-time and detailed flows with continuous user interaction.

Since the Galerkin projection technique is critical in this method, we provide a brief review here. For compressing redundant information, we may want to represent a vector in high dimensional space $\vec{u} \in R^n$ with a corresponding vector in a much lower dimensional space $\vec{r} \in R^m (m < n)$. This is called dimension reduction. To translate between the two spaces we need a projection operator $P : \vec{u} \mapsto \vec{r}$ and its inverse $P^{-1} : \vec{r} \mapsto \vec{u}$. In physical simulation, besides the state of $\vec{u}$ we are also interested in its time evolution, which is typically described by an ordinary differential equation: $\dot{\vec{u}} = F(\vec{u})$ (The dot indicates a time derivative). For model reduction, we look for an

analogous evolution for the reduced state: $\dot{\vec{r}} = \hat{F}(\vec{r})$. A standard solution is to calculate the Galerkin projection of F onto the reduced dimensional space:

$$\hat{F} = P \circ F \circ P^{-1} \tag{2.9}$$

Suppose the reduced vectors cover an m-dimensional linear subspace of R^n, which means there exists an $n \times m$ matrix B such that $\vec{u} = B\vec{r}$. Further, if B is orthonormal then $\vec{r} \approx B^T \vec{u}$, with equality when $\vec{u}$ lies in the subspace spanned by B. Given a dynamic system, a linear differential equation $\dot{\vec{u}} = M\vec{u}$, the Galerkin projection of M yields another linear differential equation $\dot{\vec{r}} = B^T MB\vec{r}$, in which the $m \times m$ matrix product $B^T MB$ can be precomputed to simulate without restoring to the full dimensional space.

In the case of fluid simulation, after the fluid domain has been spatially discretized by the grid structure (as described in the Eulerian method subsection), all velocity fields can be represented as n-dimensional vectors. In order to apply model reduction, a low dimensional orthonormal basis B must be created for the simulation. Assume that we have a set of example velocity fields $\{\vec{u}_i\}$, which form a representative basis U of the interactions expected in the system. Typically these example states are snapshots obtained from a set of off-line, full dimensional fluid simulations. Assume that each velocity field in U satisfy two properties:

- Divergence-free: $\nabla \cdot \vec{u}_i = 0$.

- Free-slip boundary conditions: for all fixed surface points $\vec{x}$ with normal $\vec{n}$, we have $\vec{u}_i(\vec{x}) \cdot \vec{n} = 0$.

Then what we look for is a low dimensional orthonormal basis $B = \left(\hat{\vec{u}}_1, \hat{\vec{u}}_2, ..., \hat{\vec{u}}_m\right)$ that minimizes the square reconstruction error $\left\| U - BB^T U \right\|_F^2$, subject to the basis states $\hat{\vec{u}}_i$ also preserving these two constraints. (Here $\| \|_F$ is the Frobenius norm, with $\| \|_F^2$ denoting the sum of squared matrix entries.) This error can be minimized by setting B to the first m eigenvectors of the matrix UU^T, actually performing a Principal Component Analysis (PCA). (More details of the PCA procedure are provided in the section of PCA based pattern matching and Chapter 5 database query acceleration.) Moreover, the vectors calculated by PCA satisfy these two properties without modification.

Armed with this low dimensional orthonormal basis B and the Galerkin projection technique, following Stam [2]'s splitting approach, which decomposed the incompressible Navier-Stokes equations into four steps, namely advection, diffusion, external forces and projection, Treuille et al. [57] simulated each step on the subspace B, with a per-timestep time complexity depending only on the basis dimensionality. Since the model reduction approach preserves those two properties automatically, they could completely skip the projection step for fixed boundaries, and approximated the local pressure forces around each moving objects. This also yielded substantial speed gains because projection is in general the most costly step of grid-based simulators. However, this speed came at the cost of precomputations, generality, accuracy and memory consumption. Since the system must be trained before simulating, changing the boundary conditions or creating new objects requires lengthy precomputations. Further, it is

difficult to determine the choice and number of basis states. If the user presents runtime inputs on which the system has not been trained, or the basis size is simply too small, the simulation result will not be satisfactory.

Although the model reduced simulation introduced by Treuille et al. [57] is very fast, it is also highly inflexible. Once the monolithic model is constructed, even small changes to the simulation domain require complete recomputation of the model. Wicke et al. [60]'s work addressed this problem by replacing the monolithic basis with a modular set of simulation primitives, called tiles, which captured spatially localized fluid behavior given specific boundary conditions such as the presence of an obstacle. To obtain these tiles, they decomposed the simulation domain. Each tile, consisting of a velocity basis representing the possible flow within the subdomain, was a reduced model precomputed from high resolution simulations. Then they precomputed coupling terms so that these tiles could be combined in a modular fashion, which means assembly and reconfiguration at runtime. Further, new tiles could be introduced without recomputing information about existing tiles. To enforce consistency between adjacent tiles (namely, velocities on the boundaries between adjacent subdomains need to be consistent for both sides), they introduced constraint reduction. This technique modifies the simulation bases so that they can satisfy a large set of consistency constraints while preserving sufficient freedom in the representation to prevent locking. Since both the dynamics and the constraints could be solved entirely in the low dimensional reduced space, their method was fast and tilings could scale to very large domains.

Chenney [58] also used flow tiles for representing and designing velocity fields. The tiles they presented were stationary, but dynamic flows could be generated by

combining tilings over time. Their flow tiles relied on the user to create plausible fields, although the constraints in designing the tiles (the way of representing and filling the velocities inside each tile) and the tiling algorithms made their resulting velocity fields divergence-free. The primary advantage of their flow tiles was that a wide variety of external and internal boundary conditions could be met by properly constructing the tilings. However, their interpolation technique for filling the interiors of tiles was not physically motivated, and made the generation of small scale turbulence difficult. Basically, using interpolation based tiles was no more than simply specifying corner velocities and edge fluxes, and then interpolating those directly.

Although in our own solution we also decompose the whole fluid domain into smaller subdomains, our method is quite different. Our solution does not calculate or assemble any bases to simulate the fluid in the subspace as Wicke et al. [60] did. In fact, we do not solve any term of the Navier-Stokes equations in the fluid generation stage. Our method is example based, and the velocity fields rely on the input simulation examples rather than a user's designing ability like Chenney [58]'s work.

2.2 Data-driven motion synthesis

Since the generation procedure of our fluid simulation is analogous to the example based motion synthesis method, we would like to provide a concise literature review in this area as well.

In the media like CG movies and video games, natural looking human motion is crucial. More lifelike characters make for more vivid environments and more believable special effects. At the same time, because people are good at discerning the subtleties of

human movement and identifying inaccuracies, realistic animation of human motion is a challenging task.

A common solution is motion capture. Motion capture (or Mocap) is the term used to describe the process of first recording motion data for an approximate skeletal hierarchy of the subject and then using the data to drive a reconstruction on the computer. However, reuse of motion capture data has proven to be difficult, and editing techniques are only reliable for small changes to a motion. If the captured data isn't sufficiently similar to what the animator wants, such as to be at a particular position at a particular time accurately, then usually there is little that can be done except acquiring more data, which is an expensive and time-consuming process.

To make motion capture widely available, the motion capture data needs to be made reusable. This may mean using previous motion capture data to synthesize new motions so that certain requirements can be met. In fact, quite a number of researchers have pursued this kind of strategy.

Lamouret and Panne [61] presented a system that stored a set of example motions in a motion database and then created new animation out from that database. Animation was generated by cutting-and-pasting together the example motion segments that best fitted into a desired motion. Tanco and Hilton [62] developed a statistical model which could be used to synthesize human motion sequences from a database of motion capture examples. The user could specify the start and end keyframes to create new movements. In order to generate motion sequences between the keyframes, segments of the original motion capture data were identified by the statistical model. Lee et al. [63] proposed a graph structure that could enable a database of motion sequences to be used effectively to

animate an avatar with three different user interfaces: a list of choices, a sketch-based interface, and a live video feed. To add flexibility, the motion data was preprocessed by creating connecting transitions where good matches in poses, velocities, and contact state of the character existed. Then the motion was clustered into groups for efficient searching and for presentation in the interfaces. This graph could be searched at run time to find appropriate paths to the behaviors and locations specified by the user. Kovar et al. [3] constructed a directed graph, motion graph, from a corpus of motion capture data. This motion graph encapsulated connections among the motion database so that both original motions and the generated transitions were included. Motion could be generated by building particular walks on the graph that meet a user's specification. Arikan and Forsyth [4] constructed a similar motion graph from a motion database and used a randomized search algorithm to generate motion sequences that could meet user constraints. The Snap-Together Motion (STM) approach, proposed by Gleicher et al. [74], also preprocesses a corpus of motion capture examples into a set of short clips that can be assembled to produce continuous streams of motion. This approach addresses controllability demands by allowing the designer to guide the graph building process to ensure that the graph has a usable structure. Ikemoto and Forsyth [75] presented a method that enriches a motion collection by cutting limbs from one motion sequence and attaching them to another. Based on a set of rules, they used randomized search to generate transplants. Since transplanting generates many frames quite close to the original frames, motions synthesized using transplants are generally better than those synthesized without using transplants, and therefore it is easier for the motion synthesis process to find a good path in the motion graph.

These methods are similar to each other. Most of them build their data structures based on existing motion capture data, and convert the motion synthesis into some kind of search in their respective structures. The differences lie in their data structure constructions and search algorithms. Moreover, Ren et al. [76] developed measures that quantify the naturalness of human motion, and the measures could be used to evaluate or improve the performance of motion synthesis approaches by identifying motion produced by those algorithms that was likely not natural.

2.3 PCA based pattern matching

In our solution, when we look for the best match fragment in the constructed database, we employ a PCA (Principal Component Analysis) based pattern matching technique, which has been used widely in the face recognition literature. For completeness, we here provide a simple review of this technique as well.

Face recognition is one of the most successful applications of image analysis and understanding, and the literature on it is vast and diverse. There are many methods of face recognition proposed during the past 40 years. What we review in this thesis is only one method that is related to our fragment matching solution. For readers who are interested in detailed and comprehensive review of the face recognition literature, we recommend the survey paper by Zhao et al. [67].

Among various face recognition methods, the holistic matching methods use the whole face region as the raw input of the system, rather than extract the local features such as eyes, nose and mouth then feed their properties into a classifier. For efficiently representing pictures of the whole face region, Sirovich and Kirby [64] and Kirby and

Sirovich [65] developed a technique based on principal component analysis, called eigenpictures. Afterward, eigenpictures became one of the most widely used representations of the face region. Since there are significant statistical redundancies in face images, any collection of face images could be approximately represented by a small collection of weights for each face and a small set of standard pictures (the eigenpictures). The weights that represent each face are obtained by projecting the face image onto the eigenpictures.

More specifically, face images can be considered as two-dimensional N by N arrays of intensity values, or N^2 dimensional vectors. For PCA to work properly, the mean of the vectors (the average face of the face image collection), has to be subtracted from each individual vector. Then the mean subtracted vector $\vec{x}$ can be represented by a linear combination of the set of orthonormal basis vectors Φ_i : $\vec{x} = \sum_{i=1}^{n} a_i \Phi_i \approx \sum_{i=1}^{m} a_i \Phi_i$ (typically $m << n$). The basis vectors are the eigenvectors, actually referred to as eigenpictures, and they can be obtained by solving the eigenproblem:

$$C\Phi = \Phi\Lambda \tag{2.10}$$

where C is the covariance matrix of the ensemble of faces, and Λ denotes the diagonal matrix with eigenvalues on the diagonal.

The first really successful face recognition system was developed by Turk and Pentland [66] using eigenpictures (also known as eigenfaces). They built up the eigenpictures by experience over time, and every face known in the database could be

represented as a vector of eigenpicture weights. The weights could be calculated by projecting the face image onto the orthonormal basis vectors (eigenfaces) with a simple inner product operation. When a new face image was inputted for recognition, firstly, they projected the image onto each of the eigenfaces and calculated its corresponding eigenface weights as well. Then the input image could be identified by locating the face image known in the database whose vector of weights had the closest Euclidean distance to the weights of the input image.

3 Chapter: Fluid Example Database Construction

In this chapter, we describe the fluid example database construction procedure in the preprocessing stage. Firstly, we introduce the grid based fluid simulation setup used in our implementation. Then we explain how to collect the fluid motion example data and how to process and store those data.

3.1 Grid based fluid simulation setup

Since we want to capture the properties of local fluid behavior by dividing the fluid domain into smaller subdomains, the grid based Eulerian method is inherently a good choice for us. The particles system of Lagrangian method would not be fit in this requirement, because particles would flow around different subdomains and make it difficult to be depicted within any subdomain. In fact, as we introduced in the literature review, the previous works of fluid modeling that decomposed the fluid domain into subdomains [58] [60], all chose the Eulerian method spontaneously.

It is worth noting that, as one example-based simulation, the work of Nugjgar et al. [14] did use Lagrangian method in their preprocessing stage, but the moving particles were used only to collect the stochastic information of their motion behavior over the underlying grid nodes, which is also an Eulerian setup.

In the previous chapter, in the Eulerian method subsection we introduced two different ways to make use of the grid to do the discretization, the staggered Marker and Cell (MAC) grid and the grid that stores different variables at the same locations. For the

simplicity, and more importantly for the consistency with the fluid solver that we use to produce the input example data, we follow the Eulerian approach's convention by dicing up a finite region of space into identical size *cells* and sample the fluid velocity at each cell's center.

Since in the preprocessing stage we use Stam's fluid solver [2] to collect the example data, for consistency and simplicity, we choose a box as our fluid modeling space as Stam did, more specifically, a square in two dimensions, a cube in three dimensions. See Figure 3. In this thesis, we describe our approach mainly in two dimensions for the sake of clarity, but our method in not constrained to two dimensions.

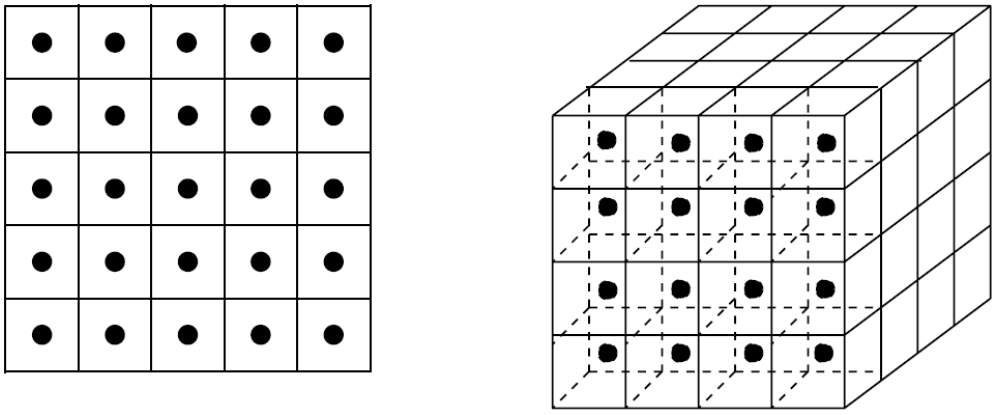

Figure 3 The velocity field is defined at the center of each cell (reprinted from [2]).

Therefore, including the additional layer of cells that account for the boundary conditions, in two dimensions we have computational grids of size $(N+2)\times(N+2)$, that is $N+2$ cells in width and $N+2$ cells in length. Our method will follow this setting throughout the whole implementation, namely in both the preprocessing stage and the fluid generation stage.

3.2 Example data collection

After the description of the setup we chose, it should be clear that the example data we want to collect are just sequence of frames over time, with each *frame* being a static state of the whole fluid space at a given instant of time, or mathematically, a velocity vector field. Obviously, according to the discretized settings we use, in each frame there are $(N+2)\times(N+2)$ cells with $(N+2)\times(N+2)$ velocity vector values sampled at the cells' centers. In order to obtain the fluid example data, or in other words, to make the velocity vector values specified for all these frames, one option is to make use of a fluid solver.

3.2.1 Fluid solver

Among the Eulerian methods reviewed in the previous chapter, which simulate fluid and approximate the Navier-Stokes equations, the stable fluid algorithm presented by Stam [2] is a milestone. This algorithm is unconditionally stable. It allows long time steps and achieves fast simulation. Although our algorithm does not limit to use any particular fluid solver or even use a solver at all (we may record and extract the fluid motion data from the nature), for simplicity we choose to employ Stam's fluid dynamics solver [2] [68] to calculate our fluid example data, and use the same settings in our own modeling approach as described in the last section.

In Stam's paper [2] [68], a framework of the two-dimensional fluid solver was provided with C code. We customized this fluid solver for obtaining our two-dimensional example data, and we also extended it to a three-dimensional version in order to collect

our three-dimensional example data. For integrity, we would like to provide a concise explanation of this stable fluid algorithm here.

The fluid solver evolves the velocity vector field from an initial state $\vec{u}_0 = \vec{u}(\vec{x}, 0)$ (with $\vec{x}$ corresponding to the spatial coordinate) by marching through time with a time step Δt. Assume that the velocity field has been resolved at a time t, then the fluid solver calculates the velocity field at a later time $t + \Delta t$ according to the Navier-Stokes equations.

The algorithm works on the basis of the splitting technique [17]. If the rate of change of one quantity is the sum of several terms, using this technique the quantity can be numerically updated by computing each term and adding them in one by one. For example, if we know $\frac{\partial q}{\partial t} = f(q) + g(q)$ (where f and g are some black box functions), we could split it up into two steps: $\tilde{q} = q^n + \Delta t f(q^n)$, and $q^{n+1} = \tilde{q} + \Delta t g(\tilde{q})$. Here q^n denotes the value of q computed at time step n. $\tilde{q}$ is an intermediate value that includes the contribution of the first term, and by adding the second term's contribution it gets to q^{n+1}, the value of q for the next time step.

In the stable fluid algorithm, this splitting technique is employed to decompose the simulation into a series of separate steps, each one tackling a different term in Equation 2.1, namely external force $\vec{g}$, advection $-\vec{u} \cdot \nabla \vec{u}$, diffusion $\nu \nabla \cdot \nabla \vec{u}$, and projection $-\frac{1}{\rho} \nabla p$. See Figure 4. Over each time span Δt, the velocity field needs to be updated in these four steps, starting from the solution $\vec{w}_0(\vec{x}) = \vec{u}(\vec{x}, t)$ of the previous time step, ending with the solution $\vec{u}(\vec{x}, t + \Delta t) = \vec{w}_4(\vec{x})$ at time $t + \Delta t$.

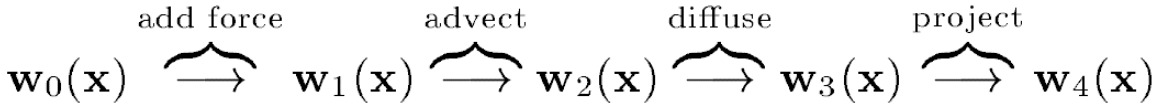

Figure 4 **Four steps need to be iterated over each time span (reprinted from [2]).**

The first step, the addition of the external force $\vec{g}$, is straightforward to solve.

$$\vec{w}_1(\vec{x}) = \vec{w}_0(\vec{x}) + \Delta t \vec{g}(\vec{x}, t) \tag{3.1}$$

The second step accounts for the effect of advection, which means that all the fluid quantities (including velocity) are moved by the velocity of the fluid itself. Instead of using finite differencing, to obtain the velocity at a point $\vec{x}$ at the new time $t + \Delta t$, the algorithm back traces the point $\vec{x}$ through the velocity field $\vec{w}_1$ over a time Δt. This defines a path $\vec{p}(\vec{x}, s)$ corresponding to a partial streamline of the velocity field $\vec{w}_1$. Then the velocity that the point $\vec{x}$ had at its previous location $\vec{p}(\vec{x}, -\Delta t)$ is assigned to the new velocity at the point $\vec{x}$. This idea is generally named as the Semi-Lagrangian technique.

$$\vec{w}_2(\vec{x}) = \vec{w}_1(\vec{p}(\vec{x}, -\Delta t)) \tag{3.2}$$

The diffusion step solves for the effect of viscosity. Since solving the explicit diffusion equation is unstable when the viscosity is large, the algorithm chooses to use an

implicit method. See Equation 3.3. After the diffusion operator $\nabla \cdot \nabla$ is discretized, it leads to a sparse linear system for the unknown field $\vec{w}_3$, which can be solved using a simple iterative technique called Gauss-Seidel relaxation.

$$\vec{w}_3(\vec{x}) - \nu \Delta t \nabla \cdot \nabla \vec{w}_3(\vec{x}) = \vec{w}_2(\vec{x}) \tag{3.3}$$

On the forth step, instead of solving the pressure field and then adding the contribution of the term $-\frac{1}{\rho}\nabla p$ directly, the algorithm performs a projection that makes the resulting field divergence-free. The projection is based on the mathematical theorem, the Helmholtz or Hodge decomposition, which states that any vector field $\vec{w}$ can uniquely be written as the sum of a divergence-free field and a gradient field. That is $\vec{w} = \vec{u} + \nabla q$, where $\vec{u}$ has zero divergence (i.e. $\nabla \cdot \vec{u} = 0$) and q is a scalar field. Since the desired velocity vector field $\vec{w}_4$ is divergence-free and the remaining pressure term $-\frac{1}{\rho}\nabla p$ is a gradient field (p is a scalar field), if we project the vector field $\vec{w}_3$ onto its own divergence-free part, we can get $\vec{w}_4$ directly. In other words, the vector field $\vec{w}_3$ can be resolved into the sum of the divergence-free field $\vec{w}_4$ and a gradient field, say ∇q, namely $\vec{w}_3 = \vec{w}_4 + \nabla q$. To solve this scalar field q, we can multiply both sides of this equation by "$\nabla \cdot$", then the $\vec{w}_4$ term becomes zero.

$$\nabla \cdot \nabla q = \nabla \cdot \vec{w}_3(\vec{x}) \tag{3.4}$$

$$\vec{w}_4(\vec{x}) = \vec{w}_3(\vec{x}) - \nabla q \tag{3.5}$$

Equation 3.4 is a Poisson equation for the scalar field q. When spatially discretized, this Poisson equation also becomes a sparse linear system, and it can be solved using Gauss-Seidel relaxation similarly to the diffusion step. Once the scalar field q is solved, the velocity vector field for the next time step, $\vec{w}_4$, is trivial to calculate according to Equation 3.5.

For the treatment of the boundary, we implemented the commonly used free-slip boundary condition. At every time step, the velocity at the extra layer of grid cells around the fluid's domain is set to compensate the normal component of the velocity field at the boundary, making sure that fluid does not penetrate through the boundary. More specifically, the horizontal component of the velocity should be zero at the vertical boundary, while the vertical component of the velocity should be zero at the horizontal boundary.

3.3 Segmentation process

After collecting, the example data that we obtained are sequence of frames over time. Each frame is a velocity vector field at a given instant of time, and it is discretized by the grid structure. According to the discretization settings we use, obviously in each frame there are $(N+2)\times(N+2)$ grid cells, with one velocity vector value sampled at the center of each grid cell.

Apparently, there would be many different ways to do the segmentation process and construct the database. We perform this procedure as follows.

In the example data, for each adjacent two frames in the same sequence, we pick out every 3×3 grid cells, which form a subdomain of the fluid space, called *fragment*. As shown in Figure 5, neighboring fragments in a frame are overlapped to each other. We store each fragment and its counterpart of the next frame together into database. Therefore, every element in our database is a pair of 3×3 fragments, and each fragment has 9 cells (with velocity vector values) inside. The first fragment in a pair comes from the prior frame, and the second fragment is obtained from same location of the successive frame. All the pairs are stored sequentially in the database so that any pair can be accessed in constant running time using its index number.

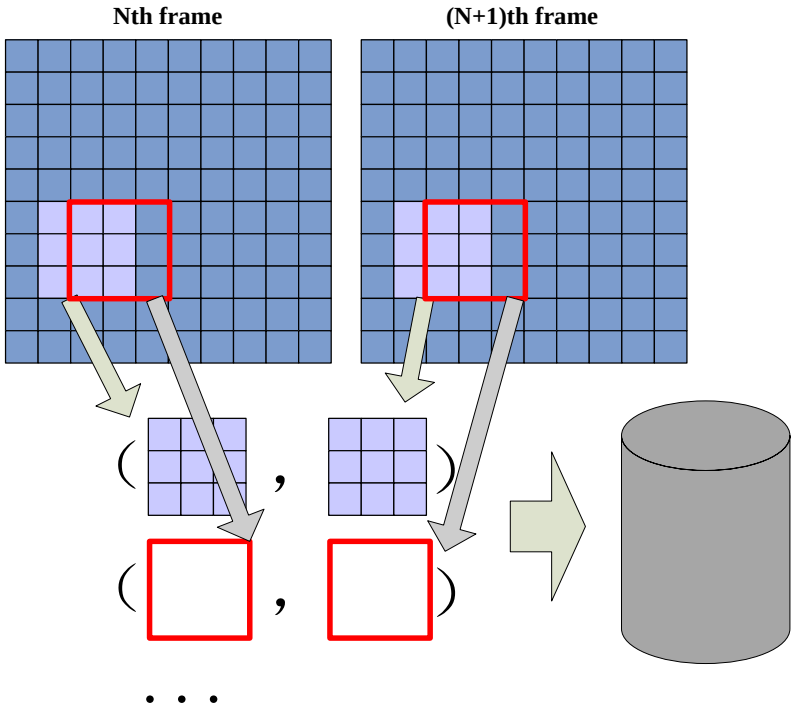

Figure 5 Fluid example database construction. One fragment from prior frame, one from successive frame. Put all the pairs into a sequential data structure for fast and random access.

The size of a fragment we chose, 3×3, is actually a tradeoff between the matching cost for one fragment and the information amount one fragment can contain. When the fragment size increases, more information can be included in one fragment but matching speed will slow down (The matching method will be explained in the next chapter). Smaller fragment size provides us higher speed matching and more flexible constructing. However, too small fragment will definitely lose information and end up with many duplicate elements in the database.

The overlap between neighboring fragments is for better connectivity and smooth propagation when generating new fluid flows.

By storing fragments in pairs, with the first fragment coming from prior frame and the second fragment from the subsequent frame, we captured the fluid motions of the local fragments. Therefore, in fluid generation stage, given a current state of the fluid, in order to properly evolve the fluid flow the next state can be found from the database.

4 Chapter: Fluid Generation

In this chapter, we describe given a current frame how to generate the next frame according to the database constructed in the previous step. From the way we build the database, it is natural to get the idea that we need to generate each frame fragment by fragment. See Figure 6.

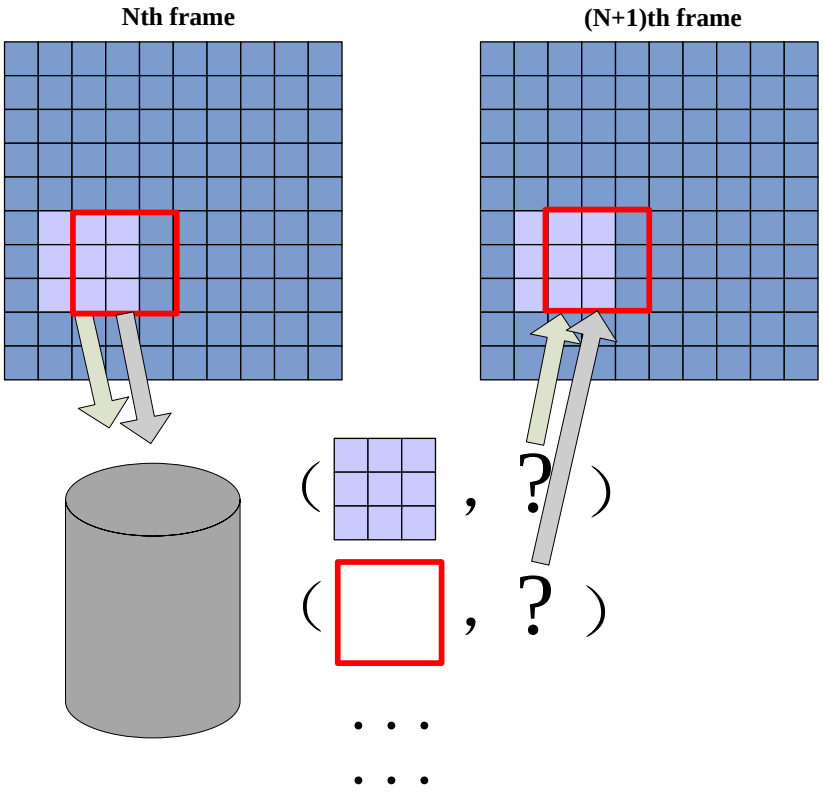

Figure 6 Fluid generation. For each fragment of the current frame, we look for a most similar one in the first fragments of the pairs in database, and use the second fragment in that pair as the next frame's corresponding fragment.

When we are given a frame of fluid, we want to obtain the next frame using the database. We adopt a divide-and-conquer strategy, and perform a fragment based approach, i.e. given a 3×3 size fragment in the current frame we find the fragment for the next frame from the database. More specifically, for each fragment in the current frame, we look for a most similar one in the first fragments of the pairs in the database, and use the second fragment in this pair as the next frame's corresponding fragment. Once all the fragments in the current frame are processed, the next frame is completely generated. We repeat this procedure until the desired number of frames is achieved.

The detailed problems and solutions involved in this procedure are explained in the following sections.

4.1 Initial state

The very first frame of fluid flow, namely the initial state of the velocity vector field, needs to be given at the beginning. For simplicity, as shown in the left part of Figure 7, we choose a frame within which only one single cell has nonzero-velocity vector (all the other cells' velocity vectors are zero) as the initial state.

There are reasons to choose a frame of such properties as the initial state. Firstly, as the input of the fluid generation procedure, it is easy and convenient to specify. Although this initial frame is clearly not divergence-free, its corresponding divergence-free frame as shown in the right part of Figure 7, which could be obtained by going through one circle of the four steps of the fluid solver described in the previous chapter, is way too complicated for the user to specify. Another reason is that the initial velocity state can also be viewed as an instantaneous force acting on the single cell which actually

triggers the whole subsequent fluid flow. This viewpoint is logically correct, because the velocity field and the force acting on it have a direct relationship demonstrated in Equation 3.1 and we assume that the fluid is not moving at the very beginning ($\vec{w}_0(\vec{x}) = 0$).

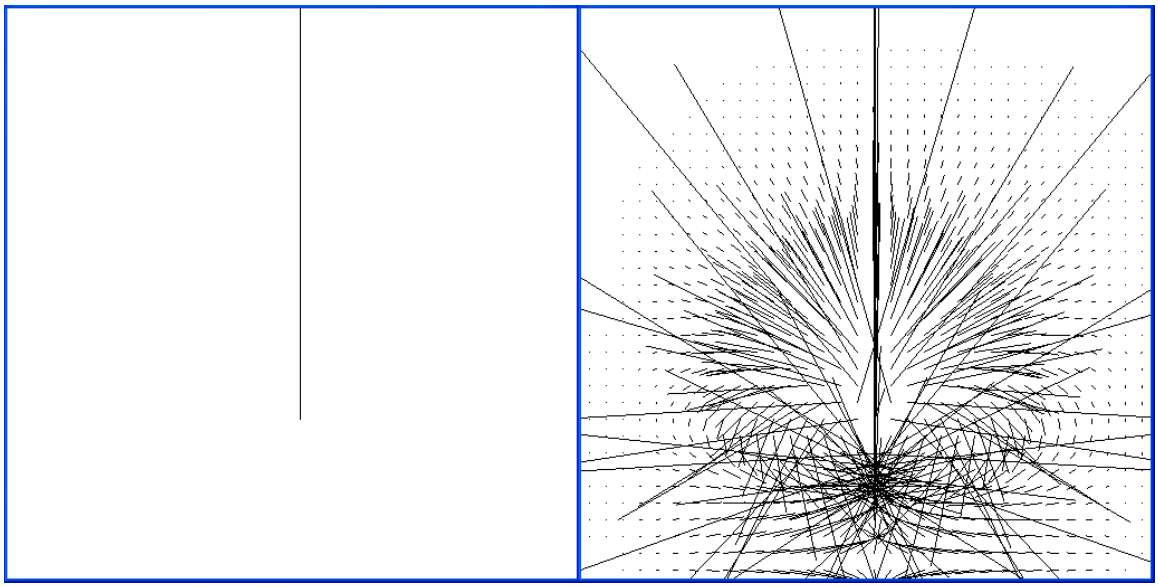

Figure 7 The left side is the initial state, a frame within which only one single cell has nonzero-velocity vector; imagine a situation where an external force acts on the cell to a direction. The right side is its corresponding divergence-free state calculated by the fluid solver from the initial state. Here velocity vector is indicated by line segment.

However, on the other hand, using this simple initial state causes a multivalued function problem. Although it may not be the only situation that has this kind of problem, if we adopt this simple initial state it is inevitable to face this problem. This multivalued function problem can be visually described in Figure 8. Two or more fragments in the current frame have exactly the same states (all zero, in this case), but in the very next frame they have totally different values.

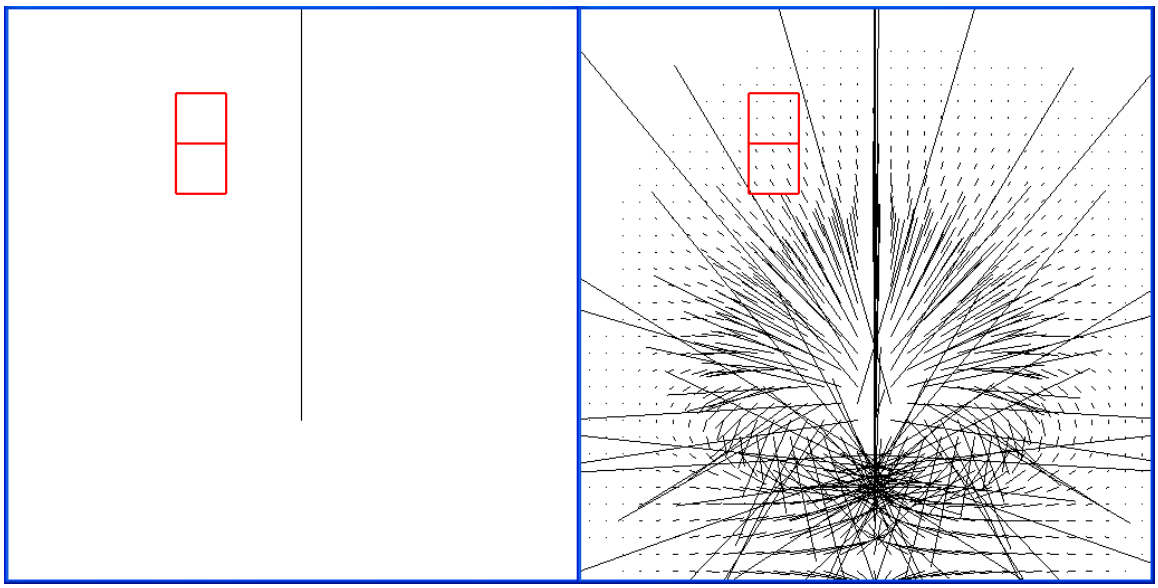

Figure 8 Multivalued function problem in the initial frame and its successive frame. Two adjacent fragments showing this problem are highlighted with red rectangles. Line segment indicates the velocity vector.

In order to tackle this problem, we use a particular order to process the fragments in the current frame, and design a flexible matching strategy to propagate the fluid flow according to the database.

4.2 Processing order

Given this kind of initial state, as described in the previous section, the multivalued function problem is inevitable. As shown in Figure 8, in the initial state many fragments have identical status (all cells' velocity vectors are zero), but their corresponding status for the next frame are completely different. Accordingly, in the database we constructed, there are as many pairs having identical first members, fragments of 9 zero-velocity vectors, but totally different second members. If we use a common order to process the

fragments in this initial state, such as raster scanning order, which starts with a fragment of 9 zero-velocity vectors, it would be impossible to find the fragment's proper state for the next frame among these candidates in the database.

Thus, to overcome this multivalued function problem, we would like to design a processing order that could let us start from the fragment which has the cell with the nonzero-velocity vector, and avoid directly dealing with the fragments that have the same state (all zero-velocity vectors, in this case).

For this purpose, besides velocity we maintain another quantity c at each cell, which is simply the sum of squared velocity differences between the cell itself and each one of its 8 adjacent neighbors:

$$c_{i,j} = \sum_{k,l} (\vec{u}_{k,l} - \vec{u}_{i,j})^2 \tag{4.1}$$

In Equation 4.1, the double subscript shows the location of the cell, with the first part being the column number and the second part being the row number (the column number increases from left to right, while the row number increases from downside to upside). The iterator k goes from $i-1$ to $i+1$, and l from $j-1$ to $j+1$. The squared vector difference can be easily replaced by the sum of the squared differences in each component (x and y components, in this case).

This quantity c indicates the changing intensity inside the fragment centered at this cell, and helps to find the fragment in which not all the cells have the same velocity. We process the fragments in the order that their center cell's c quantity declines. (In fact,

this order only matters when partial matching occurs, which will be explained in the following section.) The extra layer cell does not need this quantity, because it is not the center cell of any fragment. Therefore, we only need to maintain this c quantity inside the region $N \times N$.

More specifically, when generating a new frame, first we select one remaining fragment whose center cell has the greatest c quantity. Then we process this fragment, by looking up the database to find the velocity values for the next frame and updating the cells in the corresponding fragment of the next frame. (The following section will focus on how to find one fragment's velocity vector values for the next frame.) We repeat this procedure till the whole frame is updated. See Figure 9.

Although fragments are overlapped to each other, we choose not to repeat updating by maintaining a set of flags indicating whether each cell has been updated or not. Once a cell's velocity is updated, we set its corresponding flag, and maintain the related cells' c quantities, that is, the quantities of the cell itself and its 8 adjacent neighbors. Armed with these flags, before we process each fragment, we can check whether all the cells in this fragment have been updated. If so, we can skip this fragment and continue from the next one. This is why in Figure 9 at the 483rd substep we could process the last fragment for generating the successive frame of the initial state, instead of processing all the 1024 fragments (there are $N \times N$ fragments in total in the $(N+2) \times (N+2)$ grids, here we set N to 32).

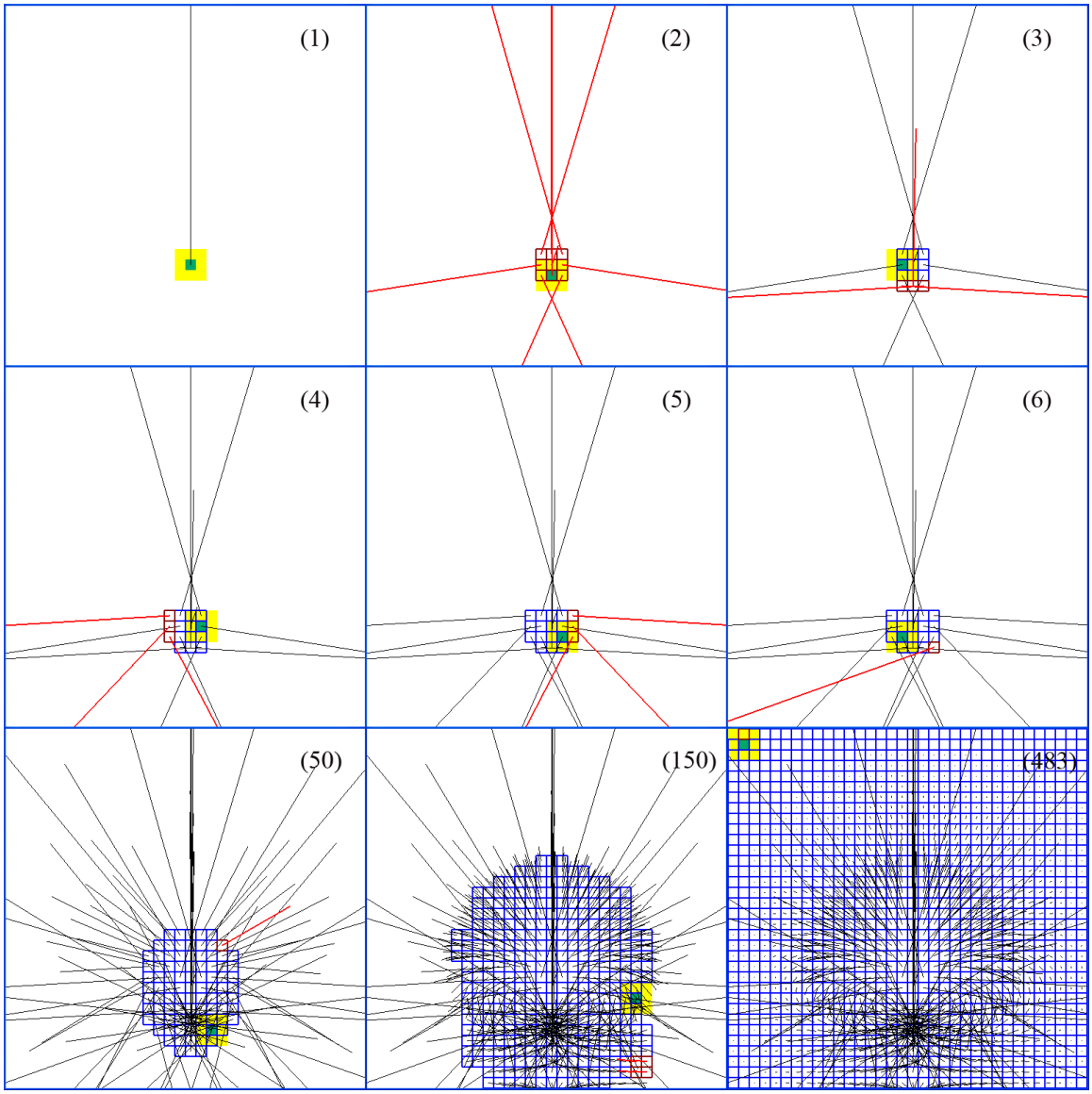

Figure 9 **Sampled substeps from the initial state to the next frame. The serial number is displayed in the upper right corner of each image. Line segment corresponds to the velocity vector, and is colored red when being updated. Grid cells are highlighted by different colors or patterns for different meanings. Yellow interior: fragment selected to be processed; Green interior: center cell of the fragment selected to be processed; Brown boundary: cells being updated; Blue boundary: cells updated.**

4.3 Fragment pattern matching schemes

In this section, we explain given an arbitrary fragment in the current frame how to achieve its corresponding velocity values for the successive frame.

According to the status of the fragment selected to be processed, when we look for the velocity vector values for the next frame, there are two kinds of situations we need to deal with differently in order to overcome the multivalued function problem.

We developed full matching and partial matching schemes to handle different situations. Full matching is employed for most of the time and represents the central idea of our algorithm, while partial matching is a modification which is designed to be used when the multivalued function problem occurs.

4.3.1 Full matching

From observations and experiments, we discovered that the multivalued function problem usually happens when we process the fragment that has zero-velocity vectors. Therefore, if there are no zero-velocity vectors in the whole fragment selected to be processed, that is all 9 cells have nonzero-velocity vectors, then we can ignore the problem temporarily and use a straightforward matching scheme, which we call full matching, to search for the corresponding velocity vector values for the next frame in our database. Fortunately, this is the situation for most of the time.

There is one exception: if none of the cells in the fragment has been updated, we also use this full matching scheme, because partial matching needs updated cells as input. Nearly only the first fragment of the initial frame fits this situation, since the initial frame

only has one cell with nonzero-velocity vector but no cell has been updated yet. See the first image in Figure 9.

The basic idea of full matching is already described visually in Figure 6 at the beginning of this chapter. When we process a fragment that fits the full matching situation described above, we look for the most similar one among the first fragments of the pairs in database, and use the second fragment in that pair as the next frame's corresponding fragment. We choose the name, full matching, because in this matching scheme, when we look for the pair element in the database, we make use of all the contents in the fragment that is being processed, namely all the 9 velocity vectors.

In order to measure the similarity between two fragments, we employ the Euclidean distance as our metric:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{4.2}$$

Here p and q correspond to two fragments. p_i and q_i stand for the velocity vectors of fragment p and q respectively, at the same relative position, such as center, top, upper left corner, etc. In two dimensions we have 9 cells in each fragment, so n equals to 9. Again, the squared vector difference can be easily substituted by the sum of the squared differences in each dimension. Because the Euclidean distance we need is only for comparison between different fragments and the square root function is monotonically increasing, for the sake of efficiency the square root calculation can be safely omitted. Therefore, in two dimensions the Euclidean distance is the sum of 9×2 squared

differences. In three dimensions, since there are 27 cells in each fragment, there would be 27×3 squared differences to compute.

Obviously, two fragments have more similarity when the Euclidean distance between them decreases. However, given a fragment, in order to find its most similar fragment in the database, it would be extremely expensive to calculate all the Euclidean distances between the given fragment and each fragment in the database and then choose the closest one. In fact, for the purpose of accelerating this procedure, we project all the fragments data onto a lower dimensional space with PCA technique and organize these data as a kd-tree of this lower dimensionality in preprocessing, and then we can project the given fragment onto this same subspace and perform a nearest-neighbor query on the run. The detailed explanations of these techniques in database query acceleration are provided in the following chapter.

It is worth noting that when we do full matching all the states that we use as matching reference to look up the database come from the current frame. This is different from partial matching scheme, which is explained in the next subsection. Since fragments are overlapped, one cell usually can be involved in different fragments. Therefore, when we process a fragment and update its cells with the following frame's velocity vectors, we update them in a new copy, in order to save the current frame's state for processing other fragments. In other words, we keep the current frame status untouched until the whole following frame is completely generated.

4.3.2 Partial matching

When we process a fragment that has zero-velocity vectors, the full matching is not safe to use, because with this scheme there is a high probability that the multivalued function problem occurs. Usually this is the situation when we generate the following frame from the initial frame. As shown in Figure 8, in the initial frame which has only a single cell with nonzero-velocity vector, there are many similar fragments with 9 zero-velocity vectors, but their corresponding states for the next frame are not the same. After the database construction step, there are many pairs in database with the exactly same first members, that is a fragment with 9 zero-velocity vectors, but their second members are completely different. Since full matching's input fragment states only come from the current frame, in the initial frame given a fragment that has 9 zero-velocity vectors, it is impossible for the full matching scheme to look for the most similar fragment in the first members of the pairs in database and fetch the proper corresponding fragment for the next frame.

Therefore, we shift to a different scheme, partial matching, when the fragment selected to be processed has zero-velocity vectors in the current frame. In fact, with the cooperation of the processing order described in the previous section and this partial matching scheme, we can successfully solve this problem as shown in Figure 9. The processing order that relies on the changing intensity prevents us from processing the fragments with only zero-velocity vectors directly at the very beginning, and lets us start with the fragment with nonzero-velocity vector, as shown at the beginning of the first substep (the first image) in Figure 9. Once this fragment is processed using the full matching scheme (this is the exception explained in the previous subsection), its 9 cells are updated, as show at the end of the first substep (the second image). Then the partial

matching uses the updated states as the matching reference rather than the states from the initial frame, and allows us to proceed in such situation, as shown in the following substeps in Figure 9.

More specifically, in the partial matching scheme, when we process a fragment we choose only the cells that have been updated as the matching reference. Then we iterate the whole database, and calculate the Euclidean distance between these updated cells' state of the processing fragment and the corresponding cells' state of the second fragment in each pair element. Here we choose the second fragment of each pair to calculate rather than the first member which would be chosen in the full matching scheme, because the matching reference we use are the updated states not the states from the current frame.

With slight modifications of the symbols' interpretations, Equation 4.2 can still be used to compute the Euclidean distance for partial matching. In partial matching, n corresponds to the number of cells that have been updated in the fragment being processed, and apparently it changes for different fragments. The subscript in the velocity vector p_i and q_i stands for the position of the updated cell in the fragment being processed, and it is distributed dynamically since the updated cells are not fixed for different fragments while the system is running. For example, at the beginning of the second substep (the second image) in Figure 9, 6 updated cells in the upper part of the yellow fragment are used to calculate the Euclidean distance, while at the beginning of the third substep (the third image) the updated cells are in the right part of the fragment that is selected to be processed.

After calculating the Euclidean distance for each pair element in the database, we choose the pair with the closest Euclidean distance, and use the remaining cells' velocity vectors (i.e. the vectors that are not used to compute the Euclidean distance) in the second fragment of that pair to fill their corresponding cells in the processing fragment, which have not been updated before. For example, at the beginning of the second substep (the second image) in Figure 9, we selected the yellow fragment to process and used its upper 6 updated cells as the matching reference. After we found the pair whose second fragment's upper 6 velocity vectors are most similar to the matching reference, we used the other 3 velocity vectors at the bottom of this pair's second fragment to update the corresponding cells in the fragment being processed as shown at the end of the second substep (the third image) in Figure 9.

Since, for the fragment being processed, both the number and the relative positions of the cells that are chosen as the matching reference are dynamically changing, we could not accelerate this procedure as we do in the full matching scheme. Fortunately, at most of the time we can use the full matching scheme, and the situation that fit the partial matching condition occurs infrequently.

4.4 Boundary condition

No matter using which matching scheme, the generated velocity vectors can cover the whole region of $(N+2)\times(N+2)$ grids, which already includes the extra layer that accounts for the boundary conditions. However, in order to reduce the errors that may be introduced during the fluid flow evolution and reinforce the boundary condition, at each time a new frame is completely generated we set its extra layer cells in the same way as

the fluid solver does at every time step. As explained in database construction chapter, the treatment for free-slip boundary condition is to set the velocity vectors at the extra layer cells such that the normal component of the velocity field at the boundary is compensated, making sure that fluid does not penetrate through the boundary. More specifically, the horizontal component of the velocity should be zero at the vertical boundary, while the vertical component of the velocity should be zero at the horizontal boundary. Once set, the extra layer cells will be used in the following frame's generation.

5 Chapter: Database Query Acceleration

In this chapter, first we introduce the techniques that are employed to efficiently query the constructed database, of which the structure is described in Chapter 3, according to the requirement of the full matching scheme defined in Chapter 4. Then we show our results at the end of this chapter.

The main techniques we use are the Principal Component Analysis (PCA) and kd-tree. With the PCA technique, we can project all the fragment data that is supposed to be used in calculating the Euclidean distance for the full matching, namely the fragment data from the first members of the pairs in the database, onto a lower dimensional subspace. Therefore, given a fragment that is selected to be processed, by projecting its data onto this same subspace we can perform the full matching procedure completely in this reduced lower dimensional subspace and still find its most similar fragment. Using the kd-tree data structure, at the preprocessing stage we can organize these reduced data properly, and then at the fluid generation stage perform the highly efficient nearest neighbor query instead of calculating the Euclidean distance to each reduced element data.

5.1 Dimension reduction using PCA

The idea of using Principal Component Analysis (PCA) technique to accelerate the full matching procedure comes from the face recognition literature. In Chapter 2, this PCA based matching method is concisely introduced from the face recognition point of view.

In this section, we provide a detailed explanation of this PCA technique and how this technique is adjusted into our system to accelerate the full matching procedure.

As described in Chapter 4, normally the full matching procedure needs to calculate the Euclidean distance between two fragments. That is to compute the sum of 18 (9×2) squared differences between corresponding velocity components in two-dimensional system, and 81 (27×3) squared differences in three-dimensional system, as expressed in Equation 4.2. Compared to the face recognition system, a two-dimensional fragment, namely 3×3 grid cells, can be considered as a vector of dimension 18 (81 for three-dimensional fragment), or, equivalently, a point in 18-dimensional space. More specifically, in the vector that represents a fragment, we set the first 9 coordinates to the x components of the 9 velocity vectors of the fragment, and set the other 9 coordinates to the y velocity components. Then, an ensemble of fragments maps to a collection of points in this 18-dimensional space.

Similarly to the idea of eigenpictures [64] [65] or eigenfaces [66] in face recognition's literature, the fragments, having some regularity in their patterns, will not be randomly distributed in this large space, and thus can be described by a relatively low dimensional subspace. Then, the Euclidean distance in this subspace can be employed to find the most similar fragment, and therefore the full matching procedure can be accelerated. In order to project the fragment data onto this lower dimensional subspace, we employ the PCA dimension reduction technique.

Principal Component Analysis (PCA) is a powerful statistical technique for identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Once we have found these patterns in the fragments, we can

compress the data by reducing the number of dimensions without much loss of information. Although, besides PCA, quite a few dimension reduction techniques (linear or non-linear) exist, PCA is the best linear dimension reduction technique in the mean-square error sense [77]. Moreover, since linear technique is generally simpler and easier to implement than the method considering non-linear transforms, PCA technique is most widely used. The following paragraphs explain how we perform the Principal Components Analysis on the set of fragments concerned. Readers interested in other dimension reduction methods, such as Factor Analysis, Projection Pursuit, and Independent Component Analysis, may wish to consult the survey by Fodor [77].

Suppose the fragments from the first members of the pairs in our database are Γ_1, Γ_2, … , Γ_n (vectors of dimension 18). For PCA to work properly, first we have to subtract the mean from each of the data dimensions. The mean subtracted is the average across each dimension. So, we define the average fragment as $\Psi = \frac{1}{n}\sum_{i=1}^{n}\Gamma_i$, and then each fragment has the average fragment subtracted and becomes the vector $\Phi_i = \Gamma_i - \Psi$. This produces a set of vectors whose mean is a zero vector.

After subtracting the mean, we can construct the covariance matrix C for this set of fragments as:

$$C = \frac{1}{n}\sum_{i=1}^{n}\Phi_i\Phi_i^T \qquad (5.1)$$

Since the fragment data has 18 dimensions, the matrix C has 18 rows and 18 columns (so it is square). Apparently, this matrix is symmetrical about the main diagonal.

The next step is to calculate the eigenvectors and eigenvalues of the covariance matrix C. Readers who are not familiar with the concepts of eigenvectors and eigenvalues can consult any text book about linear algebra. According to their mathematical properties, because the covariance matrix is square (18×18) and symmetric, there exist 18 eigenvectors and associated eigenvalues for this matrix, and all the eigenvectors are 18-dimensional and perpendicular to each other. Since there are ready-made solutions for this typical problem, instead of implementing the algorithms by ourselves, we adopt the C++ solution directly from the classic book written by William et al [69]. This method first reduces the input symmetric matrix to a tridiagonal form, and then uses the QL algorithm with implicit shifts to determine the eigenvalues and corresponding normalized eigenvectors.

After the eigenvectors and eigenvalues are found, we can choose the components and form a feature vector, which is just a fancy name for a matrix of vectors. Since the eigenvector with the highest eigenvalue is the principle component of the fragment data set, we order the eigenvectors by eigenvalue from highest to lowest. This produces the components of the fragment data set in order of significance: $\vec{u}_1$, $\vec{u}_2$, … , $\vec{u}_{18}$. Given the number of dimensions we want to reduce to, say m, we ignore the components of lesser significance and put only the first m eigenvectors in the columns to form a $18\times m$ matrix, the feature vector F:

$$F = (\vec{u}_1 \quad \vec{u}_2 \quad . \quad . \quad . \quad \vec{u}_m) \tag{5.2}$$

Once we have chosen the components (eigenvectors) that we wish to keep in our data and formed the feature vector F, we simply take the transpose of F and multiply it on the left of the mean-subtracted original fragment data as:

$$\Omega_i = F^T(\Gamma_i - \Psi) = F^T\Phi_i \qquad (5.3)$$

This simple operation transforms a fragment data Γ_i into the m most significant components we chose, i.e. it projects a fragment data Γ_i onto the m dimensional subspace. The new vector with the reduced dimensionality m, $\Omega_i^T = (\omega_1 \quad \omega_2 \quad . \quad . \quad . \quad \omega_m)$, describes the contribution of each eigenvector in representing the input fragment data Γ_i, treating the m eigenvectors as a basis set for fragment data. Reversely, that is to say, the input fragment data Γ_i can be represented by a linear combination of the set of orthonormal basis vectors (the m eigenvectors):

$$\Gamma_i \approx F\Omega_i + \Psi \qquad (5.4)$$

When m increases to 18, the exact input fragment data can be represented.

Therefore, for the purpose of accelerating the full matching procedure, at the preprocessing stage, we project all the fragments (the first members of the pair elements, which are sequentially stored in our database), Γ_1, Γ_2, … , Γ_n, onto the m dimensional

subspace using Equation 5.3, and obtain their corresponding m dimensional data, Ω_1, Ω_2, … , Ω_n. At the fluid generation stage, when we process a fragment that fits the full matching situation, we project this fragment Γ_k onto this same subspace with Equation 5.3 as well, and get its corresponding m dimensional data Ω_k. Then, instead of directly finding the most similar fragment to Γ_k, we look for Ω_k's most similar data in the range of Ω_1, Ω_2, … , Ω_n. Once we get the result, Ω_l, from its index l we can easily locate its original fragment Γ_l and the related pair element in our database, and then we can use the second member of that pair as the following frame's corresponding fragment. When we look for Ω_k's most similar data, we still use the Euclidean distance, as shown in Equation 4.2, as the metric. However, instead of calculating the Euclidean distance between the 18-dimensnional fragment data (81-dimensional in three-dimensional system), we can compute the Euclidean distance between the data with reduced dimensionality, m (m can be 3 or even smaller).

5.2 Nearest neighbor search using kd-tree

With the PCA technique, we can represent our fragment data in lower dimensional subspace and process the full matching procedure completely in this reduced subspace. But to process one fragment we still need to visit all the reduced fragment data that are transformed from the first member of every pair element in the whole database. In other words, the PCA technique we use reduces the dimensionality of each individual fragment data, but the number of the data remains unchanged. When we have large amount of pair

elements in database, visiting every first member's corresponding reduced data for each processing fragment is still considerably time consuming.

To tackle this problem and accelerate the procedure even further, one possible solution might be using the data reduction techniques to reduce the number of the elements in our database as well. But reducing redundancy or removing similar elements from the database would only give us an accelerated result that is proportional to the size of the problem, and beyond the reduction ratio limit the system would suffer from severe information loss. A better solution is to organize these elements properly such that only a small portion of the elements have to be visited during each search iteration. In fact, with this strategy, a tremendous amount of computational cost can be saved, and the accelerated result could reach to a logarithmic running time.

Following the explanation of the previous section, a two-dimensional fragment, 3×3 grid cells, can be considered as a point in 18-dimensional space (81 for three-dimensional fragment). After the projecting operation of the PCA approach, it becomes a point in m dimensional space. Thus, the full matching problem can be abstracted as: Given a database of n points in an m dimensional space, we want to find the nearest neighbor of a query point.

One of the earliest data structures proposed for this problem is the kd-tree, and it is still commonly used. Kd-tree is a useful data structure for searches involving a multidimensional search key, such as range searches and nearest neighbor searches. In our case, we use it for nearest neighbor search.

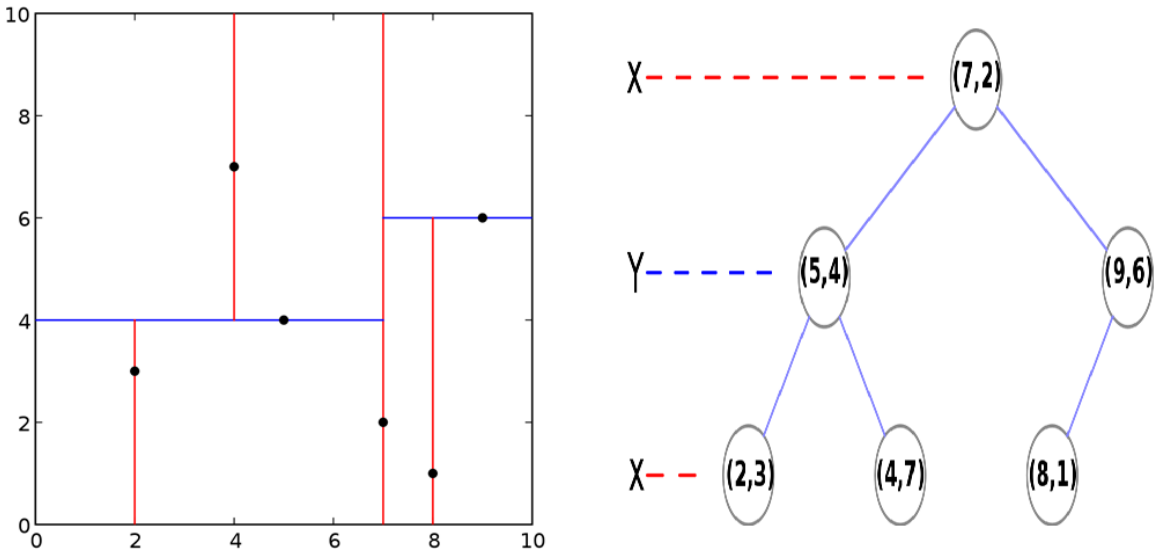

Figure 10 A 2-dimensional kd-tree example: on the left the way the plane is subdivided, and on the right the corresponding binary tree.

An *m* dimensional kd-tree (the "kd-tree" name's original meaning "k-dimensional tree" is lost already) is a binary tree in which each node stores a *m* dimensional point. At every non-leaf node, the set of points stored in the offsprings of this node are split into two subsets, by a hyperplane that goes through the point stored in this node. Points to the left of this hyperplane are stored in the left sub-tree of this node, while points to the right of the hyperplane are stored in the right sub-tree. The hyperplane's direction is decided in the following way: every node in the tree is associated with one of the *m* dimensions, with the hyperplane perpendicular to that dimension's axis. In Figure 10, a two-dimensional kd-tree example is shown. For more detailed description of the kd-tree structures and the supported range searches, readers can consult the textbook [1] on computational geometry.

Since there are many kd-tree libraries that support nearest neighbor search shared on internet, instead of implementing the algorithms by ourselves, we customized one

directly. For integrity, here we summarize the point insertion algorithm and the nearest neighbor search algorithm implemented by the source code we utilized, in a style of pseudo-code.

Algorithm 1 INSERT_POINT (*r*, *pos*, *data*, *dir*)

Input: A node *r* that indicates the root of the kd-tree (or sub-tree) we want to insert to. The position *pos* of the point, and the *data* that we want to store along with the point if any. The dimension *dir* associated with this root node, which indicates the direction of the splitting hyperplane and usually is set to 0 at the outermost call.

Output: The kd-tree with the new point inserted.

```
1. If r is null
2.     then make r a new node.*
3.          Store pos and data into r.
4.          Set r's associated dimension to dir.
5.          Set r's left child and right child to null.
6.     else new_dir ← (dir+1) modulo m (dimensionality of kd-tree).
7.          if in dimension dir, pos's coordinate < r's
8.             then INSERT_POINT(r's left child, pos, data, new_dir).
9.             else INSERT_POINT(r's right child, pos, data, new_dir).
```

* To make sure that the new created leaf node is connected to its parent node, the argument *r* is passed as a node pointer's pointer in C or C++ implementation.

This recursive algorithm adds a new point and its related data to a kd-tree as a new leaf node. It traverses the tree from the root to the leaf, moving to either the left or the right child depending on whether the new point is on the "left" or "right" side of each splitting hyperplane. Since we treat our reduced data as points in a m dimensional space, we can use this algorithm to build an m dimensional kd-tree, by inserting all these points and storing the corresponding fragment's index number along with each point.

More specifically, continuing last section's convention, at the preprocessing stage, after the PCA dimension reduction procedure we have m dimensional data, Ω_1, Ω_2, … , Ω_n, representing the fragments , Γ_1, Γ_2, … , Γ_n, which are actually the first members of all pair elements in database. Then, we consider each Ω_i as a point in an m dimensional space, and insert the point Ω_i (as the *pos* parameter) along with its index number i (as the *data* parameter) into a kd-tree using this INSERT_POINT algorithm. At the fluid generation stage, when we process a fragment that fits the full matching condition, by projecting this fragment Γ_k onto the same subspace, we get its corresponding m dimensional data Ω_k. Now, instead of visiting all the reduced fragment data, Ω_1, Ω_2, … , Ω_n, and calculating each Euclidean distance to Ω_k, to look for Ω_k's most similar data (the one that has the closest Euclidean distance) we can search for the nearest neighbor of this point Ω_k in our kd-tree structure using the SEARCH_NEAREST_NEIGHBOR algorithm described below, which only needs to visit a relatively smaller portion of the points. Once the algorithm outputs the node whose point Ω_l is nearest to Ω_k, the data stored at this node l tells us the corresponding fragment Γ_l's the index number, which is also the sequence number of the pair element that has Γ_l as the first member. Since the

pair elements are sequentially stored in database, we can locate the pair by that number in constant running time, and then use its second member as the following frame's corresponding fragment, which completes an iteration of the full matching procedure.

Algorithm 2 SEARCH_NEAREST_NEIGHBOR (*r*, *pos*, *rect*, *rn*, *rd*)

Input: A node *r* that indicates the root of the kd-tree (or sub-tree) we want to search from. The position *pos* of the given query point. A hyperrectangle *rect* that contains the information of every dimension's upper and lower bound coordinates of all the points stored in this kd-tree or sub-tree.

Output: The result, a node *rn* in the tree, whose point is nearest to the input query point. The distance between this node's point and the query point, *rd*.*

```
1. if in r's associated dimension, pos's coordinate < r's
2.     then near_child ← r's left child.
3.          far_child ← r's right child.
4.     else near_child ← r's right child.
5.          far_child ← r's left child.
6. if near_child is not null
7.     then copy rect for recovery.
8.          Update rect to fit near_child, by slicing the hyperrectangle
            with the splitting hyperplane associated with r.
9.          SEARCH_NEAREST_NEIGHBOR(near_child, pos, rect, rn, rd).
10.         Recover rect from the copy.
```

```
11.dist ← distance between pos and the point in r.
12.if dist < rd then rd ← dist, rn ← r.
13.if far_child is not null
14.   then copy rect for recovery.
15.        Update rect to fit far_child, by slicing the hyperrectangle
           with the splitting hyperplane associated with r.
16.        dist ← distance between pos and the hyperrectangle rect.
17.        if dist < rdist
18.           then SEARCH_NEAREST_NEIGHBOR(far_child, pos, rect, rn, rd).
19.        Recover rect from the copy.
```

* At the first call, we pass the resulting distance *rd* a number large enough to make sure the real assignment in Line 12 get performed.

This nearest neighbor search algorithm is also recursive. First, the algorithm checks the query point is on which side of the splitting hyperplane associated with the current root node. If the query point's coordinate is less than (/greater than or equal to) the root node in the split dimension, it finds the nearest neighbor in the left (/right) sub-tree recursively and saves it as the current best. Then, it calculate the root node's Euclidean distance to the query point, if it is closer than the current best, then it becomes the current best. Meantime, the algorithm maintains a hyperrectangle that bounds all the points in the current tree, by slicing the hyperrectangle with the splitting hyperplane and leaving the left side part for its left sub-tree and right side part for the right sub-tree. By calculating the distance from the query point to the other sub-tree's hyperrectangle, the

algorithm knows whether it is possible to have a nearer point in that sub-tree. If it is possible, the algorithm has to recursively find the nearest neighbor in the remaining sub-tree and updates the result if necessary. Otherwise, the entire branch of the remaining sub-tree of the current root node is eliminated, which actually makes this search algorithm highly efficient.

Since both the point insertion algorithm and the nearest neighbor search algorithm need to traverse the tree from root to leaf, the performance of these algorithms can be stabilized by reducing the depth of the tree. Obviously, a binary tree's depth is minimized when it is balanced. However, balancing a kd-tree is rather complicated. The tree rotation technique cannot be used, because nodes are sorted in multiple dimensions. Considering the condition that we do not need to change the tree any more once the search begins, we choose a particular order to insert the points such that the tree is kept balanced all the time, rather than balance the tree after the construction. The algorithm we designed to manage the inserting order, BUILD_KDTREE, is also described in the pseudo-code style as the following.

Algorithm 3 BUILD_KDTREE (*P*, *D*, *size*, *m*)

Input: An array *P* that consists of all the points' positions. An array *D* that contains the data to be stored along with these points.* The number of the points, *size*. The dimensionality of the points, *m*.

Output: A balanced kd-tree with all the points and data inserted.

1. Create an empty *m* dimensional kd-tree, *root*.

2. Randomly shuffle array *P* and adjust *D* accordingly.

```
3. Put (0, size-1) in waiting_list_1.
4. Set waiting_list_2 to empty.
5. level ← 0.
6. while waiting_list_1 is not empty
7.   do get two subscripts (begin, end) out from waiting_list_1.
8.      if begin = end
9.         then INSERT_POINT(root, P[begin], D[begin], 0).
10.        else median ← ⌊(begin + end) / 2⌋.
11.             PARTITION(P, D, begin, end, median, level modulo m).
12.             INSERT_POINT(root, P[median], D[median], 0).
13.             if median ≠ begin
14.                then put (begin, median-1) in waiting_list_2.
15.             if median ≠ end
16.                then put (median+1, end) in waiting_list_2.
17.     if waiting_list_1 is empty
18.        then swap waiting_list_1 and waiting_list_2.
19.             level ← level+1.
```

* In our system, array *D* stores the index numbers of the corresponding fragments.

The algorithm manages to build the kd-tree level by level, so that the tree is balanced from beginning to end. For each level, it picks up each subarray of the points in the waiting list (the list has one complete array at the beginning) and uses a PARTITION

algorithm to determine the median of the points according to their coordinates in the dimension associated with that level. The PARTITION algorithm also puts the points with smaller coordinates ahead of the median, the other points behind the median. Then, it inserts the median point into the kd-tree, and puts the two small subarrays (one ahead of the median, and one behind the median) into next level's waiting list. This way, as a median, each node has nearly equal numbers of points in its two sub-trees.

Algorithm 4 PARTITION (*P*, *D*, *begin*, *end*, *k*, *dir*)

Input: Two arrays *P* and *D*, same as the ones of BUILD_KDTREE algorithm. A range of array, with the first element's subscript, *begin*, and the last subscript, *end*. An integer *k* (*begin*≤ *k*≤ *end*). The dimension *dir* indicating which coordinates should be used when we compared different points.

Output: The subarray *P*[*begin*..*end*] rearranged, with all the elements ahead of *P*[*k*] having smaller coordinates than *P*[*k*] in dimension *dir*, and all the elements behind *P*[*k*] having greater or equal coordinates. The subarray *D*[*begin*..*end*] rearranged accordingly.

```
1. if begin = end then return.
2. x ← P[end]'s coordinate in dimension dir.
3. i ← begin-1, j ← end.
4. while i < j
5.   do repeat i ← i+1
6.      until in dimension dir P[i]'s coordinate ≥ x.
```

```
7.      repeat j ← j-1
8.      until i ≥ j or in dimension dir P[j]'s coordinate < x.
9.      if i < j then exchange P[i] with P[j].
10.                   exchange D[i] with D[j].
11.exchange P[i] with P[end].
12.exchange D[i] with D[end].
13.if i = k then return.
14.         else if i < k
15.                 then PARTITION(P, D, i+1, end, k, dir).
16.                 else PARTITION(P, D, begin, i-1, k, dir).
```

The median finding algorithm, PARTITION, is modified from a general selection algorithm. The rule for comparison is made flexible by a parameter, because we need to compare points by their coordinates in different dimensions. Selection can be done in linear time. The worst-case linear time selection algorithms, however, are rather complicated. We use an expected linear time algorithm instead, and shuffle all the points with a random permutation at the very beginning. The algorithm is modeled after the famous quicksort algorithm [70]. As in quicksort, it partitions the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, this algorithm works on only one side of the partition.

We know the expected running time of this selection algorithm, PARTITION, is $O(n)$ [70]. In the BUILD_KDTREE algorithm, PARTITION is called for all subarrays during each level's construction. No matter how points are partitioned already, the total

amount is not more than n, and therefore, for each level all the subarrays sum to $O(n)$. Since what we build is a balanced tree, it has $O(\log n)$ levels in total. Additionally, the random permutation shuffling costs linear time. Thus, the expected running time spent to rearrange the points is $O(n \log n)$. Apparently, for building the kd-tree, the INSERT_POINT algorithm is called with every point. For each insertion, INSERT_POINT costs as much time as ordinary binary search trees, which is $O(h)$ on a tree of height h. Again, since the tree is balanced, its height can be bounded by $O(\log n)$. Therefore, our balanced kd-tree for a set of n points can be constructed in expected time $O(n \log n)$. Obviously, since we only store each point and its associated data once, our kd-tree uses $O(n)$ storage.

On the other hand, this selection algorithm, PARTITION, has a worst-case running time $O(n^2)$ [70]. If we apply it in the BUILD_KDTREE algorithm, for building the top level of the tree, there is 1 array and it costs n^2 time units, at the second level there are 2 subarrays and each one costs $(n/2)^2$, at the third level there are 4 subarrays and each one costs $(n/4)^2$, and so on. They can be summed up as $n^2 + 2 \times (n/2)^2 + 4 \times (n/4)^2 + \ldots = n^2(1 + 1/2 + 1/4 + 1/8 + \ldots) < 2n^2$. So the worst-case running time for building our balanced kd-tree is $O(n^2)$. Actually, it is not very bad in our case, since kd-tree construction only takes a small portion of the overall processing time, and we randomly shuffle the points at the very beginning to reduce the possibility of that worst case. It is worth noting that there exists a more complicated selection algorithm whose running time is $O(n)$ in the worst case, and a good explanation is provided in the excellent textbook on algorithms [70].

For the nearest neighbor search algorithm, namely SEARCH_NEAREST_NEIGHBOR, in the case of randomly distributed points the expected running time is $O(\log n)$ [41]. However, the performance seems to degrade as the number of dimensions increases. In high dimensional spaces, suffering from the curse of dimensionality, the algorithm needs to visit many more branches than in lower dimensional spaces. The worst-case running time has been found by analyses of binary search trees [25]. For a k-dimensional kd-tree containing n points, the worst-case running time of the nearest neighbor search operation is

$$T_{worst-case} = O(k \cdot n^{1-\frac{1}{k}}). \qquad (5.5)$$

In particular, when the number of dimensions is very large, in the worst case the algorithm is only slightly better than a linear search of all of the points. Fortunately, in our case, with the benefit of the PCA approach, we can control the dimensionality of the data that is used for this search, and therefore, get improved worst-case results with data in a lower dimensional space.

Although we choose kd-tree because of its easier implementation and excellent cooperation with the PCA technique, it is worth noting that there are other solutions to this nearest neighbor search problem. Indyk and Motwani [78] performed approximate nearest neighbor search in high dimensions based on the concept of locality-sensitive hashing (LSH). The key idea is to use locality-sensitive hash function to map points to a discrete space where nearby points are likely to get hashed to the same value and far apart

points are likely to get hashed to different values. After that, Andoni and Indyk [79] improved LSH functions for the Euclidean distance. Since their method aims at extremely high dimensional data, it requires more than linear storage and the query time is worse than logarithmic.

5.3 Results

In our two-dimensional system, following the setup described in Chapter 3, we set N to 32, so our computational grids are 34×34 including the extra layer. We collected 600 frames of the same size simulation as the example data by Stam's fluid solver [2]. Therefore, there were $32\times32\times(600-1)=613{,}376$ pair elements constructed in the database. Given the initial frame, we synthesized 599 frames of fluid flow one after another. Figure 11 gives a validation of our synthesized fluid flow. The images on the left are different frames of the velocity field we generated. For comparison, on the right we put the corresponding frames that were obtained from the fluid solver directly. As shown in the figure, after nearly 600 frames' fluid generation, it is still hard to tell the difference between our synthesized results and the ones calculated directly by the fluid solver. In this case, the initial state we used for generating the fluid is the same as the one we employed to collect our example data. Figure 12 shows another two-dimensional result.

Figure 13 and Figure 14 show the three-dimensional results in a similar condition. In three-dimensional system, we set N to 10, so including the extra layer the computational grids become $12\times12\times12$. We collected 600 frames of the same size simulation as the example data using a 3D fluid solver. So there were $10\times10\times10\times(600-1)=599{,}000$ pair elements in the database.

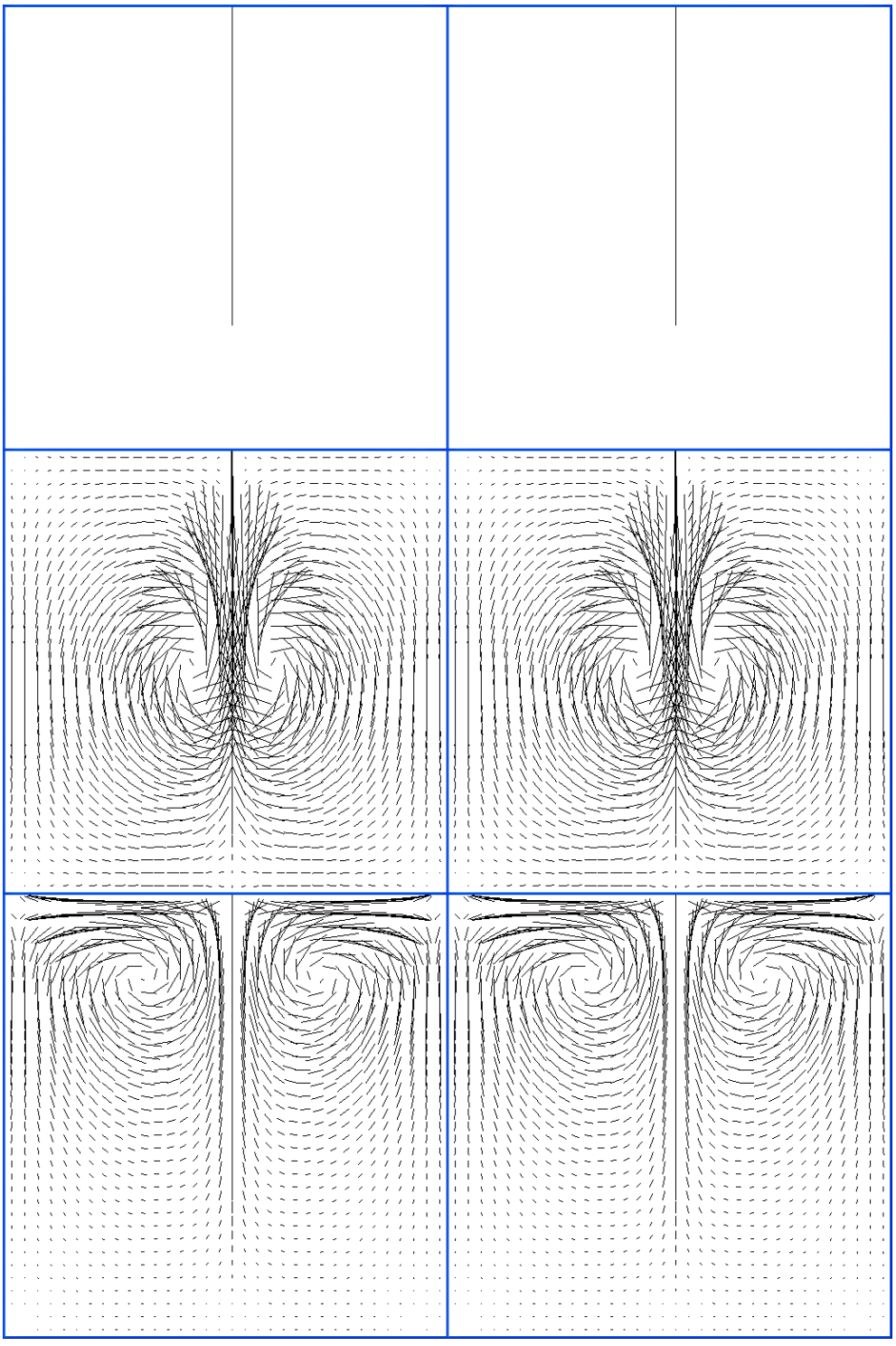

Figure 11 Validation of the synthesized fluid flow in 2D. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 100th, and 600th frames. The initial states of both sides were set identical.

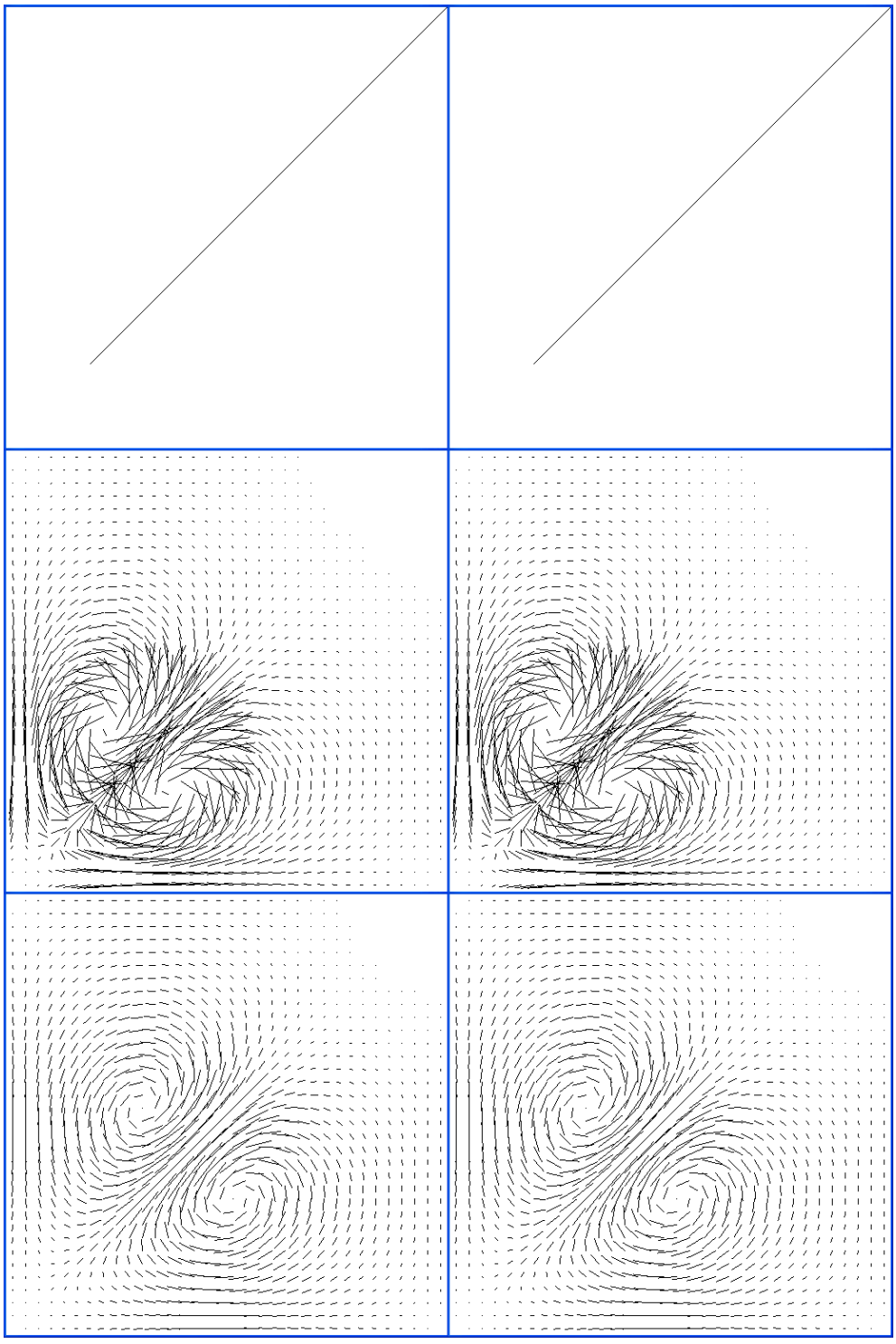

Figure 12 Another validation of the synthesized fluid flow in 2D with a different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 100th, and 600th frames. The initial states of both sides were set identical.

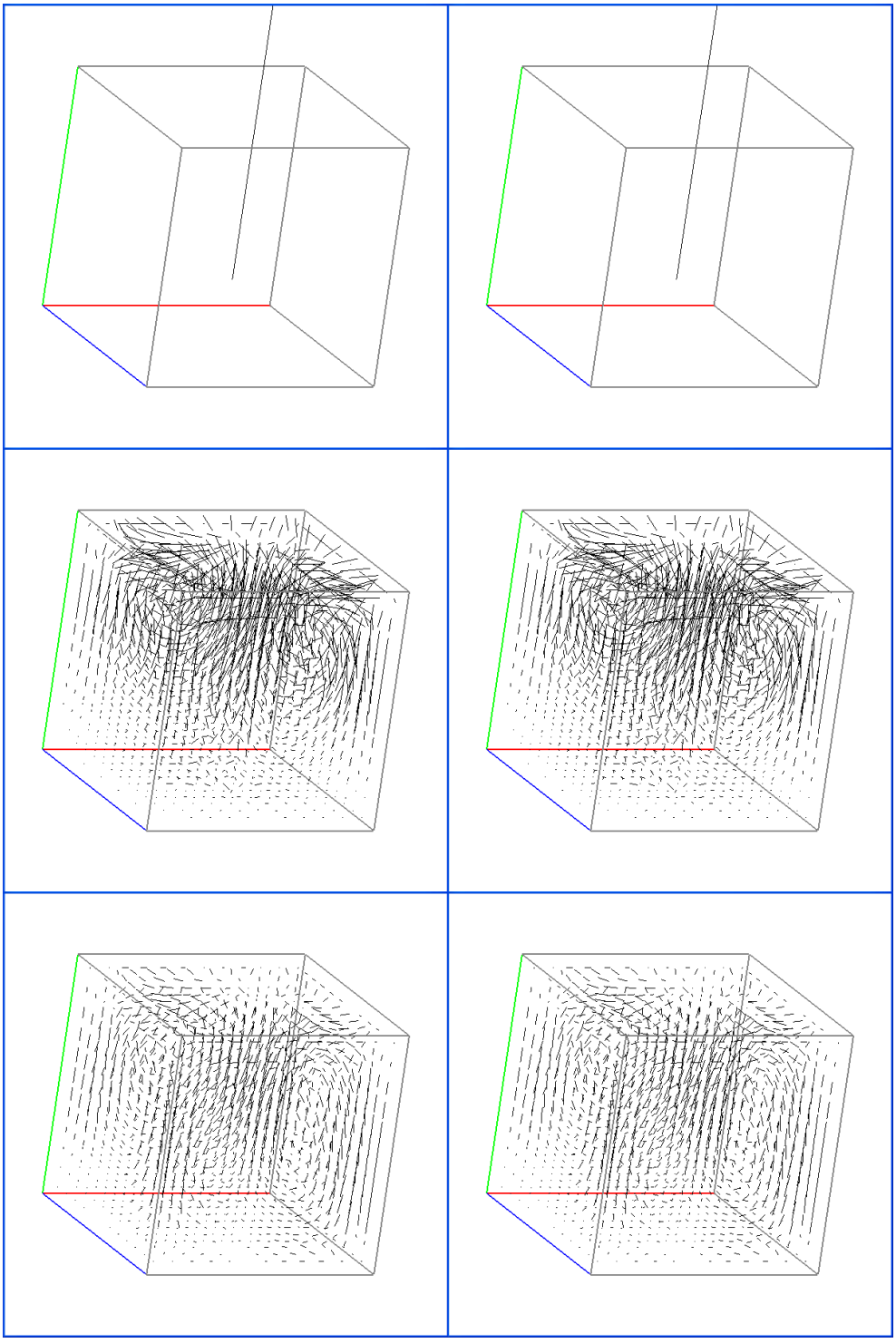

Figure 13 Validation of the synthesized fluid flow in 3D. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 30th, and 100th frames. The initial states of both sides were set identical.

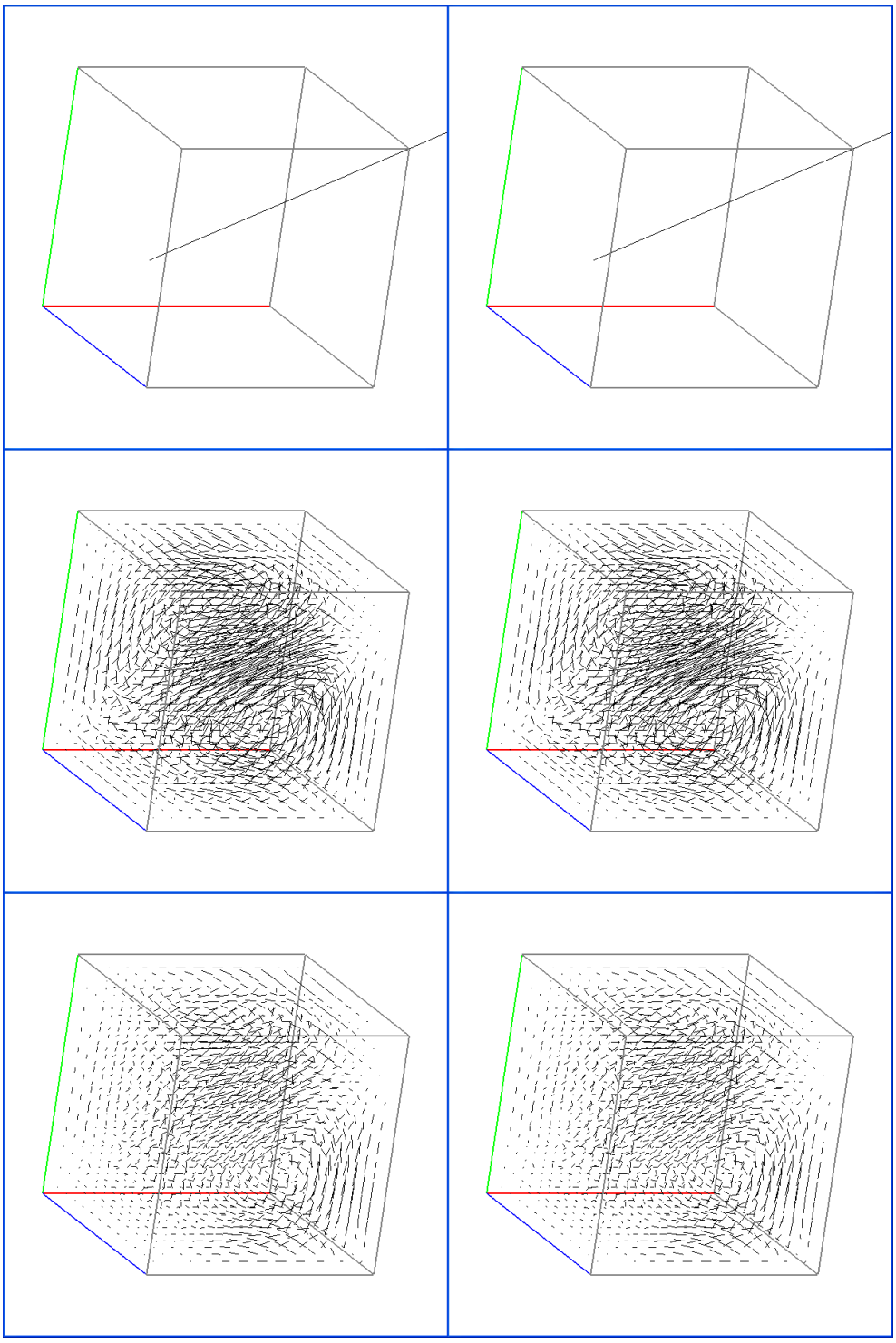

Figure 14 Another validation of the synthesized fluid flow in 3D with a different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 30th, and 100th frames. The initial states of both sides were set identical.

In Table 1 we list the computational times in seconds for both preprocessing and fluid generation. For comparison, we collected the times consumed when our system has different settings: with full acceleration approaches, namely the PCA technique and the kd-tree structure; with the PCA technique only; and without any acceleration. From the table, we can see that although the PCA approach and the kd-tree structure cost a little extra time in preprocessing, they played a significant part in accelerating the procedure of fluid generation. For example, in our two-dimensional system without any acceleration techniques, to exhaustively visit the elements in database and calculate the Euclidean distances in full space, it cost more than 13 hours CPU time on a standard PC hardware, an AMD Opteron 2.6GHz processor with 2GB RAM. With the PCA approach only, by calculating the distances in a reduced subspace, the cost was decreased to less than 2.5 hours CPU time. When both PCA approach and the kd-tree structure were employed, we got the results within 6.3 minutes of CPU time.

System dimensions	Grid size	Reduced data dimensions	PCA and Kd-tree	With PCA only	Without Acceleration
2D	34×34	from 18 to 2	8.8 / 377.2	3.4 / 8765.4	0.1 / 47239.3
3D	$12\times12\times12$	from 81 to 3	103.2 / 805.1	94.9 / 7457.6	0.8 / 195565.9

Table 1 Computational times for preprocessing and fluid generation with different settings. The "Reduced data dimensions" column shows the number of dimensions we reduced to when using the PCA approach. All times are in seconds. The time before slash shows the time for preprocessing, while the one behind slash shows the time for fluid generation.

It is worth noting that the two-dimensional examples shown in Figure 11 and Figure 12 used PCA approach to project the 18-dimensional fragment data onto a 2-dimensional subspace, as indicated in the column "Reduced data dimensions" in Table 1. Similarly, the three-dimensional results shown in Figure 13 and Figure 14 used PCA to reduce the 81-dimensional fragment data to 3 dimensions. Further reduction will result in quality degradation, and cause severe visual effect. Figure 15 and Figure 16 provide the comparisons concerned in two and three-dimensional systems respectively.

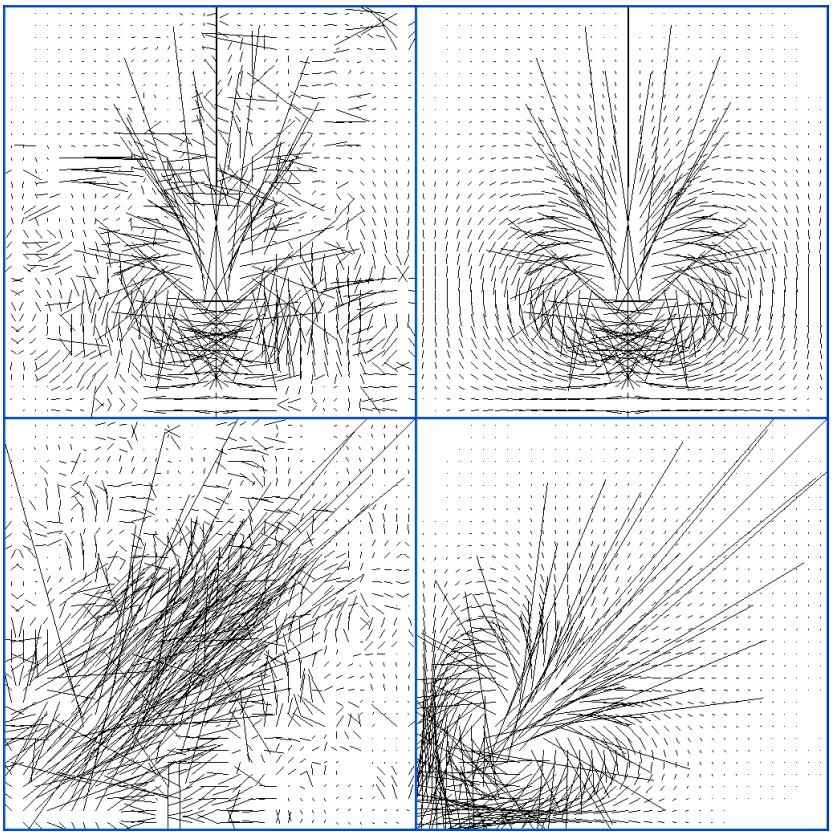

Figure 15 Quality degradation in 2D system when fragment data is reduced by PCA. On the left the frames of velocity field were generated with fragment data reduced to 1 dimension; on the right the frames were synthesized with fragment data reduced to 2 dimensions. The top two frames were generated from the initial state shown in Figure 11; the bottom two frames share the initial state shown in Figure 12. All of them are the 10th frames.

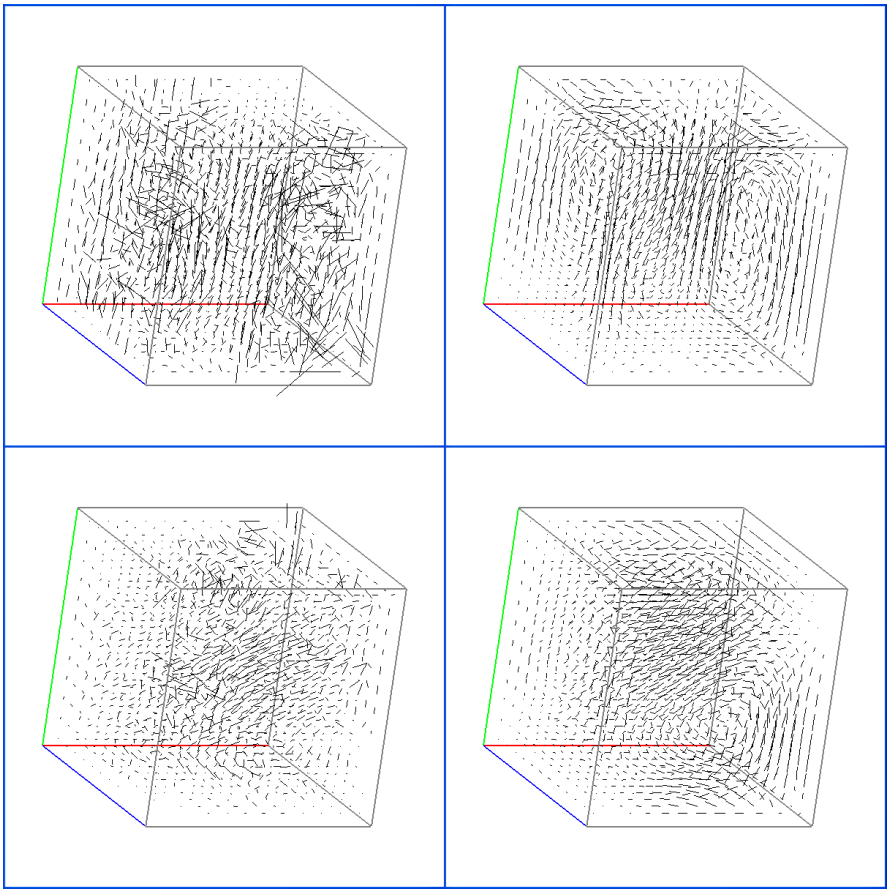

Figure 16 Quality degradation in 3D system when fragment data is reduced by PCA. On the left the frames of velocity field were generated with fragment data reduced to 2 dimensions; on the right the frames were synthesized with fragment data reduced to 3 dimensions. The top two frames were generated from the initial state shown in Figure 13; the bottom two frames share the initial state shown in Figure 14. All of them are the 100th frames.

6 Chapter: Conclusions

We have presented a novel method for example-based simulation of fluid flow. This method synthesizes fluid flow by reconstructing from the input fluid motion examples. Although our method does not limit to any particular sources, in our system the fluid motion example data, namely sequence of velocity field, is collected from a physically based fluid solver.

Given the fluid motion example data, a series of frames of velocity field, to capture the properties of local fluid behavior, we decompose the example data into fragment pairs, which are actually fluid motions of smaller size and shorter length, with the first fragment standing for previous status and the second fragment standing for the successive status. We construct a database by storing these fragment pairs sequentially so that every element can be accessed by its index number in constant running time.

Given an initial or prior fluid frame of the velocity field, our system can generate the following frame, fragment by fragment according to the database. To make this happen, we overcame the multivalued function problem by choosing a proper fragment processing order and developed two pattern matching schemes, “full matching” and “partial matching”, to handle different kinds of situations when querying the database for the following frame’s information. Then, the system can repeat this procedure and generate the frames one after another until the desired number of frames is achieved.

As “full matching” is needed for most of the time when querying the database, and it consumes dominant amount of the computational time, we make use of two

techniques, PCA and kd-tree, to accelerate this procedure. As "full matching" concerns fixed-dimension data comparison, we first utilize the PCA approach to project the elements of our database onto a lower dimensional subspace. Then, we build a kd-tree of this lower dimensionality, treat element's corresponding lower dimensional data as a point in the subspace, and insert the database index number of each element along with its corresponding point into the kd-tree. So when we need to do "full matching" of a given fragment, instead of visiting and comparing every element in the database, we can just project this fragment onto the same subspace, and find the target element's database index number by performing the nearest neighbor search on the kd-tree.

In Chapter 5, we have shown the validation of our method in synthesizing both two-dimensional and three-dimensional fluid's velocity fields.

6.1 Contribution

The following contributions were made as the result of performing the research described in the thesis:

- We developed a novel method for fluid simulation, by which fluid motion could be synthesized base on existing example data. This example based method successfully avoids solving the complicated equations of fluid dynamics, and still produces acceptable fluid flow in certain circumstances. To tackle the multivalued function problem involved in this method, we also developed a particular processing order and two different matching schemes.
- We designed an efficient pattern matching technique by coupling the PCA approach and kd-tree data structure together. With the principle components

identified by the PCA approach, the pattern matching problem is reduced into a lower dimensional subspace. Then, by organizing the reduced data into a balanced kd-tree, best match can be found by performing the nearest neighbor search operation on the tree, which can be done in logarithmic expected time. Furthermore, the operation's worst-case running time, which is sensitive to number of dimensions, is also improved, since the data is reduced to lower dimensions by the PCA approach.

6.2 Discussion

As shown in the last section of Chapter 5, if the initial state given to trigger our fluid generation process is the same as the one we applied to collect the example data on the fluid solver, our system can achieve the result as if it was obtained by the fluid solver directly. Therefore, it is naturally proven that our result satisfies the equations of fluid dynamics (i.e. the Navier-Stokes equations) if the example data is calculated properly by the fluid solver.

It is worth noting that our fluid generation method does not limit to use any particular fluid solver or even use a solver at all to collect the example fluid motion data. We choose to use Stam's fluid dynamics solver [2] [68] to calculate the example data only for the sake of simplicity. Although our method's efficiency is not comparable to this fluid solver's direct simulation, there are other simulation methods that can approximate the equations of fluid dynamics more accurately, and accordingly consume more resources. Furthermore, some complex fluid-like phenomena (e.g. due to their unknown physical or chemical properties) may be too hard to be simulated in ordinary

ways. Considering our method's specialty, with these phenomena recorded and their motion data (i.e. frames of velocity field) extracted, we believe that our method could be a promising solution.

On the other hand, our system still has some limitations. As expected from a data-driven technique, there are errors when the input initial state is different from the one we applied to collect the example data. Figure 17 shows a comparison between our result and the one simulated directly by the fluid solver, when the input initial state deviates from the one we use to collect the example data. In this case, we translated the nonzero-velocity cell of the initial state along x-axis and reduced its magnitude slightly. If we translated the cell along y-axis or rotated the velocity direction for some angle, the errors exist as well. These errors occur simply because the database we constructed with the example data does not contain enough information to generate the fluid flow from a different initial state. Thus, for this reason, when the database is constructed with more various fluid motion example cases, the errors should become smaller. The results shown in Figure 18 can validate this conclusion. With the input initial state unchanged (same as in Figure 17), when the database is constructed with one more fluid motion example that is collected with a more similar initial sate to the input (with the nonzero-velocity cell only slightly translated along x-axis and its magnitude unchanged), the errors in the generated frames decreased visually.

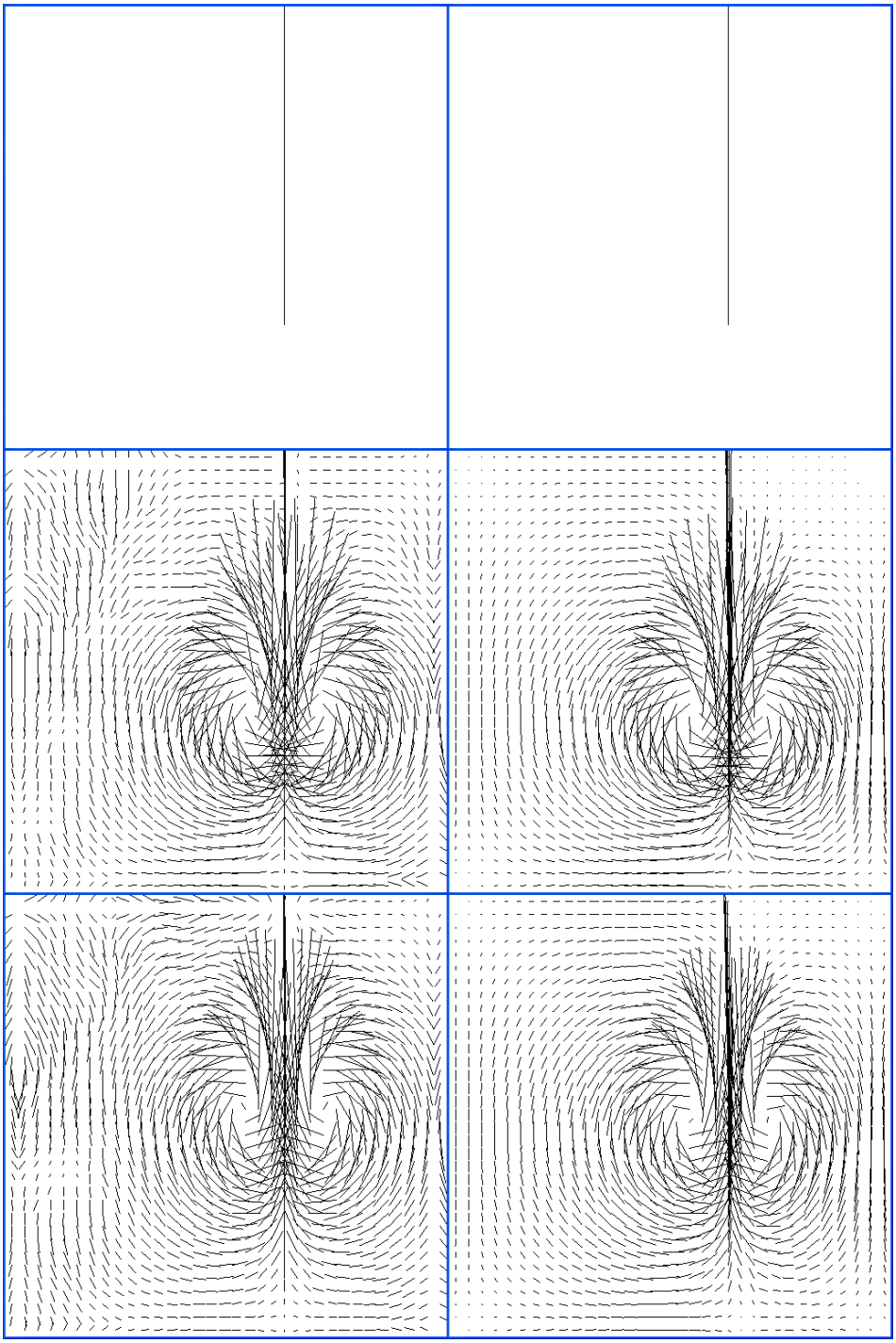

Figure 17 Errors occurred with different initial state. On the left the frames of velocity field were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 50th, and 100th frames. The initial states of both sides were set identical, but they are different from the one we applied to collect the example data, shown in Figure 7, with the nonzero-velocity cell translated along *x*-axis and its magnitude reduced slightly.

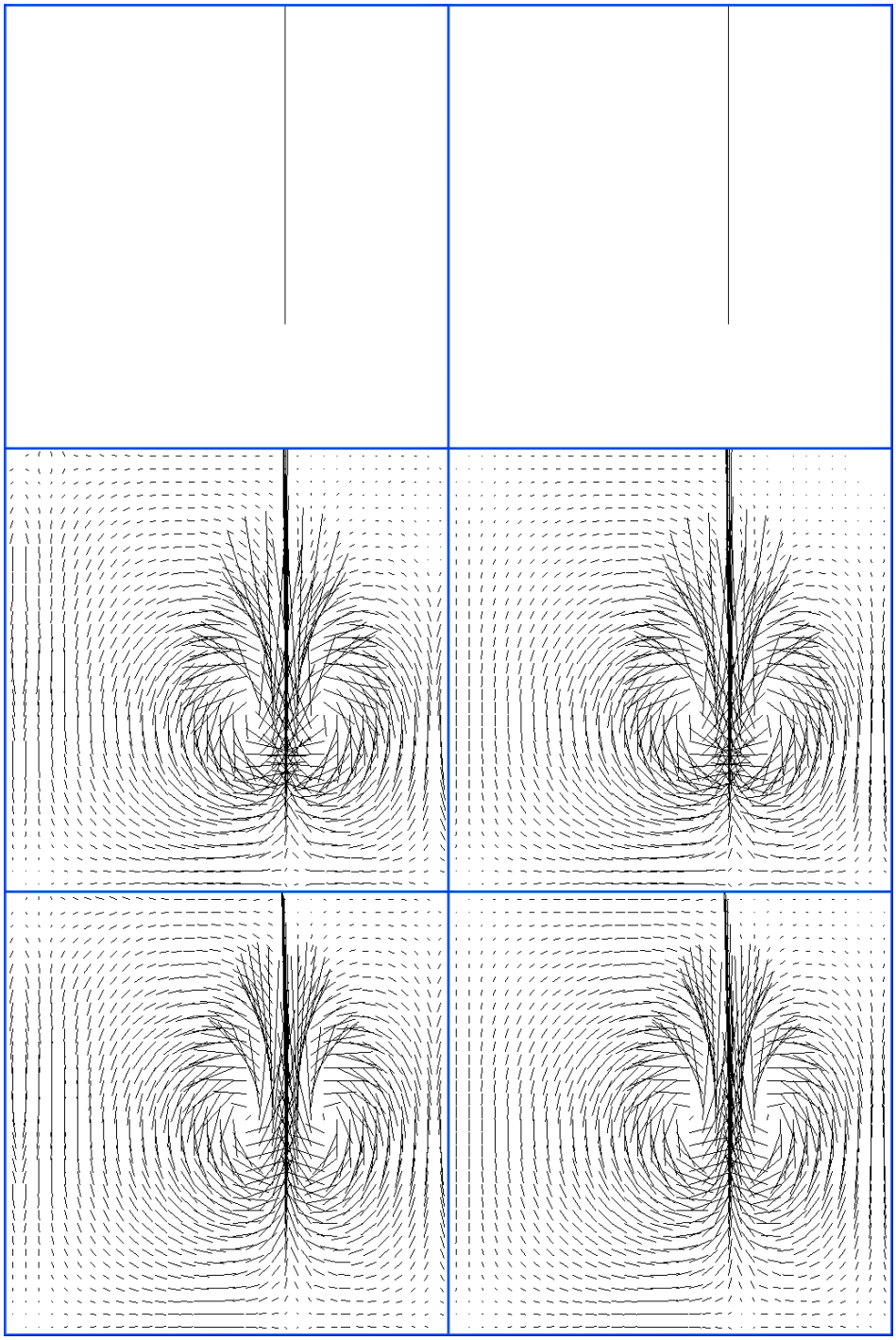

Figure 18 **Errors become smaller when more fluid motion example cases are included into database. On the left the frames were synthesized by our method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states, 50th, and 100th frames. The initial states of both sides were set identical to the one in Figure 17. The newly included fluid example has a more similar initial state, with the nonzero-velocity cell only slightly translated.**

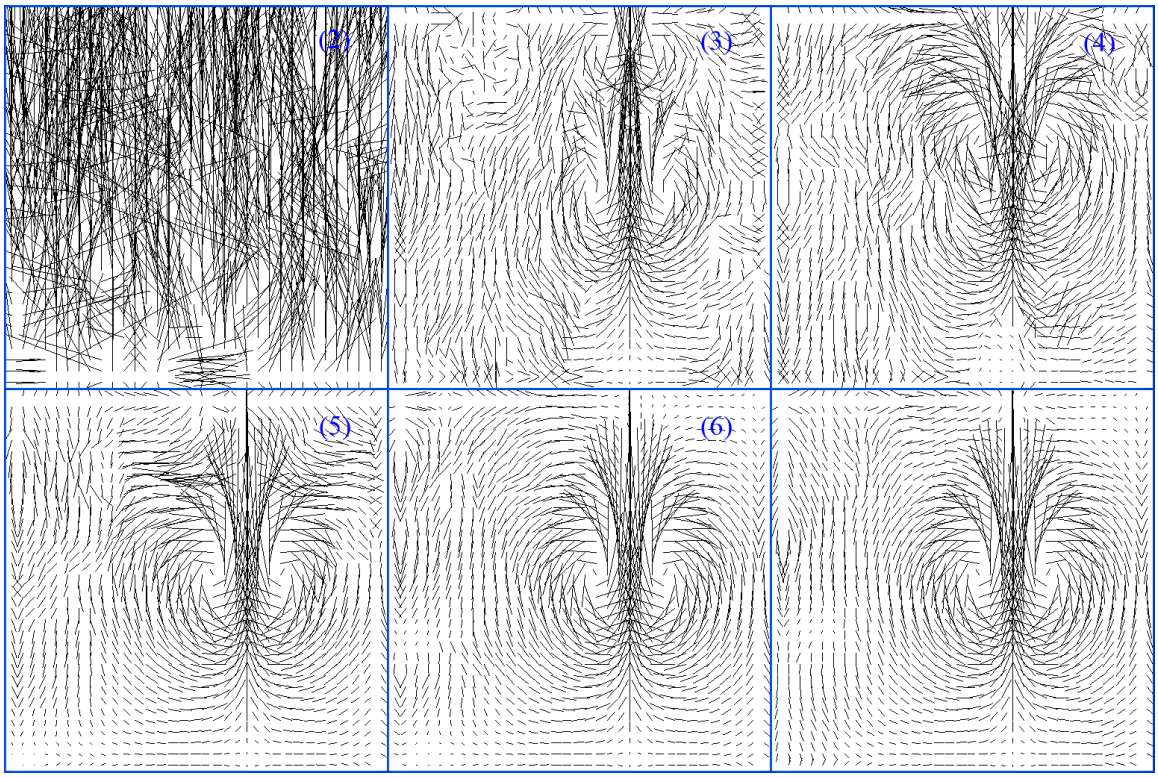

Figure 19 Frames generated with different dimension reduction ratios. All frames were generated from the same initial state and database as in Figure 17, and they are all the 100th frames. The number displayed in upper right corner of the frame indicates the dimensions PCA reduced to. For comparison, the frame generated without dimension reduction is provided at the lower right spot, identical to the bottom left image in Figure 17.

Another thing worth mentioning about the system is that the number of the dimensions to which the fragment data is reduced with the PCA technique cannot be arbitrarily set. As described in the result section of the previous chapter, our experiments indicate that when the input initial state is identical to the one of the example fluid, we can reduce our 18-dimensional fragment data to 2 dimensions in our two-dimensional system, and 81-dimensional data to 3 dimensions in three-dimensional system. Further reduction will lead to information loss, and cause severe visual effects in the result.

Experiments also indicate that when the input initial state deviates from the one of the example fluid, to generate fluid flow with smallest errors, the number of dimensions to which we reduce needs to be increased. Figure 19 depicts this trend with results of different dimension reduction ratios.

6.3 Future Research

The research documented in this thesis lays the groundwork for several areas of the extended future research.

As discussed in the previous section, if the input initial state is not included in the collected example data, this system will produce some errors in its generated fluid flow. In order to synthesize variable fluid flow of different initial states with acceptable reduced errors and still avoid exploding our database, more diverse and representative fluid motion examples should be collected. Therefore, how to define that a fluid motion example set is more diverse or representative than another, and how to obtain such an example set, become a possible future research topic.

With the collected example data extended, the database structure and the matching criterion also need to be extended. For the purpose of distinguishing the locations of the nonzero-velocity cell (force) in different initial states, one possible solution is to extend the dimensionality of each fragment data and include its center cell's coordinates in the extra dimensions. In the matching method, the Euclidean distance calculation should involve the extra dimensions as well, with certain coefficient tuned until acceptable result can be produced. However, for the initial states with multiple

nonzero-velocity cells (forces), a new extended solution remains to be found in the future research.

Figure 20 shows an example of the current method's limitation when the initial state has multiple nonzero-velocity cells (forces) of the same value. Sampled substeps of this example are shown in Figure 21. Since the fragment pairs stored in the database do not have the information of the location in their original velocity field, the current method would have identical treatments for the areas around the two nonzero-velocity cells of the initial state: the first substep updating for the left spot (from the first image to the second) is similar to the second substep updating for the right spot (from the second image to the third); the third substep updating for the left spot (from the third image to the fourth) is similar to the fourth substep updating for the right spot (from the fourth image to the fifth); and so on. However, due to the different locations of the two nonzero-velocity cells in the initial state, these two cells (and the cells around them) should have different status in the following frame, as shown at the bottom right of Figure 20.

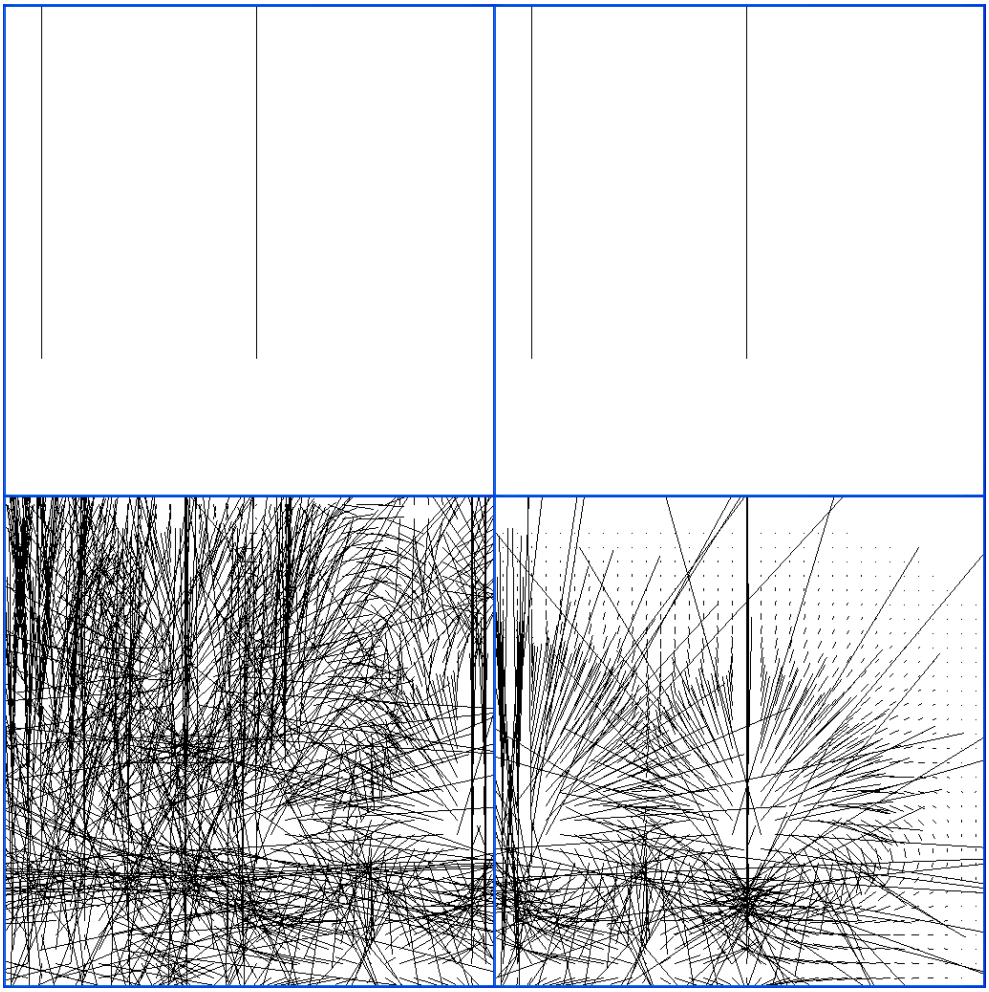

Figure 20 **A limitation of the current method when the initial state has multiple nonzero-velocity cells (forces) of the same value. On the left the frames of velocity field were synthesized by the current method; on the right the frames were obtained from fluid solver directly. From top to bottom, they are the initial states and 2nd frames. The initial states of both sides were set identical, and there are two nonzero-velocity cells with the same vector value in the initial state.**

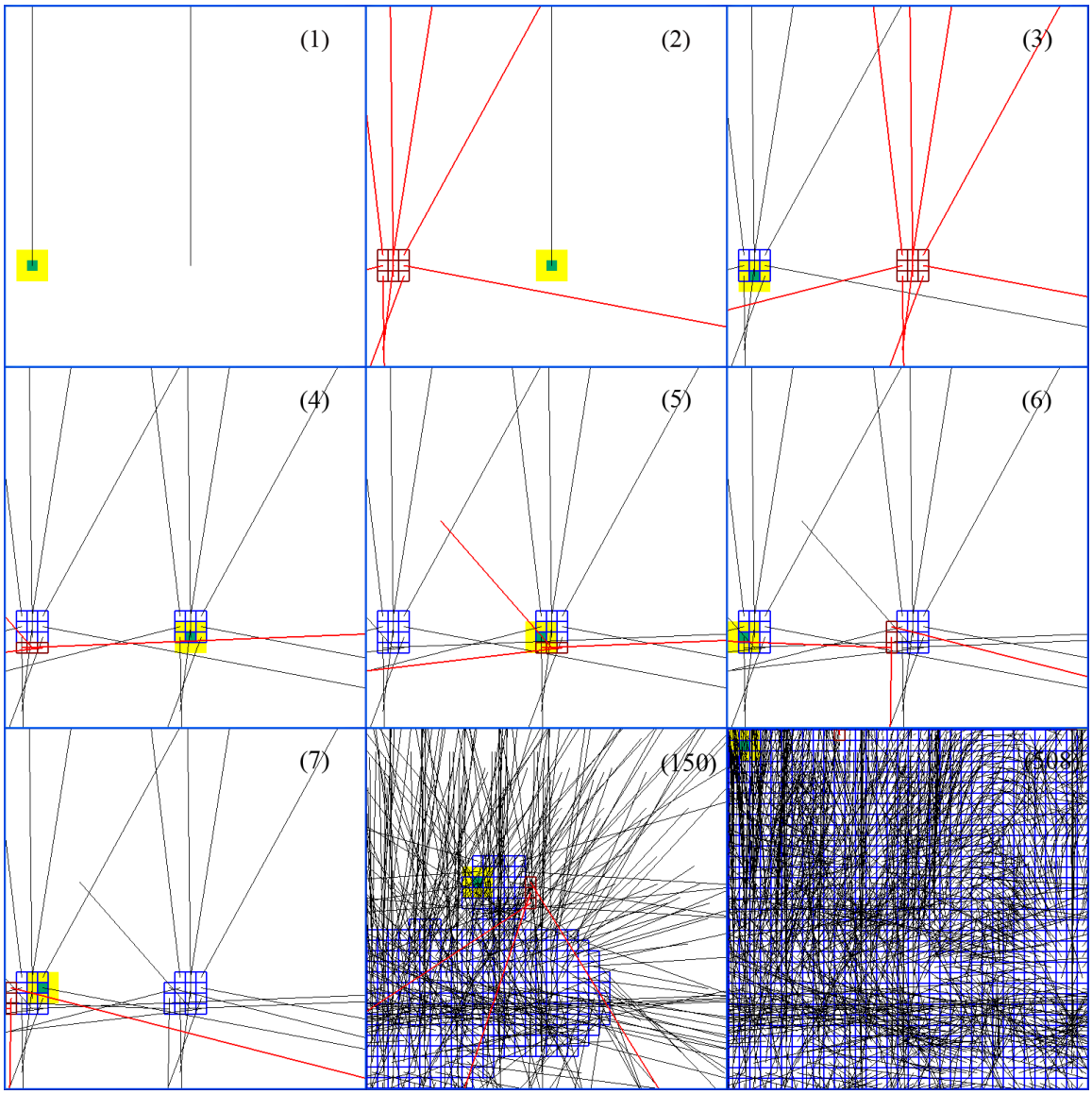

Figure 21 Sampled substeps from the initial state to the next frame for the example shown in Figure 20. The serial number is displayed in the upper right corner of each image. The grid cells and the line segments (velocity vectors) are highlighted by different colors or patterns under the same convention as in Figure 9.

Since our method's time complexity is related to the size of the constructed database, in order to make the database query even more efficient, another future research topic may focus on how to reduce redundancy and remove similar elements from the

database. Moreover, for the full matching procedure, because the processing order of the fragments does not affect the result, parallel computation may also be applied to boost the generating speed.

References

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications, Third Edition*, Springer, 2008.

[2] J. Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH '99)*, p.121-128, July 1999.

[3] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, July 23-26, 2002.

[4] O. Arikan and D. A. Forsyth. Interactive motion generation from examples. In *ACM Transactions on Graphics (TOG)*, v.21 n.3, July 2002.

[5] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. In *ACM SIGGRAPH Computer Graphics*, v.24 n.4, p.49-57, August 1990.

[6] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques (SIGGRAPH '93)*, p.369-376, 1993.

[7] M. Müller, D. Charypar, and M. Gross. Particle-Based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH / Eurographics symposium on Computer animation*, July 26-27, 2003.

[8] N. Foster and D. Metaxas. Realistic animation of liquids. In *Graphical Models and Image Processing*, v.58 n.5, p.471-483, Sept. 1996.

[9] V. Blanz and T. Vetter. A Morphable Model for the Synthesis of 3D Faces. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, p.187-194, July 1999.

[10] P. Xi, W.-S. Lee, and C. Shu. A Data-driven Approach to Human-body Cloning using a Segmented Body Database, In *Proceedings of Pacific Graphics* (PG 2007), p.139-147, Maui, Hawaii, October 29-30 & November 1-2, 2007.

[11] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra. Texture optimization for example-based synthesis, *ACM Transactions on Graphics* (TOG), v.24 n.3, July 2005.

[12] B. Allen, B. Curless, and Z. Popovic. Articulated body deformation from range scan data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, July 23-26, 2002.

[13] S. Koshizuka, H. Tamako, and Y. Oka. A particle method for incompressible viscous flow with fluid fragmentation. *Computational Fluid Dynamics Journal*, v.4 n.1, p.29-46, 1995.

[14] P. Nugjgar, T. Fujimoto, and N. Chiba. Markov-Type Velocity Field Designed for Endless Animation of Water Stream. In *Computer Graphics International* (CGI 2011), June 12-15, 2011.

[15] S. Premože, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker. Particle-based simulation of fluids. In *Computer Graphics Forum*, v.23 n.3, p.401-410, 2003.

[16] J. Tan and X. Yang. Physically-based fluid animation: A survey. *Science in China Series F: Information Sciences*, v.52 n.5, pages 723-740, 2009.

[17] R. Bridson and M. Müller-Fischer. Fluid simulation: SIGGRAPH 2007 course notes. In *ACM SIGGRAPH 2007 courses (SIGGRAPH '07)*, p.1-81, 2007.

[18] J. D. Anderson. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, 1995.

[19] A. J. Chorin and J. E. Marsden. *A mathematical introduction to fluid mechanics*. Springer-Verlag, 1990.

[20] D. R. Peachey. Modeling waves and surf. *ACM SIGGRAPH Computer Graphics*, v.20 n.4, p.65-74, August 1986.

[21] Y. Liu, X. Liu, H. Zhu, and E. Wu. Physically based fluid simulation in computer animation. *Journal of Computer Aided Design & Computer Graphics*, 2005.

[22] A. Fournier and W. T. Reeves. A simple model of ocean waves. *ACM SIGGRAPH Computer Graphics*, v.20 n.4, p.75-84, August 1986.

[23] J. Tessendorf. Simulating ocean water. In *Simulating Nature: Realistic and Interactive Techniques*. *SIGGRAPH 2001 Course Notes 47*, 2001.

[24] F. H. Harlow and J. E. Welch. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with a Free Surface. *The Physics of Fluids 8*, p.2182-2189, 1965.

[25] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, v.9 n.1, p.23–29, 1977.

[26] J. J. Stoker. *Water waves: the mathematical theory with applications*. Wiley, 1992.

[27] G.D. Crapper. *Introduction to water waves*. E. Horwood, 1984.

[28] J. X. Chen, N. V. Lobo, C. E. Hughes, and J. M. Moshell. Real-Time Fluid Simulation in a Dynamic Virtual Environment. *IEEE Computer Graphics and Applications*, v.17 n.3, p.52-61, May 1997.

[29] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, v.11 n.2, p.215-234, March 1967.

[30] S. Osher and R. P. Fedkiw. *Level-set methods and dynamic implicit surfaces*. Springer, 2003.

[31] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, p.23-30, August 2001.

[32] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, July 23-26, 2002.

[33] V. Mihalef, D. Metaxas, and M. Sussman. Textured liquids based on the marker level set. *Computer Graphics Forum*, v.26 n.3, p.457-466, 2007.

[34] W. T. Reeves. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics (TOG)*, v.2 n.2, p.91-108, April 1983.

[35] G. Miller and A. Pearce. Globular Dynamics: A Connected Particle System for Animating Viscous Fluids. *Computers and Graphics*, v.13 n.3, p.305-309, 1989.

[36] J. F. O'Brien and J. K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation*, p.198-208, April 19-21, 1995.

[37] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki. Realistic Animation of Fluid with Splash and Foam. *Computer Graphics Forum*, v.22 n.3, p.391-400, 2003.

[38] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual review of Astronomy and Astrophysics*, v.30, p.543-574, 1992.

[39] J. Stam and E. Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, p.129-136, September 1995.

[40] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of Fluids with Deformable Solids. *Computer Animation and Virtual Worlds*, v.15 n.3-4, p.159-171, July 2004.

[41] J. L. Bentley, J. H. Friedman, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, v.3 n.3, p.209-226, 1977.

[42] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, v.19 n.7, p.444-456, 2003.

[43] S. Chen and G. D. Doolean. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, v.30, p.329-364, 1998.

[44] Z. Yu and L.-S. Fan. Lattice Boltzmann method for simulating particle-fluid interactions. *Particuology*, v.8 n.6, p. 539-543, December 2010.

[45] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors.

International Conference on Parallel Processing, p.550-557, September 22-25, 2009.

[46] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, July 26-27, 2003.

[47] X. Wei, W. Li, K. Mueller, and A. E. Kaufman. The Lattice-Boltzmann Method for Simulating Gaseous Phenomena. *IEEE Transactions on Visualization and Computer Graphics*, v.10 n.2, p.164-176, March 2004.

[48] W. Li, Z. Fan, X. Wei, and A. Kaufman. GPU-based flow simulation with complex boundaries. *GPU Gems 2*, Addison-Wesley, p.747–764, 2005.

[49] J. Kim, D. Cha, B. Chang, B. Koo, and I. Ihm. Practical animation of turbulent splashing water. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, September 02-04, 2006.

[50] N. Thürey, F. Sadlo, S. Schirm, M. Müller-Fischer, and M. Gross. Real-time simulations of bubbles and foam within a shallow water framework. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, August 02-04, 2007.

[51] J.-M. Hong, H.-Y. Lee, J.-C. Yoon, and C.-H. Kim. Bubbles alive. *ACM Transactions on Graphics (TOG)*, v.27 n.3, August 2008.

[52] A. Selle, N. Rasmussen, and R. Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Transactions on Graphics (TOG)*, v.24 n.3, July 2005.

[53] B. Kang, Y. Jang, and I. Ihm. Animation of chemically reactive fluids using a hybrid simulation method. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, August 02-04, 2007.

[54] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-Way Coupled SPH and Particle Level Set Fluid Simulation. *IEEE Transactions on Visualization and Computer Graphics*, v.14 n.4, p.797-804, July 2008.

[55] A. C. Antoulas, D. C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. *Contemporary Mathematics*, v.280, p.193-219, 2001.

[56] J. L. Lumley. Stochastic Tools in Turbulence. vol. 12 of *Applied Mathematics and Mechanics*, Academic Press, New York, 1970.

[57] A. Treuille, A. Lewis, and Z. Popovic. Model reduction for real-time fluids. *ACM Transactions on Graphics (TOG)*, v.25 n.3, July 2006.

[58] S. Chenney. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH / Eurographics symposium on Computer animation*, August 27-29, 2004.

[59] N. Thürey, C. Wojtan, M. Gross, and G. Turk. A multiscale approach to mesh-based surface tension flows. In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2010*, v.29 n.4, 2010.

[60] M. Wicke, M. Stanton, and A. Treuille. Modular bases for fluid dynamics. *ACM Transactions on Graphics (TOG)*, v.28 n.3, August 2009.

[61] A. Lamouret and M. Panne. Motion synthesis by example. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, p.199-212, December 1996.

[62] L. M. Tanco and A. Hilton. Realistic synthesis of novel human movements from a database of motion capture examples. In *Proceedings of the Workshop on Human Motion (HUMO'00)*, p.137, December 07-08, 2000.

[63] J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. S. Pollard. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics (TOG)*, v.21 n.3, July 2002.

[64] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human face. *Optical Society of America*, v.4, p.519-524, 1987.

[65] M. Kirby and L. Sirovich. Application of the Karhunen-Loeve Procedure for the Characterization of Human Faces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v.12 n.1, p.103-108, January 1990.

[66] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, v.3 n.1, p.71-86, Winter 1991.

[67] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys (CSUR)*, v.35 n.4, p.399-458, December 2003.

[68] J. Stam. Real-Time Fluid Dynamics for Games. In *Proceedings of the Game Developer Conference*, March 2003.

[69] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 2007.

[70] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[71] M. Lentine, M. Aanjaneya, and R. Fedkiw. Mass and momentum conservation for fluid simulation. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '11)*, Stephen N. Spencer (Ed.). ACM, New York, NY, USA, p.91-100, 2011.

[72] N. Chentanez and M. Müller. Real-time Eulerian water simulation using a restricted tall cell grid. In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2011*, v.30 n.4, 2011.

[73] B. Solenthaler and M. Gross. Two-scale particle simulation. In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2011*, v.30 n.4, 2011.

[74] M. Gleicher, H. J. Shin, L. Kovar, and A. Jepsen. Snap-together motion: assembling run-time animations. In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003*, v.22 n.3, July 2003.

[75] L. Ikemoto and D. A. Forsyth. Enriching a motion collection by transplanting limbs. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '04)*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, p.99-108, 2004.

[76] L. Ren, A. Patrick, A. A. Efros, J. K. Hodgins, and J. M. Rehg. A data-driven approach to quantifying natural human motion. In *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005*, v.24 n.3, p.1090-1097, 2005.

[77] I. K. Fodor. A survey of dimension reduction techniques. *Technical Report UCRL-ID-148494*, Lawrence Livermore National Laboratory, 2002.

[78] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of computing (STOC '98)*, p.604-613, 1998.

[79] A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Near Neighbor Problem in High Dimensions. In *Proceedings of the Symposium on Foundations of Computer Science* (FOCS'06), 2006.