PHYSICALLY BASED SIMULATION OF EXPLOSIONS

A Thesis

by

MATTHEW DOUGLAS ROACH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2005

Major Subject: Visualization Sciences

PHYSICALLY BASED SIMULATION OF EXPLOSIONS

A Thesis

by

MATTHEW DOUGLAS ROACH

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Donald H. House (Chair of Committee)	Frederic I. Parke (Member)
William H. Marlow (Member)	Phillip J. Tabb (Head of Department)

May 2005

Major Subject: Visualization Sciences

ABSTRACT

Physically Based Simulation of Explosions. (May 2005)

Matthew Douglas Roach, B.S.; B.S., Southern Methodist University

Chair of Advisory Committee: Dr. Donald House

This thesis describes a method for using physically based techniques to model an explosion and the resulting side effects. Explosions are some of the most visually exciting phenomena known to humankind and have become nearly ubiquitous in action films. A realistic computer simulation of this powerful event would be cheaper, quicker, and much less complicated than safely creating the real thing. The immense energy released by a detonation creates a discontinuous localized increase in pressure and temperature. Physicists and engineers have shown that the dissipation of this concentration of energy, which creates all the visible effects, adheres closely to the compressible Navier-Stokes equation. This program models the most noticeable of these results. In order to simulate the pressure and temperature changes in the environment, a three dimensional grid is placed throughout the area around the detonation and a discretized version of the Navier-Stokes equation is applied to the resulting voxels. Objects in the scene are represented as rigid bodies that are animated by the forces created by varying pressure on their hulls. Fireballs, perhaps the most awe-inspiring side effects of an explosion, are simulated using massless particles that flow out from the center of the blast and follow the currents created by the dissipating pressure. The results can then be brought into Maya for evaluation and tweaking.

ACKNOWLEDGEMENTS

First of all, I want to thank my wife, Parke. She was supportive when I was discouraged, she was tough when I was lazy, and most of all, she went to bed early so I could get some work done. Plus, she explained all the equations to me. We were not married when I started this thing, but she knew so much about fluid dynamics, I had to find some way to spend more time with her.

Dr. House was a great advisor and chair. He had ideas when I was stuck and was a great sounding board for my wacky short cuts. He was always there to help, motivate, and discourage me from writing jokes into my thesis proposal.

The whole Viz Lab was a wonderful resource for my project. There was always someone there willing to stop what they were doing and chat physics, C++, or ray tracing while I tried to figure something out. For instance, Bob Moyer helped a great deal when I was trying to get my geometry from Maya to the Explosion simulator. Using a shader was a brilliant idea. Zeki Melek always had helpful ideas for my code, and almost single handedly ported me over to Windows. Will Telford was in California the whole time, but was still able to help me with my MEL scripts. Gavin McMillan contributed to my project in countless morale-boosting ways including introducing me to Halo, and co-founding Margarita Fridays.

I would also like to thank Jill Raupe for her willingness to assist this confused graduate student. She kept me enrolled in Texas A&M over the phone and filed my paperwork for graduation via email.

TABLE OF CONTENTS

		Page
ABSTRACT		iii
ACKNOWLEDGEMENTS		iv
TABLE OF CONTENTS		v
LIST OF FIGURES		vii
CHAPTER		
I	INTRODUCTION	1
	1.1 Introduction	1
	1.2 Background and Related Work	4
II	PHYSICS OF EXPLOSIONS	9
	2.1 Shock Front	9
	2.2 Non-Ideal Blast Waves	10
	2.3 Diffracted Waves	10
	2.4 Mach Stem	13
	2.5 Blackbody Radiation	14
III	MATHEMATICAL MODEL	15
	3.1 Conservation of Mass	15
	3.2 Compressible Navier-Stokes	15
	3.3 Conservation of Energy	16
	3.4 Dynamic Overpressure	16
IV	SIMULATION	18
	4.1 Discretizing Fluid Equations	18
	4.2 Boundary Conditions	19
	4.3 Initial Conditions	20
	4.4 Rigid Bodies	20
	4.5 Connecting Rigid Bodies and Fluid	23
	4.6 Particle System	25

CHAPTER			Page
	4.7	Integration Methods	26
	4.8	Interfacing with Maya	27
V	RESULTS		32
	5.1	Results	32
VI	CONCLUSION		37
	6.1	Conclusion	37
	6.2	Future Improvements	42
REFERENCES			44
APPENDIX A			47
VITA			53

LIST OF FIGURES

FIGURE Page

2.1 Ideal Blast Wave Equation.. 9

2.2 Diffraction Wave I.. 11

2.3 Diffraction Wave II.. 11

2.4 Diffraction Wave III... 12

2.5 Diffraction Wave IV... 13

4.1 Animation Inaccuracy... 22

4.2 Voxelizing a Building... 29

5.1 OpenGL Simulation Frames I... 33

5.2 OpenGL Simulation Frames II.. 34

5.3 Animated Object Frames.. 35

5.4 Particle Animation Frames.. 36

CHAPTER I

INTRODUCTION

1.1 Introduction

For decades, explosions have been the most dynamic and visually compelling special effects in film and video games. They have become so prominent in action and adventure movies that it seems unusual for a movie not to have one these days. Often they are more important than the plot or even the characters themselves. Fans of the movie *Independence Day* would be hard pressed to remember any of the characters' names but they all remember the scene when the White House exploded [Twentieth Century Fox 1996]. What would the movie *Star Wars* be without the final explosion of the Death Star [Twentieth Century Fox 1977]? Explosions are easily the most frequently used and most visually stimulating special effect in Hollywood today.

Traditional explosive effects, also called practical effects because they are done in front of the camera, not afterwards in the computer, are usually achieved in one of two ways. Either a scaled down model is built and blown up in front of high-speed cameras, or actual explosives are employed. Neil Corbould and Steven Spielberg used slurries, high explosives used for mining, covered with special sand that had been filtered to remove any possible shrapnel for the mortar blasts in the opening sequence of *Saving Private Ryan* [Magid 1998a, Dreamworks SKG 1999]. However, any blast that occurred

---

This thesis follows the style and format of *ACM Transactions on Graphics.*

remotely near an actor had to be done solely with air pipes buried under the sand. *X-Files: Fight the Future*'s [Twentieth Century Fox 1998] visual effects supervisor Mat Beck used a combination of full-size pyrotechnics and a one-eighth scale model to blow the façade off of the Unical building in downtown Los Angeles, doubling as a Dallas Federal Building for the movie [Magid 1998b]. For the movie *Spiderman* [Sony Pictures 2002], a one-eighth scale model of the Roosevelt Island Tramway wheelhouse was constructed in rural Agua Dulce, California complete with soda machines and waste receptacles [Fordham 2002]. They even had to engineer a special rig to propel the pull-wheel out of the explosion toward camera. When filming a scaled model, it is important to increase the film speed otherwise the scale becomes quite apparent. Speeding up the film, and thus slowing down the effect, makes the explosion seem larger and more massive. Incidentally, these three examples, though generally considered practical, were each sweetened in post-production with some computer-generated effects as well.

There are numerous compelling reasons for using computers to generate explosive effects instead of the more traditional practical techniques. The most significant motivation of course would be the concern for the actors' safety. When the explosion is entirely within the computer, there is no chance of someone accidentally being caught in the blast. For those who are production cost conscious, computer-assisted explosions are cheaper and quicker than precisely scaling and placing detonators, and fireproofing existing structures or building special miniatures that can only be filmed from a distance. Computer generated blasts also allow the director complete control over the camera placement, as in the movie *Swordfish* where director

Dominic Sena was able to place the camera almost at the center of a major explosion [Warner Brothers 2001]. Perhaps the most useful reason for using software to create a detonation is the iterative control over the final visual appearance of the effect. Usually when directors shoot a practical explosion, they set up for days just to get several angles on one single explosion. With computer-generated effects, directors can look at an iteration and ask for things to be changed to reflect their creative vision more closely. Another reason to opt for digital explosions has cropped up only recently. If the entire scene around an explosion were completely computer-generated, compositing in a real blast would probably expose the illusion. A couple of examples of computer-simulated explosions in computer-generated environments would be the fireballs in the movie *Final Fantasy* [Columbia Pictures 2001], by Square, and the asteroid detonation in *Star Wars: Episode II* [Twentieth Century Fox 2002].

The primary goal of my thesis is to implement a physically based explosion simulator into an existing interactive software package. Most innovative visualization solutions for explosions and blast waves have yet to be realized for the average end-user or animator because of the sheer complexity of the code and the prohibitive computational intensity of the simulation. I, therefore, have developed a solution that is simple to use and computationally swift, and integrated it into an interactive environment that many users already understand.

1.2 Background and Related Work

Scientific interest in the processes of the generation and transmission of blast waves through air and the resulting phenomenological effects dates back at least a century, and of course, research in the field intensified significantly during and after World War II [Baker 1973]. Numerical solutions of mathematical models that approximate the phenomenon of temperature and pressure discontinuity in a compressible fluid such as air were first developed in the 1950's. These original programs were expanded over the decades to encompass more of their respective models. Incidentally, in 1970, W. E. Johnson developed the original donor-acceptor method, a technique for determining the mass flux between discrete fluid cells and a more complex ancestor to the one that will be implemented in this thesis [Mader 1998]. These highly accurate numerical solutions were then too slow for efficient computer rendered explosions, however. So, more simplified techniques were initially used to approximate the visual results of an explosion.

The earliest attempts at mimicking explosions with a computer simulation used a system of infinitely small objects called particles. As William Reeves [1983] described in his paper, the particles could be moved with functions that approximated various physical laws and mathematical models. Rendering the particles as points, color streaks or volumes creates a simple approximation of water, clouds, fire, or even explosions. Reeves used this technique to create the Genesis Demo sequence from the motion picture *Star Trek II: The Wrath of Khan* [Paramount Pictures 1982]. Industrial Light and Magic ended up using the footage in the movie's sequel *Star Trek III: The Search for*

Spock [Paramount Pictures 1984], and a related simulation for the exploding objects in a computer tactical display for the movie *Return of the Jedi* [Twentieth Century Fox 1983]. In 1990, Karl Sims made particles more accessible to the industry by developing more controls and simpler pseudo code; he also devised some techniques for better visualizing fire and falling water with this system [Sims 1990]. Though they were quite revolutionary and dazzling in the eighties and early nineties, particle systems are quite computationally intensive since several million are required for a convincing frame of animation. Worse yet, the particles cannot collide with each other. Therefore any attempt to model a continuum, such as fluid flow or a blast wave, with particles is exceptionally difficult; the particles cannot maintain a volume. The fundamental weakness in particle systems is that they are simply points in space; any attempt to give them volume is really just a trick and therefore not physically accurate.

Another interesting technique for simplifying the mathematical model of a discontinuous pressure front in a compressible fluid, which is what air becomes at the temperatures and pressures of most common explosions, is to assume perfectly symmetrical blast waves that are impervious to the obstacles in the surrounding environment. These techniques use explicit functions for the pressure at any given point in a scene based on the distance from the center of the blast and the time elapsed since detonation. These functions are called blast curves. Neff and Fiume [1999] use blast curves, created by researchers in the structural engineering fields, in their paper about fracture algorithms because they need an explosion algorithm that is simple to implement and quick to run. Since blast curves are simply explicit functions, this

algorithm requires almost no computation time and is relatively simple to implement if you have the experimental data already. Oleg Mazarak et al. [1999] also use blast curves in their paper about fracture algorithms for similar reasons. However, instead of completely changing the blast curves for each type of detonation, they use a modified Friedlander equation to approximate all blast curves. This solution is particularly flexible since the equation's parameters can be tweaked for specific explosive properties. Unfortunately, blast curves vary only in one dimension, distance from the explosion center, which means they are unchanged by walls or the ground and cannot be shaped in any way. Therefore, they are only useful in very simple environments where obstacles would not be present to reflect or diffract the blast wave.

In order to achieve a complex solution that reacts to the surroundings, researchers in computer graphics turned to simulating fluids. Kass and Miller [1990] developed a shallow water simulator that modeled only the surface of a fluid. The next year, Wejchert and Haumann [1991] published a paper that simplified three-dimensional fluid flow by creating several flow primitives. These primitives could be placed into a scene to mimic the behavior of a fluid flowing around obstacles and into and out of sources and sinks. Then, in a big leap forward in physically based fluid animation, Foster and Metaxas [1996, 1997] implemented a finite difference approximation of the Navier-Stokes equations by dividing a scene into a grid of cubes called voxels. The cells generate pressure and velocity fields that are then used to transport fluid between the cells. While their research did introduce the industry of computer graphics to the field of computational fluid dynamics, Foster and Metaxas also introduced us to the "relaxation

step", the step in the algorithm where every cell is iteratively adjusted in order to maintain computational stability. The relaxation step was not reliably stable however, and Jos Stam developed an algorithm in 1999 that eliminated it entirely [Stam 1999]. His process, called "Stable Fluids", uses a technique called the *method of characteristics* to calculate convection. This procedure guarantees stability because the maximum velocities of the previous time step bound the velocities of each step. Stam's work brought fluid simulation to the masses since time steps were no longer bounded by the complexity of the scene; real-time interaction with a fluid was finally possible. Unfortunately, the price for ensured stability is the dissipation of fluid mass and vorticity. Two papers in 2001 attempted to bring the interesting turbulence back into fluid simulations. Foster and Fedkiw [2001] combat the dissipation with a hybrid surface algorithm and Fedkiw, Stam and Henrik Wann Jensen add turbulence back into the stable fluid system with a vorticity field generated by a physically based heuristic [Fedkiw et al. 2001]. While the current state of fluid simulations is quite convincing for water, smoke, and turbulent gases, all of the above papers address an incompressible fluid since at the temperature and pressures they are concerned with, the compressibility of water and air is negligible. However, at the pressures associated with explosions, air is highly compressible; in fact, a blast wave is the manifestation of that phenomenon.

A recent paper by Gary Yngve, James O'Brien, and Jessica Hodgins attempts to handle compressible flow for the very purpose of modeling the propagation of a blast wave [Yngve et al. 2000]. They model the area containing the explosion as a three-dimensional fluid divided into voxels. Each voxel is treated as a separate fluid that

experiences changes in density and energy based on the flow across each of its six borders. They also discuss ways to handle interaction between the fluid and solids in the pressure field. I relied heavily on their paper while implementing my compressible fluid simulator and even contacted them for some clarification while writing my code.

CHAPTER II

PHYSICS OF EXPLOSIONS

2.1 Shock Front

Because air is actually a compressible fluid, a high-pressure event, such as a detonation, creates velocities too high for air molecules to move out of the way of the dissipating mass. Instead, the air compresses. The leading edge of a significant pressure disturbance becomes a nearly discontinuous wave as it expands and dissipates. In a still, homogeneous atmosphere, a spherically symmetric pressure source creates very predictable results as all system characteristics becomes functions of time and distance from the blast center. This simplified case is called an ideal blast wave, and the pressure at a fixed distance over time would look similar to Figure 2.1. [Baker 1973]

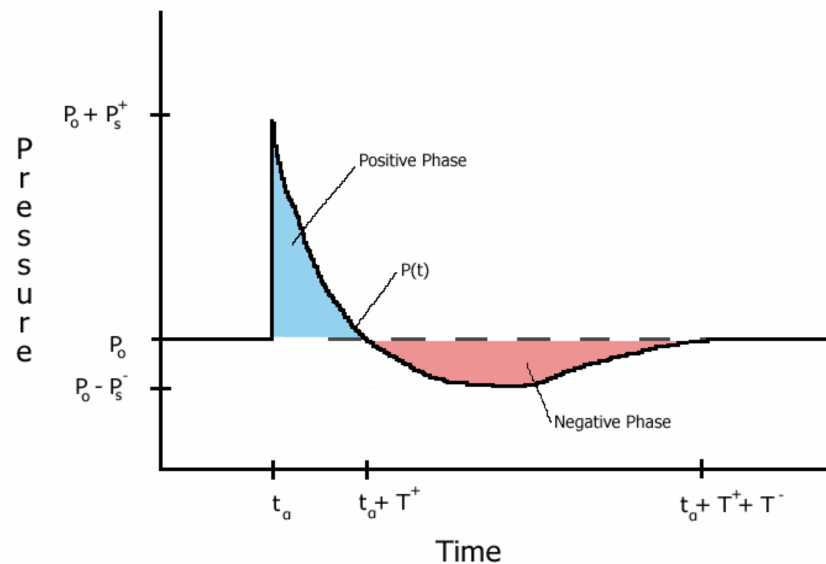

Figure 2.1: Ideal Blast Wave Equation

Many models have been proposed over the last sixty years to describe this function mathematically. In their paper, “Animating Exploding Objects,” Oleg Mazarak et al. [1999] used the modified Friedlander equation to approximate the ideal blast wave since their paper was more concerned with fracture algorithms and the resulting animation of objects.

2.2 Non-Ideal Blast Waves

Baker suggests several reasons for non-ideal blast waves including wave noise caused by the explosive’s casing and thermal radiation inhomogeneously preheating the atmosphere around an explosion. However, he also states, “small aberrations from ideal conditions usually smooth out quickly as the blast wave passes through the air, resulting in relatively ideal blast waves at a distance from the blast source.” [Baker 1973] This is good news for simulations of explosions that require only visual accuracy since the initial flash from the detonation will cover the first few milliseconds of non-ideal behavior and the remaining blast wave can be more simply modeled.

2.3 Diffracted Waves

With the interaction of a shock front with boundaries of finite extent, such as those presented by solid objects, complex behavior ensues. These reactions are collectively called diffraction.

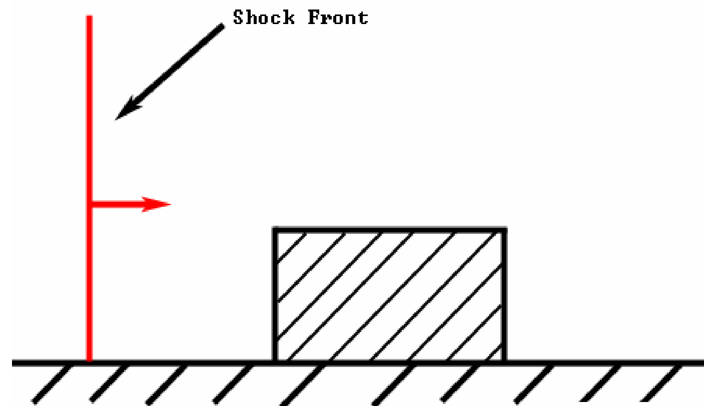

Figure 2.2: Diffraction Wave I

For ease of explanation, let us imagine a planar shock wave, *i*, approaching a rectilinear finite object. Simplifying to two dimensions leaves us with the case shown in Figure 2.2. The pressure behind the incident wave is ambient, p_o, plus the pressure of the shocked-up air, P_s.

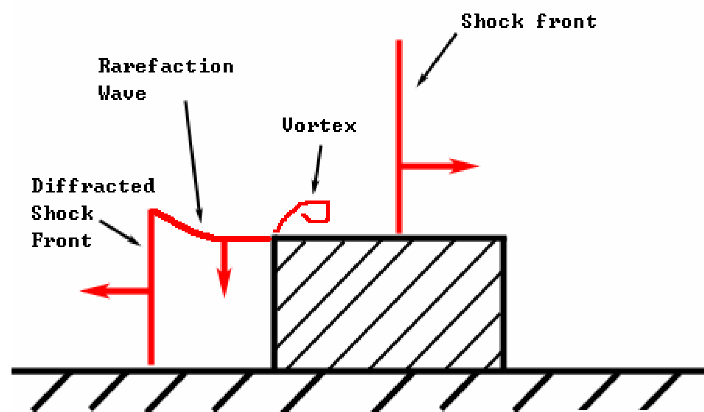

Figure 2.3: Diffraction Wave II

The pressure on the front wall is at its maximum shortly after the wave hits as the momentum of the wave pushes more and more air against the object. When the pressure mounts to the point of reversing the momentum, a reflection wave, *r*, is created. Meanwhile, the portion of the wave not hitting the front face continues unabated. As the reflected wave moves away from the wall, a rarefaction wave moves down the front face. A vortex forms at the upper left corner of the object where the Venturi effect causes significant spin. At this point, as depicted in Figure 2.3, the pressure of the reflected wave is $p_o + P_r$, the pressure of the incident wave is a little lower, $p_o + P_s$, and the pressure near the vortex is even slightly less.

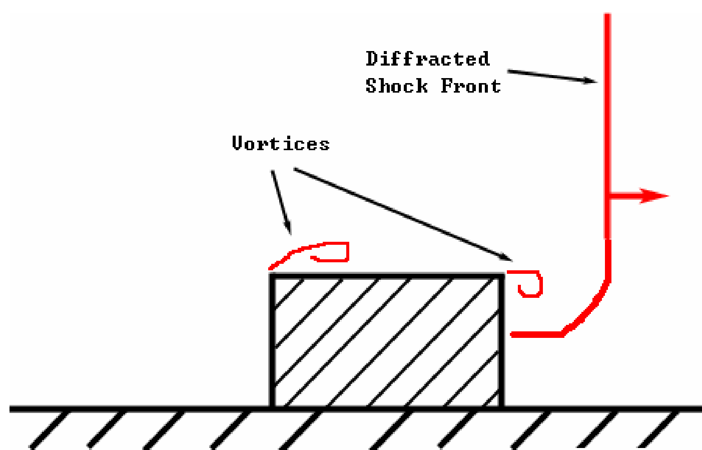

Figure 2.4: Diffraction Wave III

As the incident blast front passes beyond the rear face of the object, the wave diffracts around it, as shown in Figure 2.4. At this point, another vortex forms and the pressure on the back face begins to build up. Finally, as depicted in Figure 2.5, the shock

front eventually moves beyond the object, having lost some of its energy to the reflected wave and been reshaped by the process of diffraction. [Baker 1973]

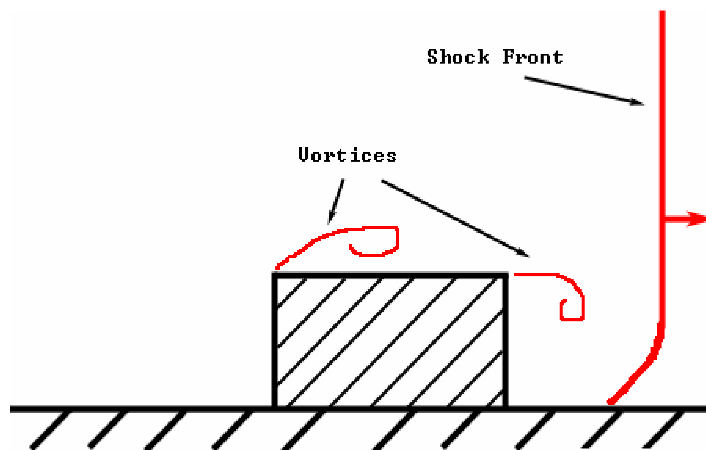

Figure 2.5: Diffraction Wave IV

2.4 Mach Stem

Due to the pressures and velocities involved in explosions of high energy, there is a critical angle of incidence beyond which normal, acoustic, wave reflection does not occur. At this angle, the reflected wave begins to merge back together with the incident wave, creating a much more intense wave front called a Mach stem. This interaction of a shock front with a ground plane causes blasts that occur slightly above the ground to be much more destructive than ones that occur in contact with the same plane. In acoustic reflections, the maximum obtainable pressure is simply twice the incident pressure. In a high-energy explosion, the multiplier can be as much as twenty. Unfortunately, much of this increased ratio comes from real gas effects such as ionization and dissociation, which are ignored in this simulation for the sake of simplicity. [Baker 1973]

2.5 Blackbody Radiation

Blackbody radiation refers to an object, or system, which absorbs all incoming radiation and emits radiation solely dependent on the characteristics of the system, thus independent of the incident radiation. For our purposes, the particulate matter pushed around by the explosion will behave as blackbodies, giving off light solely depending upon the temperature of the particle. Planck's radiation formula gives energy density in terms of wavelength, and the derivative of that formula, the Wien Displacement Law, can be used to determine the peak wavelength's relation to temperature. Thus, we have an equation for radiation in terms of body temperature. [Nave 2003]

$$\lambda_{peak} = \frac{2.898 \times 10^{3}\, m \cdot K}{T} \tag{1}$$

CHAPTER III

MATHEMATICAL MODEL

3.1 Conservation of Mass

The most basic equation affecting a compressible fluid flow is the conservation of mass. The mass conservation equation simply states that the change in mass within a given volume must be equal to the mass flux across the volume's corresponding surface. [Yngve et al. 2000]

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho v) \tag{2}$$

3.2 Compressible Navier-Stokes

The compressible Navier Stokes equations govern the conservation of momentum in a fluid. Body forces, the pressure gradient, viscous accelerations, and the convective transportation of momentum determine the change in velocity of a particular differential volume within the fluid. The body forces usually represent constant accelerations like gravity. The pressure gradient affects momentum because fluid naturally flows from higher to lower pressures. Viscosity fights both vorticity and acceleration, while convection simply reflects the change in momentum caused by fluid flowing into and out of the given volume. [Yngve et al. 2000]

$$\rho \frac{\partial v}{\partial t} = \rho \cdot f - \nabla P + \frac{\mu}{3} \nabla \cdot (\nabla v) + \mu \nabla^2 v - \rho (v \cdot \nabla) v \tag{3}$$

3.3 Conservation of Energy

The First Law of Thermodynamics governs the conservation of energy in a compressible fluid system. This law dictates that the change of energy in a system, in this case, a differential volume of fluid, is equal to the amount of heat added to the system minus the work done by the system. The work in the system is performed by pressure and viscosity, and the heat is added via thermal conductivity. The change in internal energy is thus determined by

$$\rho \frac{\partial N}{\partial t} = k \cdot \nabla^2 T - P\nabla \cdot v - \frac{2\mu}{3}(\nabla \cdot v)^2 + \frac{\mu}{2} \sum_{i,j \in \{x,y,z\}} \left(\frac{\partial v_i}{\partial j} + \frac{\partial v_j}{\partial i} \right)^2 - \rho(v \cdot \nabla)N \qquad (4)$$

The first term is thermal conduction of the system. This term reflects changes in energy due to heat flowing from high to low temperatures. The second term is the work done by pressure over the divergence of velocity; this can be thought of as the work required to maintain different velocities inside and outside of the volume. The two viscosity terms represent the energy lost as variations in velocity are damped out of the system plus the smaller amount of energy gained by the production of heat in the process. The final term represents the change in internal energy due to the convection of fluid across the surface boundary of the differential volume. [Yngve et al. 2000]

3.4 Dynamic Overpressure

The two types of forces experienced by an object surrounded by fluid are hydrostatic and dynamic. Hydrostatic forces act normal to the surface of an object, and exist because of molecules bouncing around naturally. These forces are measured as pressure. The flow of the continuous fluid creates dynamic forces. These forces act both

normally and tangentially to the surface of an object. The tangential shearing force can be safely ignored in the case of explosions since the hydrostatic pressures are so high. Assuming the object is at equilibrium in ambient pressure, the hydrostatic forces can be computed with the overpressure, $\overline{P}$. The overpressure is simply the difference between the hydrostatic pressure, P, and the ambient pressure P_{amb}. The dynamic overpressure is simply the overpressure plus the added pressure of the velocity of the fluid normal to the object's surface, like so:

$$\overline{P}_{dyn} = \overline{P} + \frac{1}{2}\rho\left(v_{rel} \cdot \hat{n}\right)^2 \tag{5}$$

The velocity in that equation represents that of the fluid relative to the surface, and the vector $\hat{n}$ is the outward surface normal. The dynamic overpressure represents the magnitude of the pressure normal to the surface. The force acting on a small portion of the surface can be calculated by multiplying that pressure by the surface area. [Yngve et al. 2000]

CHAPTER IV

SIMULATION

4.1 Discretizing Fluid Equations

The fluid is discretized into a regular lattice of cubical cells called voxels, short for volume elements. Fluid properties are stored for each voxel and considered constant across the volume. Central finite differences are used to determine spatial derivatives of values such as pressure, temperature, and velocity. The governing equations of fluid behavior hold for each finite voxel the same way they hold for differential volumes. [Yngve et al. 2000]

Once the governing equations are rewritten using finite differences, they can be used as update steps for an explicit integration method. However, according to Yngve, O'Brien and Hodgins, that scheme would fall apart under the stress of the steep pressure gradients created by a blast wave. Instead, they recommend a slightly more complex integration method. First, they suggest stability can be gained by handling the convection terms separately from the temporal ones. The steps for their integration method are as follows [Yngve et al. 2000]:

1. Use the first four terms of equation (3) to calculate fluid acceleration ($\tilde{a}_t = (dv/dt)_t$).

2. Approximate the velocity at the end of the timestep ($\tilde{v}_{(t+\Delta t)} = v_t + \Delta t \cdot \tilde{a}_t$), and then the average velocity during the timestep ($\bar{v}_t = (\tilde{v}_{t+\Delta t} + v_t)/2$).

3. Approximate ΔN using the nonconvective terms of equation (4) while substituting $\overline{v}_t$ for fluid velocity.

4. Compute the new density, ρ, using $\overline{v}_t$ for fluid velocity.

5. Calculate complete $v_{(t+\Delta t)}$ and $N_{(t+\Delta t)}$ with equations (3) and (4) using all the terms and the new value of ρ.

6. Update secondary values with state equations.

To stabilize the integration scheme further, Yngve et al. propose a specific technique for handling the convection of mass in steps 4 and 5. They use the donor-acceptor method. This procedure uses the average velocity at a boundary between cells to determine the direction of flow across it. The magnitude of the flow is proportional to the mass of the donor voxel, thus this process never empties a voxel. Preventing an empty cell stabilizes the integration by keeping the fluid densities positive and avoiding inordinately large changes to cell velocity and internal energy [Yngve et al. 2000].

4.2 Boundary Conditions

The three types of boundaries implemented in this thesis are hard, free and pseudo-free. Hard boundaries represent solid objects in the scene and force fluid velocities normal to them to be zero. Free boundaries are implemented along the edge cells of the simulation in order to let the blast wave pass out of the fluid grid without reflection. This allows the longer-term aspects of an explosion to be explored. The third type of boundary, pseudo-free, is employed purely to speed up the simulation while the blast wave is still small relative to the size of the grid. Cells with pressure differences

below a certain threshold are ignored for a time step, allowing most of the cells to be skipped while the time steps are at their smallest. [Yngve et al. 2000]

4.3 Initial Conditions

All the fluid related constants as well as the ambient pressure and temperature are specified in one place to facilitate fine-tuning of the performance of the simulation. The cells within the specified detonation sphere or spheres have their temperature and pressure values initially augmented. The default values are those recommended by Yngve et al. [2000] to simulate a typical chemical explosion, 2900K and 1000 atmospheres. The detonations can also be time delayed to allow multiple blasts to occur in succession.

4.4 Rigid Bodies

The motion of rigid bodies in this simulation is implemented using the equations presented in Witkin and Baraff's physically based modeling course notes. The details of deriving these equations have been omitted and can be found in their document [Witkin and Baraff 1999].

The rigid body's state at any point in time is defined by the state vector

$$X = \begin{bmatrix} \vec{x} \\ q \\ \vec{P} \\ \vec{L} \end{bmatrix} \tag{6}$$

where

$\vec{x}$ is a vector representing the position of the center of the body's mass

q is the quaternion representing the rotation of the body

$\vec{P}$ is a vector representing the linear momentum of the body

$\vec{L}$ is a vector representing the angular momentum of the body

The time derivative of the state vector is computed and used to calculate the state at the next timestep. Where

$$\dot{X} = \begin{bmatrix} \dot{\vec{x}} = \vec{P} / M \\ \dot{q} = \frac{1}{2}[0, \omega] \cdot q \\ \dot{\vec{P}} = \sum f_i \\ \dot{\vec{L}} = \sum \tau_i \end{bmatrix} \tag{7}$$

being

M = total mass of the body

ω = Angular Velocity = $I^{-1}\vec{L}$ (8)

I = inertial tensor of the body

f_i = the ith force being applied to the body

τ_i = the ith torque being applied to the body

Quaternions are used for maintaining body rotation in order to avoid the numerical drift and subsequent skewing inherent in rotation matrices. See reference for more explanation [Witkin and Baraff 1999].

To simplify the process of bringing objects into the simulation and handling their collisions, rigid bodies are initialized as groups of cubes. A cube object is created for every voxel that an object fills at the beginning of the simulation and those cubes are

grouped together into one rigid body. This simplifies collision detection because every concave rigid body is actually a collection of smaller convex rigid bodies. A collision detection hierarchy is setup to speed up the process so that most cubes are eliminated from the list of possible collisions very quickly. For instance, internal cubes never need to be tested at all. This technique does cause some animation inaccuracy since the forces of the fluid will be acting on approximated geometry and in only three directions at any one time. The possible inaccuracies are revealed in Figure 4.1 where in each case the bottom half of the sphere approximation is experiencing a significantly higher pressure than the top half. The resulting force on the perfect sphere in the example would be directly up. The approximations, depending on their exact orientation to the high-pressure field, could receive widely varying resulting forces.

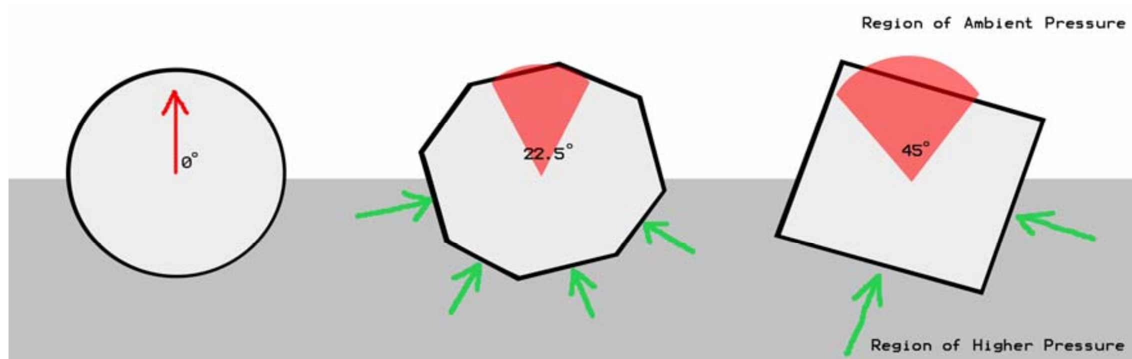

Figure 4.1: Animation Inaccuracy

4.5 Connecting Rigid Bodies and Fluid

The rigid bodies are coupled with the surrounding fluid using the techniques described in “Animating Explosions” [Yngve et al. 2000]. The three-step process involves first animating the rigid body by determining and applying the fluid forces on the faces of a body, then revoxelizing the body after animation, and finally using the change in voxelization to adjust cell values and push fluid around the scene.

4.5.1 Determining Forces

Assuming the objects in the scene are at equilibrium at atmospheric pressure, hydrostatic forces are computed using the overpressure, $\overline{P}$, which is simply the difference between pressure, P, and ambient pressure, P_{AMB}.

$$\overline{P} = P - P_{AMB} \tag{9}$$

The dynamic forces created by fluid momentum break into two types, a normal force and a tangential shearing force. The shearing force is irrelevant in this context since it is negligible compared to the hydrostatic forces near detonations. The total normal force on any point on the surface of an object is then evaluated as the dynamic overpressure.

$$\overline{P}_{dyn} = \overline{P} + \frac{1}{2}\rho\left(v_{rel} \cdot \hat{n}\right)^2 \tag{10}$$

The force is approximated to be constant over the face of an object, so the actual force, *f*, on any facet with surface area, *A*, would be

$$f = -\hat{n}A\overline{P}_{dyn} \tag{11}$$

4.5.2 Voxelization

In order for the fluid to react to the movement of the objects, it must know where the objects are. This information is generated by a technique called voxelization. Voxelization works by breaking up an object into tiny pieces and placing each one into the grid cell it occupies. Then each grid cell knows how full or empty it is. Full cells are temporarily considered hard boundaries and partially filled cells have their convection equations modified to reflect their altered volumes. For a simple unit cube, the process of voxelization would entail breaking the cube into one thousand pieces and placing each one in the grid cell that contains the center point of each smaller piece. Since each rigid body is stored in the simulation as a collection of these cubes, it is very easy to step through the sub-cubes of an object repeating this procedure.

After the forces are applied on each face of each object's surface, the rigid body motion of the objects is computed normally. Once this step is completed, each object is then revoxelized and the change in volume of each voxel is calculated. This change in volume is used to displace fluid.

4.5.3 Displacing Fluid

The change in partial volume of each voxel works like a miniature piston compressing or expanding the gas in that cell. Since the fluid is compressible, pressure does not vary directly with the change in volume. However, mass *is* conserved; so the new density, ρ, is determined by

$$\rho_2 = \rho_1 * \frac{V_1}{V_2} \tag{12}$$

To update the pressure and temperature of the voxel, I use a thermodynamic equation from Yngve's paper relating the work done to a system by changing the density,

$$\frac{P_2}{P_1} = \left(\frac{\rho_2}{\rho_1}\right)^{\gamma} = \left(\frac{T_2}{T_1}\right)^{\gamma/(\gamma-1)} \tag{13}$$

where $\gamma = 1 + R/c_v$ with R and c_v being constants specific to the fluid. If γ were set to one, the fluid would be incompressible, and in the case of air, a γ approximately equal to 1.4 is accurate. [Yngve et al. 2000]

Special circumstances arise when an object begins to exit a completely filled cell and when an object initially envelopes an entire cell. In these two cases, the voxel in question must be handled simultaneously with a neighbor. These two cells together are treated as one larger voxel so that the partial volume is never zero. The "Animating Explosions" paper suggests choosing the neighboring cell based on the largest axial component of the object's velocity [Yngve et al. 2000]. In a slight departure, I chose to use the instantaneous velocity of a point on the object near the cell's center. This change allows for better accuracy in a few specific cases. For instance, when an object is simply spinning in place it would not have any velocity at all, making the choice of neighboring cells impossible. My adjustment handles those cases where the angular velocity is more significant than the linear velocity.

4.6 Particle System

A particle system is used to simulate the fireball effect commonly associated with explosions. The system suggested by Yngve et al. [2000] is an extreme

simplification of the particle systems described in [Sims 1990, Reeves 1983]. Instead of tracking velocity, only position of each particle is stored in the state vector. Each timestep, the fluid velocity at the position of a particle is interpolated. The massless particle simply moves at the speed of the fluid for that timestep. This technique simulates detonated material from the explosion spreading out with the thermal currents and buoyancy of an explosion.

4.7 Integration Methods

In an attempt to provide a balance of simulation speed and accuracy, a combination of integration techniques is used in this program. The rigid body motion is calculated with a fourth order Runge-Kutta technique because rigid bodies become unstable with less accurate methods [Witkin and Baraff 1999]. The three-dimensional fluid is simulated by a modified Euler integration. However, the extreme pressure gradients would still create negative densities and unstable velocities with this technique, so a special donor-acceptor method keeps the simulation stable by handling the convection terms of the equations [Yngve et al. 2000]. In order to speed up the simulation after the blast wave, the most numerically complicated aspect of the simulation, has left the grid, the step size is dynamic. Before each iteration, the program calculates the maximum divergence in the fluid grid and uses it to determine the largest stable time step.

4.8 Interfacing with Maya

4.8.1 Scene Creation

The scene is entirely setup within Maya using simple scripts. The first script to run on a scene is ExploSimSetup.mel. This script sets up the approximate fluid grid and creates a Filenames node, which can be used to direct all output and input to or from Maya.

After the scene has been modeled and laid out, select all the solid objects, stationary or movable, which you want to interact with the simulation fluid. Running PrepareObjects.mel on these selected object attaches simulation attributes to them that you can adjust on a per object basis. A particular Renderman shader is also attached to each object; I will elaborate on this later. The main attributes added by the script are ObjectID and Mass. Leaving the Mass value at zero means you want that object to be a hard boundary in the simulation, whereas entering a positive value will allow the geometry to be animated by the explosion. All objects with the same ObjectID are combined into one rigid body in the simulator. Adjusting these numbers can be used to keep separate pieces of geometry from flying apart in the simulation. I should mention a few caveats. First, you have full control over the relative size of the fluid cells, but geometry thinner than a dimension of the fluid grid will likely not translate into the simulation. I recommend modeling simpler proxy geometry for each of your objects and using them to create the initial fluid grid. In addition, large concave objects, such as a wall combined with a ground plane, will slow down the simulation significantly as many of the collision detection shortcuts are undermined by this type of geometry. Simply

making the ground plane and the wall discrete convex rigid bodies can speed up the simulation by two orders of magnitude.

The CreateExplosion.mel script adds an explosion to the scene in the form of a scalable initial detonation sphere. This proxy blast geometry has some attributes on it that can be used to fine tune and adjust the timing of the simulation. For instance, the temperature and pressure range of the blast can be adjusted, as well as the detonation offset. This offset value allows for multiple and successive explosions to occur in the fluid.

The last step in scene creation is to output something from Maya that ExploSim can read. The script that handles the various forms of output is called Output2ExploSim.mel. Based on the paths and filenames stored in the Filenames node, this script writes out a mass file, an explosion file, and then renders the initial state of the voxel grid. The mass file is simply a list of object identification numbers followed by the corresponding mass and initial velocity. The explosions file is just a list of the blasts in the scene and their pertinent attributes like position, size and pressure. The initial state of the fluid grid is a bit more complicated. The PrepareObjects.mel script attached a Renderman shader called ExploSim.sl to each of the objects in the scene. This shader returns black for the exterior of an object and a color that represents the object’s identification number for the inside of an object. Then an orthographic projection camera, placed at the top of the explosion grid by ExploSimSetup.mel, animates its clip plane down through the grid at intervals related to the dimension of the fluid grid cells. Thus, each rendered frame of this animation represents a separate slice of the initial fluid

state. Figure 4.2 shows an object being voxelized from Maya. The red grid represents the animated clipping plane moving down the object, and the black and white image is the corresponding rendered image output from Renderman. Then, when ExploSim runs, it reads in these files and initializes the simulation.

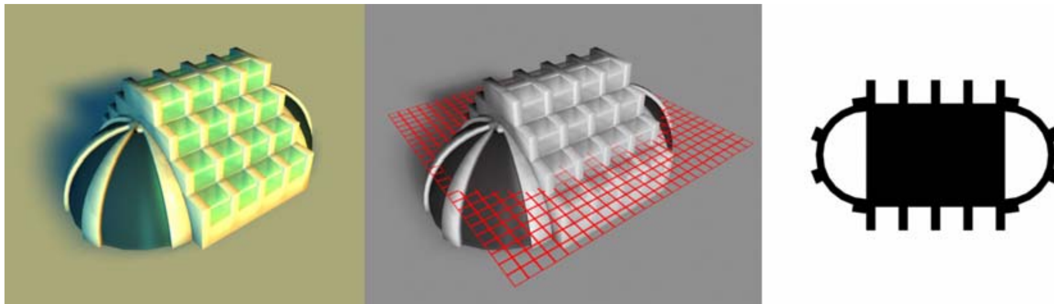

Figure 4.2:   Voxelizing a Building

4.8.2 Importing the Results

The pressure grid is turned into renderable geometry by selecting a threshold value and building triangles that approximate the surface around the volume with pressures greater or equal to that value. This technique is called the marching cubes algorithm and was first presented as a surface construction procedure by Lorensen and Cline [1987] in the conference proceedings of SIGGRAPH. The system works by looking at each voxel of the grid as eight points that either above or below the threshold value. Based on the number and position of included vertices, inclusive triangles are created that have their vertices interpolated along the cube’s edges. These triangles form a mesh approximation of the field threshold, or in our case a three dimensional isobaric contour. Relying heavily upon preexisting code from Paul Bourke’s webpage,

“Polygonising a scalar field”, the simulation exports a separate polygonal mesh for each frame of the simulation in Wavefront object file format [Bourke 2003]. I chose obj files because they are simple to create and easily imported into Maya. Once the simulation is finished, running the ImportBlastWave.mel script inside of Maya will import all the blast wave geometry and animate their visibilities so that each one is visible for one frame. This staggering of the mesh visibilities suggests animation of the blast wave. Since the blast wave is only visible due to its bending of light, the lack of motion blur on the animation should not be noticeable.

The animation of the moveable objects in the scene is stored in the form of a position vector and rotation quaternion. Since the purpose of this paper is to bring the results back into Maya, a program that as of version 4.5 does not allow direct manipulation of an object’s rotation quaternion, some conversion process must occur when animation is exported to convert the quaternions into the more common Eulerian angles. Inspired by a more comprehensive version of the code presented online by Ken Shoemake [2003], my code exports position and rotation data for each object at each frame into separate files. Another MEL script, ImportAnimation.mel, brings the animation data into Maya and assigns it to the corresponding objects. Due to the inherent limitations of the Eulerian method of rotation description, the objects will have visually disruptive motion blur artifacts since the rotations resulting from the conversion process are limited to the range of -180 to 180 degrees. If an object rotates far enough about one axis, it will shift suddenly from significantly positive to significantly negative rotation

values. I have another script that fixes these artifacts called FixRotations.mel that can be run on objects with broken motion blurring.

Maya's particle system does not allow for the keyframing of individual particles, so the most obvious solution to bringing in the particle system data would not work. What I create instead is a makeshift particle cache. A runtime expression runs every time the current frame changes that looks up the current frame's particle data and moves each of the particles to the correct place. Unfortunately, all the file accessing makes this implementation a bit unwieldy. In addition, an annoying memory bug in Maya occasionally pops up and crashes your scene. Ideally, it would be possible to write out a native Maya particle cache file, but I could not find an intuitive explanation of how or even a suggestion that it is possible to create usable Maya particle caches from another program.

CHAPTER V

RESULTS

5.1 Results

I created a few different animations in order to reveal the simulation's accuracy and the overall visual interest of the results. Frames from those animations are included below. Figures 5.1 and 5.2 show frames captured directly from the simulator itself. The OpenGL window shows a two-dimensional slice of the pressure wave expanding and diffracting around a wall. Figure 5.3 is a collection of frames from a Maya playblast, demonstrating imported blast wave geometry and objects animated by the simulation. The last image, Figure 5.4, confirms the inaccuracy of my particle system with a few frames from a Maya playblast of simulated particles.

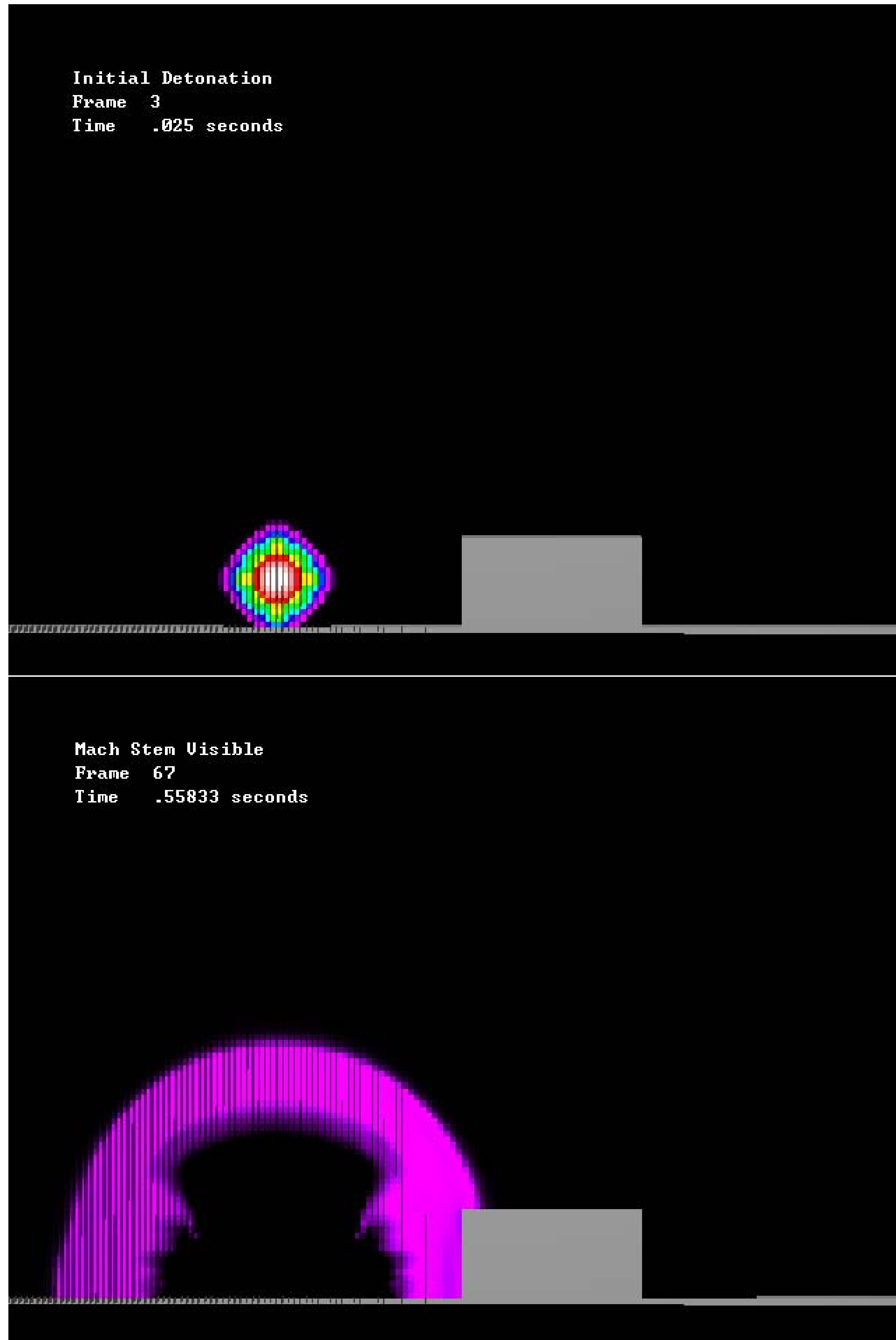

Figure 5.1: OpenGL Simulation Frames I

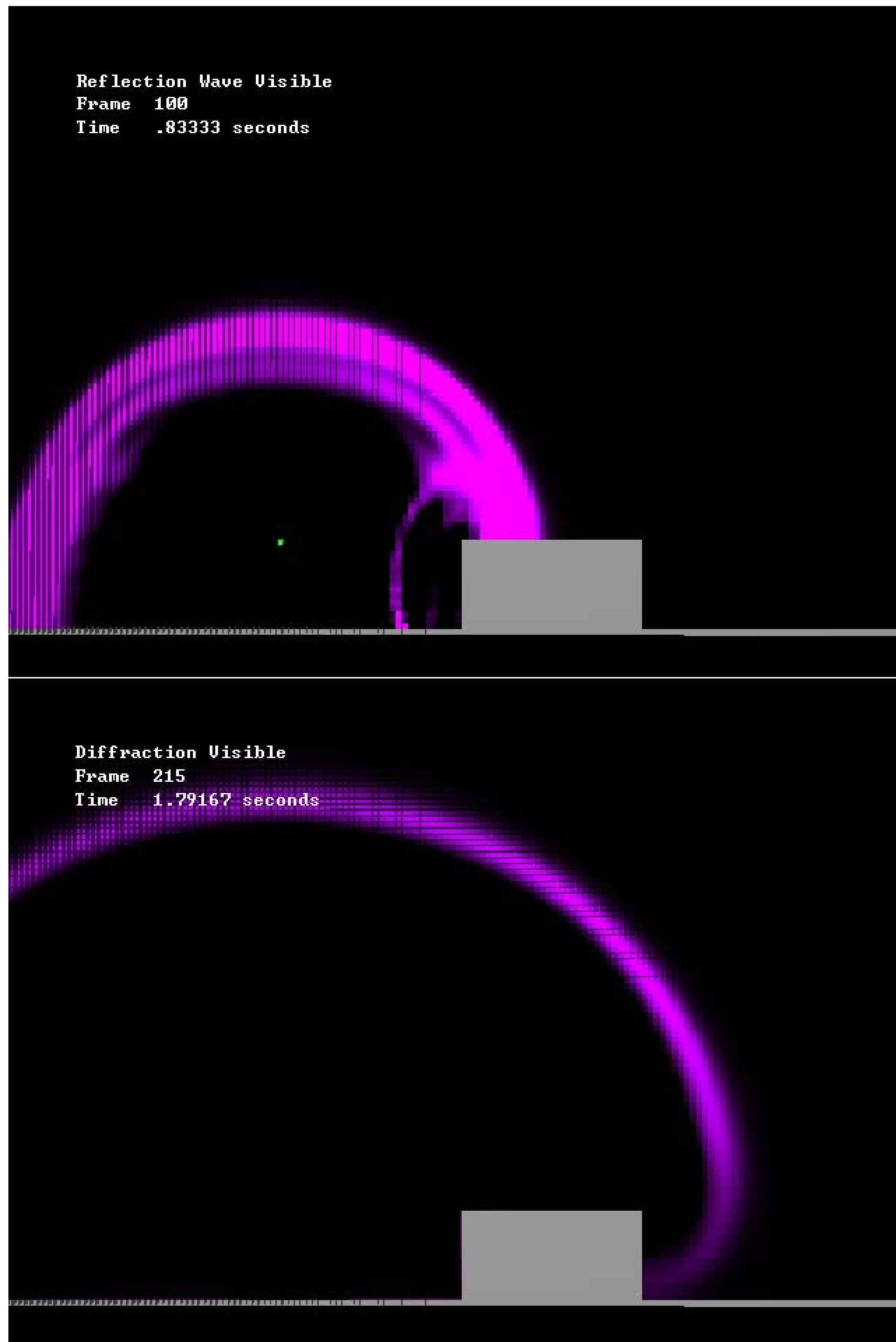

Figure 5.2: OpenGL Simulation Frames II

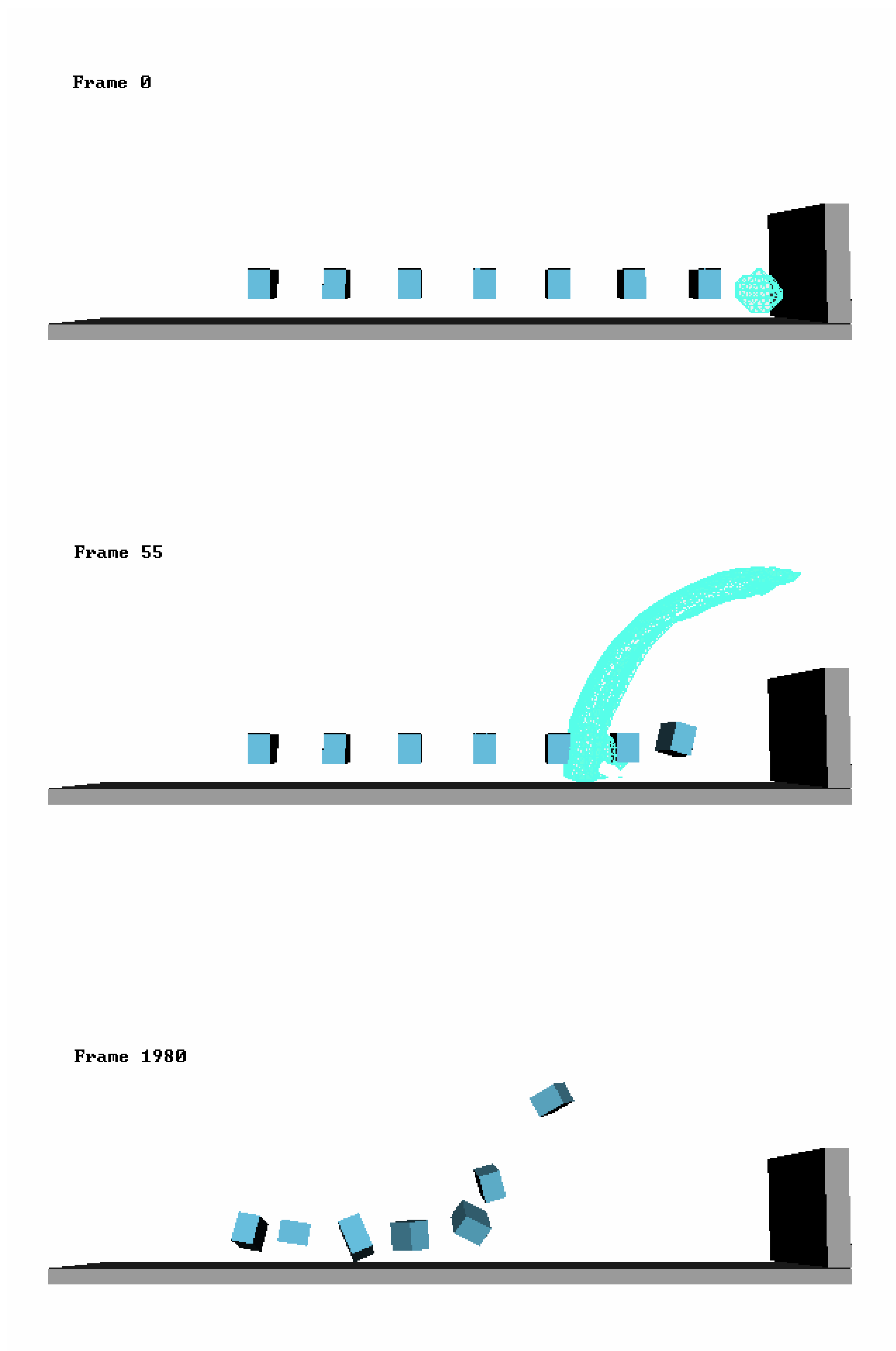

Figure 5.3: Animated Object Frames

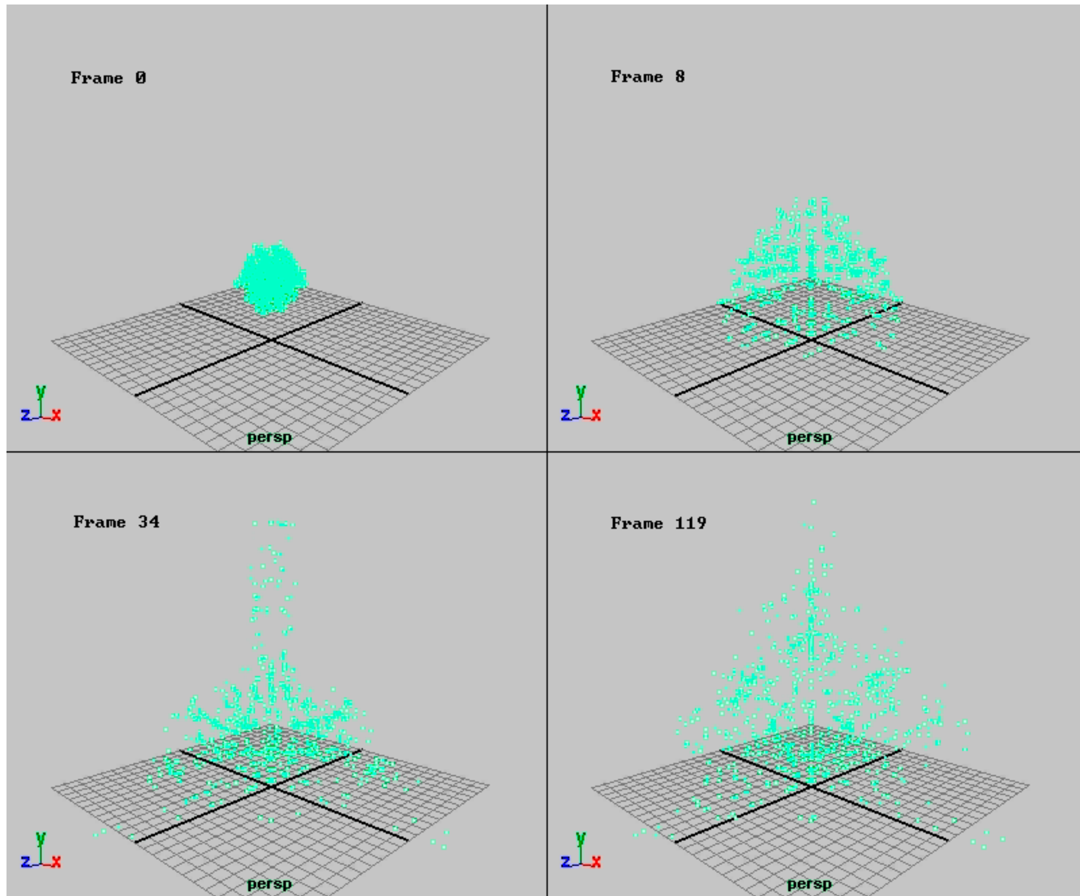

Figure 5.4: Particle Animation Frames

CHAPTER VI

CONCLUSION

6.1 Conclusion

My attempt to build a compressible three dimensional fluid simulator for the purpose of modeling visually stimulating explosions and interfacing it with Maya proved to be too ambitious and merely a partial success.

One aspect of the thesis that performed capably was the process of building three dimensional fluid grids and rigid body approximations from a Maya scene. Almost any scene can be exported to the simulator with only minor tweaks to the geometry. The scripts and the Renderman shader allow a technical animator to start an explosion shot in Maya, then move easily into the simulation. The files necessary for initializing ExploSim can also be created and edited by hand with a text editor and a simple paint program.

The most successful part of the thesis by far is the pressure wave. The frames from the OpenGL simulator reveal quite accurate diffraction behavior. Explosions occurring slightly above the ground plane generate Mach stem reflections of greater intensity than the rest of the blast wave. When a shock front hits a solid obstacle it is reflected, and the wave refracts correctly beyond an unmoving object. The accuracy of the pressure wave propagation suggests the underlying compressible fluid simulation is reliable.

The rigid body simulation was satisfactory. Even with the inaccuracy and instability introduced into the system by the simplifications I chose to make to the rigid body system, it still handled collisions and generated legitimate animation. It certainly works well enough to initialize a more robust rigid body simulator that could handle the post explosion animation quicker and more accurately.

The particle simulation was an utter failure. No usable particle data was ever generated by the ExploSim. The expected behavior of the particles would be initially expanding with the blast wave, followed by a subsequent pull back into the blast center as the particles are left behind by the super sonic blast wave and sucked in by the low-pressure area. My particles did that much; perhaps that portion worked because the velocities would be dominated by the pressure gradient and pressure dissipation was the aspect that worked best. Next, however the particles would be expected to rise into a beautiful cloud, riding the currents of thermal expansion. This behavior would suggest some bugs in the handling of the energy flow. Either the temperature differences are not having enough of an effect on the pressure gradient, or energy is being lost in the system somewhere.

The integration techniques are a bit complicated, but were necessary in order to get the simulation to run in a manageable amount of time. My initial simulations used a fixed timestep of one microsecond and finished the blast wave in forty-eight hour period. Waiting to see if the particles formed a fireball, however, took over a week. The first improvement I made was to make the timestep dynamic so it would speed up as the blast wave moved out of the grid. Using the cell divergence to approximate the eigenvalue of

the system worked well and kept the simulation stable. Some minor tweaks were necessary after rigid bodies were added since the rigid bodies affected the divergence after the timestep had already been calculated. Since each grid cell often needed calculated values from its neighbors, I added a clean/dirty matrix to the simulation so a cell never calculated its derivatives more than once per step. This process obviously saved time over repeated recalculation calls, but it also saved CPU time over one giant cell by cell sweep because only the needed cells are calculated and free boundaries are ignored. After the dynamic timestep and speed-ups were implemented, the whole simulation from detonation to complete dissipation takes less than two days. The actual speed enhancement cannot be accurately measured though due to the program being ported to a new operating system twice, and the inevitable enhancement of processing power.

Bringing data back into Maya proved erratic as I often ran into a memory bug in the Windows version of Maya. Closing a file pointer made the file unreadable, but it often did not free up the RAM. This caused a problem when reading multiple large text files for data input. The blast wave geometry could only be imported twenty to fifty frames at a time before the RAM would need to be purged by restarting Maya. Too large a guess would result in a crash and all unsaved data would be lost. This problem was not as much of a hindrance for the object animation though, and the process of importing animation went rather smoothly. Unfortunately, importing particle data was unwieldy and painful. Maya particles cannot be keyframed or reliably coerced into Maya's caching system from an expression. In the end, I used a runtime expression to read in

and set each particle's position every time the current frame changed. In essence, I created my own particle cache, and this process ran very slowly for the several thousand-particle system I originally envisioned. Even after scaling back the size of the particle system to only a few hundred, the memory bug would still crash Maya after a few frames anyway. In the end, I would have to say interfacing the simulator with Maya was a naïve endeavor motivated by a general ignorance of the software available to the computer graphics industry.

Once all the blast wave geometry was imported into Maya and set up to sequentially become visible, the limitations of the Marching Cubes algorithm revealed itself. The geometry was jagged and heavy to manipulate. Luckily, the lack of accurate motion blur or smooth edges really was obscured by the invisibility of the blast wave as I expected. Renders of the blast wave simply revealed a growing field that diffracted light and warped the background. Unfortunately, any shader applied to the surface must be a cheat since the index of refraction is constant throughout the geometry. It can be animated in the scene or UV mapped onto the blast wave or both, but the object is only a surface, not a volume, so it must be rendered as one.

The animation of objects in the scene by the pressure wave turned out to work fairly well. The objects were pushed away from the blast convincingly and the simulation even generated some exciting rotations on the objects as well. Using the state vectors from the simulation just after the blast wave has passed the objects by to initialize Maya's built-in rigid body simulator created good results in a practical amount

of time. The object animation is by far the most practical result from the simulator at this time.

In conclusion, I believe my project scope was too wide. Perhaps simply implementing a three dimensional compressible fluid simulation would have been more reasonable. Integration with the Maya software package was time consuming, marginally successful, and generally shortsighted. The particle simulation should have been exported directly to a renderer or at least imported into a piece of software specifically designed around physics simulation. The blast wave would have been better served by a volume shader that made full use of the pressure field. Admittedly, writing one could have taken just as long as bringing the marching cubes algorithm results into Maya, but the results would have been more accurate. Only the rigid body animation was successfully incorporated into Maya, but then, animation is one of Maya's strong points.

The fluid simulation worked quite well, though not perfectly. Clearly, the underperforming particles would suggest an error somewhere in the implementation. A more modest goal would have helped here as well. Besides the time lost developing realistic blackbody radiation shaders and smoke for a fireball that never culminated, I also spent a great deal of time learning the underlying physics of the problem. I even took an extra partial differential equations course to solidify my understanding, though a follow-up heat transfer class would probably have helped as well. Despite the wildly ambitious nature of my chosen problem, I believe I managed a respectable solution that addresses all issues in some manner and even performed well in several major areas.

Through research and my limited experience in the computer animation industry, I have learned that explosions are just quicker, cheaper, and simpler to create by hand in a specially adapted software package. The blast wave is far too fast for rigorous visual inspection on screen, and the fireball is simply too important to simulate. Directors are going to want complete control over the shape and color of the most visually stimulating aspect of the effect, and a simulation would take all of the control away from them. The initialization of a rigid body simulation would be the only practical result of an explosion simulation in the computer graphics industry since hand animation of rigid bodies remains a daunting, time-consuming task.

6.2 Future Improvements

The whole reason I undertook this project was to see an impressive fireball, so obviously the most important future improvement would be to get the particles working. The quickest way to do this would be to nail down the misfiring code. However, this would only get you a solution equal to the fireballs presented in the paper by Yngve et al. [2000]. These fireballs were better than mine were, but not nearly as impressive as those presented in Feldman, O'Brien and Arikan's paper "Animating Suspended Particle Explosions" [Feldman et al. 2003]. Their paper handles combustion using a simpler incompressible flow that is better suited for the fireball. Since most of the interesting flame effects occur well after the blast wave has propagated beyond visual range, the compressibility of Yngve's solution is a computational waste of time.

A volume rendered blast wave would be a significant improvement in the accuracy of the renders, though it would probably not significantly alter the final look of

the blast wave. The wave is just too fast and visually insignificant to spend the effort perfecting. Similarly, the rigid bodies could be more accurately simulated with a more complicated approach but that would probably slow down the simulation for a result that could just as easily be cheated, or even hand animated. Dust clouds could also be added to the simulation as Yngve did in “Animating Explosions”, but again they could also just be added in later in the form of timed smoke emitters.

REFERENCES

Baker, W. E. 1973. *Explosions in Air.* University of Texas Press, Austin.

Bourke, P. 2003. *Polygonising a Scalar Field*, http://astronomy.swin.edu.au/~pbourke/modelling/polygonise.

Columbia Pictures. 2001. *Final Fantasy: The Spirits Within* (film).

Dreamworks SKG. 1999. *Saving Private Ryan* (film).

Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual simulation of smoke. In *SIGGRAPH 2001 Conference Proceedings.* 15-22.

Feldman, B., O'Brien, J., and Arikan, O. 2003. Animating suspended particle explosions. In *SIGGRAPH 2003 Conference Proceedings*. 708-715.

Fordham, J. 2002. Spin city. *Cinefex 90*, 14-54,123-130.

Foster, N., and Fedkiw, R. 2001. Practical animation of liquids. In *SIGGRAPH 2001 Conference Proceedings*. 23-30.

Foster, N., and Metaxas, D. 1996. Realistic animation of liquids. *Graphical Models and Image Processing 58*, (5), 471-483.

Foster, N., AND Metaxas, D. 1997. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH 97 Conference Proceedings*. 181-188.

Kass, M., AND Miller, G. 1990. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH 90 Conference Proceedings.* 49-57.

Lorensen, W., and Cline, H. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH 87 Conference Proceedings.* 163-169.

Mader, C. L. 1998. *Numerical Modeling of Explosives and Propellants.* CRC Press, Boca Raton.

Magid, R. 1998a. *Blood on the Beach*, http://www.theasc.com/magazine/dec98/Blood/index.htm.

Magid, R. 1998b. *Paranormal Activities*, http://www.theasc.com/protect/jul98/paranormal/index.htm.

Mazarak, O., Martins C., and Amanatides, J. 1999. Animating exploding objects. *Graphics Interface 99,* 211-218.

Nave, C. 2003. *Blackbody Radiation*, http://hyperphysics.phy-astr.gsu.edu/hbase/bbcon.html.

Neff, M., and Fiume, E. 1999. A visual model for blast waves and fracture. *Graphics Interface 99,*193-202.

Paramount Pictures. 1982. *Star Trek II: The Wrath of Khan* (film).

Paramount Pictures. 1984. *Star Trek III: The Search for Spock* (film).

Reeves, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics 2,* (2), 91-108.

Shoemake, K. 2003. *Euler Angle Conversion*, http://www1.acm.org/pubs/tog/GraphicGems/gemsiv/euler_angle.

Sims, K. 1990. Particle animation and rendering using data parallel computation. In *SIGGRAPH 90 Conference Proceedings.* 405-413.

Sony Pictures. 2002. *Spider-Man* (film).

Stam, J. 1999. Stable fluids. In *SIGGRAPH 99 Conference Proceedings.* 121-128.

Twentieth Century Fox. 1977. *Star Wars: Episode IV - A New Hope* (film).

Twentieth Century Fox. 1983. *Star Wars: Episode VI - Return of the Jedi* (film).

Twentieth Century Fox. 1996. *Independence Day* (film).

Twentieth Century Fox. 1998. *The X-Files* (film).

Twentieth Century Fox. 2002. *Star Wars: Episode II - Attack of the Clones* (film).

Warner Brothers. 2001. *Swordfish* (film).

Wejchert, J., and Haumann, D. 1991. Animation aerodynamics. In *SIGGRAPH 91 Conference Proceedings.* 19-22.

Witkin, A., and Baraff, D. 1999. Physically based modeling. In *SIGGRAPH 99* Course Notes.

Yngve, G., O'Brien, J., and Hodgins, J. 2000. Animating explosions. In *SIGGRAPH 2000 Conference Proceedings.* 29-36.

Supplemental Sources Consulted

Kundert-Gibbs, J., and Lee, P. 2001. *Mastering Maya 3*, Sybex, San Francisco.

Mao, W. 1999. Producing a computer generated explosive effect. M.S. thesis, Texas A&M University, College Station.

Naithani, P. 2002. Visually simulating realistic fluid motion. M.S. thesis, Texas A&M University, College Station.

APPENDIX A

Here are the major MEL scripts used to interface the explosion simulator with Alias|Wavefront's Maya 4.5.

ExploSimSetup.mel

```
string $FileNode = `group -empty -n Filenames`;
addAttr -dt "string" -ln outputDirectory $FileNode;
addAttr -dt "string" -ln gridFilePrefix $FileNode;
addAttr -dt "string" -ln massFile $FileNode;
addAttr -dt "string" -ln explosionFile $FileNode;
addAttr -dt "string" -ln inputDirectory $FileNode;
addAttr -dt "string" -ln inputPrefix $FileNode;
string $File = `file -q -sceneName`;
$File = match( "[^/\\]*$", $File );
int $sz = size($File);
if ($sz > 1) $File = substring($File,1,($sz - 3));
setAttr ($FileNode + ".outputDirectory") -type "string" (`workspace -q -rd`);
setAttr ($FileNode + ".gridFilePrefix") -type "string" $File;
setAttr ($FileNode + ".massFile") -type "string" ($File + ".mss");
setAttr ($FileNode + ".explosionFile") -type "string" ($File + ".xpl");
setAttr ($FileNode + ".inputDirectory") -type "string" (`workspace -q -rd`);
setAttr ($FileNode + ".inputPrefix") -type "string" "simOut";
setAttr -e -keyable false ($FileNode + ".tx");
setAttr -e -keyable false ($FileNode + ".ty");
setAttr -e -keyable false ($FileNode + ".tz");
setAttr -e -keyable false ($FileNode + ".rx");
setAttr -e -keyable false ($FileNode + ".ry");
setAttr -e -keyable false ($FileNode + ".rz");
setAttr -e -keyable false ($FileNode + ".sx");
setAttr -e -keyable false ($FileNode + ".sy");
setAttr -e -keyable false ($FileNode + ".sz");
setAttr -e -keyable false ($FileNode + ".visibility");
string $gridObj = `createNode nurbsCurve`;
setAttr -k off ($gridObj + ".v");
setAttr ($gridObj + ".cc") -type "nurbsCurve" 1 15 0 no 3 16 0 1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 0.5 0.5 -0.5 -0.5 0.5 -0.5 -0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 -0.5
0.5 -0.5 -0.5 0.5 -0.5 0.5 0.5 0.5 0.5 -0.5 0.5 0.5 -0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5
-0.5 0.5 -0.5 -0.5 -0.5 -0.5 0.5 -0.5 -0.5 -0.5 -0.5 0.5 -0.5 -0.5;
string $parents[] = `listRelatives -fullPath -parent $gridObj`;
$gridObj = `rename $parents[0] ExplosionGrid`;
xform -ws -piv -0.5 -0.5 -0.5 $gridObj;
move 0.5 0.5 0.5 $gridObj;
makeIdentity -apply true -t 1 $gridObj;
setAttr -e -keyable false ($gridObj + ".rx");
setAttr -e -keyable false ($gridObj + ".ry");
setAttr -e -keyable false ($gridObj + ".rz");
addAttr -ln xth -at long  $gridObj;
setAttr -e -keyable true ($gridObj + ".xth");
setAttr ($gridObj + ".xth") 10;
```

PrepareObjects.mel

```
string $testSelected[] = `ls -sl -s -dep`;
string $each;
for($each in $testSelected)
{
	if ( "transform" != `nodeType $each` )
	{
		string $parents[] = `listRelatives -fullPath -parent $each`;
		$each = $parents[0];
	}
	if (!(`attributeExists "objectID" $each`))
	{
		string $shapes[] = `listRelatives -fullPath -shapes $each`;
		addAttr -ln objectID -at long  $each;
		setAttr -e -keyable true ($each + ".objectID");
		addAttr -ln objID -at long $shapes[0];
		connectAttr ($each + ".objectID") ($shapes[0] + ".objID");
		addAttr -ln objectMass -at double  $each;
		setAttr -e -keyable true ($each + ".objectMass");
		addAttr -ln initialVelocity -at double3  $each;
		addAttr -ln initialVelocityX -at double -p initialVelocity  $each;
		addAttr -ln initialVelocityY -at double -p initialVelocity  $each;
		addAttr -ln initialVelocityZ -at double -p initialVelocity  $each;
		setAttr -type double3 ($each + ".initialVelocity") 0 0 0;
		setAttr -e -keyable true ($each + ".initialVelocity");
		setAttr -e -keyable true ($each + ".initialVelocityX");
		setAttr -e -keyable true ($each + ".initialVelocityY");
		setAttr -e -keyable true ($each + ".initialVelocityZ");
	}
}
```

CreateExplosion.mel

```
string $newX[] = `sphere -p 0 0 0 -ax 0 1 0 -r 1 -d 3 -ut 0 -s 24 -nsp 12 -ch 0 -n
"Explosion"`;
int $number = 0;
if ( `gmatch $newX[0] "*[0-9]"` )
{
  $number = `match "[0-9]+$" $newX[0]`;
}
addAttr -ln explosionID -at long  $newX[0];
setAttr -e -keyable true ($newX[0] + ".explosionID");
setAttr ($newX[0] + ".explosionID") $number;
addAttr -ln size -at double  $newX[0];
setAttr -e -keyable true ($newX[0] + ".size");
connectAttr -f ($newX[0] + ".size") ($newX[0] + ".scaleX");
connectAttr -f ($newX[0] + ".size") ($newX[0] + ".scaleY");
connectAttr -f ($newX[0] + ".size") ($newX[0] + ".scaleZ");
setAttr ($newX[0] + ".size") 1;
addAttr -ln pressure -at double  $newX[0];
setAttr -e -keyable true ($newX[0] + ".pressure");
setAttr ($newX[0] + ".pressure") 101305000;
addAttr -ln pressureRange -at double  $newX[0];
setAttr -e -keyable true ($newX[0] + ".pressureRange");
addAttr -ln temperature -at double  $newX[0];
setAttr -e -keyable true ($newX[0] + ".temperature");
setAttr ($newX[0] + ".temperature") 2900;
addAttr -ln timeOffset -at double  $newX[0];
```

```
setAttr -e -keyable true ($newX[0] + ".timeOffset");
setAttr -e -keyable false ($newX[0] + ".rotateX");
setAttr -e -keyable false ($newX[0] + ".rotateY");
setAttr -e -keyable false ($newX[0] + ".rotateZ");
```

Output2ExploSim.mel

```
if (`objExists Filenames`)
{
	string $outputFilename = ((`getAttr Filenames.outputDirectory`)+(`getAttr
Filenames.massFile`));
	float $masses[], $Vx[], $Vy[], $Vz[];
	$masses[0] = 0;
	string $Transforms[] = `ls -dep`;
	string $each;
	for ($each in $Transforms)
		if (`attributeExists "objectID" $each`)
		{
			int $ID = `getAttr ($each + ".objectID")`;
			$masses[$ID] = `getAttr ($each + ".objectMass")`;
			$Vx[$ID] = `getAttr ($each + ".initialVelocityX")`;
			$Vy[$ID] = `getAttr ($each + ".initialVelocityY")`;
			$Vz[$ID] = `getAttr ($each + ".initialVelocityZ")`;
		}
	int $massFileID = fopen($outputFilename,"w");
	int $iter;
	for($iter=0;$iter<size($masses);$iter++)
		fprint $massFileID ($masses[$iter]+" "+$Vx[$iter]+" "+$Vy[$iter]+"
"+$Vz[$iter]+"\n");
	fclose($massFileID);
	print ("Wrote file " + $outputFilename);

	$outputFilename = ((`getAttr Filenames.outputDirectory`)+(`getAttr
Filenames.explosionFile`));
	float $Px[], $Py[], $Pz[], $Pr[], $PR[], $T[], $Size[], $Off[];
	float $offsetX, $offsetY, $offsetZ;
	$offsetX = (`getAttr ExplosionGrid.tx`) + 0.5;
	$offsetY = (`getAttr ExplosionGrid.ty`) + 0.5;
	$offsetZ = (`getAttr ExplosionGrid.tz`) + 0.5;
	string $Transforms[] = `ls -dep`;
	string $each;
	for ($each in $Transforms)
		if (`attributeExists "explosionID" $each`)
		{
			int $ID = `getAttr ($each + ".explosionID")`;
			$Px[$ID] = `getAttr ($each + ".translateX")`;
			$Py[$ID] = `getAttr ($each + ".translateY")`;
			$Pz[$ID] = `getAttr ($each + ".translateZ")`;
			$Pr[$ID] = `getAttr ($each + ".pressure")`;
			$PR[$ID] = `getAttr ($each + ".pressureRange")`;
			$T[$ID] = `getAttr ($each + ".temperature")`;
			$Size[$ID] = `getAttr ($each + ".size")`;
			$Off[$ID] = `getAttr ($each + ".timeOffset")`;
		}
	int $exploFileID = fopen($outputFilename,"w");
	int $iter;
	for($iter=0;$iter<size($Px);$iter++)
		fprint $exploFileID (($Px[$iter]-$offsetX)+" "+($Py[$iter]-
$offsetY)+" "+($Pz[$iter]-$offsetZ)+" "+$Pr[$iter]+" "+$PR[$iter]+" "+$T[$iter]+"
"+$Size[$iter]+" "+$Off[$iter]+"\n");
```

```
        fclose($exploFileID);
        print ("Wrote file " + $outputFilename);

        float $orthoWidth = `getAttr ExplosionGrid.xth`;
        string $orthoCam[] = `camera -orthographic 1 -orthographicWidth $orthoWidth
-n GridCam`;
        move -a ((`getAttr ExplosionGrid.translateX`) + ((`getAttr
ExplosionGrid.xth`)/2.0)) ((`getAttr ExplosionGrid.translateY`) + (`getAttr
ExplosionGrid.yth`)) ((`getAttr ExplosionGrid.translateZ`) + ((`getAttr
ExplosionGrid.zth`)/2.0)) $orthoCam[0];
        rotate -a -90 0 0 $orthoCam[0];
        setKeyframe -at "nearClipPlane" -v 0.5 -t ((`getAttr ExplosionGrid.yth`)-1)
$orthoCam[1];
        setKeyframe -at "nearClipPlane" -v ((`getAttr ExplosionGrid.yth`)-0.5) -t 0
$orthoCam[1];
        mtor control setvalue -rg dspyName -value (`getAttr
Filenames.gridFilePrefix`);
        mtor control setvalue -rg camName -value $orthoCam[1];
        mtor control setvalue -rg dspyRez -value ((`getAttr ExplosionGrid.xth`) + "
" + (`getAttr ExplosionGrid.zth`));
        mtor control setvalue -rg pixelSamples -value "1 1";
        mtor control setvalue -rg filterWidth -value "1 1";
        mtor control setvalue -rg jitter -value 0;
        mtor control setvalue -rg doAnim -value 1;
        mtor control setvalue -rg computeStart -value 0;
        mtor control setvalue -rg computeStop -value ((`getAttr ExplosionGrid.yth`)-
1);
        mtor control renderspool;


}
else
{
        error "Initialize scene as an Explosion first.";
}
```

ImportBlastWave.mel

```
if (`objExists Filenames`)
{
        int $start = `playbackOptions -q -min`;
        int $end = `playbackOptions -q -max`;
        int $timeOffset = 0;
        string $filePrefix = ((`getAttr Filenames.inputDirectory`)+(`getAttr
Filenames.inputPrefix`)+".");
        string $fileNum;
        int $frame = $start;
        string $bwGroups[];
        while($frame <= $end)
        {
                $fileNum = $frame;
                while (size($fileNum) < 4) $fileNum = "0" + $fileNum;
                if ((`file -q -ex ($filePrefix + $fileNum + ".obj")`)==1)
                {
                        file -r -type "OBJ" -rpr ("bw"+$frame) -options "mo=0"
($filePrefix + $fileNum + ".obj");
                        file -sa ($filePrefix + $fileNum + ".obj");
                        $bwGroups[($frame-$start)] = `group -n ("bw"+$frame)`;
                        file -ir ($filePrefix + $fileNum + ".obj");
                        $frame++;
                }
                else
```

```
                {
                        warning ("Stopped at "+$frame);
                        $frame = $end+1;
                }
        }

        string $bwParentGrp = `group -em -n "BlastWave"`;
        int $grpIter;
        for($grpIter = 0; $grpIter <= size($bwGroups); $grpIter++)
                {
                int $kTime = $timeOffset + $grpIter;
                setKeyframe -attribute "visibility" -v 0 -t ($kTime - 1) -t ($kTime +
1) ($bwGroups[($grpIter)]);
                setKeyframe -attribute "visibility" -v 1 -t ($kTime)
($bwGroups[($grpIter)]);
                parent ($bwGroups[($grpIter)]) $bwParentGrp;
                }
}
else
{
        error "Initialize scene as an Explosion first.";
}
```

ImportAnimation.mel

```
if (`objExists Filenames`)
{
        string $selected[] = `ls -sl -dep`;
        for ($tnode in $selected)
        if (`attributeExists "objectID" $tnode`)
        {
        string $filename = ((`getAttr Filenames.inputDirectory`) + (`getAttr
Filenames.inputPrefix`) + ".");
        string $obj = `getAttr ($tnode + ".objectID")`;
        while (size($obj)<4) $obj = ("0"+$obj);
        $filename = ($filename + $obj + ".rbs");
        int $FileID = fopen($filename,"r");
        if ($FileID == 0)
                error ($filename + " Not Found");
        float $startTime = `currentTime -q`;

        float $temp[6], $offset[6];

        $offset[0] = `getAttr -t $startTime ($tnode + ".tx")`;
        $offset[1] = `getAttr -t $startTime ($tnode + ".ty")`;
        $offset[2] = `getAttr -t $startTime ($tnode + ".tz")`;
        $offset[3] = `getAttr -t $startTime ($tnode + ".rx")`;
        $offset[4] = `getAttr -t $startTime ($tnode + ".ry")`;
        $offset[5] = `getAttr -t $startTime ($tnode + ".rz")`;
        int $i;
        for($i=0;$i<3;$i++)
        {
                $temp[($i)] = `fgetword $FileID`;
                $offset[($i)] -= $temp[($i)];
        }
        for($i=3;$i<6;$i++)
        {
                $temp[($i)] = `fgetword $FileID`;
                $temp[($i)] = $temp[($i)] * (180/3.1415);
                $offset[($i)] -= $temp[($i)];
        }
```

```
        int $frame = $startTime;
        while (!(`feof $FileID`))
         {
         setKeyframe -t $frame -at translateX -v ($temp[0]+$offset[0]) $tnode;
         setKeyframe -t $frame -at translateY -v ($temp[1]+$offset[1]) $tnode;
         setKeyframe -t $frame -at translateZ -v ($temp[2]+$offset[2]) $tnode;
         setKeyframe -t $frame -at rotateX -v ($temp[3]+$offset[3]) $tnode;
         setKeyframe -t $frame -at rotateY -v ($temp[4]+$offset[4]) $tnode;
         setKeyframe -t $frame -at rotateZ -v ($temp[5]+$offset[5]) $tnode;
         for($i=0;$i<3;$i++) $temp[($i)] = `fgetword $FileID`;
         for($i=3;$i<6;$i++){ $temp[($i)] = `fgetword $FileID`;$temp[($i)] =
$temp[($i)] * (180/3.1415);}
         $frame++;
         }

        fclose $FileID;
        }
}
else
{
        error "Initialize scene as an Explosion first.";
}
```

VITA

Name	Matthew Douglas Roach
Education	Texas A & M University M.S. in Visualization Sciences March 2005
	Southern Methodist University B.S. in Mathematics B.S. in Computer Engineering May 2000
Address	5055 Addison Circle #717 Addison, TX, 75093