Graph-Based Fracture Models for Rigid Body Explosions

by

Jessica Leigh Socha

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Explosions are one of the most powerful and devastating natural phenomena. The pressure front from the blast wave of an explosion can cause fracture of objects in its vicinity and create flying debris. In this thesis, I outline a previously proposed explosion model. An explosion is treated as a fluid with its behaviour governed by the Navier-Stokes equations and the gaseous products modeled using particles. Explosions are simulated as a means for initiating fracture of rigid bodies in the vicinity of an explosion.

In contrast to fracture models that are based on physics, I propose a new approach to simulating fracture which treats fracturing the rigid body as a pre-processing step. A rigid body can be pre-fractured by treating it as graph and using one of the two proposed graph partitioning algorithms to divide the object into the desired number of pieces. By treating fracture as a pre-processing step, much less computation need be done during the simulation than models based on physics.

It is shown that the recursive breadth-first search graph partitioning algorithm produces physically realistic results for shattering windows that are consistent with observations of real broken windows. The curvature-driven spectral partitioning algorithm fractures objects into two pieces where the object is weakest, where weakest is defined by the area with largest curvature. Numerical simulations of explosions and fracture were conducted to produce data that was used by a ray tracer and volume renderer to create images which were assembled into animations.

Acknowledgments

First and foremost I would like to thank my supervisor, Professor Justin Wan for his guidance and supervision. This work would not have been possible without his support and friendship. I thank him for always being available to talk to and providing excellent direction with my thesis work.

I would like to extend sincerest gratitude towards Professor Gladimir Baranoski. The knowledge I have gained through his graduate courses and informal meetings with him has been very valuable. I would also like to thank Professor Stephen Mann and Professor Bruce Simpson for being a part of my thesis committee.

Thank you to my family for your continued love and support. Thank you to my brother Steve for creating the background images used in the final animations. A special thank you to my boyfriend Edwin Vane for providing moral support throughout my thesis work. It is greatly appreciated.

Last, but not least, I would like to thank my colleagues in the Scientific Computation Lab and the Computer Graphics Lab for their friendship and providing an enjoyable environment to work in.

The meshes used in this thesis were obtained from the Introduction to Computer Graphics course CS 688.

Dedication

For my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Explosions are one of the most powerful and devastating natural phenomena. Explosions are a sudden release of energy creating an outward-propagating blast wave. This blast wave is capable of causing other secondary effects such as soot, fireballs, flying debris and the fracture and deformation of objects in the explosion's vicinity. It is desirable to simulate explosions for a variety of reasons. Real explosions are dangerous and costly to produce. There is very little danger involved in simulating an explosion. A simulation is also repeatable and can be changed in an iterative fashion until the desired effect is achieved.

Widely seen in movies and videos games, most people are fortunate to have not witnessed an explosion in person. In movies and video games, the explosions are generally much larger than would actually occur in the observed setting to increase dramatic effect and create a visually appealing explosion. The simulation of explosions, both the blast wave and secondary effects, is one area of computer graphics that has been explored. The focus of these simulations is to produce realistic visual effects using physical equations. This thesis outlines a recently proposed explosion model that was implemented as a means for initiating fracture.

The fracture and shattering of objects is another special effect seen in movies and video games. Fracture has also been studied from a physics point of view, and simulated in computer graphics applications. Physically based models include those that compute stress and strain when determining where a fracture will occur, and those that are dependent on the presence of a crack to determine where it will spread. While these models produce physically realistic results, they are also computationally expensive. This thesis proposes a different approach to fracture.

The objects to be fractured are rigid bodies, where a rigid body is one that does not deform when force is applied to it. A rigid body is represented as a polygonal

mesh. This thesis proposes treating a rigid body as a graph and using a graph partitioning algorithm to divide the mesh into the desired number of pieces prior to the simulation. This allows the simulation to be concerned with only updating the pieces of the rigid body each time-step without calculating stress or strain on the object, which is time consuming. This approach to fracture is justified in the fact that explosions happen on a very short time scale. Objects will fracture almost instantaneously when in the vicinity of an explosion and the pieces will quickly fly away from the source of the explosion. The same visual results are obtained with a graph-based approach as would be obtained with a physically based simulation, but the graph-based approach to pre-fracture objects results in a much faster simulation.

My contribution in this thesis is the proposal of the recursive breadth-first search graph partitioning algorithm and the curvature-driven spectral partitioning algorithm as graph-based approaches to simulating fracture. The breadth-first search graph partitioning algorithm classifies the vertices of the mesh into levels, and splits the vertices based on this classification. To demonstrate the appropriateness of this algorithm for fracturing objects in the vicinity of explosions, a mesh representing a piece of glass is placed in front of an explosion and the fracture is observed. This animation shows that while not physically based, the breadth-first search graph partitioning algorithm is capable of producing physically realistic results for fracture when used with explosions. The curvature-driven spectral partitioning algorithm is based on the spectral bisection algorithm which is motivated by the harmonics of a vibrating string. The curvature-driven spectral partitioning algorithm uses curvature of each edge to determine the weakest place to break the mesh, where the weakest region of the mesh is defined as the region with largest curvature. To demonstrate that this model produces physically realistic results, a complex polygonal is fractured resulting in two pieces that would intuitively be expected.

The remainder of this thesis is organized as follows. Chapter 2 outlines the equations used in the explosion model, and how the motion of the explosion is tracked through the use of particles. Chapter 3 introduces the concepts of rigid bodies, including a description of how they are represented and how their motion is updated. The chapter concludes with a discussion of triangulation algorithms for creating 2D meshes. Chapter 4 proposes two graph-based approaches to fracture, the breadth-first search graph partitioning algorithm and the curvature-driven spectral partitioning algorithm. Chapter 5 presents results of simulations followed by conclusions and future work. Appendix A provides a discussion of the ray tracer and volume renderer used to produce images from the simulation results.

Chapter 2

Modeling Explosions

A model for visualizing explosions must consist of three parts: a representation of the explosion, a model for its behaviour, and a graphical renderer to visualize its appearance. The explosion is modeled as a fluid, with the behaviour of an explosion dictated by a set of physical equations for fluid flow known as the Navier-Stokes equations. Particles are placed in the fluid to represent the gaseous products of the explosion such as fuel and soot. Their position is tracked to determine how the explosion is moving. Rendering of explosions is discussed in Appendix A.

2.1 Previous Work

Stam and Fiume [43] proposed one of the first models for animating gaseous phenomena such as flames and the spread of fire. In their model, a user specifies a wind field to govern the behaviour of a gas. The temperature and expansion of the gas is computed using an advection-diffusion equation, with the assumption of constant density which captures the main characteristics of a gas. The source of the gas is either user-defined or based on a model of chemical reactions. Their model is advantageous if user-control is desirable. However, the simple advection-diffusion equation does not give a complete physical model.

Uhl and Blanc-Talon [46] proposed an explosion model that improves on smooth particle hydrodynamics. Smooth particle hydrodynamics governs the evolution of density, thermal energy, particle speed, radius and position. Uhl and Blanc-Talon modified these equations to account for different types of gases and behaviours by adding chemical rules and measured data. Since the model is an N-body problem,

the complexity is $\mathcal{O}(N^2)$. However, it is capable of producing physically realistic images, which was the goal of the authors.

The work by Neff and Fiume [30] models the blast wave of an explosion while ignoring the explosion cloud. They simulated the detonation of a spherical charge of trinitrotoluene (TNT) and studied how the wave propagated. The blast wave is approximated by a triangular pressure pulse that is calculated using blast curves. The blast curve approach is less computationally expensive than a simulation involving physical equations, and data is readily available. Mazarak et al. [29] also proposed a blast wave model to study the pressure changes across a shock front. They used the Friedlander equation [6] to compute pressure changes at any distance from the source of the explosion. Both models use a simplified set of equations that can capture some key aspects of a blast wave. However, accuracy of the simulation is sacrificed for speed.

Yngve et al. [49] proposed an explosion model based on computational fluid dynamics. Their model assumes that a detonation has occurred according to some initial conditions. The explosion is modeled post-detonation as a fluid using the equations for compressible, viscous flow. To simplify the equations, it assumed that changes in vibrational energies or molecules is negligible and that air is at chemical equilibrium. This model was one of the first to use a computational fluid dynamics approach to model explosions. The model is physically-based and able to capture many of the effects of an explosion that blast wave models cannot. However, it is computationally more expensive than approaches that are not based on physical equations.

The work conducted by Martins et al. [28] expands on Mazarak's model to simulate the blast wave and its effects. The goal of the authors was to create a physically realistic model capable of executing in real time. Their blast wave model is a combination of simplified physically-based equations and experimental data. They are able to capture some relevant physical properties of an explosion, such as the blast wave, smoke and dust, while simplifying and optimizing others. As a result, Martins et al. produce a "physically believable" model that is not too computationally expensive.

Feldman et al. [12] describe a model for animating suspended particle explosions. Their work disregards the blast wave in favour of modeling the motion of the gas with the incompressible fluid equations. The divergence field is adjusted at a specified location to simulate a detonation and allow the gaseous products to expand. Many of the parameters that define the behaviour of the explosion are user-specified. Although their explosion model does not include the blast wave, it provides a physically-based simulation of the gas that was comparable to observa-

tions from real explosions.

2.2 Modeling Fluid Flow

The explosion model described in this thesis follows the work of Feldman et al. The explosion is modeled as an incompressible, inviscid fluid that evolves over time according to the incompressible Navier-Stokes equations. The use of the compressible fluid equations would lend itself to a more physically-based model of explosions. However, the incompressible Navier-Stokes equations used by Feldman et al. were also used here because they lend themselves to easier implementation and are suitable for the purpose of this thesis.

The Navier-Stokes equations approximate physical laws governing motion, temperature and combustion. They have been used to animate the motion of natural phenomena such as water [14][15][41], gas [16], smoke [11][42], fire [31] and flames [22], viscoelastic fluids such as soap and clay [17], and the melting of wax [9].

Incompressible flows are characterized by the property that density changes are negligible [18, page 11]. In the proposed explosion model, density is assumed to be constant at a value of 1 and the density term, ρ, has been removed from the equations for simplicity. The law of conservation of momentum is given by

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} - \frac{\partial (uw)}{\partial z} + g_x$$
$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} - \frac{\partial (vw)}{\partial z} + g_y$$
$$\frac{\partial w}{\partial t} + \frac{\partial p}{\partial z} = \frac{1}{Re}\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) - \frac{\partial (uw)}{\partial x} - \frac{\partial (vw)}{\partial y} - \frac{\partial (w^2)}{\partial z} + g_z,$$

where u, v and w is the fluid velocity in the x, y and z direction respectively, t is time, p is pressure, Re is the Reynold number, and g_x, g_y and g_z are external forces in the x, y and z directions respectively.

For an incompressible fluid, the divergence of the flow field is zero. The fluid obeys conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0.$$

To simulate the detonation of an explosion, a rapid change in pressure is required. However, to cause an explosion to expand, fluid needs to be added to the

environment which affects the conservation of mass equation. To use the incompressible equations to simulate explosions, the conservation of mass equation is modified to include the term ϕ,

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = \phi,$$

where ϕ controls the rate of expansion. The value of ϕ is initialized to zero everywhere except in the region of the detonation. To allow the gaseous products to add heat to the fluid, the energy transport equation

$$\frac{\partial T}{\partial t} = c_k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) - \frac{\partial (uT)}{\partial x} - \frac{\partial (vT)}{\partial y} - \frac{\partial (wT)}{\partial z} + \dot{H} - c_r\left(\frac{T - T_a}{T_{max} - T_a}\right)^4$$

is incorporated, where T is temperature, T_a is the ambient temperature of the fluid, T_{max} is the maximum temperature in the environment, c_k is a thermal conductivity constant, c_r is a cooling constant and $\dot{H}$ is the heat transfered into the fluid. Diffusive transfer, the expansion of heat, is approximated by the first term. The second term models the transfer of heat by the fluid. The third term represents heat transfered into the fluid from external sources. The last term approximates radiative loss into the environment.

2.3 Modeling Gaseous Products

The motion of the gaseous products, fuel and soot, in the explosion is modeled using a particle system. The seminal work by Reeves [38] introduced particle systems and their use for modeling "fuzzy" objects without hard boundaries such as smoke, fire and clouds. A particle system is responsible for generating particles, moving and transforming them, and possibly allowing them to die over time. An in-depth look at the implementation of particle systems is given in [47].

The basic attributes for a particle include position, velocity, radius and mass. In addition to these variables, the particles used to model the gaseous products have the attributes listed in Table 2.1, which also gives the variables used for each attribute used in equations that follow. In addition to these variables, a particle has a 'type' denoting whether it is fuel or soot. A particle also has two flags, one for whether it is ignited or not, and the other for whether soot has been created from the particle or not.

At the beginning of the simulation, a user-defined number of fuel particles is created. Fuel particles can also be created at the end of each time-step. A fuel particle's position is initialized to a random location on a sphere of radius r_d centered

Attribute	Variable
position	pos
velocity	vel
mass	m
temperature	$\dot{Y}$
thermal mass	c_m
radius	r
soot mass	s
ignition temperature	T_i

Table 2.1: Particle attributes

at the position of the detonation, pos_{det}. Velocity, temperature, and soot mass of a fuel particle are initially zero. The flags for whether a fuel particle is ignited, and whether soot has been created are both initially set to false. The mass, thermal mass, and ignition temperature of both fuel and soot particles are set to user defined parameters. A particle's radius is 0.1.

The motion of the fuel particles is dictated by the movement of underlying fluid flow of the explosion. The fuel particles also affect the fluid flow. Fuel particles, once ignited, will start to consume their own mass at the burning rate, z. Ignited particles generate heat at a rate

$$\dot{H}_{i,j,k} = b_h z$$

where b_h is the amount of heat released per unit combusted mass of fuel. An ignited particle also accumulates soot mass at a rate

$$s = b_s z$$

where b_s is the mass of soot produced per unit combusted mass of fuel. Ignited particles add gaseous products to the fluid according to

$$\phi_{i,j,k} = \phi_{i,j,k} + \frac{1}{V} b_g z$$

where V is the volume of a cell and b_g is the volume of gas released per unit combusted mass of fuel. Lastly, if an ignited particle has accumulated soot mass reaching a set threshold, then a soot particle is created. The initial position and velocity of the soot particle is set to the same position and velocity of the fuel particle from which it was created, with a small perturbation. The movement of soot particles is also dictated by the fluid flow, but soot particles do not affect the flow field and do not affect the conservation of mass equation.

2.4 Numerical Solutions

The Navier-Stokes equations cannot be solved exactly. Instead, numerical solutions are used to approximate the solution of the equations at each time-step. The numerical method consists of a series of steps: determine the size of the time-step, set boundary conditions, solve for the right-hand side of the equation, update pressure, update velocity.

2.4.1 Discretization

To solve the Navier-Stokes equations numerically, the continuous domain must be discretized. The values of the solution are determined at a finite number of points in the domain. Consider a three dimensional region

$$\Omega := [0, a] \times [0, b] \times [0, c] \subset \mathbb{R}^3.$$

The region is divided into $i_{max} \times j_{max} \times k_{max}$ uniformly sized *cells* or *voxels* of width Δx, height Δy and depth Δz, where $\Delta x = \frac{a}{i_{max}}$, $\Delta y = \frac{b}{j_{max}}$ and $\Delta z = \frac{c}{k_{max}}$.

Scalar values, such as pressure, p, and temperature, T, are located in the cell centers. The values for the components of the velocity vectors are stored using a *staggered grid*; each component is located in the center of one of the faces of the cell. The u component is located at the left face of the cell, the v component is located at the bottom face of the cell and the w component is located at the rear face of the cell, see Figure 2.1. As a result, the discrete values of u, v, w and P are located on four separate grids, each shifted by half a grid spacing to the bottom, to the left, to the rear and to the rear lower left, respectively. This arrangement prevents possible oscillations that could occur if all unknowns are evaluated at the same grid points.

2.4.2 Finite Differences

Finite differences [44] can be used to solve the Navier-Stokes equations on the staggered grid. The idea of finite difference schemes is to replace derivatives by finite differences. This can be accomplished in many ways. For example, a second derivative appearing in the Navier-Stokes equation can be solved using a central-differencing scheme

$$\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j,k} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{(\Delta x)^2}.$$

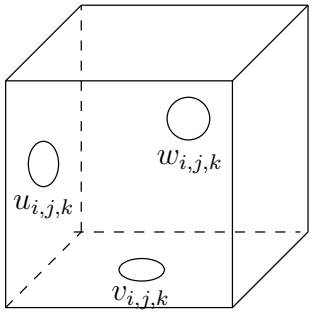

Figure 2.1: A cell at position (i, j, k) in the staggered grid

Other second derivates are treated analogously. The spatial pressure terms are also solved using a central difference scheme, for example

$$\left[\frac{\partial p}{\partial x}\right]_{i,j,k} = \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}.$$

The discretization of the convective terms is more involved. For example, to discretize $\partial(uv)/\partial x$, points are needed in both the u and v directions. The average of the u and v terms is used, giving

$$\left[\frac{\partial(uv)}{\partial x}\right]_{i,j,k} = \frac{1}{\Delta x}\left(\left(\frac{u_{i,j,k} + u_{i,j+1,k}}{2}\frac{v_{i,j,k} + v_{i+1,j,k}}{2}\right) - \left(\frac{u_{i,j-1,k} + u_{i,j,k}}{2}\frac{v_{i,j-1,k} + v_{i+1,j-1,k}}{2}\right)\right).$$

Similarly, to discretize $\partial(u^2)/\partial x$, central differencing of averaged values is used

$$\left[\frac{\partial(u^2)}{\partial x}\right]_{i,j,k} = \frac{1}{\Delta x}\left(\left(\frac{u_{i,j,k} + u_{i+1,j,k}}{2}\right)^2 - \left(\frac{u_{i-1,j,k} + u_{i,j,k}}{2}\right)^2\right).$$

The convective terms in the momentum equations become dominant when Re is large or the velocity is high. To prevent this from happening, the donor-cell method can be incorporated. The donor-cell method computes a weighted sum of two finite difference techniques.

Euler's method is used to discretize the time derivates as follows:

$$\left[\frac{\partial u}{\partial t}\right]^{n+1} = \frac{u^{n+1} - u^n}{\Delta t}.$$

The finite difference techniques described are also applicable to the temperature equation. However, since temperature is defined at the center of cells, there is no

need to average the u, v and w terms in the evaluation of $\partial(uT)/\partial x$, $\partial(vT)/\partial y$ and $\partial(wT)/\partial z$, for example

$$\left[\frac{\partial(uT)}{\partial x}\right] = \frac{1}{\Delta x}\left(u_{i,j,k}\frac{T_{i,j,k}+T_{i+1,j,k}}{2} - u_{i-1,j,k}\frac{T_{i-1,j,k}+T_{i,j,k}}{2}\right).$$

The momentum equation discretized by time is

$$u^{n+1} = u^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} - \frac{\partial(uw)}{\partial z} + g_x - \frac{\partial p}{\partial x}\right]$$

$$v^{n+1} = v^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}+\frac{\partial^2 v}{\partial z^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} - \frac{\partial(vw)}{\partial z} + g_y - \frac{\partial p}{\partial y}\right]$$

$$w^{n+1} = w^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2}+\frac{\partial^2 w}{\partial z^2}\right) - \frac{\partial(uw)}{\partial x} - \frac{\partial(vw)}{\partial y} - \frac{\partial(w^2)}{\partial z} + g_z - \frac{\partial p}{\partial z}\right].$$

The variables F^n, G^n and H^n are introduced as abbreviations defined by

$$\begin{aligned}
F^n &= u^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} - \frac{\partial(uw)}{\partial z} + g_x\right] \\
G^n &= v^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}+\frac{\partial^2 v}{\partial z^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} - \frac{\partial(vw)}{\partial z} + g_y\right] \\
H^n &= w^n + \Delta t\left[\frac{1}{Re}\left(\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2}+\frac{\partial^2 w}{\partial z^2}\right) - \frac{\partial(uw)}{\partial x} - \frac{\partial(vw)}{\partial y} - \frac{\partial(w^2)}{\partial z} + g_z\right].
\end{aligned}$$

F^n, G^n and H^n are evaluated at time t_n while the pressure terms, $\partial p/\partial x$, $\partial p/\partial y$ and $\partial p/\partial z$ are associated with time t_{n+1}. This gives

$$\begin{aligned}
u^{n+1} &= F^n - \Delta t\frac{\partial p^{n+1}}{\partial x} \\
v^{n+1} &= G^n - \Delta t\frac{\partial p^{n+1}}{\partial y} \\
w^{n+1} &= H^n - \Delta t\frac{\partial p^{n+1}}{\partial z}.
\end{aligned}$$

The Courant-Friedrichs-Lewy (CFL) condition,

$$\Delta t = \min\Bigg(\frac{Re}{2}\left(\frac{1}{(\Delta x)^2}+\frac{1}{(\Delta y)^2}+\frac{1}{(\Delta z)^2}\right)^{-1}, \frac{RePr}{2}\left(\frac{1}{(\Delta x)^2}+\frac{1}{(\Delta y)^2}+\frac{1}{(\Delta z)^2}\right)^{-1}, \frac{\Delta x}{|u_{max}|}, \frac{\Delta y}{|v_{max}|}, \frac{\Delta z}{|w_{max}|}\Bigg)$$

is imposed on the value of Δt to ensure numerical stability and avoid generating oscillations.

2.4.3 Initial and Boundary Conditions

The driving force behind the explosion is the incorporation of ϕ to create outward pressure. Consequently, it is desirable to have the velocity vanish at the boundary to satisfy a no-slip condition. The no-slip boundary conditions are

$$\begin{aligned}
u_{(i,j,0)} &= -u_{(i,j,1)} & u_{(i,j,k_{max+1})} &= -u_{(i,j,k_{max})} & i &\in [0, i_{max}], j \in [0, j_{max+1}] \\
u_{(i,0,k)} &= -u_{(i,1,k)} & u_{(i,j_{max+1},k)} &= -u_{(i,j_{max},k)} & i &\in [0, i_{max}], k \in [0, k_{max+1}] \\
v_{(i,j,0)} &= -v_{(i,j,1)} & v_{(i,j,k_{max+1})} &= -v_{(i,j,k_{max})} & i &\in [0, i_{max+1}], j \in [0, j_{max}] \\
v_{(0,j,k)} &= -v_{(1,j,k)} & v_{(i_{max+1},j,k)} &= -v_{(i_{max},j,k)} & j &\in [0, j_{max}], k \in [0, k_{max+1}] \\
w_{(i,0,k)} &= -w_{(i,1,k)} & w_{(i,j_{max+1},k)} &= -w_{(i,j_{max},k)} & i &\in [0, i_{max+1}], k \in [0, k_{max}] \\
w_{(0,j,k)} &= -w_{(1,j,k)} & w_{(i_{max+1},j,k)} &= -w_{(i_{max},j,k)} & j &\in [0, j_{max+1}], k \in [0, k_{max}].
\end{aligned}$$

Neumann boundary conditions are used for temperature:

$$\begin{aligned}
T_{(i,j,0)} &= T_{(i,j,1)} & T_{(i,j,k_{max+1})} &= T_{(i,j,k_{max})} & i &\in [1, i_{max}], j \in [1, j_{max}] \\
T_{(i,0,k)} &= T_{(i,1,k)} & T_{(i,j_{max+1},k)} &= T_{(i,j_{max},k)} & i &\in [1, i_{max}], k \in [1, k_{max}] \\
T_{(0,j,k)} &= T_{(1,j,k)} & T_{(i_{max+1},j,k)} &= T_{(i_{max},j,k)} & j &\in [1, j_{max}], k \in [1, k_{max}].
\end{aligned}$$

Temperature, heat, pressure and ϕ are initialized to zero. The velocities are initialized to small random numbers very close to zero to reduce symmetry in the pressure.

2.4.4 Solving for Pressure

The Poisson equation for pressure at time t_{n+1} is

$$\frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial^2 p^{n+1}}{\partial y^2} + \frac{\partial^2 p^{n+1}}{\partial z^2} = \frac{1}{\Delta t}\left(\frac{\partial F^n}{\partial x} + \frac{\partial G^n}{\partial y} + \frac{\partial H^n}{\partial z}\right).$$

To enforce the modified conservation of mass equation, a modified version of the Poisson equation,

$$\frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial^2 p^{n+1}}{\partial y^2} + \frac{\partial^2 p^{n+1}}{\partial z^2} = \frac{1}{\Delta t}\left(\frac{\partial F^n}{\partial x} + \frac{\partial G^n}{\partial y} + \frac{\partial H^n}{\partial z} - \phi\right).$$

is used when solving for pressure. This equation is discretized using the techniques described in section 2.4.2 resulting in $i_{max}j_{max}k_{max}$ equations in $i_{max}j_{max}k_{max}$ unknowns. Using a direct method such a Gaussian elimination to solve this system

of equations is too costly. Large, sparse systems such as the one obtained from the Poisson equation for pressure tend to be solved efficiently using iterative techniques such as conjugate gradient [44, page 327]. The conjugate gradient method begins with an initial guess for the solution and computes successive approximations to the solution by computing residuals which determine the search direction. The conjugate gradient method for solving $Ax = b$ is outlined in Algorithm 1, where A is the three-dimensional Laplacian, b is the right-hand side of the pressure equation and x is the value being solved for.

Algorithm 1 Conjugate Gradient

$x \Leftarrow 0$
$r \Leftarrow b$
$d \Leftarrow r$
$\delta_0 = r^T r$
$\delta_1 = \delta_0$
while $\delta_1 > tolerance$ **do**
  $q \Leftarrow Ad$
  $\alpha \Leftarrow \frac{\delta_1}{d^T q}$
  $x \Leftarrow x + \alpha d$
  $r \Leftarrow r - \alpha q$
  $\delta_0 \Leftarrow \delta_1$
  $\delta_1 \Leftarrow r^T r$
  $\beta \Leftarrow -\frac{\delta_1}{\delta_0}$
  $d \Leftarrow r + \beta d$
end while

2.4.5 Updating Particles

At the beginning of each time-step, the particles must be updated according to the underlying flow field determined by the equations of the explosion model. First, particles with a mass below a specified threshold are deleted. Remaining particles are considered in turn. The velocity and temperature of the fluid at the particle's current position are calculated by interpolating the velocity and temperature at the corners of the cell is occupies. The indices into the flow field, i, j and k, and scaling terms, dx, dy and dz, are computed based on the position of the particle (p_u, p_v, p_w). Since the fluid is being calculated over a staggered grid, see Figure 2.1, the computation of the velocity components and temperature are dealt with uniquely. The calculations in each case need to be offset in certain dimensions so that the

flow field is being indexed correctly. The indices and scalars for the velocity in the u direction are

$$\begin{aligned} i &= \texttt{int}(p_u/\Delta x) \\ dx &= p_u/\Delta x - i \\ j &= \texttt{int}((p_v + 0.5\Delta y)/\Delta y) \\ dy &= (p_v + 0.5\Delta y)/\Delta y - j \\ k &= \texttt{int}((p_w + 0.5\Delta z)/\Delta z) \\ dz &= (p_w + 0.5\Delta z)/\Delta z - k, \end{aligned}$$

where the operator `int` means to take the integer value of the result. The values Δx, Δy and Δz denote the size of a cell in each of the x, y and z dimensions. The indices and scalars for the velocity in the v direction are

$$\begin{aligned} i &= \texttt{int}((p_u + 0.5\Delta x)/\Delta x) \\ dx &= (p_u + 0.5\Delta x)/\Delta x - i \\ j &= \texttt{int}(p_v \Delta y) \\ dy &= p_v/\Delta y - j \\ k &= \texttt{int}((p_w + 0.5\Delta z)/\Delta z) \\ dz &= (p_w + 0.5\Delta z)/\Delta z - k. \end{aligned}$$

The indices and scalars for the velocity in the w direction are

$$\begin{aligned} i &= \texttt{int}((p_u + 0.5\Delta x)/\Delta x) \\ dx &= (p_u + 0.5\Delta x)/\Delta x - i \\ j &= \texttt{int}((p_v + 0.5\Delta y)/\Delta y) \\ dy &= (p_v + 0.5\Delta y)/\Delta y - j \\ k &= \texttt{int}(p_w \Delta z) \\ dz &= p_w/\Delta z - k. \end{aligned}$$

Lastly, the indices and scalars for temperature are

$$\begin{aligned} i &= \texttt{int}((p_u + 0.5\Delta x)/\Delta x) \\ dx &= (p_u + 0.5\Delta x)/\Delta x - i \\ j &= \texttt{int}((p_v + 0.5\Delta y)/\Delta y) \\ dy &= (p_v + 0.5\Delta y)/\Delta y - j \\ k &= \texttt{int}((p_w + 0.5\Delta z)/\Delta z) \\ dz &= (p_w + 0.5\Delta z)/\Delta z - k. \end{aligned}$$

Each component of the interpolated velocity, $vel_{interp} = (u_{interp}, v_{interp}, w_{interp})$, is calculated using the appropriate indices and scalars. For example,

$$\begin{aligned} u_{interp} = \quad & (1-dz) \quad \Big((1-dy) \Big((1-dx) w_{i,j,k} + dx w_{i+1,j,k} \Big) + \\ & \qquad dy \Big((1-dx) w_{i,j+1,k} + dx w_{i+1,j+1,k} \Big) \Big) + \\ & \quad dz \qquad \Big((1-dy) \Big((1-dx) w_{i,j,k+1} + dx w_{i+1,j,k+1} \Big) + \\ & \qquad dy \Big((1-dx) w_{i,j+1,k+1} + dx w_{i+1,j+1,k+1} \Big) \Big). \end{aligned}$$

The remaining two components of the interpolated velocity are computed analogously using the appropriate values for the indices and scalars. The interpolated velocity is used with the current velocity of a particle, vel, to determine the force

$$f = \alpha_d r^2 |vel_{interp} - vel|(vel_{interp} - vel) \tag{2.1}$$

on a particle, where α_d is the drag coefficient. If the mass of a particle is below a threshold then this calculation is ignored and the force is zero. The position of a particle is updated according to

$$pos = pos + \Delta t vel,$$

where Δt is the size of the time-step. If this calculation results in a particle's position lying outside of the computational grid, then the particle is deleted. The velocity of a particle is updated by

$$vel = vel + \frac{\Delta t f}{m},$$

where f is the force calculated in equation (2.1) and m is the mass of the particle.

A particle's temperature, $\dot{Y}$, is updated by

$$\dot{Y} = \dot{Y} + \frac{\Delta t \alpha_h r^2 |\dot{Y}_i - \dot{Y}|}{c_m},$$

where $\dot{Y}_i$ is the interpolated temperature and α_h is the coefficient of thermal conductivity. If the thermal mass of a particle is below a threshold then this calculation is ignored. The entire process of updating particles is summarized in Algorithm 2.

Algorithm 2 Updating of particles

```
for each particle p do
  if p is of type fuel and mass is below threshold then
    Delete p
  end if
  Calculate interpolated velocity of p from flow field
  Calculate force on p
  Update position of p
  if the position of p is outside the boundary then
    Delete p
  end if
  Calculate acceleration of p
  Update velocity of p
  Calculate interpolated temperature of p
  if p is of type fuel then
    if p's temperature is above ignition temperature and p not ignited then
      Ignite p
    end if
    if p is ignited then
      p will consume its own mass
      p accumulates soot mass
      p generates heat
      p adds gaseous products into the fluid
    end if
    if soot mass of p is above threshold then
      Create a soot particle from p
      Flag that soot has been created from p
    end if
  end if
end for
```

2.4.6 The Time-Stepping Algorithm

The entire procedure to solve the Navier-Stokes equations and update the particles and rigid bodies is adapted from [18, page 135] and given in Algorithm 3. Rigid bodies are listed here for completeness. The implementation details for rigid bodies are given in Chapter 3.

Algorithm 3 Navier-Stokes solver [18, page 135]

$t \Leftarrow 0$
create initial particles and rigid bodies
initialize $u, v, w, p, T, \dot{H}, \phi$
while $t < t_{end}$ **do**
  Export the particle and rigid bodies for rendering
  Update particles and rigid bodies
  Select Δt according to the CFL condition
  Set boundary values for u, v, w, T
  Compute F^n, G^n, H^n, T^n
  Compute the right-hand side of the pressure equation
  Perform conjugate gradient to solve the pressure equation
  Compute u^{n+1}, v^{n+1} and w^{n+1}
  Reset $\dot{H}$ and ϕ to 0
  Create new particles
  $t \Leftarrow t + \Delta t$
end while

A summary of the variables used in the Navier-Stokes equations is given in Table 2.2.

c_k	thermal conductivity constant
c_r	cooling constant
g_x	external force in x direction
g_y	external force in y direction
g_z	external force in z direction
$\dot{H}$	heat transfered into the fluid
p	pressure
Re	Reynold number
t	time
T	temperature
T_a	ambient temperature
T_{max}	maximum temperature in the environment
u	fluid velocity in x direction
v	fluid velocity in y direction
w	fluid velocity in z direction

Table 2.2: Variables used in the Navier-Stokes equations

Chapter 3

Rigid Bodies

To enhance the explosion model and study the effects of flying debris and the fracture of objects, rigid bodies were added to the simulation in the form of spheres, cubes and polygonal meshes. Rigid bodies are objects that produce negligible deformation when a force is applied to them. A force can only produce a change in the position of the center of mass and rotational motion. In the simulations produced with the explosion model, the motion of the rigid bodies is unconstrained, that is that the simulation is not concerned with collisions between rigid bodies. Since explosions happen so fast and debris is moving quickly away from the detonation, collision detection is not considered.

3.1 Modeling Rigid Bodies

A rigid body is formally defined by a set parameters as presented in the work by Jansson [20] and given in Table 3.1. The variable names given will be used throughout the remainder of the section in descriptions and equations that follow. Each type of rigid body (sphere, cube and polygonal mesh) contain the parameters in the table in addition to other unique variables for each type. All three types of rigid bodies are initialized with a velocity, orientation and angular velocity of zero. The mass is defined by the user when creating the rigid body. The initialization of the remaining variables, and a discussion on the representation of each type of rigid body follows.

Spheres are created by specifying the center of mass and radius r. The moment of inertia tensor for a sphere is

$$\tilde{I} = \frac{2mr^2}{3}.$$

pos	position of center of mass
vel	velocity of center of mass
m	mass of the rigid body
α	orientation of the rigid body
ω	angular velocity
$\tilde{I}$	moment of inertia tensor
n_v	number of vertices
$verts$	list of vertices

Table 3.1: Parameters that define a rigid body

Points are sampled on the surface of a sphere at each of the poles, giving $n_v = 6$ and $verts = \{(r,0,0),(-r,0,0),(0,r,0),(0,-r,0),(0,0,r),(0,0,-r)\}$, where n_v is the number of vertices and $verts$ is the list of vertices.

Cubes are created by specifying the rear, left, lower corner, $corner = (c_u, c_v, c_w)$, and a $width$, $height$ and $depth$. The remaining seven vertices are calculated based on the specified corner and dimensions of the cube giving $n_v = 8$ and $verts = \{(c_u, c_v, c_w), (c_u, c_v, c_w+depth), (c_u, c_v+height, c_w), (c_u, c_v+height, c_w+depth), (c_u+width, c_v, c_w), (c_u+width, c_v, c_w+depth), (c_u+width, c_v+height, c_w), (c_u+width, c_v+height, c_w+depth)\}$. The $corner$ and $width, height$ and $depth$ are used to calculate the center of mass

$$pos = (c_u + width/2, c_v + height/2, c_w + depth/2). \tag{3.1}$$

All vertices are then updated by translating them by $(pos - corner)$ so that they are stored with respect to a center of mass at the origin. However the value of pos does not change from that calculated in equation (3.1). This is done to make the mechanics of updating the cube easier. The moment of inertia tensor for a cube is

$$\tilde{I} = \frac{2m \min(width, height, depth)^2}{3}.$$

Polygonal meshes are the most complicated of the three types of rigid bodies. A polygonal mesh is defined as follows. The number of vertices, n_v, is specified, follow by a list of vertices, $verts$. Then the number of faces, n_f, is given, followed by the faces, $faces$, which are indices into the $verts$ list. For simplification, polygonal meshes are only allowed to be specified using triangles. An example of a polygonal mesh of a cow is given in Figure 3.1.

The center of mass is approximated by

$$pos = (sumX/n_v, sumY/n_v, sumZ/n_v), \tag{3.2}$$

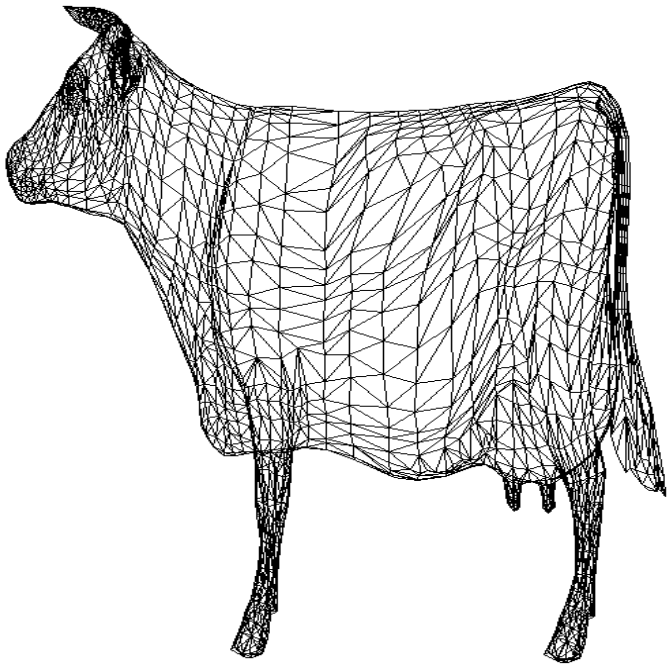

Figure 3.1: A polygonal mesh of a cow

which requires sums; $sumX$, $sumY$, $sumZ$; for each of the x, y and z components of each vertex respectively. As with cubes, all the vertices of the mesh are adjusted so that they are stored with respect to a center of mass at the origin by translating each vertex by $-pos$. The calculated pos from equation (3.2) remains unchanged. The moment of inertia for a mesh is

$$\tilde{I} = \frac{2m \min(width, height, depth)^2}{3}.$$

The $width$, $height$ and $depth$ of the mesh are calculated by finding the difference of the minimum and maximum value of each x, y and z co-ordinate respectively.

3.2 Mechanics of Rigid Body Motion

The motion of a rigid body is defined as translations of and rotations about the center of its mass. The mechanics of rigid body motion describe how a rigid body's position, $pos = (pos_u, pos_v, pos_w)$, and orientation, $\alpha = (\alpha_u, \alpha_v, \alpha_w)$, change over time according to linear velocity, vel, and angular velocity, ω, respectively. The velocity vectors are depicted in Figure 3.2.

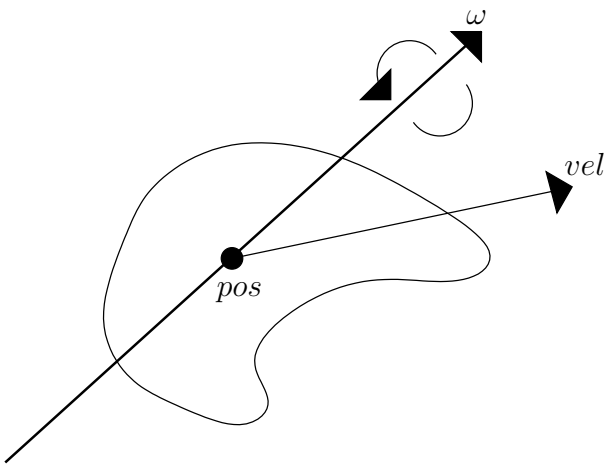

Figure 3.2: Linear and angular velocity of a rigid body

The position and orientation of the rigid bodies in the fluid volume are influenced by the underlying flow field determined by the explosion model. However, rigid bodies do not have any affect on the flow field. Explosions are so vigorous that the effect of the motion of rigid bodies is negligible.

Forces act on rigid bodies at a particular geometrical location on the body due to some external source, for example gravity. The total force on a rigid body is the sum of the forces at each vertex. Torque refers to the forces that produce rotation. The calculation of torque is dependent on the position of the vertex relative to the center of mass. As with force, the total torque acting on a body is the sum of the torque at each vertex.

To update the parameters associated with a rigid body, the force and torque at each vertex is calculated. For an in-depth derivation of the equations to follow, the reader is referred to [7].

Each vertex, $vert = (vert_u, vert_v, vert_w)$, of a rigid body is stored with respect to the center of mass considered at the origin. First, the actual position of the vertex,

$$pos_{actual} = R_u R_v R_w \begin{pmatrix} vert_{i_u} + pos_u \\ vert_{i_v} + pos_v \\ vert_{i_w} + pos_w \end{pmatrix}$$

is calculated based on the position of the center of mass, and the rotational matrices R_u, R_v, R_w which represent rotation about the center of mass in the x-axis, y-axis

and z-axis respectively, and are defined as:

$$R_u = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha_u) & -\sin(\alpha_u) \\ 0 & \sin(\alpha_u) & \cos(\alpha_u) \end{pmatrix}, \qquad R_v = \begin{pmatrix} \cos(\alpha_v) & 0 & \sin(\alpha_v) \\ 0 & 1 & 0 \\ -\sin(\alpha_v) & 0 & \cos(\alpha_v) \end{pmatrix},$$
$$R_w = \begin{pmatrix} \cos(\alpha_w) & -\sin(\alpha_w) & 0 \\ \sin(\alpha_w) & \cos(\alpha_w) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The interpolated velocity, vel_{interp}, at position pos_{actual} is calculated analogously to particles. The interpolated velocity is then used to calculate the force,

$$F = \frac{F_c vel_{interp}}{m},$$

at position pos_{actual}, where F_c is a user-defined constant. The vector,

$$r = pos - pos_{actual}$$

is a vector from a vertex's actual position to the center of mass. The force at pos_{actual} is decomposed into two components, parallel and perpendicular to r as depicted in Figure 3.3. The parallel vector,

$$F_{\parallel} = \frac{F \cdot r}{|r|^2} r,$$

is calculated by computing a vector projection of F onto r. The total parallel force is accumulated in a sum, $\|_{\sum}$. The perpendicular vector,

$$F_{\perp} = F - F_{\parallel},$$

is calculated by computing a vector difference of the force and its parallel component. The torque on a rigid body is determined by

$$\tau = \frac{1}{\tilde{I}}(r \times F_{\perp}).$$

The total torque is accumulated in a sum, $\tau_{\sum}$. Once the force and torque have been calculated at each vertex resulting in $\|_{\sum}$ and $\tau_{\sum}$, the motion of the rigid body is updated. First the position of the rigid body is updated:

$$pos = pos + \Delta t vel.$$

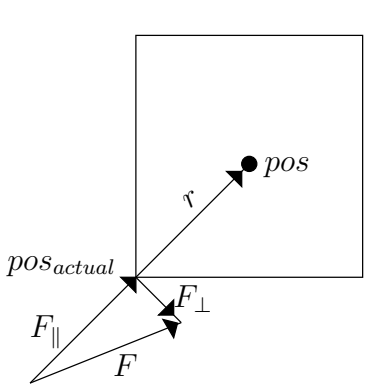

Figure 3.3: Decomposition of Force vector

The orientation is updated:

$$\alpha = \alpha + \Delta t \omega.$$

The velocity is updated:

$$vel = vel + \frac{\Delta t \parallel_{\sum}}{m}.$$

Lastly, angular velocity is updated:

$$\omega = \omega + \Delta t \tau_{\sum}.$$

The entire process of determining and updating the motion of a rigid body is summarized in Algorithm 4.

3.3 Triangulation

To simulate the fracturing of windows, mirrors and other planar objects, an algorithm that performs planar triangulation is used to create a 2D mesh for such objects. Two triangulation algorithms are considered to achieve different effects. Each algorithm starts with a number of random points and computes a triangulated mesh in the xy-plane.

The trisection triangulation algorithm is a divide and conquer algorithm. The algorithm begins by computing the convex hull of the points. The convex hull is the boundary of the smallest convex polygon containing all the points [37, page 18]. A random interior point is selected and used to divide the convex hull into triangles. For each triangle created, it is further trisected by selecting a random interior point until no such point exists. This algorithm does not result in a minimum-weight

Algorithm 4 Updating of rigid bodies

for each rigid body rb **do**
  $\sum_{\parallel} \Leftarrow 0$
  $\sum_{\tau} \Leftarrow 0$
  for each vertex v_i of rb **do**
    Calculate position pos_{actual} of v_i in volume based on pos and α
    Calculate interpolated velocity at pos_{actual}
    Determine the force F at pos_{actual}
    Decompose F into $F_{\parallel}$ and $F_{\perp}$
    $\sum_{\parallel} \Leftarrow \sum_{\parallel} + F_{\parallel}$
    Calculate torque τ on v_i
    $\sum_{\tau} \Leftarrow \sum_{\tau} + \tau$
  end for
  Update position of rb
  Update orientation of rb
  Update velocity of rb
  Update angular velocity of rb
end for

triangulation, one in which the total length of the triangulation edges is minimized. A triangulation of 200 points produced from the trisection triangulation algorithm is given in Figure 3.4.

The Delaunay triangulation [37, page 203] is a minimum-weight triangulation. The Delaunay triangulation first computes the Voronoi diagram of the points. A Voronoi diagram gives for each point p, the region for which p is the closest point. By connecting pairs of points that share a Voronoi edge, a triangulation is created. A Voronoi diagram, and the resulting Delaunay triangulation are given in Figures 3.5 and 3.6 respectively. The figures use the same 200 points that were used to obtain the trisection triangulation in Figure 3.4.

To use a 2D mesh from a triangulation algorithm to simulate glass, it is desirable that the glass have some thickness for light to pass through. Each piece of glass is a separate mesh with its own set of vertices and triangles. First, the list of triangles is duplicated. The triangles in first copy of the list are assigned a constant z coordinate, $depth_1$. The triangles in the second copy of the list are assigned another constant z coordinate, $depth_2$,

$$depth_2 = depth_1 + thickness$$

according to a user-defined constant for $thickness$.

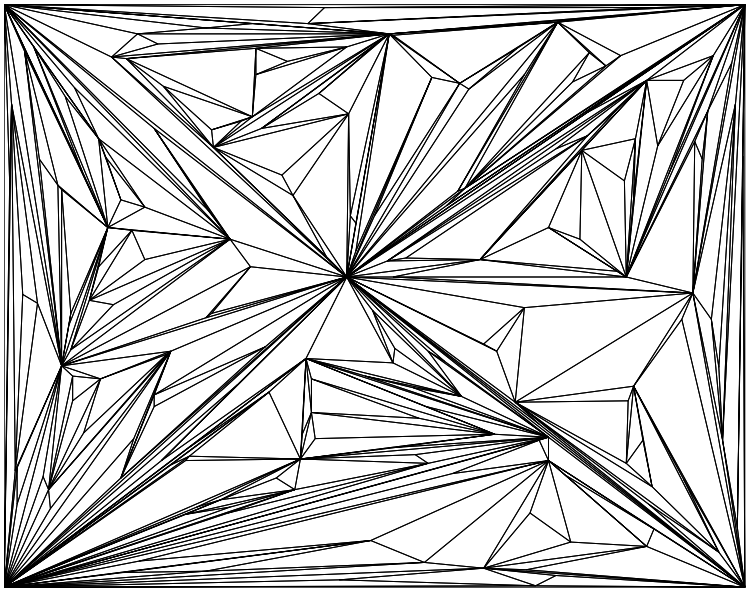

Figure 3.4: Trisection triangulation

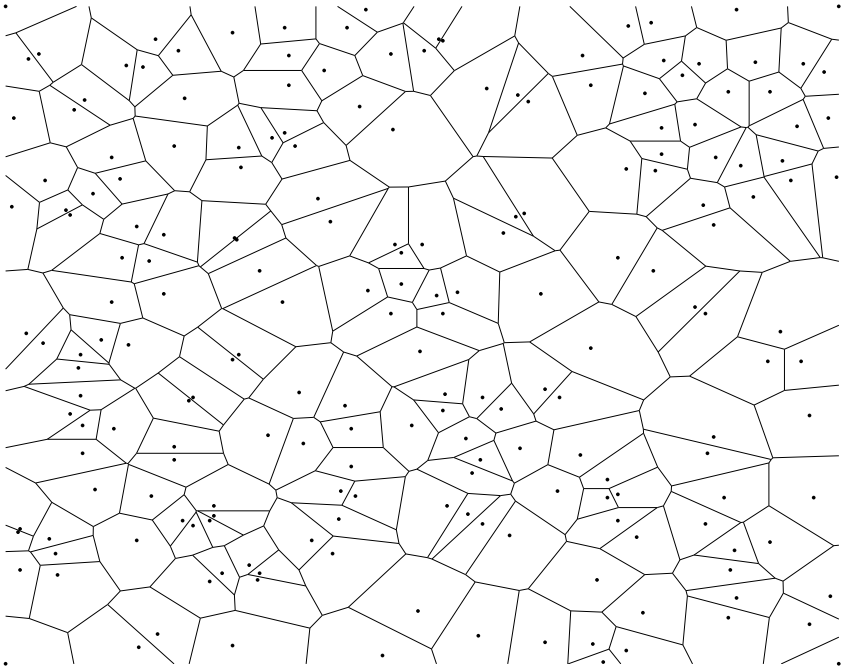

Figure 3.5: Voronoi diagram

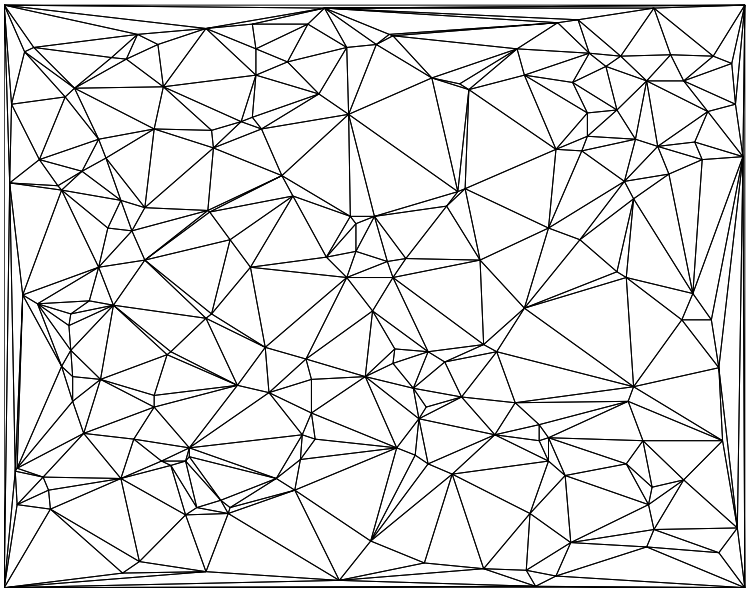

Figure 3.6: Delaunay triangulation of points in Figure 3.5

To join the two planar meshes together, the vertices on the exterior of the mesh are determined, and triangles are created joining exterior points. This is accomplished by first considering one of the triangle lists and determining which edges of each triangle are shared with other triangles. For each unshared edge, (x_1, y_1, $depth_1$), (x_2, y_2, $depth_1$) in the first list, there is a corresponding unshared edge, (x'_1, y'_1, $depth_2$), (x'_2, y'_2, $depth_2$) in the second list. Two triangles

$$(x_1, y_1, depth_1), (x'_2, y'_2, depth_2), (x'_1, y'_1, depth_2)$$

and

$$(x_1, y_1, depth_1), (x_1, y_2, depth_1), (x'_2, y'_2, depth_2)$$

are created to join these edges and create sides of the mesh so it will be become a solid 3D object.

Chapter 4

Graph-Based Fracture Models

Cracking and fracture are common effects of the destruction of objects. Cracks result from internal stress on an object caused by external forces or nonuniform expansion or contraction. The proposed models study the fracture of rigid bodies induced by explosions.

4.1 Previous Work

The seminal work by Terzopoulos et al. [45] introduced the mechanics of the deformation and fracture of a rigid body due to an external force. Norton et al. [32] implemented a physical simulation of the fracture of objects based on the work by Terzopoulos et al. The authors used a spring-mass model for the rigid bodies where the object is broken into equally sized cubes connected by springs. When the force on a spring connecting two nodes is greater than a threshold, the connection between the two nodes breaks. Fracture occurred only along the boundaries in the mesh creating artifacts.

Hirota et al. [19] proposed a physically-based simulation of crack patterns. Objects are modeled by specifying both the surface that is allowed to crack, and the layer below the surface where cracks do not reach. The surface layer is constructed of nodes representing volume elements with springs that represent the physical connections between the elements. Cracks are generated by contracting the object and simulated by the cutting of springs according to force constraints presented in the paper. Since the structure of an object relates to where it is likely to crack, a finely detailed surface model of the object is required to generate realistic crack patterns.

Mazarak et al. [29] model the blast wave of an explosion to simulate debris. Rigid bodies are modeled with connected voxels and links between adjacent voxels. The links are infinitely stiff, maintaining the rigidity of the object. The blast wave model is used to calculate pressure between links. A link is broken if its pressure exceeds a predetermined yield limit, simulating cracks. The complete fracture of the object may occur if a fragment of the object becomes disjoint from the remainder of the object. The proposed fracture model also takes into account multiple explosions. If an object is in the vicinity of the first explosion, the links are weakened, making it easily susceptible to damage from subsequent explosions. The proposed model is unable to account for non-uniform pressure waves.

Neff and Fiume [30] proposed a technique for the fracture of objects as the result of a blast wave. The fracture model is dependent on an initial crack being present in the model, determined by pressure calculated from the blast wave. This crack is then propagated in each direction, creating a crack tree. Objects are modeled using polygonal meshes. The geometry of the mesh is mapped to a panel on which forces and torques are calculated. A panel consists of the position of its center, its normal and its area. Since panels are two dimensional planes, the algorithm is not easily extensible to three dimensions.

O'Brien and Hodgins [33] animated fracture of objects based on linear elastic fracture mechanics. Objects are represented by a set of differential equations which are discretized for use with a finite element method. Local deformation is calculated by using Green's strain tensor. A stress tensor is calculated based on the material of the object. Together the stress and strain are used to calculate where cracks begin and where they will propagate. Once the placement of a crack has been determined, the elements must be modified to reflect the crack. This requires a local remeshing of the object in the vicinity of the crack. The fracture and distance a fracture can travel during a time-step is dependent on the number of elements used to represent the object.

In the work by Smith et al. [40], the authors present an efficient method for controlling the shattering of brittle objects. In addition to vertices specified on the surface of an object, interior vertices are added to create connected tetrahedra. The tetrahedra are converted into a representation of constraints connecting point masses. The distance-preserving constraints are associated with an ultimate strength by user-specified functions and heuristics based on the geometry of the object. The force on the constraints is calculated by solving for Lagrange multipliers using a conjugate gradient method. These forces indicate where and when an object will break. When a connection is broken, the constraints around it are weakened so that pre-existing cracks are more likely to spread rather than new cracks form-

ing. This user-controlled method is fast and produces realistic results. However, fractures can only occur along mesh boundaries, and a fully tetrahedralized mesh is required.

Martinent et al. [27] propose a fracture model in which the user can control the pattern of cracks, and the shape and size of the fragments. The model employs a Hybrid Tree model in which objects are represented as a hybrid of skeletal implicit surfaces and triangular meshes. To cause the fracture of an object, the user defines a crack pattern that is converted to a graph representation. The graph is mapped on the object to give the skeleton of the crack which is then converted into a volumetric representation. The crack is then removed from the original object. Their model allows for a user to efficiently simulate cracks and create interlocking fragments.

4.2 Graph Partitioning

The main difference in previously proposed fracture models is how meshes are modeled. Some authors employ a spring-mass model, some authors depend on the presence of a crack and determine the direction in which it spreads, while other authors compute stress and strain on objects to determine weakest points and allow them to crack. The drawbacks of all these models is twofold. First, these models are dynamic in nature. They need to compute forces at each time-step of the simulation to determine the nature of cracks and fracture. This can be a computationally expensive process. Secondly, these models all require some knowledge of physics to understand crack formation and fracture patterns.

The models proposed here treat fracturing as a pre-processing step. The mesh is subdivided into n pieces before the simulation begins so that it initially appears to be one solid object at the beginning of the simulation. However, as the simulation progresses, the individual pieces are each updated independently causing them to move apart and simulate fracture of the object. Since explosions happen so quickly, the fractured pieces will rapidly be moving away from the source of the explosion. In this situation, the pre-fracturing step obtains the same visual results as a physically based fracture model while using less computation time.

The following section proposes two fracture models that treat meshes as graphs. The breadth-first search graph partitioning algorithm works by classifying nodes into levels and splits the nodes based on level number. The curvature-driven spectral partitioning algorithm uses curvature as a measure of the weakness of edges to determine where to break the mesh.

Definition 1 *A* graph G *is a pair $G = (V, E)$, where V is a finite set of* nodes *or* vertices *and E has as elements subsets of V of cardinality two called* edges. *The vertices of V are usually called v_1, v_2, ... If v_1 and v_2 are joined by an edge, we say that v_1 is* adjacent *to v_2 (and vice versa).*

The vertices that define the polygonal mesh become the set V. Each triangle $T = (v_i, v_j, v_k)$ consists of three edges, $e_1 = (v_i, v_j)$, $e_2 = (v_j, v_k)$ and $e_3 = (v_k, v_i)$. The triangles are divided into two groups N_1 and N_2 based on a graph partitioning algorithm, where $T = N_1 \cup N_2$. This has the effect of fracturing the object into two pieces.

A graph-based approach to fracture has several advantages over previously proposed models. A graph partitioning algorithm is used a pre-processing step. As a result, during the simulation, there is no need to compute stress, strain or other forces on the object. It is only necessary to update the position of each piece of the mesh according to the flow of the fluid volume, which is much less computationally expensive. The graph-partitioning algorithms proposed here are easier to implement than dynamic methods and require no knowledge of physics. Both approaches are applicable to planar meshes and 3D meshes.

4.2.1 Recursive Breadth-First Search Graph Partitioning

To divide a mesh according to a recursive breadth-first search graph partitioning algorithm, a user-defined variable, m, which is the number of desired pieces, is used to determine the maximum level of recursion,

$$maxRec = \log_2(m).$$

The initial call to the algorithm uses the complete list of triangles in the mesh, T, and an initial level of 0. The algorithm progresses by selecting a node, r, from the list T, assigning it level 0 and adding it to the queue, Q, and the list of classified nodes, C. While the queue is not empty the first node in the queue, n, is removed. All of the children of n are identified, where children are all other triangles that share a vertex with n. For each unprocessed child, it is assigned a level as one more than its parent, and then added to the queue and the node list. Once the queue is empty, the list C contains each triangle and its level. This list is divided according to a value L, where nodes with level less than or equal to L belong in one partition, and nodes with level greater than L belong in the second partition. The complete algorithm is given in Algorithm 5. Figure 4.1 gives the levels of the triangles in a

Algorithm 5 procedure RecBFS(T, $level$)

```
if level = maxRec then
  Output mesh data
  return
else
  Q ⇐ empty queue
  C ⇐ {(r, 0)}
  Add (r,0) to Q
  Flag r as processed
  while Q is not empty do
    (n,level) ⇐ remove from Q
    for all unflagged children c of n in T do
      C ⇐ C ∪ (c,level + 1)
      Add (c,level + 1) to Q
      Flag c as processed
    end for
  end while
  L ⇐ maxLevel/2
  N1 ⇐ nodes with level ≤ L
  N2 ⇐ nodes with level > L
  RecBFS(N1, level + 1)
  RecBFS(N2, level + 1)
end if
```

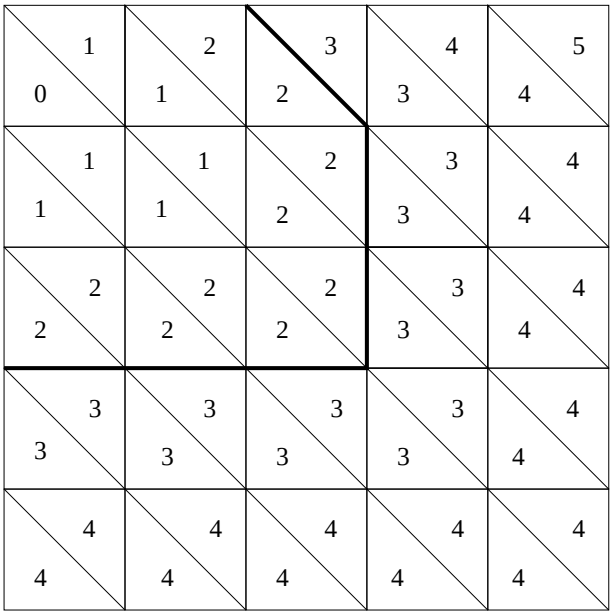

Figure 4.1: The first step of the breadth-first search graph partitioning algorithm

mesh after one level of recursion of the breadth-first search graph partitioner. The bold line shows the division of the mesh into two pieces using $L = 2$.

In some cases, the breadth-first search may fail to classify all triangles to a level and consequently these triangles are lost and not passed on to the next level of recursion. Consider the right partition of the mesh given in Figure 4.1. This piece is recursed on to again partition it. However, the partitioner may classify the triangles into levels as given in Figure 4.2. If the mesh is split using $L = 2$ as shown with the bold lines, the result is two partitions in which one consists of disjoint pieces. When the pieces at levels 3, 4 and 5 are again passed to the graph partitioner, the partitioner is only able to classify the triangles in one of the pieces as it will not be able to identify any of the triangles in the other piece as children, and these triangles will essentially be lost resulting in an incomplete mesh.

To overcome this problem, an addition was made to Algorithm 5. After the `while` loop, the algorithm checks for lost triangles by comparing the list of triangles C with the list of triangles T that was passed in. Any triangles in T that do not appear in C are added to the end of C. C now contains a list of triangles sorted by level, with unclassified triangles at the end. To partition the triangles, the list C is divided in half and each piece is recursed on.

The breath-first search graph partitioner is capable of creating both small homo-

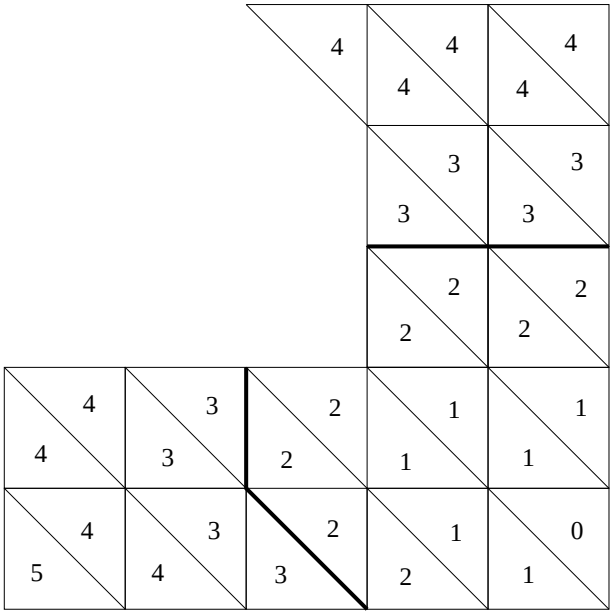

Figure 4.2: The second step of the breadth-first search graph partitioning algorithm

geneous pieces from the mesh object or pieces of varying sizes depending on where the triangle lists are divided. Although this algorithm is not physically-based, it is still capable of generating realistic looking results. In particular, the simulations of broken glass using the breadth-first search graph partitioning algorithm match the behaviour of the observed fracture of glass [1] [2].

4.2.2 Curvature-Driven Spectral Partitioning Algorithm

The spectral bisection algorithm is based on techniques proposed by Fiedler [13] in 1973, and made popular by Pothen et al. [3] in 1990. The technique is motived by the modes of a vibrating string, as depicted in Figure 4.3. The dotted line represents the rest position of the string. Each part of the string is labeled as (+) if it is above the rest position and (-) if it is below. Notice that the second frequency divides the string into two equal sized connected parts. If a graph is treated as a string, where the nodes represent a set of identical masses and the edges the springs connecting the masses, then a graph can also be partitioned in a similar fashion.

For rigid bodies, it is desirable to be able to break the mesh (partition the corresponding graph) in such a way that it is related to the geometry of the object. Curvature can be used as a measure of how weak the edge connecting two vertices

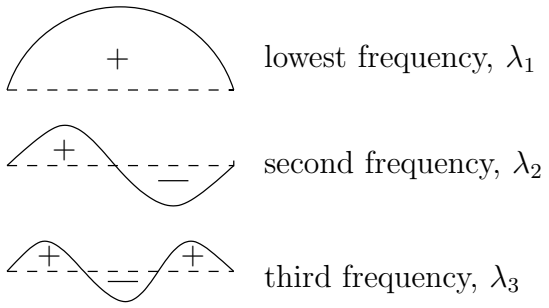

Figure 4.3: Modes of a vibrating string

is. Gaussian curvature [50], K, of a vertex is approximated by

$$K = \frac{A}{3\Delta\theta},$$

where A is the total area of the adjacent triangles T_i for $i = 1, 2, 3, \ldots$ and $\Delta\theta$ is the angle deficit calculated as

$$\Delta\theta = 2\pi - \sum_i \theta_i.$$

The curvature of an edge is the average of the curvature of the vertices connecting it. Edges of largest curvature are interpreted as the weakest regions of the mesh so that an area of large curvature is more susceptible to fracture than areas with small curvature. The curvature of each edge is used to create the Modified Laplacian matrix of G.

Definition 2 *The* Modified Laplacian matrix*, $ML(G)$, of a graph G is an $|N|\times|N|$ symmetric matrix with one row and one column for each vertex. It is defined as follows:*

$$ML(G)(i,j) = \begin{cases} K_e \equiv -\frac{K_i+K_j}{2} & \text{if there is an edge between } i \text{ and } j,\ i \neq j \\ -\sum_{k,i\neq k} ML(G)(i,k) & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

A value of 0 in entry (i, j) means there is no connection between i and j. Edges with large curvature contain the smallest entries in the matrix to show that there is a small connection between the vertices, and edges with small curvature contain the largest values to show that the connection is strong. Note that in graph theory, the Laplacian matrix, $L(G)$, contains a -1 for where there is an edge between i and j for $i \neq j$.

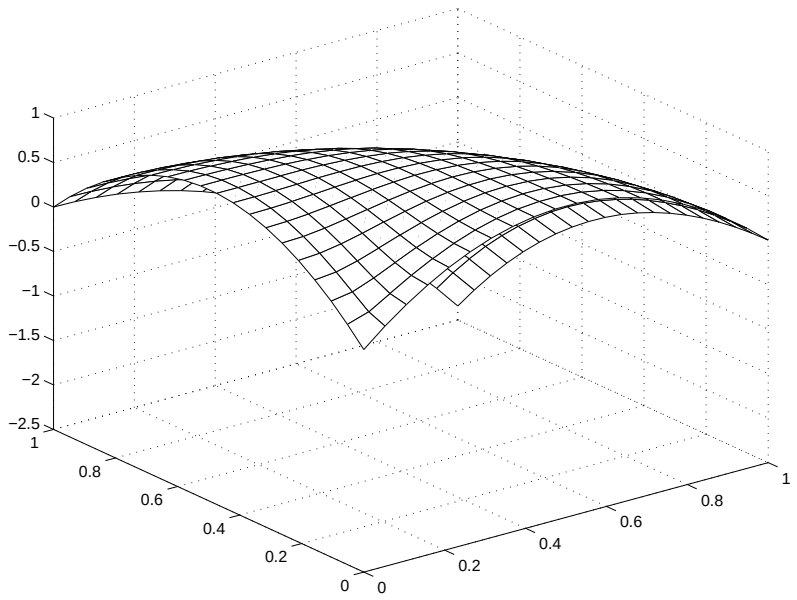

Figure 4.4: A curved surface

Definition 3 *The* Modified Incidence Matrix, $MIn(G)$, *of a graph* G *is an* $|N| \times |E|$ *matrix, with one row for each node and one column for each edge. For an edge* $e = (i, j)$, *column* j *of* $MIn(G)$ *is zero except for the* i_{th} *and* j_{th} *entries, which are* $\sqrt{K_e}$ *and* $-\sqrt{K_e}$ *respectively.*

The curvature-driven spectral partitioning algorithm uses the idea behind the spectral bisection algorithm and incorporates the use of curvature as a means to direct the algorithm to the weakest place to partition the graph. Consider the curved surface in Figure 4.4. Consider the surface to be a rigid body. If the rigid body were to be dropped, intuitively it should break where the curvature is highest. Using the curvature-driven spectral partitioning algorithm, Figure 4.5 demonstrates that the surface does in fact break at the region of largest curvature, by depicting the two partitions of the mesh. The use of the curvature of an object allows the object to fracture in a physically realistic way. The concept of using curvature to determine weaknesses has been explored by Richards [39] who used curvature as a measure of fracture density to predict fracture and faults in reservoirs to reduce exploration risk.

The curvature-driven spectral partitioning algorithm is motivated by the following theorem.

Theorem 1 *For a given graph* G, *its associated matrix* $ML(G)$ *has the following*

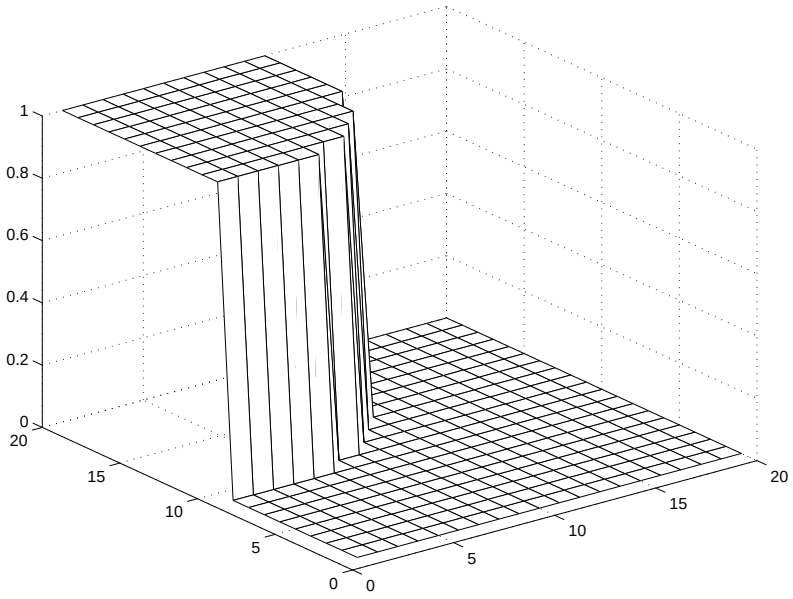

Figure 4.5: The two partitions of the curved surface

properties:

1. *Let* $e = [1, ..., 1]^T$*, i.e., the column vector of all ones. Then* $ML(G)e = 0$*.*

2. $ML(G) = MIn(G)MIn(G)^T$*. This is independent of the signs chosen in each column of* $MIn(G)$*.*

3. *Suppose* $ML(G)v = \lambda v, \lambda \neq 0$*, so that* v *is an eigenvector and* λ *an eigenvalue of* $ML(G)$*. Then*
$$\lambda = \frac{\sum_{e=(i,j)} K_e(v(i) - v(j))^2}{\sum_i v(i)^2}$$

4. *The eigenvalues of* $ML(G)$ *are nonnegative:*
$$0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$$

5. *The number of connected components of* G *is equal to the number of eigenvalues of* $ML(G)$ *equal to 0. In particular,* $\lambda_2 \neq 0$ *if and only if* G *is connected.* λ_2*, is referred to as the* algebraic connectivity*.*

Proof of part 1. The i^{th} entry of $ML(G)e$ is the sum of the entries of the i^{th} row of $ML(G)$. The diagonal entry of the i_{th} is the negative of the sum of the off-diagonal entries. Therefore the sum of the i_{th} row is exactly zero.

Proof of part 2.

$$\begin{aligned}
(MIn(G)MIn(G)^T)(i,j) &= \sum_k MIn(G)(i,k)MIn(G)^T(k,j) \\
&= \sum_k MIn(G)(i,k)MIn(G)(j,k) \\
&= MIn(G)(i,k)MIn(G)(j,k) \\
&= \sqrt{K_e}(-\sqrt{K_e}) && \text{where } e=(i,j) \\
&= -K_e
\end{aligned}$$

$$\begin{aligned}
(MIn(G)MIn(G)^T)(i,i) &= \sum_k (MIn(G)(i,k))^2 \\
&= \sum_{\text{edges } e \text{ that contain } i} (MIn(G)(i,e))^2 \\
&= \sum_e K_e
\end{aligned}$$

Proof of part 3. Suppose $ML(G)v = \lambda v$ where λ is an eigenvalue of $ML(G)$ and v is a nonzero eigenvector of $ML(G)$. Then $v^T ML(G)v = \lambda v^T v$, where $v^T v$ is a positive scalar. Then

$$\begin{aligned}
\lambda &= \frac{v^T ML(G)v}{v^T v} \\
&= \frac{v^T MIn(G)MIn(G)^T v}{v^T v} \\
&= \frac{y^T y}{v^T v} && \text{where } y = MIn(G)^T v \\
&= \frac{\sum_{e=(i,j)} (\sqrt{K_e}v(i) - \sqrt{K_e}v(j))^2}{\sum_i v(i)^2} \\
&= \frac{\sum_{e=(i,j)} K_e (v(i)-v(j))^2}{\sum_i v(i)^2}
\end{aligned}$$

Proof of part 4. This follows from part 3. Since each eigenvalue λ is the quotient of two nonnegative quantities, λ must be nonnegative.

Proof of part 5. For λ to equal 0, each $v(i) - v(j)$ in the expression

$$\lambda = \frac{\sum_e \sqrt{K_e}(v(i) - v(j))^2}{\sum_i v(i)^2}$$

must be zero. This means $v(i) = v(j)$ for each edge $e = (i, j)$. For any node i, the fact $v(i) = v(j)$ is repeatedly applied. Any node k reachable from i also satisfies $v(k) = v(i) = c$ for some constant c. The eigenvector v has the value c for each connected component. Since $ML(G)$ is symmetric, the number of unique eigenvectors corresponding to $\lambda = 0$ is equal to the number of eigenvalues equal to 0. If there is exactly m connected components, there are exactly m unique eigenvectors. Choosing m constants $c(1), \ldots, c(m)$ determines each eigenvector.

As a consequence of this theorem, partitioning a graph is a matter of computing λ_2 for $ML(G)$. For each node in G, if the corresponding entry in the eigenvector is negative, then the node belongs in the first partition, N_1. If the corresponding entry in the eigenvector is positive, then the node belongs in second partition, N_2. This partitions the graph into two components with the following consequences.

Theorem 2 *(M. Fiedler [13]). Let G be connected and N_1 and N_2 be defined as above. Then N_1 is connected. If no entry in the second eigenvector is equal to 0, N_2 is also connected.*

Theorem 3 *(M. Fiedler [13]). Let $G_1(N, E_1)$ be a subgraph of $G(N, E)$ so that G_1 is "less connected" that G. Then the algebraic connectivity of G_1 is less than or equal to the algebraic connectivity of G.*

With the existence of these theorems, the curvature-driven spectral partition algorithm has a theoretical advantage over the breadth-first search algorithm. As was demonstrated with the breadth-first search algorithm, there may be cases where it fails to classify nodes into a partition. However, with the curvature-driven spectral partition algorithm it can always be determined whether the two partitions will be connected. In general, if the graph was connected to begin with, the algorithm will split the graph into two connected pieces.

What remains to be accomplished is reconstructing the meshes into two groups of triangles based on the partition of the nodes. For each triangle, if two or three of its vertices are in N_1, then the triangle belongs in the first partition, otherwise it belongs in the second partition. The curvature-driven spectral partitioning algorithm is given in Algorithm 6.

Algorithm 6 Curvature-Driven Spectral Partitioning

Read in mesh data (vertices and triangles)
Determine the curvature, K, of each vertex
Create the Modified Laplacian matrix, ML
Find the second smallest eigenvalue of ML and the corresponding eigenvector v_2
for each vertex $vert_i$ **do**
  if $v_2(i) < 0$ **then**
    $N_1 \Leftarrow N_1 \cup vert_i$
  else
    $N_2 \Leftarrow N_2 \cup vert_i$
  end if
end for
for each triangle T_i **do**
  if T_i has 2 or 3 vertices in N_1 **then**
    $T_1 \Leftarrow T_1 \cup T_i$
  else
    $T_2 \Leftarrow T_2 \cup T_i$
  end if
end for
Output the triangles of T_1 as a mesh
Output the triangles of T_2 as a mesh

As an example, consider the mesh in Figure 4.6. The list of vertices is given by

$$\begin{array}{lll} v_1 = (0.0, 0.0, 0.0) & v_2 = (1.0, 0.0, 0.0) & v_3 = (0.5, 0.5, 1.0) \\ v_4 = (1.0, 1.0, 0.0) & v_5 = (0.0, 1.0, 0.0) & v_6 = (0.5, 0.5, -1.0). \end{array}$$

The triangles created from these vertices are

$$\begin{array}{lll} T_1 = (v_1, v_2, v_3) & T_2 = (v_2, v_3, v_4) & T_3 = (v_3, v_4, v_5) \\ T_4 = (v_1, v_3, v_5) & T_5 = (v_1, v_2, v_6) & T_6 = (v_2, v_4, v_6) \\ T_7 = (v_4, v_5, v_6) & T_8 = (v_1, v_5, v_6). & \end{array}$$

The corresponding Modified Laplacian matrix is given by:

$$ML(G) = \begin{pmatrix} 1.699 & -0.474 & -0.372 & 0 & -0.477 & -0.376 \\ -0.474 & 1.703 & -0.373 & -0.479 & 0 & -0.377 \\ -0.372 & -0.373 & 1.498 & -0.377 & -0.376 & 0 \\ 0 & -0.479 & -0.377 & 1.720 & -0.482 & -0.381 \\ -0.477 & 0 & -0.376 & -0.482 & 1.716 & -0.380 \\ -0.376 & -0.377 & 0 & -0.381 & -0.380 & 1.515 \end{pmatrix}.$$

The second eigenvector computed using Matlab is given by:

$$[-0.538, -0.020, -0.329, 0.521, -0.179, 0.546].$$

This means that vertices v_4 and v_6 belong in one partition, and vertices v_1, v_2, v_3 and v_5 are in another partition. This splits the mesh into one partition of two triangles (T_6 and T_7), and another with six (T_1, T_2, T_3, T_4, T_5, T_8) as depicted in Figure 4.7.

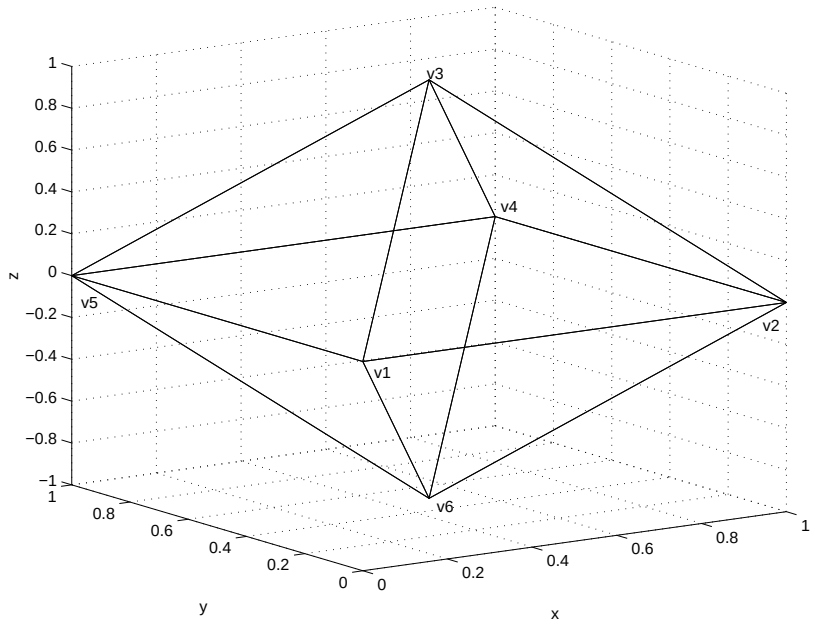

Figure 4.6: A sample mesh

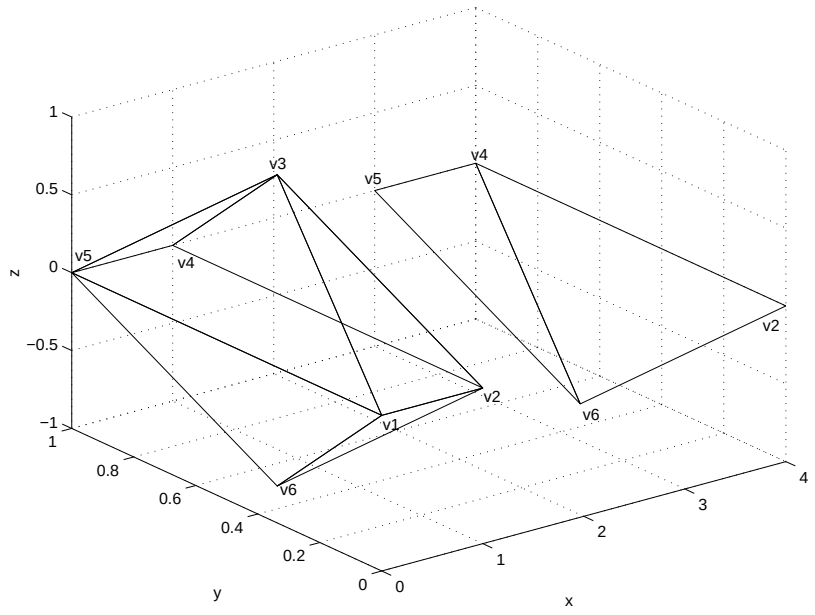

Figure 4.7: The sample mesh after curvature-driven spectral partitioning

Chapter 5

Results

5.1 Animations

To demonstrate the explosion and the proposed fracture models, animations were created from sequences of images using the rendering techniques described in Appendix A. Creating animations consisted of determining values for all user-controllable parameters, determining the vertices and faces that make up rigid bodies, and setting up the scene. Table 5.1 lists all user-controllable parameters.

To demonstrate different aspects of the simulations independently, simple animations highlighting certain features were first created. These animations demonstrate rigid body motion, and verify that a piece of glass using the breadth-first search graph partitioning algorithm achieves correct visual results, and the curvature-driven spectral partitioning algorithm partitions a complex polygonal mesh in a physically realistic location. More detailed animations demonstrating both fracture models follow.

To verify that the movement of rigid bodies is in accordance with the underlying flow field, a rectangle is placed inside a circular fluid flow. The fluid moves as if it were a glass of water being stirred. Objects at the center of the fluid move with the flow in a circle while moving towards the boundary of the fluid. A sequence of images evenly spaced in time showing the movement of the rectangle is shown in Figure 5.1. The arrows show the direction of the fluid flow. The magnitude of the arrow is in relation to the size of the velocity vector at that location. Observe that both translations and rotations are used to update the position of the rectangle and that the behaviour of the rectangle moves as predicted by moving in a circular path towards the boundary of the fluid.

To demonstrate the breadth-first search graph partitioning algorithm, a mesh representing a piece of glass is created from the trisection triangulation. Note that the trisection triangulation tends to produce highly irregular shapes of shards, for example the piece in the lower left corner. The piece of glass mesh is placed in front of an explosion and its movement is tracked. The piece of the glass fractures from the center and the shards move away from the source of the detonation. A sequence of images showing the movement of the pieces of the fractured window is shown in Figure 5.2. The explosion has been omitted from the images for clarity.

In demonstration of the curvature-driven spectral partitioning algorithm, complex polygonal meshes were partitioned as depicted in Figure 5.3. The cow and the teddy bear are partitioned at the neck so that the head forms one piece and the rest of the body forms the other. The statue of Venus is fractured at the waist and the pumpkin is divided roughly in half through the stem. The nature of the algorithm is to fracture meshes in a region of large curvature while also aiming to achieve as much of a bisection as possible. The total number of vertices in each mesh and the number of vertices in each partition is given in Table 5.2.

Complex animations using the explosion model and the breadth-first search partitioning algorithm were created. The first animation shows the front-view of a building with a window. An explosion is placed behind the window so that the pieces will fly out towards the viewer away from the position of the detonation. The window is created using a trisection triangulation and giving it depth (see Section 3.3). The window is both refractive and reflective. An image representing the opposite side of the street to the building is texture mapped onto a plane behind the viewer so that it reflects in the pieces of the window. The mesh of the window was partitioned into 16 pieces to be able to see this effect better in fewer large pieces. To add realism to the movement of the pieces of the window, the mass of each piece is proportional to its surface area. This allowed smaller pieces to fly away from the explosion faster than larger pieces. Gravity was incorporated so that the pieces of the window fly outwards from the explosion and then eventually fall to the ground. Figure 5.4 gives a sequence of images from the animation. Observe that the shattering of the window creates shards that break from the center that are consistent with observed images of shattered glass. Observe also that blackbody radiation assigns colour to the explosion that is consistent with observations of real explosions and that the colour progress through red, orange and yellow as the temperature rises.

The second animation is set in the same building as the first with a close-up view. The Delaunay triangulation was used to create the mesh for the window, and the window was fractured into 512 pieces. This was to further demonstrate the breadth-

α	Gaussian blur rate
min_d	minimum density of a cell
max_d	maximum density of a cell
nf	Gaussian filter size
τ	constant controlling conversion of density to opacity
α_d	drag coefficient
α_h	coefficient of thermal conductivity
b_h	amount of heat released per unit combusted mass of fuel
b_g	volume of gas released per unit combusted mass of fuel
b_s	mass of soot produced per unit combusted mass of fuel
c_m	thermal mass of a particle
c_k	thermal conductivity constant
c_r	cooling constant
F_c	force constant
m	mass of a particle
n_p	number of particles
n_r	number of rigid bodies
$nx \times ny \times nz$	fluid grid size
r_d	radius of detonation
T_a	ambient temperature
T_i	particle's ignition temperature
T_{max}	maximum temperature in the environment
z_f	burn rate for fuel particles
z_s	burn rate for soot particles
$vx \times vy \times vz$	density and colour grid size
pos_{det}	location of detonation
$thres_{m_f}$	fuel mass threshold
$thres_{m_s}$	soot mass threshold
$thres_{c_m}$	thermal mass threshold
c_{soot}	soot creation constant
np_i	number of particles initially ignited
np_{it}	number of new particles created each iteration

Table 5.1: User controllable parameters

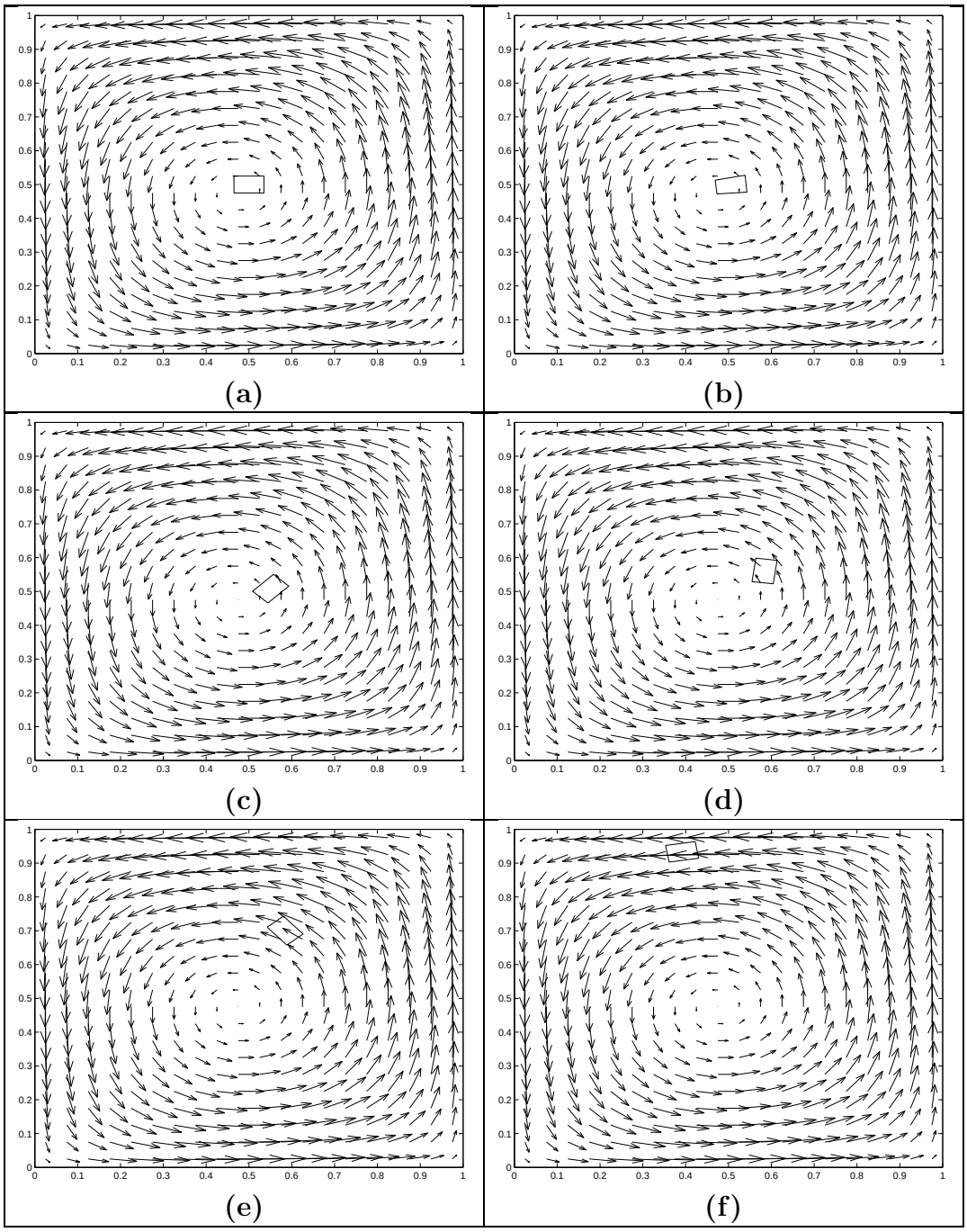

Figure 5.1: Rigid body motion of a rectangular block

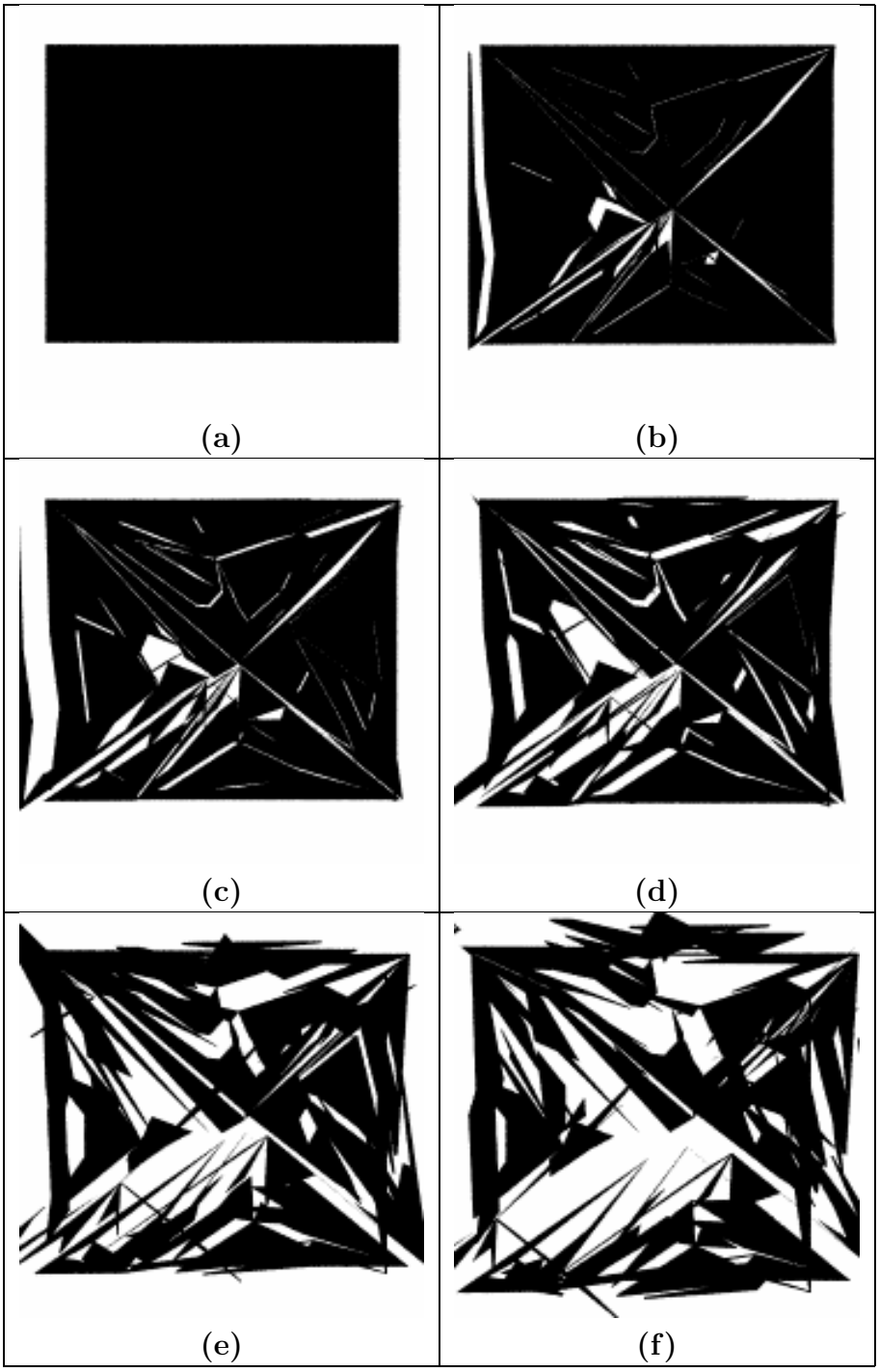

Figure 5.2: Fracture of a piece of glass using the breadth-first search partitioning algorithm

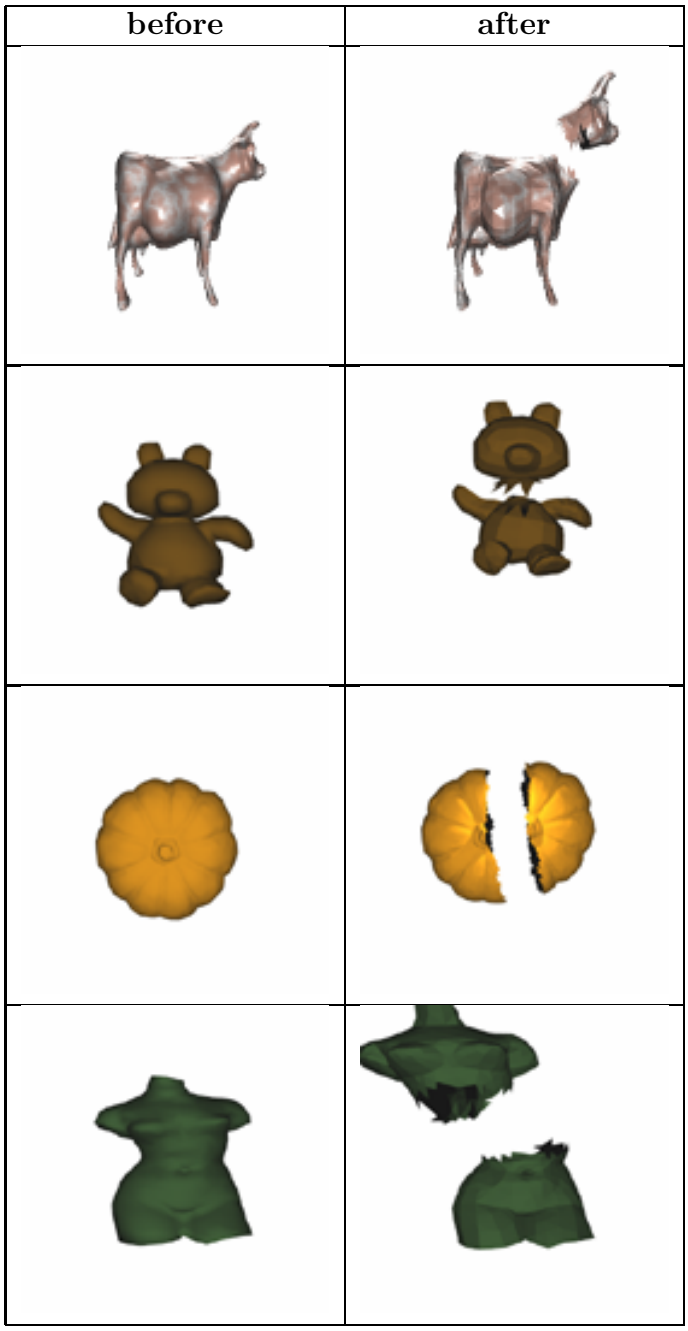

Figure 5.3: Fractured meshes using the curvature-driven spectral partitioning algorithm

	cow	teddy	pumpkin	Venus
total vertices	5804	3192	10000	1418
vertices in partition 1	1884	708	4826	698
vertices in partition 2	3920	2484	5174	720

Table 5.2: The total number of vertices in each mesh and the number in each partition after curvature-driven spectral partitioning

first search partitioning fracture model. In this instance, many small homogeneous pieces are created which again fly away from the source of the detonation. A sequence of images from the animation are given in Figure 5.5. The values of the parameters used in the window animations are given in Table 5.3. The parameters presented by Feldman [12] were used as an initial set of parameters for the animation and were fine-tuned to create the desired results. On average, 892 seconds were required to simulate one frame and 1680 seconds were required to render one 300×300 image using a sample size of 2 for anti-aliasing on an Intel Pentium 4 3.00GHz processor with 1Gb of RAM.

Two complex animations were created to demonstrate the curvature-driven spectral partitioning algorithm. The first animation features the polygonal mesh of a cow viewed from the side. The curvature-driven spectral partitioning algorithm partitioned the cow mesh into two pieces at the neck as demonstrated previously. The mesh representing the head was further partitioned with the breadth-first search partitioning algorithm. In practice, small regions do not vary much in terms of curvature, so curvature is less important and the breadth-first search partitioning algorithm gives an acceptable partition. An explosion is placed inside the cow's head which results in the head separating from the body and breaking apart while the body remains intact. A sequence of images from the animation is given in Figure 5.6. The explosion is not rendered so the effect of the partitioning can be observed without obstruction.

The second animation also makes use of the cow mesh. However, in this animation the mesh is deformed prior to fracturing. To deform the mesh, an explosion is placed inside the cow's body and the position of each vertex is updated based on the underlying flow field as is done with particles. This allows the cow to expand like it were a balloon before it fractures. The entire mesh is fractured by first splitting the cow into two pieces (the head and the body) with the curvature-driven spectral partitioning algorithm, and then running the breadth-first search partitioning algorithm on the body. Figure 5.7 gives a sequence of images from the animation. As with the first cow animation, the explosion is not rendered so the fracturing can be observed without obstruction. However, the colour of each piece

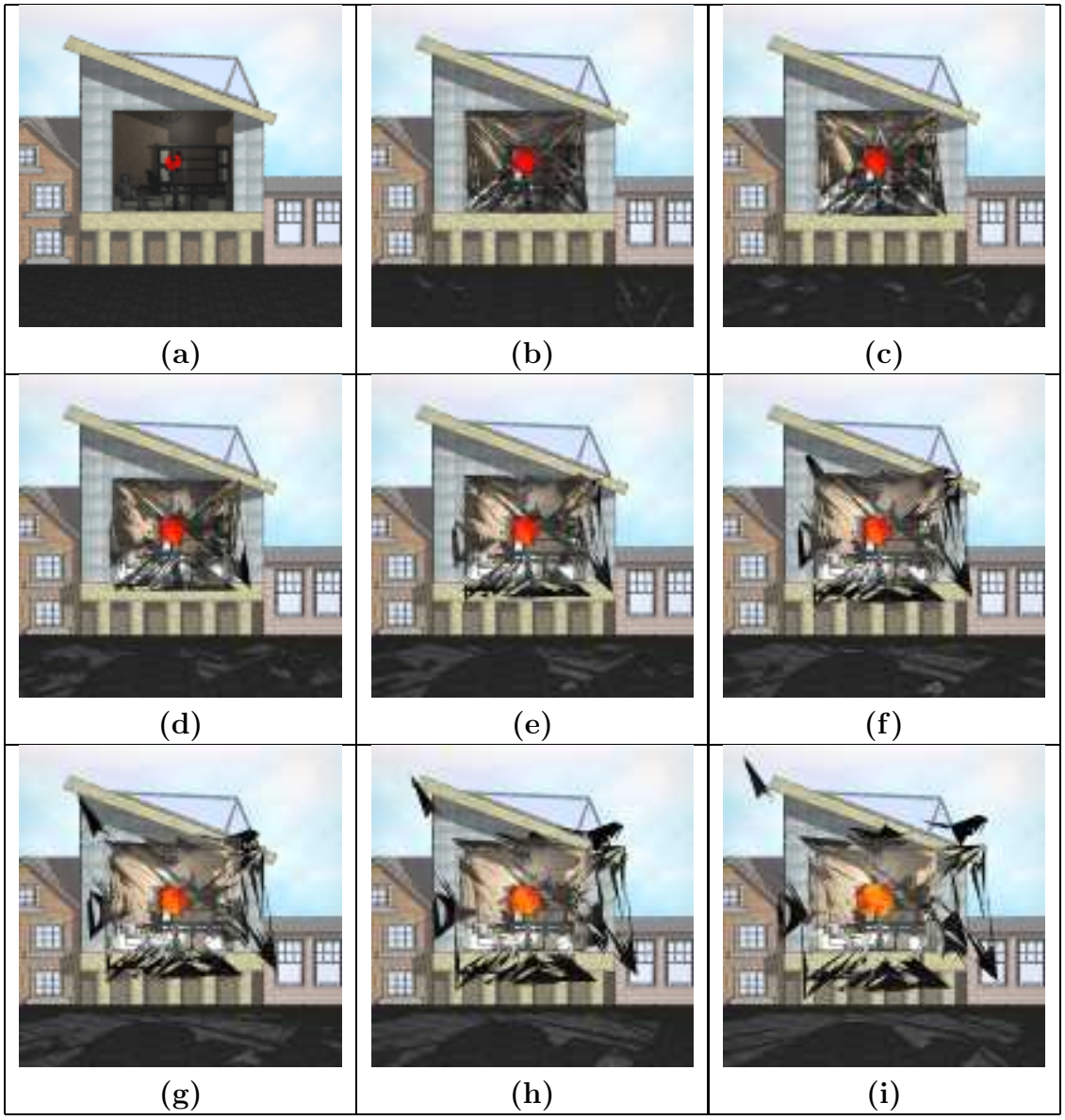

Figure 5.4: A sequence of frames from the fracturing window animation with trisection triangulation

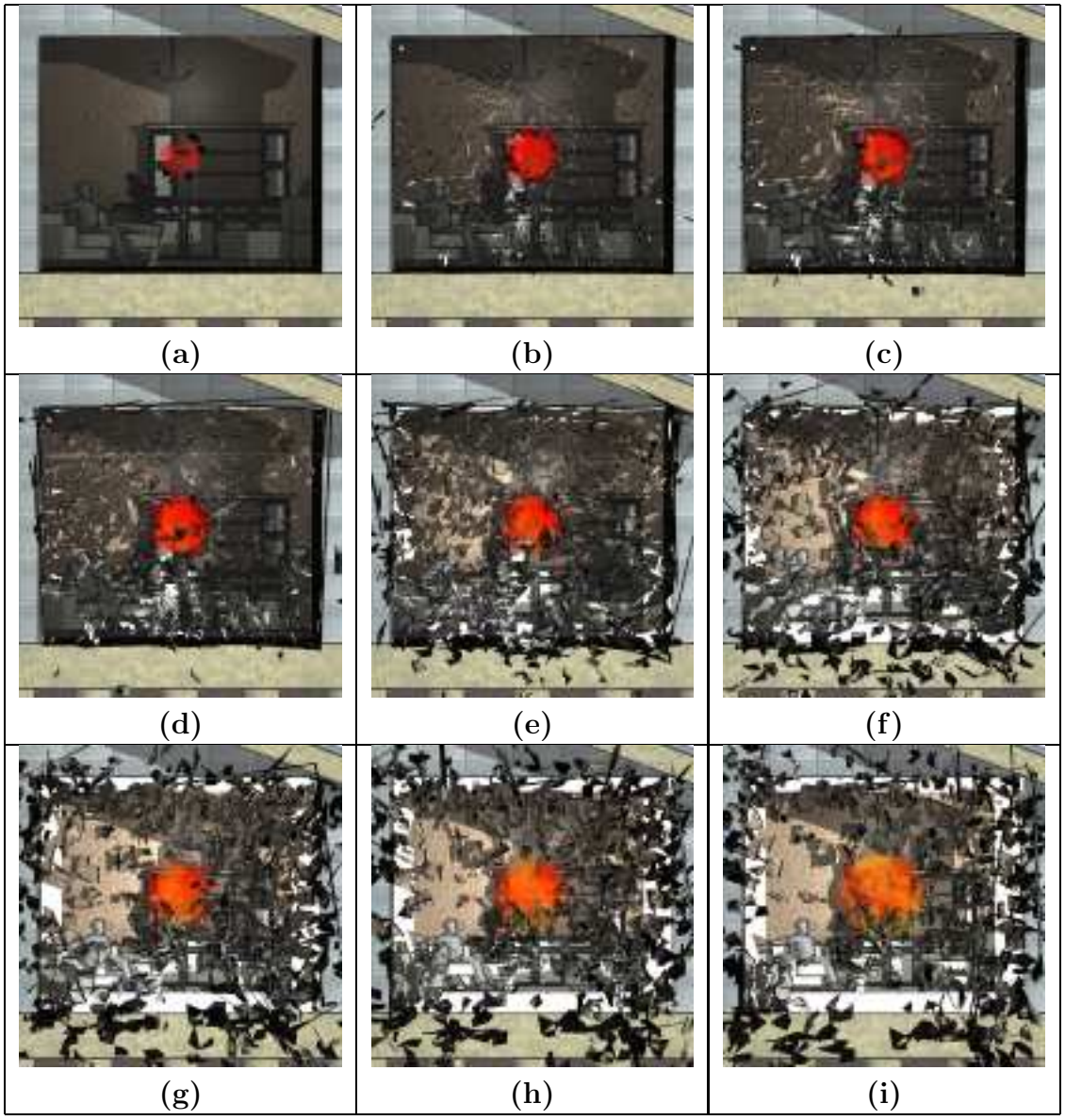

Figure 5.5: A sequence of frames from the fracturing window animation with Delaunay triangulation and a close-up view

of the cow is changed by determining the average temperature over all the vertices and using that temperature with blackbody radiation to determine the colour. On average, less than one second was required to simulate one frame. This is because explosions were not rendered and computing and storing the volume data was not necessary. On average, 125 seconds were required to render one 300×300 image using a sample size of 2 for anti-aliasing on an Intel Pentium 4 3.00GHz processor with 1Gb of RAM.

5.2 Conclusions and Future Work

This thesis proposed graph-based approaches to simulating fracture and used explosions to initiate the fracture process. Treating an explosion as a fluid, the incompressible Navier-Stokes equations were used to model its behaviour. To allow for the incompressible equations to be used, the divergence of the flow field was adjusted to account for the addition of gaseous products to the fluid. The models were used with a ray tracer and volume renderer to create a sequence of images that were assembled into animations. The animations demonstrate that explosions rendered using blackbody radiation create realistic looking results.

Both graph-based approaches for fracture are much less computationally expensive than models based on physics which require stress and strain to be calculated at each iteration of the simulation. The breadth-first search graph partitioning algorithm achieves physically accurate results for shattering windows and glass. To fracture three dimensional rigid bodies, curvature was used to determine the weakest area on the rigid body. The curvature-driven spectral graph partitioning algorithm splits meshes into two pieces in a physically realistic way.

One possible avenue for future work involves incorporating some form of solid-fluid interaction. The incorporation of collision detection in the simulations is also left as future work.

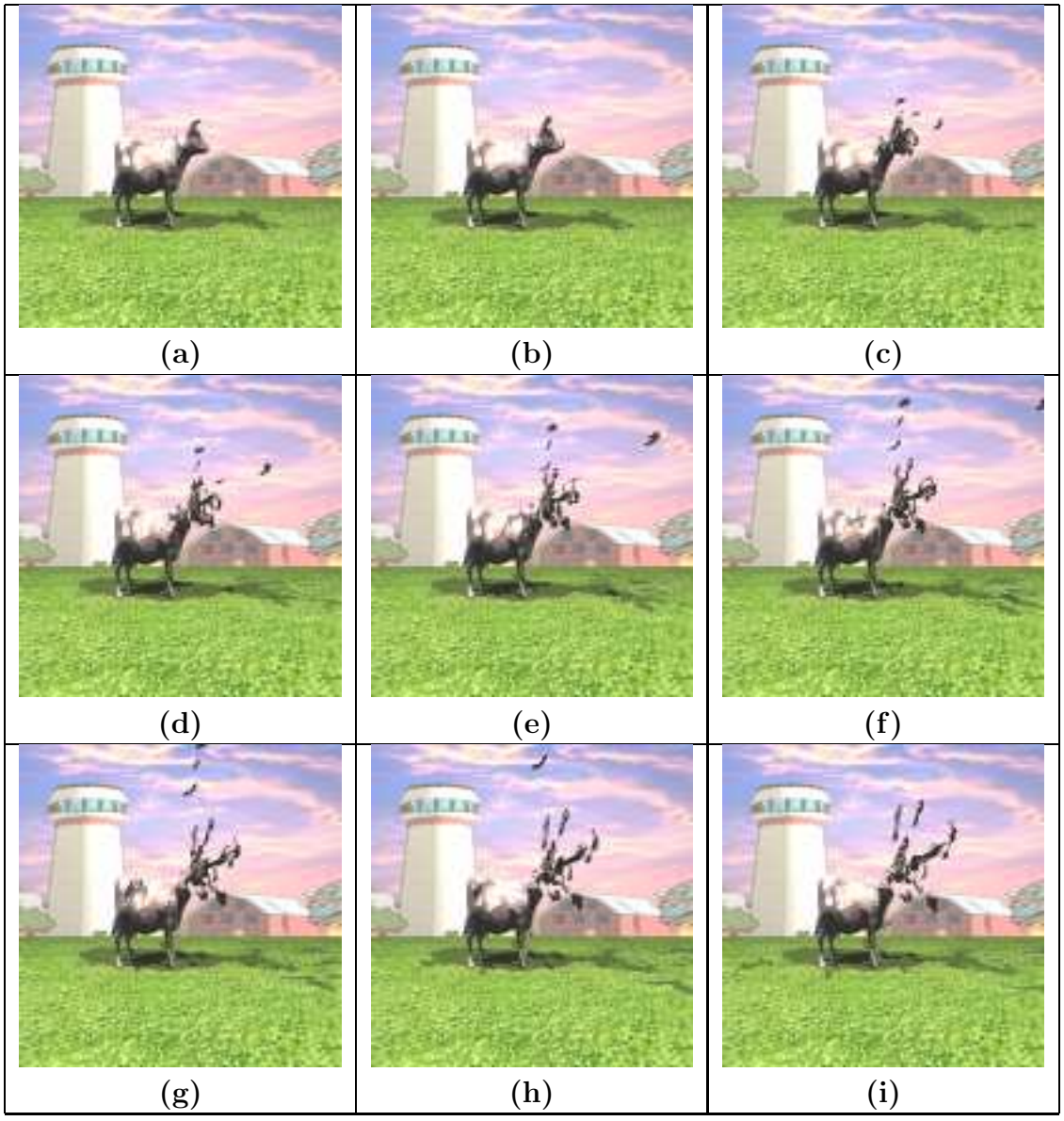

Figure 5.6: A sequence of frames from the first fractured cow animation

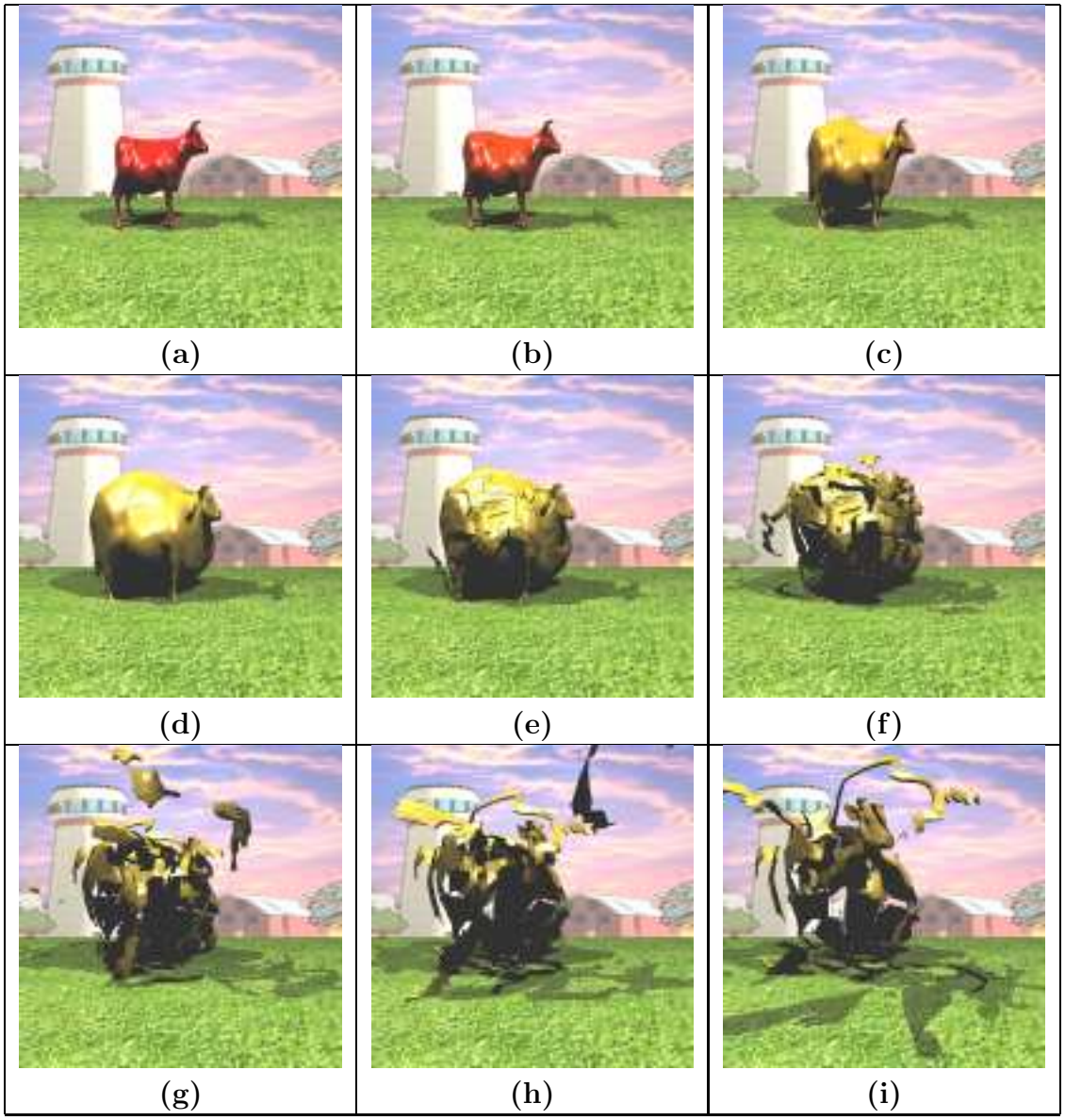

Figure 5.7: A sequence of frames from the second fractured cow animation

parameter	fractured windows	fractured cow 1	fractured cow 2
α	1.4	N/A	N/A
min_d	0.05	N/A	N/A
max_d	0.95	N/A	N/A
nf	5	N/A	N/A
τ	0.01	N/A	N/A
α_d	0	0	0
α_h	200000	50000	30000
b_h	23000	3000	6000
b_g	2.61	2.61	2.61
b_s	1.0	1.0	1.0
c_m	20.54	10	10
c_k	0.0	0.0	0.0
c_r	5.0	5.0	5.0
F_c	30.0	40.0	10.0
m	0.36	0.36	0.36
n_p	1000	100	100
n_r	16/512	17	33
$nx \times ny \times nz$	$32 \times 32 \times 32$	$20 \times 20 \times 20$	$20 \times 20 \times 20$
r_d	0.1	0.01	0.05
T_a	5.0	5.0	5.0
T_i	10.0	10.0	10.0
T_{max}	100.0	100.0	100.0
z_f	0.21	0.21	0.21
z_s	0.5	0.5	0.5
$vx \times vy \times vz$	$300 \times 300 \times 300$	N/A	N/A
pos_{det}	(0.46573, 0.52375, 0.48)	(0.5, 0.55, 0.5)	(0.525, 0.525, 0.48)
$thres_{m_f}$	0.000001	0.000001	0.000001
$thres_{m_s}$	0.00001	0.00001	0.00001
$thres_{c_m}$	0.0001	0.0001	0.0001
c_{soot}	0.02	0.02	0.02
np_i	10	1	1
np_{it}	50	5	25

Table 5.3: Animation parameters

Appendix A

Rendering

In order to visually capture the effects of the explosions, fracture and flying debris, a rendering algorithm is needed to create images from the simulation data. There are several different major techniques for rendering scenes. One technique may sacrifice certain features to make other features more accurate and efficient. The rendering of explosions with solid objects requires the ability to produce an image from both rigid bodies and particle data. Rigid bodies are most easily ray traced since they have a defined surface, while particle data is often used with a volume renderer. These two techniques are combined to achieve realistic explosion images. The following section outlines the implementation details of the ray tracer and volume renderer used to produce the images that were used to create the animations presented in Chapter 5.

A.1 Ray Tracing

Ray tracing is an approach to rendering scenes in which rays are cast from the eye through a pixel in the image plane, and into the scene. The path of the ray is traced to determine which object in the scene was intersected with first. A lighting model is used to compute the colour at the point of intersection, which in turn is the colour of the pixel. Algorithm 7 gives the basic steps of a ray tracer. For an in-depth introduction to ray tracing, the reader is referred to [21, Chapter 14].

Ray tracing is advantageous due to its inherent ability to do automatic hidden-surface removal. It is also easy to add features such as reflection, refraction and shadows. However, ray tracers do not give complete global illumination. Although

Algorithm 7 Ray Tracing [21, page 736]

Define the objects and light sources in the scene
Set up the camera
for each row r **do**
  for each column c **do**
    1. Build the rc^{th} ray
    2. Find all intersections of the rc_{th} ray with objects in the scene
    3. Identify the intersection that lies closest to, and in front of, the eye
    4. Compute the "hit point" where the ray hits this object, and the normal vector at this point
    5. Find the colour of the light returning to the eye along the ray from the point of intersection
    6. Place the colour in the rc_{th} pixel
  end for
end for

some optimization techniques exist, ray tracers are can only be accelerated to a certain point.

A.1.1 Intersections

For each of the rigid bodies considered, spheres, cubes and polygonal meshes, an intersection routine is required. A ray is given by

$$r(t) = r_0 + r_d t \tag{A.1}$$

where r_d is the ray's direction and r_o is the ray's origin. A sphere of radius r has the implicit form

$$F(r) = r^2 - 1. \tag{A.2}$$

To determine the intersection of a ray and a sphere, equation (A.1) is substituted into equation (A.2) giving

$$t = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$

where $A = |r_d|^2$, $B = r_o \cdot r_d$ and $C = |r_o|^2 - 1$. The smallest positive value for t is computed, which corresponds to the closest intersection point on the sphere that is in front of the eye.

A cube is aligned with each of the axis in its object space. The equation of the plane that each face lies on, and its normal, can be calculated. To intersect with a cube, each of the six faces are tested for intersection. It is first determined if the ray intersects with the plane the face lies on by substituting the equation of the ray into the equation of the plane and looking for a positive t value. If such a t is found, then a further check that the point at this t value actually lies on the face of the cube is performed. The smallest positive t value over all the faces is the closest intersection point on the cube that is in front of the eye. Cubes are also important in that they can be used to optimize intersection tests for more complicated objects such as polygonal meshes. A polygonal mesh can be placed inside a cube, referred to in this case as a `bounding box`. Only when the bounding box is intersected with, does the polygonal mesh inside then start to compute intersections. Bounding spheres can also be considered. This reduces the number of times the computationally expensive ray-mesh intersection test need to be done. The intersection routine for polygonal meshes employed Badouel's efficient ray-polygon intersection routine [5].

A.1.2 Lighting Models

The actual colour at any point on the surface of a rigid body is dependent on what kind of light the surface is receiving. Diffuse reflection refers to light coming from a source at a particular direction in space. Diffuse light is reflected equally in all directions. Specular reflection is the amount of shiny reflection which causes highlights to occur when light hits the surface of an object and is reflected back towards the viewer. The colour of a point on a surface is the sum of the diffuse and specular reflection components. Some lighting models also include an ambient light component. Ambient light is background lighting found throughout the environment and is usually assumed to be constant. The following two sections describe how the diffuse and specular terms are computed using the Phong and the Ashikhmin lighting models.

Phong Lighting

The Phong lighting model [36] is one of the most commonly used models in computer graphics. The Phong model improves on previously proposed models by more accurately imitating real physical shading situations. The Phong lighting equation

$$I = R_a I_a + \sum_{i=1}^{m} I_s(R_d(N \cdot L) + R_s(V \cdot R)^f)$$

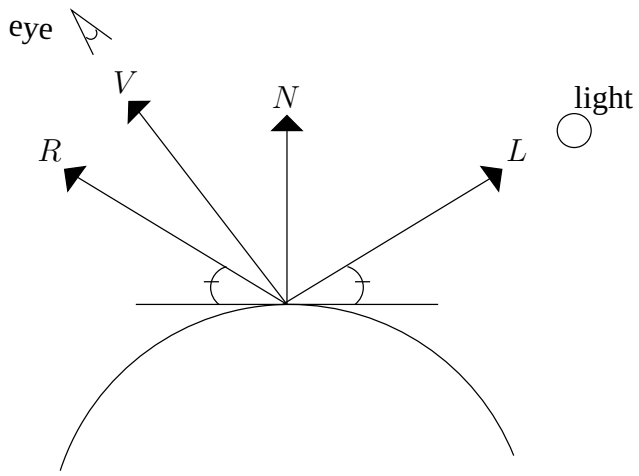

Figure A.1: Vectors in the Phong lighting model

is depicted in Figure A.1. Variables and their meaning in the Phong lighting model are given in Table A.1. Note that all vectors should be normalized.

Ashikhmin Lighting Model

The Ashikhmin anisotropic lighting model [4] is a view-dependent model that creates a brushed-metal effect. This model improves on other proposed models because is obeys energy conservation and reciprocity law and has a non-constant diffuse term. The Ashikhmin lighting model is used on a per-object basis. A *diffuse* and *specular* term are specified, as well as nu and nv, Phong-like exponents that control the shape of the specular highlight.

When a rigid body is intersected with, its uv coordinates are calculated (described in Section A.1.3). The partial derivatives $uVec$ and $vVec$ in the direction of u and v respectively are calculated for each type of rigid body. The partial derivatives form the following relations with the normal, n:

$$n = uVec \times vVec.$$

The partial derivatives for a sphere are

$$uVec = (-\sin(2\pi u), 0, -\cos(2\pi u))$$
$$vVec = (\cos(\pi v)\cos(2\pi u), -\sin(\pi v), -\cos(\pi v)\sin(2\pi u)).$$

When intersecting with a cube, the intersection routine knows which face the ray intersected. The $uVec$ and $vVec$ are set to the directions in which the u and

R_a	ambient coefficient
R_d	diffuse surface coefficient
R_s	specular surface coefficient
I_a	intensity of ambient light
I_s	intensity of light source
I	total computed intensity
N	surface normal
L	light vector
V	view vector
R	reflection vector
f	specular exponent
m	number of lights in the scene

Table A.1: Variables in the Phong lighting model

v values are moving. For example, if a ray intersects with the front face, then u corresponds to the x coordinate and v corresponds to the y coordinate. As x increases, u increases, resulting in a value of (1, 0, 0) for $uVec$. Similarly, as y decreases, so does v resulting in a value of (0, -1, 0) for $vVec$.

For polygonal meshes, the partial derivatives of each face are constant across the face. As a result, the partial derivatives can be calculated when the mesh is created and stored for later use. Using the vertices of a triangle, (p_1, p_2, p_3), and the uv coordinates at each of these vertices, (u_0, v_0), (u_1, v_1), (u_2, v_2), the partial derivatives for the face are

$$uVec = \frac{v_2 - v_0}{M}(p_1 - p_0) + \frac{v_0 - v_1}{M}(p_2 - p_0)$$
$$vVec = \frac{u_0 - u_2}{M}(p_1 - p_0) + \frac{u_1 - u_0}{M}(p_2 - p_0)$$

where

$$M = (u_1 - u_0)(v_2 - v_0) - (v_1 - v_0)(u_2 - u_0).$$

To determine pixel colour based on the Ashikhmin lighting model, the final colour is the sum of the diffuse term,

$$\rho_d(k_1, k_2) = \frac{28R_d}{23\pi}(1 - R_s)(1 - (1 - \frac{n \cdot k_1}{2})^5)(1 - (1 - \frac{n \cdot k_2}{2})^5),$$

and the specular term,

$$\rho_s(k_1, k_2) = \frac{\sqrt{(n_u + 1)(n_v + 1)}}{8\pi} \frac{(n \cdot h)^{\frac{nu(h \cdot uVec)^2 + nv(h \cdot vVec)^2}{1 - (h \cdot n)^2}}}{(n \cdot k)\max((n \cdot k_1), (n \cdot k_2))} F(k \cdot h),$$

R_d	diffuse colour of the object
R_s	specular colour of the object
nu, nv	Phong-like exponents for controlling the highlight
k_1	light vector
k_2	view vector
n	surface normal
$uVec$, $vVec$	tangent vectors that form an orthonormal basis with n
$\rho(k_1, k_2)$	BRDF
h	half vector between k_1 and k_2
$F(\cos(\theta))$	Fresnel reflectance for incident angle θ

Table A.2: Variables used in the Ashikhmin lighting model

where

$$F(k \cdot h) = R_s + (1 - R_s)(1 - (k \cdot h))^5.$$

is an approximation to the Fresnel coefficient. The variables and their meaning in the Ashikhmin lighting model are summarized in Table A.2. Note that all vectors should be normalized. Further note that when k appears with no subscript, either k_1 or k_2 can be used.

A.1.3 Features

Ray tracers typically exhibit extra features in addition to a standard lighting model. The following sections describe the features used to obtain the results presented in this thesis.

Anti-aliasing

Aliasing is the sampling error caused by representing a continuous quantity with discrete signals [34]. It describes the effect of seeing "stair-casing" where a straight line is intended, see Figure A.2, and is caused because a raster display is only able to display a discrete number of pixels in a fixed rectangular area. Anti-aliasing is the name for techniques designed to reduce or eliminate this effect. The three main strategies for anti-aliasing are prefiltering, supersampling and postfiltering. The approach used in this work is stochastic jittered supersampling. For an excellent discussion on the concept of aliasing and an in-depth look at anti-aliasing techniques, the reader is referred to [10].

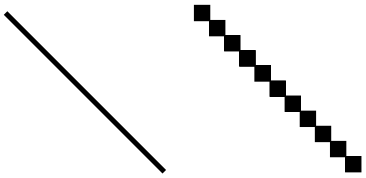

Figure A.2: Aliasing of a straight line

Jittered supersampling attempts to reduce the effects of aliasing by dividing each pixel into an $n \times n$ array of sub-pixels, where n is referred to as the sample size. A ray is then cast to the center of each sub-pixel and jittered. Jittering is accomplished by generating a random number ζ between 0 and $1/n^{th}$ of the width of a pixel and multiplying ζ by the components of the ray. To further increase the effectiveness of this technique, the final colour for each pixel is computed using a weighted sum. Sub-pixels at the edges of a pixel are given a weight of 1, while interior sub-pixels are given a weight 1.5.

Phong Shading

When shading the triangles in a polygonal mesh, it may be most efficient to choose a colour from one of the vertices and shade an entire face with that colour, a technique known as flat shading. However, this results in each face being easily distinguished from its neighbors, producing a "faceted" appearance. Flat shading also suffers from the inability to capture specular highlights.

Phong shading strives to reduce the faceted appearance and make polygonal meshes look smooth. This is accomplished by interpolating the normal at each point in the face and recomputing the lighting at each pixel. Although this technique is computationally more expensive that flat shading, it allows for more accuracy, smoother surfaces and better highlighting.

The polygonal mesh must do a preprocessing step to use Phong shading. The mesh must create an array of normals, one per vertex to later be used in the interpolation. This process is outlined in Algorithm 8.

When intersecting with each triangle of a mesh, the barycentric coordinates, α and β, for the intersection point are computed. These coordinates are used with the vertices of the triangle, v_0, v_1 and v_2, to calculate the interpolated normal,

$$n_i = (1 - (\alpha + \beta)) \cdot normals(v_0) + \alpha \cdot normals(v_1) + \beta \cdot normals(v_2).$$

The interpolated normal is then used in lighting calculations in place of the face normal.

Algorithm 8 Phong Shading Pre-processing Step

```
for each vertex v_i do
  initialize vector n_i;
  for each face f_j do
    for each vertex v_k in face f_j do
      if v_k = v_i then
        normal_{f_j} ⇐ f_j's normal
        n_i ⇐ n_i + normal_{f_j}
      end if
    end for
  end for
  normalize n_i
  add n_i to array at position i
end for
```

Texture Mapping

Texture mapping is the process of adding a separately defined 2D texture or image to a surface to change its colour [8]. It is a mathematical mapping from the texture domain to the object space. For each type of rigid body, the intersection point (x, y, z) is converted into uv coordinates that are used in texture mapping. Each type of rigid body calculates its uv coordinates uniquely.

For spheres, the north pole is assigned $v = 0$ and the south pole is assigned $v = 1$. At the positive x-axis, $u = 0$ and increases counter-clockwise around the y-axis. The values of u and v are determined by

$$u = \begin{cases} \frac{\arccos(\frac{x}{r\sin(v\pi)})}{2\pi} & \text{if } z \leq 0 \\ 1 - \frac{\arccos(\frac{x}{r\sin(v\pi)})}{2\pi} & \text{otherwise} \end{cases}$$

$$v = \frac{\arccos(\frac{y}{r})}{\pi}$$

where r is the radius of the sphere. The calculation of u takes into account the dual solution of inverse trigonometric functions.

To assign uv coordinates to a cube, each face will have values for u and v ranging from 0 to 1. As a result, the entire texture will be mapped to each face of the cube rather than warping it around the box. During the intersection process, it is known which of the six faces was intersected with and the normal. The coordinate corresponding to the non-zero component of the normal is dropped and the

remaining two are used as the u and v values. These values may need to be scaled to the range [0, 1] depending on the dimensions of the box.

To texture map the triangles of a polygonal mesh, the uv coordinates of each vertex are optionally specified with the vertices and faces of the mesh. This allows for the freedom to apply a texture over the whole mesh, or have each triangle texture mapped independently. Using the vertices of a triangle, (u_0, v_0), (u_1, v_1), (u_2, v_2), and the barycentric coordinates of the point of intersection, α and β, the u and v coordinates for the point of intersection are determined by

$$u = u_0 + \alpha(u_1 - u_0) + \beta(u_2 - u_0)$$
$$v = v_0 + \alpha(v_1 - v_0) + \beta(v_2 - v_0).$$

To use the texture map to determine the colour of a pixel, the uv values are converted into texture coordinates s and t,

$$s = u(width - 1)$$
$$t = v(height - 1).$$

The dimensions of the texture map are specified by *width* and *height*. The texture coordinates are used to index into the texture map to determine the diffuse colour of a pixel.

Procedural Textures

Procedural textures define a texture by a mathematical function based on the 3D point of intersection. Perlin Noise [35] is commonly used in procedural textures, and is a function of the form PerlinNoise3D(x, y, z, α, β, n) which computes a harmonic summing function from 1 to n. The parameters x, y and z are the coordinates of the intersection point, α is the weight when the sum is formed and β is harmonic scaling/spacing.

Each procedural texture has two colours, *colour*1 and *colour*2 associated with it. The variable *texture* is the resultant diffuse colour for each procedural texture. The procedural texture for wood is created using

$$noise = (1 + PerlinNoise3D(x, y, z, 2, 1.2, 8)/2) * 20$$
$$texture = colour1 + (colour2 + (colour1 * -1)) \cdot (noise - \texttt{int}(noise)),$$

where `int` is an operation that takes the integer value of a variable. The procedural texture for marble is given by

$$noise = |\sin(180 * (0.01x + 0.02PerlinNoise3D(x, y, z, 2, 2, 8)))|$$
$$texture = noise \cdot colour1 + (1 - noise) \cdot colour2,$$

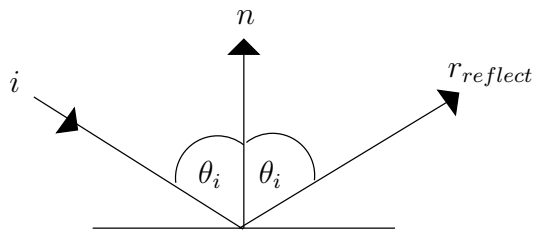

Figure A.3: The reflection vector

and the procedural texture for a checkerboard uses the uv coordinates of the intersection point with a constant `SIZE` that defines the width and height of a checker, giving the function

$$texture = \begin{cases} colour1 & \text{if } (SIZE * u \mod 2) = (SIZE * v \mod 2) \\ colour2 & \text{otherwise.} \end{cases}$$

Reflection and Refraction

When a ray is intersected with a reflective or refractive object, a secondary ray is generated in the direction of perfect reflection or perfection refraction respectively. To control the amount of secondary rays that are cast, a maximum recursion depth is set. A reflective ray is

$$r_{reflect} = i - 2n\cos(\theta_i),$$

where i is the incoming ray, θ_i is the angle of incidence, and n is the surface normal. The reflection ray is depicted in Figure A.3.

A refractive ray, $r_{refract}$ is depicted in Figure A.4 and calculated using Snell's law

$$r_{refract} = \frac{\eta_i}{\eta_r} i - (\cos(\theta_r) - \frac{\eta_i}{\eta_r}\cos(\theta_i))n.$$

In addition to the same variable definitions as reflection, η_i and η_r are the indices of refraction for the incident and refracting material respectively, θ_r is the angle of refraction, with

$$\cos(\theta_r) = \sqrt{1 - (\frac{\eta_i}{\eta_r})^2(1 - \cos^2(\theta_i))}.$$

If this quantity is negative then there is total internal reflection. Instead of casting a refractive ray, a reflection ray is cast.

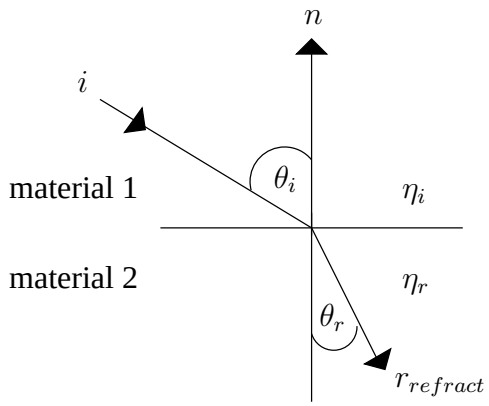

Figure A.4: The refraction vector

To simplify the refraction process, it is assumed that when a ray enters or leaves a refractive object that is is always from/to air. Objects with different refractive indices do not intersect.

A.2 Volume Rendering

Volume rendering is a technique to display surfaces from three-dimensional sampled data [23]. Rather than fitting a surface to the data, rays are traced from the eye through the volume data and projected onto the image plane. Volume rendering is advantageous because of its ability to display "fuzzy" surfaces, for example explosions, smoke and clouds. To overcome possible aliasing artifacts and loss of resolution, the volume grid must be the same dimensions as the image. However, when the volume is finely divided and there exist many cells with non-zero density, volume rendering is computationally expensive.

Two grids of the same size as the fluid grid are created, one for densities and one for colour. However, the density and colour grids are discretized much finer ($300 \times 300 \times 300$) than the fluid grid to prevent aliasing in the rendered images. As each particle is processed, its current position in the density and colour grid is computed and used to accumulate density and colour for each cell.

Once the density and colour is obtained for each cell, this data needs to be smoothed. Without a smoothing function, the data may contain a large positive value next to an empty cell resulting in a drastic change or sharp edge in the

resulting image. A Gaussian blur filter

$$G(x,y,z) = \frac{1}{(2\pi)^{3/2}\alpha^3} \mathrm{e}^{-\frac{x^2+y^2+z^2}{2\alpha^2}}$$

is used to smooth or blur the data to remove detail and sharpness where the parameter α controls the amount of decay. The function is evaluated over the range $-\frac{n-1}{2}$ to $\frac{n-1}{2}$ for each of x, y and z, where n is the filter size, n odd.

The Gaussian blur function is applied to both the density and colour data

$$R(i,j,k) = \frac{1}{S} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} D(i,j,k) F(i,j,k)$$

where $R(i,j,k)$ is the filtered data, S is the sum of the entries in the Gaussian filter, n is the size of the filter, $D(i,j,k)$ is the data (either density or colour), and $F(i,j,k)$ is the value in the Gaussian filter at index (i,j,k).

Once both the density and colour values have been blurred, the blurred colour is divided by the blurred density of each cell. The final step in preparing the volume data for rendering is to map the blurred density values to a specified range $[min, max]$ where $0 \leq min < max \leq 1$. The density and colour values are then passed to a volume renderer. To accomplish this, a cube rigid body is created for each cell with non-zero density, and designated to hold volume data by specifying that it is made of volume material.

The volume renderer uses the ray tracer to intersect with the cells in the volume. As the ray passes through the volume from front to back, colour and opacity are accumulated. The opacity of a cell is

$$a = 1 - \mathrm{e}^{-\tau D(x) dz}$$

where $D(x)$ is the density of the cell, dz is the depth of the cell in the direction of the ray, and τ is a constant controlling the conversion from density to opacity. Instead of using a lighting model, the accumulated colour is used to shade the pixel.

Opacity is accumulated along a ray using

$$A_{m+1} = A_m + a(1 - A_m) \tag{A.3}$$

and colour is accumulated according to

$$C_{m+1} = C_m + a(1 - A_m) I(x) \tag{A.4}$$

where $I(x)$ is stored illumination of the cell. If the opacity becomes greater than 0.999 then no more intersections need be considered and the colour accumulated to that point is returned. Other methods for improving the efficiency of volume rendering include octrees and adaptively terminating the ray [24].

A hybrid algorithm proposed by Levoy [25] was used to combine the volume data with the rigid bodies. All intersections with both rigid bodies and the cubes representing volume data are intersected with. All the intersection points are sorted in increasing order from front to back and then processed in order. If an intersection was with volume data, equations A.3 and A.4 are used as with normal volume rendering. However if the intersection was with a rigid body, then either the Phong or Ashikhmin lighting model is used to determine $I(x)$ for the object, and used with equations A.3 and A.4. After all intersections have been processed, or the opacity becomes greater than 0.999, the accumulated colour is used to shade the pixel.

A.2.1 Blackbody Radiation

A blackbody is an object that absorbs all light that hits it. It reflects no light and appears perfectly black [48]. All blackbodies heated to a certain temperature will emit thermal radiation with the same spectrum. Planck's law,

$$P(T) = \frac{2\pi hc^2}{\lambda^5(\mathrm{e}^{\frac{hc}{\lambda kT}} - 1)}(\mathrm{J} \cdot \mathrm{m}^{-3}) \tag{A.5}$$

gives the energy radiated by a blackbody as a function of temperature. Planck's formula can be used to create blackbody radiation curves which show the amount of energy radiated at each wavelength for a certain temperature, T in kelvin. In equation (A.5), h is Planck's constant (6.626176×10^{-34} J $\cdot$ s), c is the speed of light (2.997924×10^8 m/s) and k is the Boltzmann constant (1.380662×10^{-23} J/K). A true blackbody radiator does not exist, but in practice blackbody radiation can be used as an approximation to determine the colour of certain physical phenomena such as fire [31] and planetary nebulae [26]. Blackbody radiation is used to determine the colour of the particles in the explosion model. The resultant colour from Planck's formula is adjusted according to

$$2P(T_\lambda)^2 + 100P(T_\lambda) + 1000$$

so that it matches the observed range of colour in real explosions.

Bibliography

[1] [online]Available from: `http://www.ariastechltd.com/arias_tech_english/graphics/broken_window.g%if`.

[2] [online]Available from: `http://www.brypix.com/pic_of_day/24-Jan-2005_shattered_glass.jpg`.

[3] Khang-Pu Liou Alex Pothen, Horst Simeon. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):403–452, 1990.

[4] Michael Ashikhmin and Peter Shirley. An anisotropic Phong BRDF model. *Journal of Graphics Tools: JGT*, 5(2):25–32, 2000.

[5] Didier Badouel. An efficient ray-polygon intersection. *Graphics gems*, pages 390–393, 1990.

[6] W. E. Baker. *Explosions in air*. University of Texas Press, 1973.

[7] David Baraff. An introduction to physically based modeling: Rigid body simulation i - unconstrained rigid body dynamics. 1997.

[8] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542 – 547, 1976.

[9] Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004.

[10] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG)*, 5:51 – 72, 1986.

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

[12] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. In *Proceedings of ACM SIGGRAPH 2003*, pages 708–715, August 2003.

[13] M. Fiedler. Algebraic connectivity of graphs. *Czech. Math. Journal*, 23:298–205, 1973.

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

[15] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471–483, 1996.

[16] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 181–188. ACM Press/Addison-Wesley Publishing Co., 1997.

[17] Tolga G. Goktekin, Adam W. Bargteil, and James F. O'Brien. A method for animating viscoelastic fluids. *ACM Trans. Graph.*, 23(3):463–468, 2004.

[18] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*. Society for Industrial and Applied Mathematics, 1998.

[19] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Generation of crack patterns with a physical model. In *The Visual Computer*, number 3, pages 126 – 137. Springer-Verlag Heidelberg, July 1998.

[20] Johan Jansson and Joris S.M. Vergeest. Combining deformable- and rigid-body mechanics simulation. In *The Visual Computer*, pages 280 – 290. Springer-Verlag, February 2003.

[21] Francis S. Hill Jr. *Computer Graphics Using OpenGL*. Prentice Hall, second edition, 2001.

[22] Arnauld Lamorlette and Nick Foster. Structural modeling of flames for a production environment. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 729–735. ACM Press, 2002.

[23] Marc Levoy. Display of surfaces from volume data. In *IEEE Computer Graphics and Applications*, volume 8, 1988.

[24] Marc Levoy. Efficient ray tracing of volume data. In *ACM Transactions on Graphics*, volume 9, pages 245–261, 1990.

[25] Marc Levoy. A hybrid ray tracer for rendering polygon and volume data. In *IEEE Computer Graphics and Applications*, volume 10, pages 33–40, 1990.

[26] Marcus Magnor, Gordon Kindlmann, and Charles Hansen. Constrained inverse volume rendering for planetary nebulae. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 83–90, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Aurélien Martinet, Eric Galin, Brett Desbenoit, and Samir Akkouche. Procedural modeling of cracks and fractures, June 2004. Technical Sketch, Proceedings of Shape Modeling International. Available from: `http://artis.imag.fr/Publications/2004/MGDA04`.

[28] Claude Martins, John Buchanan, and John Amanatides. Animating real-time explosions. In *The Journal of Visualization and Computer Animation*, volume 13, pages 133–145, 2002.

[29] Oleg Mazarak, Claude Martins, and John Amanatides. Animating exploding objects. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 211–218. Morgan Kaufmann Publishers Inc., 1999.

[30] Michael Neff and Eugene Fiume. A visual model for blast waves and fracture. In *Graphics Interface*, pages 193–202, 1999.

[31] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728. ACM Press, 2002.

[32] Alan Norton, Greg Turk, Bob Bacon, John Gerth, and Paula Sweeney. Animation of fracture by physical modeling. *Vis. Comput.*, 7(4):210–219, 1991.

[33] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 137–146. ACM Press/Addison-Wesley Publishing Co., 1999.

[34] Alan V. Oppenheim and Alan S. Willsky. *Signals and Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[35] Ken Perlin. Ken Perlin's homepage [online]. Available from: `http://mrl.nyu.edu/~perlin/doc/oscar.html\#noise`.

[36] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[37] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[38] William T. Reeves. Particle systems. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375. ACM Press, 1983.

[39] David Richards. Fractured reservoirs: software and workflow advancements in fracture and fault prediction, characterization, and connection to reservoir modeling. *CSEG Recorder*, April 2000.

[40] Jeffrey Smith, Andrew Witkin, and David Baraff. Fast and controllable simulation of the shattering of brittle objects. In *Graphics Interface*, May 2000.

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

[42] Jos Stam. Interacting with smoke and fire in real time. *Commun. ACM*, 43(7):76–83, 2000.

[43] Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 129–136. ACM Press, 1995.

[44] John C. Strikwerda. *Finite difference schemes and partial differential equations*. Wadsworth Publ. Co., Belmont, CA, USA, 1989.

[45] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1988. ACM Press.

[46] Fabrice Uhl and Jacque Blanc-Talon. Rendering explosions. In *SCS, Military, Government, and Aerospace Simulation*, volume 29, pages 121–126, 1997.

[47] Andrew Witkin. An introduction to physically based modeling: Particle system dynamics. 1997.

[48] Gnther Wyszecki and W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. John Wiley & sons, second edition, 1982.

[49] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins. Animating explosions. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 29–36. ACM Press/Addison-Wesley Publishing Co., 2000.

[50] Laizin Zhou and Alex Pang. Metrics and visualization tools for surface mesh comparison. *SPIE Proceedings on Visual Data Exploration and Analysis*, 4302:99–110, 2001.