EDITING EXPLOSION SIMULATIONS

BY

FONG HEUNG WAH

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science May 2004, Hong Kong

© FONG HEUNG WAH
All Rights Reserved, 2004
I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

FONG HEUNG WAH
EDITING EXPLOSION SIMULATIONS

BY

FONG HEUNG WAH

This is to certify that I have examined the above MPhil thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Dr. Michael S. Brown, Supervisor

Prof. Lionel M. Ni, Head of Department

Computer Science Department
May 2004, Hong Kong
ACKNOWLEDGEMENT

I would like to take the opportunity to thank the people who have supported me through my MPhil experience.

First and foremost, I would like to thank my supervisor Dr. Michael Brown for his encouragement and competent advice during the time I have worked with him. We have had many interesting and valuable discussions and I look forward to future collaboration.

Thanks also go to the committee members of my thesis, Dr. Chiew-Lan TAI and Dr. Long QUAN, who took time to read my thesis and offer important comments.

I am profoundly grateful to my parents and sister for the unwavering encouragement and support they have provided over the years.

Last but not least, I would like to thank all the members of Vision and Graphics (Visgraph) Laboratory for fruitful discussions and conversations.

FONG HEUNG WAH

The Hong Kong University of Science and Technology
May 2004
TABLE OF CONTENTS

Title Page
Authorization Page
Signature Page
Acknowledgements Page
Table of Contents
List of Figures
Abstract
Chapter 1  Introduction
  1.1 Background and Motivation
  1.2 Thesis Organization
Chapter 2  Simulation of Explosion
  2.1 Background
  2.2 Physics of Fluid
    2.2.1 Descriptions of Fluid
    2.2.2 Fluid dynamics and Temperature
  2.3 Physics of Particles
  2.4 Interactions between Fluid and Particles
  2.5 Simulation Methods
    2.5.1 Modelling Fluid, Particles and Objects
2.5.2 Initial state 14
2.5.3 Simulating a time step 14

2.6 Rendering Methods 20
2.6.1 Rendering objects with shadows 20
2.6.2 Rendering particles 27

Chapter 3 Explosion Editing 30
3.1 Related Work 30
3.2 Warping Process 31
3.2.1 Scatter Point Interpolation 34
3.2.2 Coons Patch Interpolation 38
3.3 Propagate Editing from 2D to 3D 42

Chapter 4 Results 47
4.1 Results of Simulation 48
4.2 Results of Editing 55
4.2.1 Example of warping 1 55
4.2.2 Example of warping 2 57
4.2.3 Example of warping 3 59
4.2.4 Example of cut-and-paste 61
4.2.5 Example of cut-and-paste and cylindrical interpolation 63
4.2.6 Example of aggregating simulations 65

Chapter 5 Conclusion and Future Extensions 67
5.1 Conclusion 67
5.2 Future Extensions 68

References 69
# LIST OF FIGURES

1.1 An example of original explosions and versions edited 3
2.1 Discretization of the 3D animation space 12
2.2 Voxelization of objects in the environment 13
2.3 The flow of simulation in a time step 15
2.4 The shadow camera 21
2.5 The transmittance function of light at two different environments 22
2.6 The volume of cells for shadow rendering 24
2.7 The evolution of a transmittance function in an explosion 25
2.8 Shadows rendered with different filter radii 26
2.9 Simulation examples with and without shadows 27
2.10 Textures for particles 27
2.11 Relationship between camera and the rendering quads for particles 28
2.12 Rendering example I 29
2.13 Rendering example II 29
3.1 The interface for playing back simulations 31
3.2 Velocity, temperature and explosion played back 32
3.3 The editing interface 33
3.4 Relationships between simulation spaces and local spaces 33
3.5 Source and target edited with scatter point interpolation 34
3.6 Forward warping $W$ approximated by a 2D array of homographies 37
3.7 Relationship between the source velocity and the target velocity 37
3.8 An editing example using scatter point interpolation 38
3.9 Two editing ways with using Coons patch interpolation
3.10 Relationships between simulation, local and parameter spaces
3.11 Mapping points from source Coons patch to target Coons patch
3.12 An editing example using Coons patch interpolation
3.13 The volume of selection and a cut-and-paste example
3.14 Propagating warping along the normal direction
3.15 The interface for 3D interpolation and aggregation of 3D simulations
3.16 Parameters used in the cylindrical interpolation
3.17 Examples of cylindrical interpolation
4.1 The camera in the playback interface
4.2 Simulation example 1
4.3 Simulation example 2
4.4 Simulation example 3
4.5 Simulation example 4 - environment
4.6 Simulation example 4 - explosion
4.7 Simulation example 4 - velocity and temperature
4.8 Warping example 1
4.9 Result of warping example 1
4.10 Warping example 2
4.11 Result of warping example 2
4.12 Warping example 3
4.13 Result of warping example 3
4.14 A cut-and-paste example
4.15 Result of the cut-and-paste example
4.16 A cut-and-paste and cylindrical interpolation example
4.17 Result of the cut-and-paste and cylindrical interpolation example
4.18 An example of aggregating simulations

4.19 Result obtained from aggregating two simulations
EDITING EXPLOSION SIMULATIONS

BY

FONG HEUNG WAH

COMPUTER SCIENCE DEPARTMENT
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

ABSTRACT

Explosion effects are widely used in movies and the game industry. Popular methods in computer graphics synthesize these effects by computing a velocity and temperature field based on fluid dynamics, such that smoke and fire behave realistically. In this kind of physics-based simulation, long computation time and accurate setting of parameters are generally required in order to obtain a desirable result. Due to this restriction, it is usually difficult for the animator to control and model a particular explosion.

This thesis illustrates a straight-forward method for editing physics-based explosions. Rather than controlling an explosion’s behavior via simulation parameters and external forces, we allow the user to directly warp the flow-field and temperature values computed by an initial simulation. A particle-system is then played back through the altered simulation to produce a new explosion that reflects the look and behavior specified by the user. This simple approach provides a flexible method to edit and author convincing explosions that are otherwise difficult to predict and control.
CHAPTER 1

INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

Explosions are physical phenomena popularly used in entertainment media such as television, movies, and video games. A great deal of work has focused on producing realistic virtual explosion with computers. Such work is an essential alternative to the production of real explosion scenes, which is technically difficult and has potential risks.

It is not a trivial task to synthesize an explosion by using computer graphics techniques, as this type of phenomena always involves a large set of complex reactions and processes. When an explosion occurs, tons of gas, dusts and millions of burning particles are generated and pushed out from the center of the explosion. The sudden release of energy heats the surrounding air and gases start to rise. The gases at the middle of the explosion absorb more energy and rise at a faster speed than those further from the center. This difference in velocities causes a rotational effect at the boundary between hot gases and cold gases, which is called vorticity (discussed in Chorin’s book [6]). As the gases move, the particles are blown upwards and burn due to the high temperature. Heat, light and more gases are then generated during their combustion. After a certain period, the clouds of particles cool down and vanish into the air.

In a realistic synthesized explosion, these phenomena should take place reasonably at different positions in the space. This can be achieved only if the simulation parameters (such as gravity and thermal conductivity and so on) are preset correctly and the governing attributes, including wind velocity and temperature, are computed physically at each time moment. Currently, techniques for generating realistic synthetic explosions (and related phenomena) use fluid dynamics as the
cornerstone of their simulations [1], [10], [20], [29], [30] and [35].

Such an explosion simulation is inherently an initial boundary problem: initial simulation parameters govern the outcome of Partial Differential Equations (PDEs) that subsequently dictate the evolution of the explosion over time. In order to produce a desired look and behavior, a great deal of trial and error is normally required. Even small modifications to an explosion’s appearance may require several parameter tweaks along with re-simulations until a satisfactory result is obtained.

The focus of this thesis is to provide the user a better way to modify an explosion’s behavior. Instead of making parameters adjustments and redoing the simulation, in our system the user can alter a current explosion by warping the velocity and temperature fields directly, to get a new result he wants. Since the user can interactively see the result after each edit with the user interface, explosion modeling becomes much more efficient and effective.

Moreover, sometimes the desired behavior of explosion may not be within the simulation’s capabilities and deviations from the governing physics-based equations are needed. And sometimes the required explosion is too large, the physics-based simulation is impractical due to the extensive computation is and large memory requirement. With our editing system, the user can create explosions in new styles and in a larger scale by editing and combining a number of pre-computed simulations together. Figure 1.1 shows an example of explosion and its edited version generated by the proposed system. Here we see an explosion that has been edited to behavior as if an obstacle was placed in a scene. The result was obtained without re-simulation.

1.2 THESIS ORGANIZATION

This thesis is organized as follows. Chapter 2 describes the physics background of an explosion and the simulation methods chosen for the explosion synthe-
Figure 1.1: An example of original explosion and version edited by the proposed approach. Left: Original version. Right: edited version.

sis. Chapter 3 introduces our proposed approach for editing flow fields and the method for propagating 2D explosions to 3D in order to produce special explosion effects. Chapter 4 presents a number of results produced by simulation and editing tools. The last chapter concludes this thesis, mentions several limitations of the developed system and proposes some future extensions.
CHAPTER 2

SIMULATION OF EXPLOSION

2.1 BACKGROUND

The explosion simulation used in this thesis is based on the fluid-dynamics approach of Stam [29] with the added heuristics for explosions outlined by Feldman et al. [10]. There are two major challenges in the synthesis of such a phenomenon: modelling and rendering. The purpose of modelling is to define a way to describe the status of the environment (the temperature at a position and the velocity of a particle, and so on) and establish a set of rules that determine how this status varies as time varies. Rendering controls how the objects in the environment are visualized based on their status at a particular time.

While the fluid state of an actual explosion simulation includes several properties including temperature, velocity, divergence pressure and so on, the final rendered output can be distilled down to only two attributes, velocity field and temperature. The velocity field directs the motion of particles used in rendering and dictates the overall look and feel of the explosion, while temperature values govern color.

In this chapter, the physics-based rules behind an explosion simulation are introduced. They include the rules of temperature and velocity (fluid dynamics) and the interactions between fluid and particles. Afterwards, a model satisfying the physics rules is discussed. Finally, the rendering methods for the synthesized results will be mentioned.
2.2 PHYSICS OF FLUID

2.2.1 DESCRIPTIONS OF FLUID

The synthesis of fluid’s motion is the major part of explosion simulation as its motion and the external force (gravity) completely determines the motion of particles, and the final shape of the explosion. Hence it is important to know what factors influence fluid’s motion. A governing factor is its velocity. For example, when a jet of hot smoke mixes with slow-moving air, it experiences drag and starts to rotate. Another important factor is temperature. When an explosion occurs, it produces a sudden release of energy that dramatically increases the temperature of surrounding air. The air then rises due to thermal buoyancy affecting fluid’s velocity. The velocity field and temperature field can be defined either by the Eulerian description or the Lagrangian description as follows:

**Eulerian description:** The quantities of the flow are assigned to fixed points in space as time varies. For example, velocity and temperature can be represented as \( \mathbf{v}(x, y, z, t) \) and \( T(x, y, z, t) \) in 3-dimensional space, where

\[
\mathbf{v}(x, y, z) = u(x, y, z) \mathbf{i} + v(x, y, z) \mathbf{j} + w(x, y, z) \mathbf{k}.
\]

**Lagrangian description:** The quantities are given for a particular moving particle as time varies. For example, velocity and temperature for a particle \( p \) are represented as \( \mathbf{v}_p(t) \) and \( T_p(t) \).

In Eulerian terms, at a particular point in space, the acceleration of fluid includes not only the change in fluid quantity occurring there, but also the change in fluid quantity due to a fluid particle moving through that point. Hence the substantial/total derivatives of the quantities (velocity and temperature) are:

\[
\frac{D\mathbf{v}}{Dt} = \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla) \mathbf{v} = \frac{\partial \mathbf{v}}{\partial t} + \mathbf{u} \frac{\partial \mathbf{v}}{\partial x} + \mathbf{v} \frac{\partial \mathbf{v}}{\partial y} + \mathbf{w} \frac{\partial \mathbf{v}}{\partial z} \tag{2.1}
\]

\[
\frac{DT}{Dt} = \frac{\partial T}{\partial t} + (\mathbf{v} \cdot \nabla) T = \frac{\partial T}{\partial t} + \mathbf{u} \frac{\partial T}{\partial x} + \mathbf{v} \frac{\partial T}{\partial y} + \mathbf{w} \frac{\partial T}{\partial z}, \tag{2.2}
\]

where \( \nabla = \left( \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \). Other than velocity and temperature, there are still
other quantities of fluid that are relevant to an explosion, for example, pressure, vorticity and divergence. In the following subsections, it is shown that velocity and temperature are the most fundamental factors while the others have a strong dependence on them.

### 2.2.2 FLUID DYNAMICS AND TEMPERATURE

The Navier-Stokes equations (NS equations) are used frequently in previous and recent work [1], [10], [14], [19], [29], [30] and [35] for describing the motion of Newtonian fluid. The original equation is:

\[
\rho \left( \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} \right) = -\nabla p + \rho \mathbf{f} + \mu \nabla^2 \mathbf{v},
\]

\[
\frac{\partial \mathbf{v}}{\partial t} = - (\mathbf{v} \cdot \nabla) \mathbf{v} - \frac{\nabla p}{\rho} + \mathbf{f} + \frac{\mu}{\rho} \nabla^2 \mathbf{v}. \tag{2.3}
\]

Here \( \mathbf{v} \) denotes the velocity of fluid, \( \rho \) density, \( p \) pressure, \( \mathbf{f} \) the total external vector force applied to the unit volume of fluid, \( \mu \) the coefficient of viscosity and \( \nabla^2 = \left( \frac{\partial^2}{\partial x^2}, \frac{\partial^2}{\partial y^2}, \frac{\partial^2}{\partial z^2} \right) \). Notice that the first four values are functions of position and time. This equation is derived from Newton’s second law of motion:

\[
\mathbf{F} = m \mathbf{a},
\]

and is used to enforce momentum conservation. The reader can find the actual derivation from any standard text on fluid mechanics. Equation 2.3 is important because it describes how the velocity at each position is updated from one moment to next moment. Each term at its right-hand-side represents a specific update. The first term is advection, which is the change caused by the movement of fluid. The second term is the change produced by pressure. The third term is the influence of the external force, which is discussed more in the next paragraph. The last term is the change introduced by viscosity, the friction between fluid particles.

In the fluid simulation engine of this project, three kinds of external forces acting on the fluid are considered: Gravity, thermal buoyancy and the vorticity confinement force. The magnitude and direction of these forces are determined by different factors. Gravity and thermal buoyancy acting on a point are given by
the equation mentioned by Foster et al. [11]:

\[ f_{tb} = -\beta g(T_a - T), \]  

(2.4)

where \( g \) is the gravity vector, \( \beta \) is the coefficient of thermal expansion, \( T_a \) is the ambient temperature (28°C is chosen in this project) and \( T \) is the temperature at that point.

The vorticity confinement force is used to add the small scale detail back to the fluid. When simulating an explosion, large spatial deviations produced in the velocity field are supposed to cause a significant amount of turbulent or rotational structures. However during the numerical processes, this kind of small scale structures are usually damped out. Fedkiw et al. [9] gets rid of this problem by applying the vorticity confinement force. At each point, this force can be computed based on the original velocity field:

\[ \Omega = \nabla \times \mathbf{v}, \]

(2.5)

\[ \eta = \nabla |\Omega|, \quad N = \frac{\eta}{|\eta|}, \]

(2.6)

\[ f_{wc} = \epsilon h (\mathbf{N} \times \Omega), \]

(2.7)

where \( \Omega \) is the vorticity, \( \eta \) is the vorticity location vector that points from where the vorticity is lower to where the vorticity is higher, \( N \) is the normalized vorticity location vector, \( \epsilon \) controls the amount of detail added and \( h \) is the edge length of a cell in the computational grid (Section 2.5 discusses how the world space is partitioned into computational grids for simulation.).

Hence, the total external force for equation 2.3 is the sum of these forces:

\[ \mathbf{f} = f_{tb} + f_{wc} \]

(2.8)

To simplify Equation 2.3, Foster et al. [11] and Stam [29] assume fluid to be incompressible when doing simulations of smoke. For an incompressible fluid, mass conservation is required - at each time step, for each point, the volume
of fluid going out is the same as the volume going in it, or in other words, the velocity divergence has to be zero:

\[ \nabla \cdot \mathbf{v} = 0 \quad \text{(2.9)} \]

In this case, they assume the density \( \rho \) to be constant at each point and \( \rho = \rho_0 \). Hence the Navier-Stokes equations for incompressible fluid is:

\[ \frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla)\mathbf{v} - \nabla q + \mathbf{f} + \nu \nabla^2 \mathbf{v}, \quad \text{(2.10)} \]

where \( q = \frac{\rho}{\rho_0} \) and \( \nu = \frac{\mu}{\rho_0} \).

One difference between the simulation of explosion and the simulation of general smoke is that in the former case, gases can be generated when floating particles burn. Feldman et al. [10] thus suggests that positive divergence of fluid velocity should be allowed:

\[ \nabla \cdot \mathbf{v} = \phi, \quad \text{(2.11)} \]

where \( \phi \) is zero everywhere except where gases are being generated from fire. A point with positive velocity divergence is called a source.

In this thesis, equation 2.10 and 2.11 are used to control the evaluation of velocity in the space over the simulation. Two questions remain: how is the variable \( q \) in equation 2.10 determined at each moment? How are the two equations combined? A general method is used to solve these questions in most of the work on smoke and explosion simulation [9], [10], [11], [12] and [29]. All of them use \( q \) (or \( p \) in equation 2.3) to enforce mass conservation. Suppose the velocity \( \mathbf{v} \) at a point in time \( t \) is known and we are going to solve the velocity \( \mathbf{v}' \) at \( t + \Delta t \), then equation 2.10 can be approximated as follows:

\[
\mathbf{v}' = \mathbf{v} + \Delta t[-(\mathbf{v} \cdot \nabla)\mathbf{v} - \nabla q + \mathbf{f} + \nu \nabla^2 \mathbf{v}]
\]

\[ \mathbf{v}' = \mathbf{v}' - \Delta t \nabla q, \quad \text{(2.12)} \]

where \( \mathbf{v}' \) is the intermediate velocity obtained after advection, external forces and viscosity are applied. As \( \mathbf{v}' \) has to satisfy equation 2.11, a gradient operator is multiplied to both sides of equation 2.12 and a poisson equation can be obtained:

\[ \nabla^2 q = \frac{1}{\Delta t} (\nabla \cdot \mathbf{v}' - \phi) \quad \text{(2.13)} \]
Therefore, to enforce equation 2.11, one can solve $q$ from this equation and compute the new velocity with equation 2.12. A poisson equation’s solver is discussed in section 2.5.

Notice that equation 2.12 shows that a discrete physical transition rule is used for the simulation of velocities at each time step. There is another model called Lattice Boltzmann Model [32] that performs discretization on the level of the NS equations. We choose to discretize the transition rules because this is a general approach for smoke and explosion simulations in the previous work.

Until now, we have discussed how to model the velocity of fluid. We also need to consider temperature. There is only one equation, which is introduced in Feldman et al. [10], that describes how the temperature is evaluated:

$$\frac{\partial T}{\partial t} = - (v \cdot \nabla)T - c_r \left( \frac{T - T_a}{T_{\text{max}} - T_a} \right)^4 + c_k \nabla^2 T + H, \quad (2.14)$$

where $T$ is the temperature of field, $T_a$ ambient temperature, $T_{\text{max}}$ the maximum temperature in the environment and $H$ heat energy produced by the combustion. Similar to the Navier-Stokes equations, the first term at the right-hand-side represents advection, meaning that the temperature changes due to the movement of fluid. The second term approximates the effect of radiative loss and $c_r$ controls the rate of cooling. The third is the diffusion term and $c_k$ represents the thermal conductivity. This coefficient is set to large in an explosion since both of the radiative and diffusive transfer have to be considered. Notice that $T$, $v$ and $H$ are the functions of position and time, $T_{\text{max}}$ is a function of time, while other values are constants.

The following two sections illustrate the particle model and the interactions between fluid and particles, which are based on the rules mentioned in Feldman et al. [10].
2.3 PHYSICS OF PARTICLES

The properties of a particle related to an explosion include position, velocity, temperature, mass, radius and thermal mass, to name a few. Like fluid, velocity and temperature are the most important factors of the simulation as the former determines the motion and position of the particle at each moment and the latter affects how it is visualized. In a simplified case, a particle can be classified into fuel particles and soot. Fuel particles burn and generate gases when its temperature rises over a given threshold. Soot particles, which represents dust or smoke particles, are generated when fuel is burning. They are inflammable and cannot trigger gas generation.

The velocity and temperature of particles are governed by:

\[ \dot{u} = \frac{f_d}{m} + g \quad \text{and} \quad \dot{Y} = \frac{K}{c_m} \]

In the first equation, \( u \) is velocity, \( f_d \) is the drag force from fluid, which is discussed in the next section, \( g \) is the gravity vector and \( m \) is mass. In the second one, \( Y \) is the temperature of a particle, \( K \) is the heat transferred to it and \( c_m \) is its thermal mass.

2.4 INTERACTIONS BETWEEN FLUID AND PARTICLES

When particles and fluid mix with each other, momentum and heat energy are transferred between each other. In order to advect particles according to the flow of fluid, a drag force is applied on each particle:

\[ f_d = \alpha_d r^2 (v - u) \| v - u \|, \quad (2.15) \]

where \( \alpha_d, r \) and \( u \) are the drag coefficient, radius and velocity of the particle, and \( v \) is the velocity of fluid at where it is.

When the temperature of a particle is different from fluid, heat is transferred from fluid to the particle at a rate:

\[ \dot{K} = \alpha_h r^2 (T - Y), \quad (2.16) \]
where $\alpha_h$ is the thermal conductivity between fluid and particles, $T$ is the temperature of fluid at where the particle is.

As mentioned in section 2.3, a fuel particle will ignite when its temperature raises over a threshold. During the combustion, heat, gaseous products and soot are generated. At the same time, this process consumes the mass of the particle itself at a rate $z$, which is its burning rate. When the mass reaches zero, the particles will be deleted from the system. The rate of generation for the combustion products is proportional to the burning rate. The rate of heat generation is determined by:

$$\dot{K} = b_k z,$$

(2.17)

where $b_k$ is the amount of heat generated per unit consumed mass of the fuel. The generation of gas can be achieved by adding an increment to $\phi$ in equation 2.11. This triggers an expansion effect in the velocity field. The rate of change of $\phi$ is:

$$\dot{\phi} = b_g z,$$

(2.18)

where $b_g$ determines the volume of gaseous products generated per unit consumed mass. Finally the generation of soot is controlled by a variable $s$, called soot mass, associated with each fuel particle. After a fuel is ignited, its soot mass keeps accumulating in the following rate:

$$\dot{s} = b_s z,$$

(2.19)

where $b_s$ is the amount of soot mass added per unit consumed mass. When a sufficient amount of soot mass is accumulated a soot particle will be generated. The position of this new soot particle is the same as the fuel and the direction and magnitude of its velocity is slightly different from the fuel particle.

### 2.5 SIMULATION METHODS

In the last three sections, the basic physics rules of fluid and particles in an explosion were described. In this section, the actual simulation model is described; i.e. how the information is stored at each time moment and how the physics
equations are discretized and solved over the animation space. The procedures of simulation follows the approach introduced by Feldman et al. [29] and the numerical computation is based on the finite difference method used in Foster et al. [11].

2.5.1 MODELLING FLUID, PARTICLES AND OBJECTS

In order to solve the Navier-Stoke equation (2.10), the 3D animation space has to be discretized and represented with a number of cubic cells (Figure 2.1), whose edge length is h. The discretized space is called a computational grid. Properties of fluid like velocity, temperature and divergence(\phi) are assigned only to the center of each cell, such that these continuous functions are discretized over the space. Velocity and temperature are denoted by \( \mathbf{v}_{i,j} \) and \( T_{i,j} \) in a 2D grid or \( \mathbf{v}_{i,j,k} \) and \( T_{i,j,k} \) in 3D, where i, j and k are the index numbers of the cell along x, y and z directions. As a particle is advected over the fluid, the velocity and temperature of fluid at a position \( \mathbf{x} \) of the particle have to be known. These properties can be obtained by linear interpolation through the center of neighbor cells. Such interpolated properties are represented as \( \mathbf{V}(\mathbf{x}) \) and \( T(\mathbf{x}) \).

![Discretization of the 3D animation space.](image)

In our simulation, each particle object represents a cloud of rendered parti-
Figure 2.2: Voxelization of objects in the environment. Left: The objects in the environment. Right: The voxelized objects.

cles [25]. Each object contains information such as velocity, position and temperature which are shared by all particles in the group. This approach is taken because a simulation workstation cannot handle the update of millions of particles in a short time.

To increase simulation realism, objects including boxes, spheres and cylinders are inserted to the environment and the collisions between particles, fluid and objects have to be handled. The collisions between particles and objects are detected by intersection checking. If a collision occurs, the particle is bounced away from the object. The direction of a particle can be updated based on the surface normal at the point of collision.

The collision between fluid and an object is a bit tricky. Firstly, the objects are voxelized (Figure 2.2) to fit the computational grid. This means some cells in the grid are occupied by the objects. During a simulation, the velocity of a cell next to those occupied cells are solved in a slightly varied way such that the fluid will not move into the objects. It is pressed to other cells and roll around the object. The method to solve this will be talked about in subsection 2.5.3. In the next subsection, we discuss about how to setup a simulation initially.
2.5.2 INITIAL STATE

There are a number of components that have to be setup before a explosion simulation starts, including the environment objects and the parameters of the physical equations. Objects are inserted to the environment by specifying their position and size. They are then voxelized for fluid simulation as described above.

Another important procedure is to assign values to the large set of parameters related to an explosion. The major parameters include the amount of heat produced and the number of particles generated. Instead of transferring a large amount of heat or inserting a large set of particles at the first step, our system gradually reduce the amount of generated heat and the number of inserted particles down to zero over a number of steps. The result produced in this way is relatively smooth.

Other parameters include the coefficient of fluid equations and the properties of particles. The coefficients such as thermal buoyancy, thermal conductivity and viscosity are set to have constant values initially, while the properties of particles are generally assigned to be random values selected within a given range. These properties include mass, thermal mass and radius of the particle. The values of the coefficients and the range of particles' properties are chosen after a number of tests in order to produce a particular explosion result.

2.5.3 SIMULATING A TIME STEP

After setting up the initial state, the animation can be simulated step by step. The time difference between each step (\(\Delta t\)) is set to be 1/30 seconds. As shown in figure 2.3, the simulation in each step contains three components: Fluid, particles and rendering. This subsection focuses on fluid simulation, while particle simulation is not discussed since it can be easily derived from the physical rules. The rendering methods are discussed in the next section.
The task of fluid simulation in each time step is to update the velocity and temperature at each center of a cell. These updates are based on equation 2.10 and 2.14 with respect to velocity and temperature. These two equations can be decomposed into a series of updating processes:

\[
\begin{align*}
\mathbf{v} & \rightarrow \text{Apply Force} \rightarrow \text{Advect} \rightarrow \text{Diffuse} \rightarrow \text{Project} \rightarrow \mathbf{v}' \\
T & \rightarrow \text{Apply Heat} \rightarrow \text{Advect} \rightarrow \text{Diffuse} \rightarrow \text{Dissipate} \rightarrow T'
\end{align*}
\]

Since advection and diffusion of velocities and temperatures can be handled together, there are five passes in the flow of fluid simulation: (1) apply force and
heat, (2) advection, (3) diffusion, (4) velocity projection and (5) temperature dissipation. In some updating passes, it is necessary to use the old properties of neighboring cells when computing the new properties for a cell. To handle such cases, two computational grids are prepared for storing the previous and updated properties at the same time.

In the following paragraphs, the numerical methods for these updates are introduced. \( \mathbf{v} \) and \( T \) represent the input velocities and temperatures in each pass, while \( \mathbf{v}' \) and \( T' \) are the output. For the updates of velocities, all velocity components \((u, v, w)\) are updated individually. In some passes, the derivatives of properties are approximated with finite difference method. Those approximations are listed below (Let \( a \) be a property of fluid, \( h \) be the edge length of cells in the grid):

\[
\begin{align*}
\left( \frac{\partial a}{\partial x} \right)_{i,j,k} &= \frac{1}{2h}(a_{i+1,j,k} - a_{i-1,j,k}), \\
\left( \frac{\partial a}{\partial y} \right)_{i,j,k} &= \frac{1}{2h}(a_{i,j+1,k} - a_{i,j-1,k}), \\
\left( \frac{\partial a}{\partial z} \right)_{i,j,k} &= \frac{1}{2h}(a_{i,j,k+1} - a_{i,j,k-1}), \\
\left( \frac{\partial^2 a}{\partial x^2} \right)_{i,j,k} &= \frac{1}{h^2}(a_{i+1,j,k} - 2a_{i,j,k} + a_{i-1,j,k}), \\
(\nabla^2 a)_{i,j,k} &= \begin{cases} 
\left( \frac{\partial^2 a}{\partial x^2} \right)_{i,j} + \left( \frac{\partial^2 a}{\partial y^2} \right)_{i,j} & \text{in 2D}, \\
\left( \frac{\partial^2 a}{\partial x^2} \right)_{i,j,k} + \left( \frac{\partial^2 a}{\partial y^2} \right)_{i,j,k} + \left( \frac{\partial^2 a}{\partial z^2} \right)_{i,j,k} & \text{in 3D}, \\
\frac{1}{h^2}(a_{i+1,j,k} + a_{i-1,j,k} + a_{i,j+1,k} + a_{i,j-1,k} - 4a_{i,j,k}) & \text{in 2D}, \\
\frac{1}{h^2}(a_{i+1,j,k} + a_{i-1,j,k} + a_{i,j+1,k} + a_{i,j-1,k} + a_{i,j,k+1} + a_{i,j,k-1} - 6a_{i,j,k}) & \text{in 3D}.
\end{cases}
\end{align*}
\]

In the first pass, external force and heat are applied to influence the velocity and temperature. These influences can be computed using the finite difference method. The time step \( \Delta t \) used \((1/30 \text{ seconds})\) is sufficiently small such that values in the velocity and temperature fields do not become unstable [11]. Here are the equations:

\[
T' = T + \Delta t H, \quad \mathbf{v}' = \mathbf{v} + \Delta t \mathbf{f}
\]

Heat is simpler since the position and value of the heat source (at the center of explosion) is specified at the beginning and only the temperature at that position is changed. The external forces include gravity, thermal buoyancy and vorticity.
confinement force. The first two components can be computed trivially with equation 2.4, while the last one is a bit complex as it involves derivatives of the velocity field. First, equation 2.5 and 2.6 for the vorticity confinement force are represented as:

\[
\Omega = \nabla \times \mathbf{v} = \begin{cases} 
\left( \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right) \hat{k} & \text{in 2D or} \\
\hat{i} & \text{in 3D.} \\
\frac{\partial u}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial w}{\partial z} \\
u & v & w
\end{cases}
\]

\[
\eta = \nabla |\Omega| = \begin{cases} 
\frac{\partial \Omega^i}{\partial x} \hat{i} + \frac{\partial \Omega^i}{\partial y} \hat{j} & \text{in 2D or} \\
\frac{\partial \Omega^i}{\partial x} \hat{i} + \frac{\partial \Omega^i}{\partial y} \hat{j} + \frac{\partial \Omega^i}{\partial z} \hat{k} & \text{in 3D.}
\end{cases}
\]

The force is then computed by approximating all partial derivatives with the finite difference method.

The second pass of the simulation is to handle the advection term \(-(\nabla \cdot \mathbf{v})\mathbf{v}\) and \(-(\nabla \cdot \mathbf{v})T\). Although a finite difference method can also be applied when solving this terms, a better method, called a back-tracing method, is suggested by Stam [29]. This method is used to handle the advection term and move the velocity (or temperature) values according to the velocity field at the current moment. In order to know what the current velocity (temperature) at a position \(\mathbf{x}\) should be, one can back-trace from \(\mathbf{x}\) to find, at one time step before, where the original location of that fluid particle was. The velocity (temperature) at \(\mathbf{x}\) is then updated to be the velocity (temperature) linearly interpolated from that position. The major advantage of this approach is stability. While the finite difference method may introduce large velocities due to the numerical oscillations, this method never does so as the updated values are always limited by the maximum values of the properties at the last step.

Mathematically, let \(p(\mathbf{x}, \mathbf{v}, t)\) be a function that outputs the position got from back-tracing, the update of properties can be represented as:

\[
\mathbf{v}_{i,j,k}' = \mathbf{V}(p(x_{i,j,k}, \mathbf{v}_{i,j,k}, -\Delta t)) \quad \text{and} \quad T_{i,j,k}' = T(p(x_{i,j,k}, \mathbf{v}_{i,j,k}, -\Delta t)), \forall i, j, k,
\]

where \(\mathbf{V}(\cdot)\) and \(T(\cdot)\) denote the velocity and temperature interpolated over the
fields obtained from the first pass. The back-tracing function, based on the second order Runge-Kutta method [33], is defined as:

\[ p(x, v, t) = x + tV(x + \frac{1}{2}tv) \]  \hspace{1cm} (2.20)

The third pass is diffusion. In terms of velocity, it introduces the friction effect between the fluid particles. In terms of temperature, it transfers heat from hotter regions to cooler regions. In this pass, the updates of velocities and temperatures can be approximated by (h is the edge length of cells):

\[ v'_{i,j} = v_{i,j} + \Delta t \nu (\nabla^2 v)_{i,j} \]
\[ = v_{i,j} + \frac{\Delta t \nu}{h^2} (v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{i,j}), \]
\[ T'_{i,j} = T_{i,j} + \Delta t c_k (\nabla^2 T)_{i,j} \]
\[ = T_{i,j} + \frac{\Delta t c_k}{h^2} (T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j}), \forall i, j. \]

(The method illustrated here and in the fourth pass are in 2D, while the method in 3D is similar.) However, it is not preferable to approximate in this way as unstable values may be produced when the diffusion rates (ν and c_k) are high. A more stable method that solves a linear system is proposed by Stam [30]. The idea is to find the velocities (temperatures) which when diffused backward in time generate the velocities (temperatures) we started with. Mathematically,

\[ v_{i,j} = v'_{i,j} - \alpha (v'_{i+1,j} + v'_{i-1,j} + v'_{i,j+1} + v'_{i,j-1} - 4v'_{i,j}), \]

where \( \alpha = \frac{\Delta t \nu}{h^2} \). This yields \( N \times M \) equations where \( N \) and \( M \) are the number of cells on the edges of the grid and \( v'_{i,j}, \forall i, j \) are the unknowns. Here, we choose to use a simple iterative method, Gauss-Seidel method [8], to solve this system. The algorithm is shown as follows:
ALGORITHM OF GAUSS-SEIDEL METHOD

for k = 1 to #ITERATIONS do
    for i = 1 to M do
        for j = 1 to N do
            \[ v'_{i,j} = \frac{1}{1 + 3 \alpha} \cdot [v_{i,j} + \alpha \cdot (v'_{i+1,j} + v'_{i-1,j} + v'_{i,j+1} + v'_{i,j-1})] \]
        end
    end
end

#ITERATIONS: The number of iterations for the algorithm. The more times it iterates, the more the result converges to the real solution, but also the slower the algorithm is. In our system, this number is set to 20.

One extra point we need to consider is the boundary cases. Except that the computational grid has its boundary, the voxelized object also introduce boundary cells to the grid. For the boundary cells, it is not necessary to update, while for the cells next to the boundary, update is required but the algorithm has to be varied since the derivatives of velocity from normal cells to boundary cells are undefined and should be ignored. For example, if the left cell of a cell is boundary, the velocity evaluation of that cell in each iteration becomes:

\[ v'_{i,j} = \frac{1}{1 + 3 \alpha} \cdot [v_{i,j} + \alpha \cdot (v'_{i+1,j} + v'_{i,j+1} + v'_{i,j-1})] \]

The fourth pass is velocity projection, which is to enforce the mass conservation property on the velocity field. This pass involves two steps. Firstly, we have to solve for the field \( q \) using Poisson equation (2.13) as follows:

\[ \nabla^2 q = \frac{1}{\Delta t} (\nabla \cdot v - \phi) \]

Here, \( \phi \) is the divergence value assigned to each cell which represents the amount of gas generated from the cell. When a particle is burning, the value \( \phi \) of the four closest cells (eight in 3D case) will increment based on the distance between the center of cells and the particle. To solve equation 2.13, a method similar to diffusion is used. It can be approximated by:

\[ q_{i+1,j} + q_{i-1,j} + q_{i,j+1} + q_{i,j-1} - 4q_{i,j} = \frac{1}{\Delta t} \left[ (\nabla \cdot v)_{i,j} - \phi_{i,j} \right], \forall i, j, \]

which becomes a linear system that can be solved using Gauss-Seidel method. Again, for the cells next to the boundary, some particular derivatives have to
be ignored from the equation. After getting \( q \), the velocities can be updated by applying finite difference method to equation 2.12:

\[
v' = v - \Delta t \nabla q.
\]

With this mass conservation process, when the moving fluid meets an object, it will be enforced to change direction according to the field \( q \) and roll around the object.

The last pass of fluid simulation is temperature dissipation. According to equation 2.14, the following temperature update has to be handled:

\[
T' = T + \Delta t c_r \left( \frac{T - T_a}{T_{max} - T_a} \right)^4.
\]

where \( c_r \) and \( T_a \) are constant, \( T_{max} \), the maximum temperature, is found by checking the temperature field. The update is approximated by the finite difference method.

### 2.6 RENDERING METHODS

Rendering is another essential component for an explosion simulation as it directly affects whether the result is visually realistic. The rendering system for our explosion simulation consist of two parts: Rendering environment objects with shadows and rendering particles. In the following subsection, a method called deep shadow map [18], which is used to cast shadows on objects, is briefly introduced. Readers can refer to Lokovic's work [18] for a detailed description. In the subsection 2.6.2, a method that uses semitransparent quads to represent and render particles is discussed.

#### 2.6.1 RENDERING OBJECTS WITH SHADOWS

Since the objects in our simulation are constructed with simple primitives (including boxes, spheres and cylinders), the major challenge for their rendering is shadow mapping. Consider every point in the environment has a brightness
value $B(x, y, z)$, which is a binary function - it is 0 if the point is in a shadow and 1 otherwise. To render shadows for each object, we have to determine the brightness at a set of points lying on the object so that a shadow texture can be constructed. By assigning darker colors to the positions with $B = 0$ in the texture, a shadow can be drawn on the object.

![Diagram](image)

Figure 2.4: The light source is treated as a shadow camera. $d(x, y)$ is the distance from the light source to the closest visible object when looking through the point $(x, y)$ on the image plane.

According to this method, it is essential to develop an approach to compute $B$ for the points concerned. Suppose there is only one light source, which is typical for many shadow mapping methods [34], this light source is treated as a camera, called a shadow camera. The shadow camera views objects in the environment through an image plane (see Figure 2.4). We assume that for each point $(x, y)$ on the shadow camera's image plane, the distance of the closest visible object from the shadow camera, $d(x, y)$, is known. In order to determine the brightness, $B$, at an arbitrary point in the space, the point is transformed into the coordinate system of the shadow camera (This gives us a coordinate $(x, y, z)$. $(x, y)$ is a point on the image plane of the shadow camera and $z$ is the depth - the distance between the point and the camera.). If the depth $z$ is larger than $d(x, y)$, this point is invisible from the light source and is in a shadow (with $B = 0$). Otherwise, the brightness $B$ is 1.

21
In practice, we can only store a limit number of samples for \( d(x, y) \). Generally, a map of pixels is defined on shadow camera’s image plane and the value of \( d \) is found at the center of each pixel. Such a map is called a shadow map. When the brightness of a point in the world space is computed, the point is transformed but the resulting point \((x, y)\) on the image plane of shadow camera may not match with the center of a pixel in the shadow map. In this case, the depth \( z \) can be compared with the value \( d \) at a set of nearby pixels, which results in a set of brightness values (equal to 0 or 1). The weighed average of them can be used as the brightness for the point in the world space.

This method, however, cannot produce shadows cast by an explosion, which consists of semitransparent elements like smoke. When a beam of light passes through the cloud of smoke in an explosion, a fraction of it is absorbed or scattered by the particles, while some light penetrates the particles. The fraction of light that penetrates to a depth \( z \) (distance to the light source) is defined as transmittance, \( \ell(z) \). Figure 2.5 shows an example of \( \ell(z) \)'s behavior. This function drops to zero immediately when the light meets an opaque object, while it drops gradually when the light passes through smoke.

![The image plane of shadow camera](image)

**Figure 2.5:** The transmittance function of light when it meets an opaque object (left) and a cloud of smoke (right).

Due to this reason, it is insufficient to store only the distance of the closest object \( d(x, y) \) in the shadow map. For each position \((x, y)\) on the shadow camera’s image plane, a transmittance function, \( \ell(x, y, z) \), has to be defined. Similar to the traditional method, for an input point \((x, y, z)\), the corresponding point \((x', y')\)
and the depth $z'$ are found in the coordinate system of the shadow camera. The brightness of the input point is assigned to be the value of the transmittance function at depth $z'$:

$$B(x, y, z) = \ell(x', y', z')$$

where $B$ is no longer a binary function but has its values between 0 and 1. This brightness value determines how dark a shadow is in a texture. In the discrete case, a transmittance function is defined at the center of each pixel in the shadow map. A shadow map defined in this way is called the deep shadow map [18].

Here, we discuss how this technique is applied in our explosion simulation for shadow mapping. There are three major procedures: Constructing the deep shadow map, updating the map in each time step and updating the textures for the objects.

**Constructing the deep shadow map**

In general, the transmittance function is represented as:

$$\ell(x, y, z) = \exp\left(-\lambda \int_0^z \rho(z')dz'\right), \quad (2.21)$$

where $(x, y, z)$ is a point in the coordinate system of the shadow camera, $\rho(z)$ is the density of particles along the light ray shot from the light source and passing through $(x, y)$ on the image plane, and $\lambda$ is the attenuation rate of transmittance. To construct the transmittance function for each pixel in the map, an approach has to be developed for defining and updating the density function $\rho(z)$ along each light ray. For this purpose, the space is partitioned into a volume of cubic cells, which is similar to the computational grid for fluid simulation. When a light ray intersects with the volume, a set of intersection points $p_0, p_1, p_{n+1}$ are produced at depths $z_0, z_1, ... z_{n+1}$ and they define a set of light ray segments $r_0, r_1, ... r_n$ (see Figure 2.6). By assuming constant density in each cell, the densities, $\rho_0, \rho_1, ... \rho_n$, corresponding to the light ray segments can be computed with the
number of particles in each penetrated cell. Thus, the function $\rho(z)$ is defined as:

$$
\rho(z) = \begin{cases} 
\rho_0, & z_0 < z \leq z_1 \\
\rho_1, & z_1 < z \leq z_2 \\
\vdots & \\
\rho_n, & z_n < z \leq z_{n+1} \\
0, & \text{otherwise.} 
\end{cases}
$$

(2.22)

Figure 2.6: Left: The intersections and segments introduced when a ray of light (The blue line) passes through the cells (in 2D). Right: The density values used for the computation of transmittance.

The discrete transmittance function for $\text{pixel}_{i,j}$ can be defined as:

$$
\ell_{i,j,k} = \exp \left( -\lambda \sum_{a=0}^{k-1} \rho_a \right), \quad k \in [0, n + 1].
$$

(2.23)

which are the transmittances at depths $z_0, z_1, \ldots, z_{n+1}$. The complete function can be approximated by connecting these values with linear segments. Each pixel in the deep shadow map thus stores an array of segments, $s_0, s_1, \ldots, s_n$, which are sorted in the increasing order of depth. Each segment is represented by the depth and the transmittances of its first endpoints and its slopes. (Do not confuse these segments with the light ray segments. Light ray segments are decomposed from a light ray in the space of shadow camera, while the segments of transmittance functions are used to represent the function.) If the ray intersects an opaque object at depth $z_o$, a segment is inserted into the array to show that the transmittance drops to zero at this depth and the other segments with larger depths
than \( z_0 \) are neglected. After constructing the deep shadow map, for each segment the transmittance is set to 1 and slope is set to 0, as there are no particles initially and the light does not attenuate in the space.

Figure 2.7: The evolution of a transmittance function in an explosion. The first row of figures shows that a light ray penetrates the region of an explosion. The second row shows how the transmittance function varies at different time slots.

**Updating the map in each time step**

When the simulation in a time slot is finished, the density of each cell may change and the segments representing the transmittance functions have to be updated. In each cell, we record the set of segments and the corresponding transmittance functions affected by its density. If this density changes, each affected functions are updated. For example, in Figure 2.7, cell \( c \) in the space volume is penetrated by a light ray and the light ray segment \( r \) is the portion of the ray in \( c \). \( s \) is a segment in the transmittance function \( \ell(z) \) that represents how the transmittance varies along the ray segment \( r \). When the density of cell \( c \) increases from time slot 1 to time slot 2, the slope of segment \( s \) is updated to be steeper and the transmittance values for the segments at larger depth than \( s \) are set to be lower. Later on, the density decreases from slot 2 to slot 3. Following this, the slope of segment \( s \) becomes flatter and the transmittance values of the segments at larger depth rise back to higher values.
Updating the textures for the objects

After updating the deep map, the textures of objects are redrawn by sampling the map. For each pixel on each texture, the corresponding world coordinate of its center \((x, y, z)\) is found. Then the brightness \(B\) at that point is computed and the color of the pixel is adjusted. To compute \(B\) at a point, the point is transformed and the point \((x', y', z')\) in the space of shadow camera is obtained. A filter centered at \((x', y')\) on the image plane of shadow camera is applied on the deep shadow map:

\[
B = \frac{\sum w_i \ell(x_i, y_i, z')}{\sum w_i},
\]

(2.24)

where \((x_i, y_i)\) represents the centers of the pixels within the filter radius on the image plane of shadow camera, \(z'\) is the depth of the input point and \(w_i\) represents the weights for a gaussian filter. In this sampling process, we have to obtain the transmittance at depth \(z'\) from function \(\ell(x, y, z)\). First, the segment containing this depth value has to be identified. Remember that for each segment representing a transmittance function, the depth of the first endpoint is stored, and these segments are sorted in an increasing order of depth. By comparing \(z'\) with these depth values, the segment containing \(z'\) can be found (For efficiency, a binary search is applied). The transmittance at \(z'\) is interpolated linearly by using the transmittance at the endpoint and the slope of the segment. In order to achieve different shadowing effects, the filter radius can be adjusted to control the number of samples used (see Figure 2.8).

![Figure 2.8: Shadows rendered with different filter radius values. Sharper shadows are produced by using a smaller filter while softer results can be obtained by using a larger filter.](image)

radius = 1  
radius = 3  
radius = 5
Figure 2.9 shows the images of an explosion simulation containing shadows cast from smokes.

Figure 2.9: Simulation with and without shadows. Shadows are not drawn at the left image while they are drawn at the right one.

2.6.2 RENDERING PARTICLES

Precise explosion simulations requires a complex rendering system for particles which can handle the illumination caused by the combustion. The system by Feldman et al. [10] consider the scattering of light passing through the burning particles with using a hierarchical method. Such system can produce accurate result but the processing time is too long for millions of particles and the implementation is also complicated.

Figure 2.10: The list of images showing how the texture are rendered corresponding to different temperatures. The left most image is the original texture, while others are the rendered textures in the simulation with the temperature values increase from left to right.
For our simulation, we make use of a simpler and faster texture mapping method. Each particle is rendered as a small quad accompanied with a circular texture pattern (figure 2.10). When the particles move, each quad is adjusted such that it faces directly to the camera, which means it is perpendicular to the line between camera and the particle (figure 2.11). In this way, each particle looks like a sphere no matter where the camera and the particle are.

![Diagram](image.png)

Figure 2.11: Relationship between camera and the rendering quads for particles. Each quad is perpendicular to the line from the particle to camera.

As mentioned in the last section, there are two types of particles - fuel and soot. Different textures are assigned to them. Since it is assumed that all fuel particles burn when the explosion starts, the texture tends to be red like fire. For soot particles, the texture pattern is similar but it is gray in color. When the fuel particles burn, the intensity of particles should varies according to the temperature (figure 2.10). This can be achieved by adjusting the brightness of the quads representing the particles. Moreover, those quads has to be semitransparent and accumulative such that a dense set of burning particles can produce large illumination. The accumulation of quads means that the quads of fuel should not occlude each other. When a quad goes in front of another, their colors are added together to get a brighter result. This accumulation effect of the semitransparent quads can be observed at the left side of figure 2.12. Finally, figure 2.13 shows the difference between rendering particles as points and as texture-mapped quads. The region close to the center of explosion, where the particles are dense, is brighter when the texture-mapping is used.
Figure 2.12: A simulated example with a sparse set of texture-mapped particles.

Figure 2.13: An example of simulated explosion. Left: Result made by point rendering method. Right: Result made by texture mapping method.
CHAPTER 3

EXPLOSION EDITING

3.1 RELATED WORK

After building the simulation system, the explosion editing system is introduced. Early work by Gates [13] allowed flow fields to be manually authored via control splines to aid in the rendering of water. Work by Foster et al. [12] allowed users to interject several animation controls, such as explosive forces and vorticity, into a simulation to produce various behaviors for water. Later, Fedkiw et al. [9] utilizes user-specified velocities to aid in directing the behavior of water. More recently, techniques have been proposed that try to compute the necessary simulation parameters that satisfied user-specified constraints. Most noticeable are Popovic [22] and Popovic et al. [23] whose work targeted rigid body dynamics and Chenney et al. [5] work on probabilistic sampling for multi-body dynamics. Recently, Treuille et al. [31] presented elegant work to allow keyframe specification for smoke simulations. In this work, an objective function is used to compute the necessary time-varying wind forces such that particles in the simulation approximate key-frames (expressed as particle densities) while minimizing the amount of wind forces used.

Our approach edits physics-based explosions directly in the spirit of Gates [13]. Since the velocity and temperature field of fluid directs the result of the explosion simulation, these two attributes at each time step can be saved for each pre-computed fluid simulation. With the system's editing interface, the user can alter the saved attributes in a specific simulation. The techniques used are similar to image warping and image metamorphosis [2], [17], [27]. The particle simulation and the rendering system (whose flow is shown in Figure 2.3) can then be played back again based on this altered fluid data in order to render a new explosion. Figure 3.1 shows the interface for playing back simulations and producing
animations.

In the next section, the approaches to warp the velocity and temperature field are introduced. These warping approaches are built on 2D velocity fields and temperature fields. The method to propagate the edited results to 3D are discussed in 3.3.

3.2 WARping PROCESS

First, we can consider the computed simulation as a set of states, $S_0, S_1, ..., S_n$, where the index $i$ represents a time step, $n$ is the total number of steps, and the state $S_i$ represents the simulation outcome over the computational grid. For our purposes, we are interested in only two of the computed attributes per grid, the velocity and temperature. These are denoted as:

$$S_i(x, y) = (v, T), \quad (3.1)$$

where $(x, y)$ is the coordinate of the simulation space at time step $i$, $v = (v_x, v_y, v_z)$ is the vector representing the velocity, and $T$ is the temperature. Figure 3.2
Figure 3.2: (a) computed flow-field at some time step (b) computed temperature (c) explosion with particle-system played back.

shows an example of a computed velocity, temperature and corresponding explosion generated using particle playback. In our warping process, we focus on 2D simulations, which means the grid contains only a slice of cells. This is due to two reasons. Firstly, the warping approaches (including Coons Patch Interpolation and Scatter Point Interpolation) for velocity and temperature target 2D fields, and they make use of homographies, which transforms points from one 2D space to another. This is discussed more in the following subsections. The other reason is the difficulty for the user to manipulate the velocity and temperature field in 3D.

After putting our focus on 2D simulation, a simple user interface is provided to allow the user to warp the velocity field and temperature field (Figure 3.3). Several warps can be applied in tandem to produce a desired behavior. As with image morphing, key-frames can be warped manually, with the tween warps interpolated. To specify a deformation, the user draws a local coordinate frame in the source simulation space, denoted as:

\[(x, y) = L(s, t),\]  
\[(s', t') = W(s, t),\]
Figure 3.3: The editing interface.

where the function $W$ gets the parameters $(s,t)$ from the local coordinate frame and returns a new position $(s',t')$ in the target grid. Then the warped position in the target simulation space can be obtained by:

$$(x',y') = L^{-1}(s',t').$$

(3.4)

The relationship between each space is illustrated in figure 3.4, where $W^{-1}$ is the inverse of warping function $W$. Notice that the source and target spaces have to

Figure 3.4: Relationships between simulation spaces and local spaces.
be linked through the parameter space \((s, t)\). Here, we introduce two deformation approaches: scattered point interpolation and Coons patch interpolation.

### 3.2.1 SCATTER POINT INTERPOLATION

Scatter point interpolation examples

Example 1: Source and edited temperature fields.

Example 2: Source and edited temperature fields.

Figure 3.5: This figure shows the interface for warping the explosion state and how the temperature values are warped using scatter point interpolation. The blue points represent the source positions of control points while the green ones represent the target positions. In these two examples, different transformations of a local frame (the orange box) are applied in order to obtain different warping results.

Scatter point interpolation is useful in defining deformations with only a few control points. The positions of the 2D control points can be specified both in the source and target space, which allows the user to define how the space should be warped. Ruprecht et al. [27] works on image warping with scattered point interpolation. Similarly, this technique can be applied to the warping of velocity fields and temperature fields.
For convenience, the control points are defined inside of a local coordinate frame (the orange box in figure 3.5), which is associated with the local coordinate system \((s, t)\). This frame not only describes the region for warping, but also allows the user to spatially transform a group of control points by moving, rotating or scaling it. The blue points (connected by lines) in figure 3.5 represent the source positions of the control point while the green ones show the target positions. Let \(p_1, p_2, \ldots, p_n\) and \(q_1, q_2, \ldots, q_n\) be the source positions and target positions, a forward warping, \(W_{\text{forward}}\), using radial basis functions discussed in Ruprecht et al. [26] can be defined as:

\[
\mathbf{x}' = W_{\text{forward}}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i R(d_i(\mathbf{x})),
\]

where \(\mathbf{x} = (s, t)\), which is a point in the source local space, \(\mathbf{x}' = (s', t')\), which is a point in the target local space, and \(d_i(\mathbf{x})\) is the distance of \(\mathbf{x}\) to \(p_i\) and \(n\) is the number of control points. The radial basis functions, \(R(d)\), are Hardy's multiquadratics [15],

\[
R(d) = (d^2 + r^2)^{\frac{1}{2}} \quad \text{with} \quad r = \min_{i \neq j} d_i(\mathbf{x}_j).
\]

The coefficients, \(\alpha_i\), can be uniquely computed using the positions of control points. This involves solving the linear system,

\[
\mathbf{q}_i = \sum_{j=1}^{n} \alpha_j R(d_j(p_i)), \quad \forall i \in [1, n],
\]

which can be done by using methods like the LU and Cholesky Factorization [8].

After solving the coefficients, the warping can be used to maps points from the source space to the target space.

In order to guarantee that the velocity and temperature at every cell in the target space are computed, a backward warping process is a better choice instead of a forward one. For the backward warping, we can solve the coefficients in an inverse way:

\[
\mathbf{p}_i = \sum_{j=1}^{n} \alpha_j R(d'_j(q_i)), \quad i \in [1, n],
\]

where \(d'_i(\mathbf{x})\) is the distance between \(\mathbf{x}\) and \(q_i\). Then the backward warping
$W_{\text{backward}}$ is:

$$x = W_{\text{backward}}(x') = \sum_{i=1}^{n} \alpha_i R(d'_i(x')),$$

(3.9)

In this way, for each point $x'$ in the target space, a corresponding $x$ can be found and the fluid attributes can be moved from $x$ to $x'$.

As mentioned before, the source and target simulation space is linked through the parameter space $(s,t)$, where the warping function is defined. According to the relationships between different spaces shown in figure 3.4, the algorithm for computing the warped velocity field and temperature field can be deduced as:

**ALGORITHM OF WARPING USING SCATTER POINT INTERPOLATION**

for each cell$_{i,j}$ do

$(x',y') \gets$ center of cell$_{i,j}$

$(x,y) \gets L(W^{-1}(L^{-1}(x',y')))$

$(T_{\text{target}})_{i,j} \gets T_{\text{src}}(x,y)$

$(v_{\text{target}})_{i,j} \gets L(W(L^{-1}((x,y) + v_{\text{src}}(x,y)))) - (x',y')$

end

In this algorithm, $W^{-1}$ corresponds to the backward warping using the scatter point interpolation, which means $W^{-1} = W_{\text{backward}}$. Warping the scalar temperature component, $T$, is straight-forward, while the computation of velocity is not trivial which makes use of both backward and forward warping. We first explain how to obtain the forward warping $W$, which is the inverse of $W_{\text{backward}}$. At the beginning, $W_{\text{forward}}$ defined in equation 3.5 is tested for this purpose. Unfortunately, it is found that, in this scatter point approach, $W_{\text{forward}}$ is the inverse of $W_{\text{backward}}$ only at the control points:

$$x = W_{\text{forward}}(W_{\text{backward}}(x)), \text{ for } x \in \{q_i : i = 1, 2..., n\}$$

Thus, the forward mapping is performed using dense forward sampling of $(s,t)$ together with re-sampling in the target space $(s',t')$. Each small quadratic region in $(s,t)$ space is mapped to another quadratic region in $(s',t')$ space through a homography (figure 3.6), which is a $3\times3$ transformation matrix. Hence, there is a 2D array of homographies over the space, which is pre-computed based on the backward warping.
Figure 3.6: The forward warping $W$ is approximated by a 2D array of homographies.

Figure 3.7: Relationship between the source velocity and the target velocity. The new velocity is constructed with using $p'$ and $q'$, where $p'$ is an input point and $q'$ is the warped point corresponding to $q$.

For the warping of velocity at a cell, instead of using the velocity at the source point, more attractive result can be obtained by warping both the magnitude and direction of the source velocity. Suppose we start from point $p' = (x', y')$ (the center of cell$_{i,j}$) at the target space and $p = (x, y)$ is the corresponding point at the source space, when the velocity, $v = v_{src}(x, y)$ is obtained, point $q$ can be found by tracing away from $p$ using $v$. The corresponding warped point $q'$ is then used to find the warped velocity at the target space (Figure 3.7). That is,

$$(v_{target})_{i,j} = q' - p'$$
$$= L(W(L^{-1}(q))) - p'$$
$$= L(W(L^{-1}((x, y) + v_{src}(x, y)))) - (x', y').$$
A warping example of velocity field using scatter point interpolation is demonstrated in Figure 3.8.

![Figure 3.8: The source and target velocity field in an editing example using scatter point interpolation.](image)

### 3.2.2 COONS PATCH INTERPOLATION

Coons patch interpolation examples

![Example 1: Source and edited temperature fields.](image)

Example 2: Source and edited temperature fields.

![Example 2: Source and edited temperature fields.](image)

Figure 3.9: Two ways to edit with Coons patch interpolation. The user can warp the fluid attributes by changing the source Coons patch (example 1) or target Coons patch (example 2).
Coons patch interpolation [7] allows deformations to be specified using four boundary curves. Given curves, \( c_1(v), c_2(u), c_3(v), c_4(u) \), where \( (s, t) = c_4(\cdot) \); the two opposite-curve pairs (left-right, top-bottom) are parameterized by variables \( u \) and \( v \) respectively, where a Catmull-Rom Spline [4] is chosen as the representation of these curves. The interior space inside the boundary curves can be parameterized by \( (u, v) \) using the following Coons patch equation:

\[
W_c(u, v) = [1 - u \ u] \begin{bmatrix} c_4(v) \\ c_2(v) \end{bmatrix} + [c_1(u) \ c_3(u)] \begin{bmatrix} 1 - v \\ v \end{bmatrix} - [1 - u \ u] \begin{bmatrix} c_1(0) & c_4(1) \\ c_2(0) & c_3(1) \end{bmatrix} \begin{bmatrix} 1 - v \\ v \end{bmatrix},
\]

(3.10)

where \( W_c(u, v) \) returns an \((s, t)\) coordinate in the local frame. In our system, \((u, v)\), with \( u, v \in [0, 1] \), is a new parameter space that links the source and target spaces. Figure 3.9 shows the interface for the warping using Coons patches, where the user can create the patches and drag the control points. Notice that there are two separate patches associated with the source space and the target space, which means the user can edit by adjusting the points in either space. In order to achieve this function, two warping functions \( W_{c1} \) and \( W_{c2} \) are defined for mapping the parameter space \((u, v)\) to the source local frame \((s, t)\) and target local frame \((s', t')\). The diagram (Figure 3.10) shows the relationships between each space.

![Figure 3.10: Relationships between simulation spaces, local spaces and the parameter space.](image)

39
The warping function $W$, which maps points from the source Coons patch to the target Coons patch, can be approximated by a set of homographies similar to the scatter point interpolation approach. First of all, we start from a grid of sample points in the $(u, v)$ space. By warping these points with $W_{c1}$ and $W_{c2}$, the point correspondences between source space and target space can be established, based on which the required homographies can be constructed (Figure 3.11). Notice that when the warping function is used, it is insufficient for just providing a point in the $(s, t)$ space as input, since the corresponding $(u, v)$ has also to be known for identifying which particular homography should be used.

![Diagram](image)

Figure 3.11: Homographies are used to map points from source Coons patch to target Coons patch.

Now we discuss the procedures for handling the warping of Coons patch interpolation. One major problem of the Coons patch approach is that there is no defined backward warping. Since an inverse Coons patch interpolation cannot be easily computed, we can only start from points in $(u, v)$ space and warp them to get the corresponding points in $(s, t)$ or $(s', t')$ space. Also we do not try to establish a set of backward homographies for mapping points from $(s', t')$ to $(s, t)$ and start the process in $(x, y)$ like the scatter point approach, because the required $(u, v)$ point for identifying the backward homography cannot be found easily. To handle warping under these constraints, a forward approach is used:
ALGORITHM OF WARping USING COONS PATCH INTERPOLATION

for \( i = 0 \) to \( N_u \) do
  for \( j = 0 \) to \( N_v \) do
    \((x, y) \leftarrow L(W_{c1}(u_i, v_j))\)
    \((x', y') \leftarrow L(W_{c2}(u_i, v_j))\)
    \(T \leftarrow T_{vec}(x, y)\)
    \(v \leftarrow L(W(L^{-1}((x, y) + v_{vec}(x, y)))) - (x', y')\)
    \(Distribute\_Attr\_To\_Cells(T, v, x', y')\)
  end
end

\(N_u, N_v\): The number of samples along \( u \) and \( v \).

The warping starts from a grid of sample points in \((u, v)\) space. To avoid missing data that can occur in the target space, a dense set of samples are used. For each sample, the corresponding \((x, y)\) and \((x', y')\) points are acquired after applying warping and local frame transformation. The source attributes at \((x, y)\) can then be used to compute the warped attributes at \((x', y')\) like the scatter point approach. Finally, the function \(Distribute\_Attr\_To\_Cells\) distributes the attributes \(T\) and \(v\) to the neighbor cells of \((x', y')\) according to the distance between their centers and point \((x', y')\).

Deformation using Coons patch interpolation is intuitive to manipulation and useful in specifying large deformations. Figure 3.12 is an editing example of velocity field.

![Figure 3.12: The source and target velocity field in an editing example using Coons patch interpolation.](image)

41
Figure 3.13: (a) The volume is defined by the selection boxes at the starting and ending frames in the \((2D + t)\) space. (b) An new explosion generated by combining several fluid simulations together with the cut-and-paste method.

After finishing the warping of an explosion simulation, we consider how to combine several 2D results together in order to produce special explosion effects. Hence the system is enhanced to support the cutting and pasting of fluid attributes from one simulation to another. In Figure 3.13a, selection boxes at the start and end frames of a selected clip in a simulation can be specified by the user, while those at the intermediate frames are interpolated. This defines a volume of selection over the \((2D + t)\) space, where \(t\) is the simulation time. In order to avoid sharp edges that may appear at the field of fluid attributes, when this volume of data is cut-and-pasted to another simulation, cut boundaries are blended according to an alpha function. Figure 3.13b is a new explosion made by combining several simulations together with this method.

### 3.3 PROPAGATE EDITING FROM 2D TO 3D

In contrast with the previous section where the editing method was based on 2D simulation, here, a method to handle 3D explosions is discussed. A simple method is to allow the user to specify warping in a 2D slice only, and propagate it to other 2D slices along the normal direction of the edited slice. Figure 3.14
shows an example of 3D editing using this approach, where the smoke and fire at the right side is split into two. Two Coons patches are used in this example so that a replicate of attribute fields is created.

![Image of 3D editing example]

Figure 3.14: An example that propagates warping along the normal direction to the edited slice. (a) Original explosion. (b) Warped explosion. (c) A slice of the temperature field for the original explosion (d) The edited temperature field and the Coons patches that guide the warping.

Another method is to use the interpolation technique suggested by Rasmussen et al. [24] for simulating large scale phenomena. In this method, two slices of 2D simulation are placed to the 3D space and an interpolation method is developed to fill in the empty regions between these cross-sections. An interface is provided for the user to import and arrange two slices in a cylindrical fashion (Figure 3.15). Both slices are put at the origin and have different angles to the $xy$-plane. Each of these is cut into two half slices and the attributes at each position in the empty space are interpolated based on the attributes inside.
Figure 3.15: The four-viewpoint interface built for 3D interpolation and aggregation of simulations. For interpolation, two simulation slices are displayed in each window. For aggregation, the boundary boxes of several 3D simulations can be displayed.

Figure 3.16: The figure shows the parameter used to interpolate attributes in the cylindrical approach. (Top view)

As can be seen in Figure 3.16, for each target point, the source positions of the two closest half slices are found and these attributes are used to compute the values at the target point:

\[ T = \frac{T_1\theta_2 + T_2\theta_1}{\theta_1 + \theta_2}, \quad v = \frac{M_1(v_1)\theta_2 + M_2(v_2)\theta_1}{\theta_1 + \theta_2}, \]  

(3.11)
where $T_1, T_2, v_1, v_2$ are the source temperatures and velocities, $\theta_1, \theta_1$ are the angular distances from the target point to two half slices, and $M_1, M_2$ are the transformation functions for velocities. These functions adjust the $x$ and $z$ components of $v_1$ and $v_2$ such that in the top view, their directions are parallel to the direction from the origin to the target point.

For advecting the particles in the 3D space, a 3D computational grid has to be built and the velocity and temperature at each cell have to be computed. However, this is also time consuming and requires a large memory space for the grid. Rasmussen et al. [24] suggests a better method to enable us to simply compute the attributes at the positions of particles. Instead of preparing a large data grid, the particles are advected based on two 2D simulations. At each time step, the fluid velocity and temperature at the position of each particle is interpolated according to Eq. 3.11. They are then used for updating the attributes of particles, such that the particles are moved and rendered correctly.

This approach is also good for generating symmetric results and both simulated 2D explosion and warped explosion can be used as inputs in order to produce desirable 3D effects. Figure 3.17 two results generated using cylindrical interpolation.
Figure 3.17: Examples of cylindrical interpolation. (a) Simulated 2D explosion. (b) Warped version of (a). (c) Result of cylindrical interpolation that uses (a) for both input slices (Front view). (d) Result that uses both (a) and (b) as inputs (Front view). (e) Top-front view of (c). (f) Top-front view of (d).
CHAPTER 4

RESULTS

In this chapter, the simulation and editing results generated by our system are demonstrated. The simulation system is developed using C++ programming language and OpenGL graphics package, on a Pentium 4, 2.4GHz, 512MB RAM workstation. It firstly synthesizes a set of bitmap frames and the animation (with 30Hz frame rate) can be produced with a simple MPEG encoder or another video editing software, Premiere. In order to enhance the visual effect, the user can preset the parameters of the camera such that its position and view angle varies during the simulation (Figure 4.1).

Figure 4.1: The starting, ending and current position of the camera can be displayed in the playback interface in one shot. The orange pyramid represents the current camera while the grey ones show its starting and ending positions.
4.1 RESULTS OF SIMULATION

Four simulation results are shown. Simulations are first saved and then played back for rendering. Their information and the system performance are listed below (Table 4.1), where the simulation time includes the time spent on simulation and saving data, and the rendering time includes the time for loading fluid data, playing back the particle simulation and rendering objects and particles. Afterwards, the rendered explosion, velocity field and temperature field are shown in the following pages (Figure 4.2-4.7). In some examples, the shadow is mapped on the objects in the environment. For those cases, the rendering time is increased by 30 to 90 seconds per frame, depending on the resolution of the deep shadow map and the number of cells used to represent the space.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>2D</td>
<td>3D</td>
<td>3D</td>
<td>3D</td>
</tr>
<tr>
<td>Size of grid</td>
<td>50×50×1</td>
<td>45×30×40</td>
<td>45×30×40</td>
<td>70×45×70</td>
</tr>
<tr>
<td>Number of frames</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>400</td>
</tr>
<tr>
<td>Simulation time per frame</td>
<td>90 ms</td>
<td>1.4 s</td>
<td>1.3 s</td>
<td>5.6 s</td>
</tr>
<tr>
<td>Rendering time per frame (no shadows)</td>
<td>5 s</td>
<td>8 s</td>
<td>8 s</td>
<td>5.5 s</td>
</tr>
<tr>
<td>Maximum number of particles</td>
<td>1.5 millions</td>
<td>3 millions</td>
<td>3 millions</td>
<td>2.1 millions</td>
</tr>
<tr>
<td>Average number of particles per frame</td>
<td>1.2 millions</td>
<td>2 millions</td>
<td>2 millions</td>
<td>1.3 millions</td>
</tr>
</tbody>
</table>

Table 4.1: Information of the simulations and system performance.
Figure 4.2: Simulation 1 - A 2D simulation. Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the explosion, the velocity field and the temperature field.
Figure 4.3: Simulation 2 - A 3D simulation blocked by a "∏"-shaped object (The ceil of the object is not shown in the figures). Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the explosion, a horizontal slice of the velocity field and a horizontal slice of the temperature field.
Figure 4.4: Simulation 3 - A 3D simulation with one more obstacle than simulation 2. Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the explosion, a horizontal slice of the velocity field and a horizontal slice of the temperature field.
In simulation 4, cylindrical obstacles and background textures are used. Figure 4.5 below shows how the cylindrical obstacles are voxelized and the simulation results are provided in the following pages (Figure 4.6 and 4.7).

Obstacles and background.

Voxelized obstacles and background.

Figure 4.5: The cylindrical obstacles and background of simulation 4.
Figure 4.6: Simulation 4 - A 3D simulation with cylindrical obstacles. Frame 10, 60, 100, 160 and 300 are shown from top to bottom. The three columns correspond to animation generated with different view points. In the last column, variable camera position and view angle are used. Shadows are rendered in this example.
Figure 4.7: The velocity and temperature field of simulation 4. The left and right columns show a vertical velocity slice and a vertical temperature slice of the simulation, while the middle one shows a horizontal velocity slice in the top view (The obstacles are not displayed).
4.2 RESULTS OF EDITING

After getting a set of simulations, warping, cut-and-paste and cylindrical interpolation for 3D simulations can be applied to generate more interesting results. In each example, we first illustrate how the functions in the system are used when editing. Then the results and processing time are also reported.

4.2.1 EXAMPLE OF WARPING 1

The first example (Figure 4.8) is the editing of simulation 1 in the last section, where the right side of the explosion is stretched. The time spent for warping the 300 frames of this simulation is 26.3 seconds. The rendering time is 22 minutes, which is similar to simulation 1. The figure below shows the editing based on Coons patch interpolation. More frames of the edited simulation, velocity field and temperature field are shown on next page (Figure 4.9).

![source and target coons patches and temperature field](image1)

Source and target Coons patches and the temperature field.

![source and target explosion](image2)

Source and target explosion.

Figure 4.8: Warping example 1. The source and target Coons patches and the corresponding explosions are shown.
Figure 4.9: The result of warping example 1. Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the edited explosion, the velocity field and the temperature field.
4.2.2  EXAMPLE OF WARPING 2

This example is similar to example 1 but it applies two warpings to simulation 1 instead of one (Figure 4.10). In this example, the explosion is firstly bent using a Coons patch. Then the left side is stretched up. The processing time for these two passes are 26 s and 17.4 s respectively and the rendering time is the same as the last example. The second pass takes less time since the warped area is smaller.

![Source and target field in the first pass.](image)

Source and target field in the second pass.

![Source and target explosion.](image)

Figure 4.10: Warping example 2. Two editing passes using Coons patch and source and target explosions are shown.
Figure 4.11: The result of warping example 2. Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the edited explosion, velocity field and temperature field.
4.2.3 EXAMPLE OF WARPING 3

Example 3 is the example of 3D propagation mentioned in Section 3.3 (Figure 3.14). Two Coons patches are used to split the fluid such that it travels around the new obstacle. Since the warping has to be propagated to each slice and the warped attributes for two Coons patches have to be computed, the processing time is 176 seconds. But it is significantly faster than redoing the simulation, which takes about 420 seconds. The rendering time for the edited explosion is 40 minutes. Figure 4.12 shows that the warping result behaves as if an additional obstacle is in the environment, which is comparable to the result obtained from a full simulation with the obstacle in place (see the simulation 3 in the last section).

Figure 4.12: Warping example 3. (a) Result of simulation 2. (b) Result of simulation 3. (c) The edited version of simulation 2 which has similar behavior to simulation 3.
Figure 4.13: The result of warping example 3. Frame 20, 60, 100, 140 and 180 are shown from top to bottom. The left, middle and right columns correspond to the edited explosion, velocity field and temperature field.
4.2.4 EXAMPLE OF CUT-AND-PASTE

The fourth example is the cut-and-paste example that appears in Figure 3.13b. The source simulations are a mushroom-shaped 2D simulation and a circular 2D simulation (Figure 4.14). Selections are defined in these two simulation, which is then cut and pasted to a new simulation. In the bottom-left figure below, the orange boxes represent the pasted fields in an empty simulation. Finally, the system combines these fields and generates the explosion in the bottom-right image. The time for this synthesis process is 90 seconds. The animation is rendered in 34 minutes.

Figure 4.14: A cut-and-paste example. The figures on the first row shows the source simulation. The bottom-left one illustrates how the simulations are combined, while the bottom-right one shows the result.
Figure 4.15: The result of the cut-and-paste example. Frame 30, 80, 130 and 180 are shown from top to bottom. The left and right columns correspond to the edited explosion, and the velocity field.
4.2.5 EXAMPLE OF CUT-AND-PASTE AND CYLINDRICAL INTERPOLATION

In this example, different parts of a source 2D simulations are cut out and combined using the cut-and-paste method. Cylindrical interpolation is applied to generate a 3D explosion with the 2D simulation obtained. Figure 4.16 illustrates the processes to create the result. To generate the 2D result in figure 4.16c from the source simulation, 66.4 s processing time is needed. The particle simulation is played back for rendering the explosion in figure 4.16d based on the cylindrical interpolation. This takes 8.3 seconds per frame on average and 48 minutes in total. This example shows that the cylindrical approach is an efficient way to

Figure 4.16: A cut-and-paste and cylindrical interpolation example. (a) Source simulation. (b) The pasted fields. (c) The 2D simulation produced from those pasted fields. (d) The 3D simulation obtained from applying cylindrical interpolation using (c) as inputs.
generate 3D simulations, while the cut-and-paste method can be used to create new explosions that exceed the capabilities of a physics-based simulation. These are some frames of the example:

Figure 4.17: The result of the cut-and-paste and cylindrical interpolation example. Frame 40, 80, 120, 160, 200 and 240 are shown.
4.2.6 EXAMPLE OF AGGREGATING SIMULATIONS

In this example, simulation 4 and a horizontal circular explosion are aggregated to generate a more complicate explosion. First, a 400-frame circular simulation is generated, which takes 360 seconds. Then simulation 4 and the circular simulation are imported to the interface for aggregation (Figure 4.18). Finally, the particle simulation is played back directly with the velocity and temperature field of both 3D simulations (No processing time of data is needed) and the rendering time is 50 minutes for the new animation without shadows and 6 hours for the one with shadows. The aggregation results can be viewed on next page.

Figure 4.18: Example of aggregating simulations. Two simulations are imported in the interface for aggregation.
Figure 4.19: Result obtained from aggregating two simulations. Frame 20, 60, 130, 200, 270 and 340 are shown.
CHAPTER 5

CONCLUSION AND FUTURE EXTENSIONS

5.1 Conclusion

In this thesis, a general framework of explosion simulation, which includes fluid simulation, particle simulation and rendering, is presented. This involves discussion of the essential physical rules for ensuring the realism of explosion motion and the handling of the interactions between fluid and environment objects. The core attributes - velocity and temperature - of the fluid can be saved for the purpose of explosion editing and simulation playback. Then the resultant simulation can be rendered based on particles by applying the texture-mapping method.

Further on, approaches to editing the 2D explosion simulation are proposed, the fundamental idea of which is to deform the velocity and temperature field of the simulations. The deformation of temperature involves only warping the temperature values, while for velocity, the warping of both direction and magnitude is also considered. For deformation, two approaches are discussed. The scatter point interpolation allows the user to specify point correspondences and is good for defining small scale warping, while the Coon patch interpolation uses two patches to define the deformation, which is easy for manipulation and suitable for large deformations. After editing, there are functions for propagating the 2D results to 3D. The user can choose to propagate the editing along the normal direction of an edited slice or to do interpolation with a cylindrical approach. All these editing methods are developed for avoiding the re-simulations and trials of parameters, which can save much time for the production of an explosion.

Finally a set of explosion results is shown. They include the results obtained
from a physical-based simulation and from editing. Although the warped explosions may behave incorrectly in terms of physics, they are still smooth and convincing.

5.2 FUTURE EXTENSIONS

Since current movie and entertainment industries require highly realistic explosions, it may not be acceptable to introduce non-physical components during the generation of a synthesized explosion. Thus a major extension for this project is to develop a physics-based editing approach. That is, the edited fields of attributes should not violate the conservation rules of fluid, such that the behavior of edited explosion is physically correct.

Other possible enhancements of the system include the rendering scheme and simulation performance. The current approach does not consider the illuminations caused by the burning effect. In order to handle the illuminations, a new rendering system has to be developed to handle the light emitted and scattered by each particle [16], [21]. On the other hand, small scale turbulence can also be added to our system to improve the realism of explosion simulations with using turbulent wind field [28]. Finally, the performance for our simulation system is unsatisfactory when the grid size is large. A possible extension is to use a matrix solver based on the GPU [3] to enhance the simulation speed.
REFERENCES


