Visual Simulation of Shockwaves

Jason Sewall*,[a], Nico Galoppo[b], Georgi Tsankov[a], Ming Lin[a]

[a] *University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175 USA*
[b] *Intel Corporation, Jones Farm 2, 2111 N.E. 25th Avenue, Hillsboro, OR 97124-5961 USA*

Abstract

We present an efficient method for visual simulations of shock phenomena in compressible, inviscid fluids. Our algorithm is derived from one class of the finite volume method especially designed for capturing shock propagation, but offers improved efficiency through physically-based simplification and adaptation for graphical rendering. Our technique is capable of handling complex, bidirectional object-shock interactions stably and robustly. We describe its applications to various visual effects, including explosion, sonic booms and turbulent flows. Furthermore, we explore parallelization schemes and demonstrate the scalability of our method on shared-memory, multi-core architectures.

Key words: physically-based modeling, animation, fluid simulation, parallel algorithms, shockwaves

1. Introduction

Recent developments in simulating natural phenomena have made it possible to incorporate stunning, realistic animations of complex natural scenes filled with flowing, bubbling, and burning fluids. Computer-animated and live-action films alike have made great use of these advances in modeling to recreate familiar and interesting effects. Notably, little investigation has been made on how to properly capture shocks and propagate discontinuities in visual simulation. These remarkable phenomena give rise to dramatic events such as explosions, turbulent flows, and sonic booms. Such effects are common in films and are notoriously difficult to handle with numerical methods. Additionally, many state-of-the-art simulation techniques do not fully take advantage of the kind of new, powerful hardware that is emerging; these algorithms are often not designed to handle large domains efficiently and many that are based on specially simplified formulations often are not applicable to phenomena occurring at large spatial scales.

This paper presents a method for efficient simulations of nonlinear, compressible gas dynamics and describes how it may be best utilized to generate visually interesting, plausible animations. Many natural phenomena are nonlinear but can often be reasonably approximated through linearization; one example is the linear formulation of elasticity that is commonly used in graphics-targeted simulation. The equations of fluid motion are not generally suitable for linearization — waves crashing on the beach, curling smoke, and surging shockwaves all arise from the nonlinear characteristics of the system. To solve these highly nonlinear equations in a reasonable amount of time, numerical methods typically discretize simplifications of the true equations that still capture the nonlinearity of the system.

Furthermore, shocks that arise in problems of gas dynamics themselves present a numerical challenge; a shock is a region of rapid spatial variation in a small interval that propagates with tremendous speed — the blast wave that emanates from an explosion or the bow shock that forms around a supersonic projectile are some examples of these phenomena. These have a striking effect on the fluid motion but are very difficult to simulate properly with traditional numerical methods; the scale of motion we desire to capture (namely the space the shock traverses) is at odds with the need to represent the shock itself. Many numerical techniques behave poorly or fail completely in the presence of discontinuous solutions — to simulate shocks with such methods, the resolution of the discretization must be high enough for the shock to appear as a smooth transition, and thus can be prohibitively expensive to compute.

Physically correct methods for shockwave modeling focus less on conventional metrics of accuracy (such as order of convergence) and emphasize the ability to propagate discontinuities stably and with minimal diffusion. Specifically, techniques based on the finite volume method (FVM) have been developed that handle discontinuities well and allow for relatively coarse grids to capture shock behavior. Our method, through judicious simplification and application, adapts and improves the efficiency of a class of FVM techniques designed to capture shocks on coarse grids efficiently.

We have demonstrated our method on generating animations of complex fluid motion, including chambered explosions, nuclear detonations, and the turbulence and bow shock around a supersonic projectile (see Figs. 5, 6, 7).

*Corresponding author
Email addresses: sewall@cs.unc.edu (Jason Sewall), nico.galoppo@intel.com (Nico Galoppo), gtsankov@cs.unc.edu (Georgi Tsankov), lin@cs.unc.edu (Ming Lin)

Our method is also able to describe the interaction of coupled fluids and objects; we demonstrate shockwaves knocking over stacked objects and blowing a brick house to pieces, as well as the effects of an explosion within a tower of heavy blocks. Our method is able to considerably reduce the computational complexity of these highly complex effects to the level comparable to existing fluid animation techniques in graphical simulation.

Furthermore, while a naïve parallelization of our method achieves only mediocre scalability, it is possible to achieve much better scaling with a more carefully constructed parallelization scheme. We explore the components of such a near-optimal scheme and its application to the shared-memory, multi-core architectures that are becoming commonplace. The essential locality of the numerical schemes used in our method allows us to achieve parallel performance far greater than that typical of methods for fluid simulation in graphics.

2. Previous Work

In addition to many decades of research in computational fluid dynamics, there is a considerable amount of literature on the modeling of fluid phenomena in computer graphics [1, 2, 3, 4, 5]. The seminal works of Foster and Metaxas [6], Stam [1], and Foster and Fedkiw [7] on incompressible fluid simulation were among the first to examine this topic for visual simulation. For a detailed explanation of the numerical methods and the mathematics behind them, we refer the readers to the recent book by Bridson [8].

Recent work on fluid simulation based on Finite Volume Methods has been discretized on irregular grids [9, 10, 11]. Subsequent improvements [12, 13, 4] to these methods have combined the best features of the initial publications to achieve impressive results.

The finite volume method has received much attention from the aeronautics community; our technique uses numerical Riemann solvers based on the work done by Roe [14], van Leer [15], and others. For a superb introduction to the topic of the finite volume method and Riemann solvers, see [16].

The problem of describing the evolution of shocks — known as "shock capturing" — has been addressed from a variety of directions. Our work follows the vein of *Riemann-solver based* approaches that strive to treat areas with and without shocks with the same numerical technique. Another family of approaches, generally known as *front-tracking* methods, uses standard solvers in areas away from shocks and explicitly models shocks as evolving surfaces in the domain. Front-tracking approaches have been successful, but are extremely complicated for two- and three-dimensional simulations and have difficulty handling situations where multiple shocks interact. The survey of Fedkiw et al. [17] gives a good overview of the topic.

Relatively little work in computer graphics has utilized the Euler equations — that is, the compressible, inviscid simplification of the Navier-Stokes system of equations — all of the aforementioned methods from the graphics community are based on an incompressible simplification of the equations. Yngve et al. [18] present a method for high-energy, compressible fluid simulation based on finite differences, which they use to simulate explosions and their secondary effects. Sewall et al. [19] use a method similar to finite volume on irregular grids to simulate compressible flow.

Feldman et al. [20] simulate combustive phenomena based on an incompressible model of flow with additional density tracers, and Selle et al. [21] present an approach that generates what they describe as "rolling explosions". Like Feldman et al., they use an incompressible model of fluid, which precludes the presence of shocks. Our approach aims to model phenomena similar to those addressed by Yngve et al. [18]. The greater fidelity and higher efficiency afforded by our method opens up a wide range of new applications of these phenomena to visual effects.

Work done by Müller et al. [22] and Adams et al. [3] on particle-based fluid with the Smooth Particle Hydrodynamics (SPH) method strives to represent incompressible flow. The natural tendency of the space between particles to expand and collapse suggests potential future application to compressible phenomena.

Several methods have addressed the considerable challenge of coupling fluids to objects. Geneveaux et al. [23] suggest an explicit method for the bidirectional coupling of grid-based fluids to solid bodies using a particle representation of the surface. Carlson et al. [24] use distributed Lagrange multipliers to achieve stable fluid-object coupling and Guendelman et al. [25] describe how to handle the interaction of infinitely thin shells with fluids. Chentanez et al. [13] use an implicitly-coupled model of fluid and elastic bodies to obtain stable interactions. Batty et al. [5] use variational principles to develop a simple extension to the pressure projection step to achieve stable two-way coupling in incompressible fluids. We achieve bidirectional coupling through voxelization (as in [24] and [5]). We use simple modifications to the Riemann solvers on boundary interfaces to affect the interaction.

There has been some work on simulating the effects of blast waves through analytical models of blast propagation. Mazarak et al. [26] used an expanding ball to determine forces on bodies to fracture or propel them. Neff and Fiume [27] use similar analytic models of blast waves to fracture objects and are unable to generate the aforementioned effects of shock dynamics. These approaches are typically quite fast, but their extremely simple model of blast dynamics does not allow for the effects of shock-object interaction — notably reflection and vortex shedding — nor do they have the ability to visualize the blast itself.

3. Method

The key challenge is to simulate shockwave and compressible gas dynamics by designing a practical numerical method that can stably handle moving boundary conditions in three-dimensional space and is efficient enough to be used in a visual simulation production pipeline. We present a basic introduction to the finite volume method and refer the readers to [16] for more detail.

3.1. Conservation laws

We seek solutions to the Euler equations of gas dynamics. These equations form a *hyperbolic conservation law*, the general, three dimensional form of which is:

$$\mathbf{q}_t + \mathbf{F}(\mathbf{q})_x + \mathbf{G}(\mathbf{q})_y + \mathbf{H}(\mathbf{q})_z = 0 \tag{1}$$

where subscripts indicate partial differentiation.

Here $\mathbf{q}$ is the vector of unknowns and $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{H}$ are vector-valued *flux functions* specific to each conservation law. A conservation law states that a quantity of unknowns $\mathbf{q}$ over an arbitrary domain S changes in time due only to the flux across the boundary ∂S of the domain.

3.1.1. Integral form

The derivatives found in partial differential equations such as Eq. (1) are not defined around discontinuities; to capture them properly we use an integral form of the equations.

In one dimension, consider an interval $[a, b]$; the change in $\mathbf{q}$ over that interval is due to the flux at a and the flux at b. More formally,

$$\frac{d}{dt}\int_a^b \mathbf{q}(x,t)\,dx = \mathbf{F}(\mathbf{q}(a,t)) - \mathbf{F}(\mathbf{q}(b,t)) \tag{2}$$

where $\mathbf{F}$ is a flux function such as $\mathbf{F}$, $\mathbf{G}$, or $\mathbf{H}$ from Eq. (1) and we follow the convention that 'positive flux' is left-going and 'negative flux' right-going.

3.2. The finite volume method

The finite volume method (FVM) on regular grids follows directly from Eq. (2); the presentation here is for scalar equations in one dimension with scalar unknowns q and scalar fluxes f, but the formulae for systems of equations in multiple dimensions are straightforward extensions of these.

We discretize the spatial interval $[a, b]$ into m intervals ("cells") of equal size $\Delta x = \frac{b-a}{m}$. For each time t_n, we have m quantities Q_i^n defined as the average value of q over the cell:

$$Q_i^n = \frac{1}{\Delta x}\int_{\chi_i^l}^{\chi_i^r} q\,(x, t_n)\;dx \tag{3}$$

where we have $\chi_i^l = a + i\Delta x$, and $\chi_i^r = \chi_i^l + \Delta x$ as the positions of the cell boundaries. Observe that $\chi_{i-1}^r = \chi_i^l$. We will also occasionally refer to these boundary positions with half-index increments; for example, the flux at χ_i^l is $F_{i-\frac{1}{2}}$.

We apply Eq. (2) to each of the intervals i

$$\frac{d}{dt}\int_{\chi_i^l}^{\chi_i^r} q(x,t)\,dx = f\Big(q\big(\chi_i^l,t\big)\Big) - f\Big(q\big(\chi_i^r,t\big)\Big) \tag{4}$$

and integrate Eq. (4) from t_n to t_{n+1}

$$\int_{\chi_i^l}^{\chi_i^r} q\,(x,t_{n+1})\;dx - \int_{\chi_i^l}^{\chi_i^r} q\,(x,t_n)\;dx = \int_{t_n}^{t_{n+1}} f\Big(q\big(\chi_i^l,t\big)\Big) - f\Big(q\big(\chi_i^r,t\big)\Big)\,dt \tag{5}$$

Observe that we can substitute Eq. (3) if we divide Eq. (5) by Δx.

$$Q_i^{n+1} - Q_i^n = \frac{1}{\Delta x}\int_{t_n}^{t_{n+1}} f\Big(q\big(\chi_i^l,t\big)\Big) - f\Big(q\big(\chi_i^r,t\big)\Big)\,dt \tag{6}$$

The right-hand side of this equation is a *flux difference* that cannot generally be evaluated exactly; we approximate the integrals with averages over the cell interfaces from $[t_n, t_{n+1}]$:

$$F_{\chi_i^l}^n \approx \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f\Big(q\big(\chi_i^l,t\big)\Big)\,dt \tag{7}$$

Substituting Eq. (7) into Eq. (6), we obtain the most basic FVM update scheme (Eq. (8)):

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{\chi_i^l}^n - F_{\chi_i^r}^n\right) \tag{8}$$

This scheme is first-order and is subject to the numerical viscosity typical of first-order methods. Second-order schemes such as the Law-Wendroff scheme [28] can be employed with comparable computation effort; we complement this with a *flux limiter*, which minimizes diffusive and dispersive artifacts. We have used the MC limiter of van Leer [15] in our method.

3.3. The Riemann problem

According to Eq. (7), the flux at $\mathbf{F}_{i-\frac{1}{2}}$ is dependent on the state $\mathbf{Q}_{i-\frac{1}{2}}$ at the interface between $\mathbf{Q}_{i-1}$ and $\mathbf{Q}_i$ over the interval $(t_n,\, t_{n+1})$; thus we must determine the value at this interface as it evolves in time. $\mathbf{Q}_i$ here is the vector version of the discrete unknowns first introduced in 3.2.

Given the initial data:

$$\mathbf{Q}(x,0) = \begin{cases} \mathbf{Q}_l, & x < 0 \\ \mathbf{Q}_r, & x \geq 0 \end{cases} \tag{9}$$

we wish to solve for $\mathbf{Q}(x,t)$ for $t > 0$ subject to the governing equations. This formulation is known as the *Riemann problem*; the resulting $\mathbf{Q}(0,t)$ obtained can then be used to compute the flux at the cell interface.

Numerical methods based on Riemann solvers can often succeed where other methods fail because their solution is achieved through analysis of the governing equations. Whereas many techniques are developed through the mechanical application of numerical stencils to the terms of an equation, Riemann solvers inherently incorporate more information about the equation in their formulation.

3.3.1. Riemann problem for linear systems

Let us consider linear, constant-coefficient (but not necessary scalar) hyperbolic conservation laws, i.e. Eq. (1) where the flux function $\mathbf{F}$ takes the form $\mathbf{F}(\mathbf{q}) = \mathbf{A}\mathbf{q}$, where $\mathbf{A}$ is a *flux matrix*. (Assume that the other flux functions $\mathbf{G}$, and $\mathbf{H}$ are of the same form).

Such a system of order n can be diagonalized into n decoupled equations $\mathbf{Q}^+_t + \Lambda \mathbf{Q}^+_x = 0$, where $\mathbf{Q}^+ = \mathbf{R^{-1}Q}$. Here $\mathbf{R}$ is the matrix of right eigenvectors of $\mathbf{A}$, and Λ is a diagonal matrix of the eigenvalues of $\mathbf{A}$ satisfying $\mathbf{A} = \mathbf{R}\Lambda\mathbf{R}^{-1}$.

The solution to the Riemann problem for these equations is given by n weighted eigenvectors $W_i = \alpha_i \mathbf{r}_i$ (also known as *waves*) propagating with speeds λ_i, the corresponding eigenvalues.

The waves W_i are determined by projecting the jump in the initial states $\Delta\mathbf{Q} = \mathbf{Q_r} - \mathbf{Q_l}$ onto the space formed by the eigenvectors of the system:

$$\sum_i W_i = \sum_i \alpha_i \mathbf{r_i} = \Delta\mathbf{Q} \tag{10}$$

$$\mathbf{R}\mathbf{a} = \Delta\mathbf{Q} \tag{11}$$

$$\mathbf{a} = \mathbf{R}^{-1}\Delta\mathbf{Q} \tag{12}$$

where $\mathbf{a} = [\alpha_0, \alpha_1, \ldots, \alpha_{n-1}]^{\mathrm{T}}$

The waves define k intermediate states $\mathbf{Q}^{*_i} = \mathbf{Q}_l + \sum_{j=0}^{i} W_j$, and the solution to the Riemann problem is therefore the piecewise-constant function

$$\mathbf{Q}(x,t) = \begin{cases} \mathbf{Q}_l, & x < \lambda_1 t \\ \vdots & \vdots \\ \mathbf{Q}^{*_i}, & \lambda_i t < x \le \lambda_{i+1} t \\ \vdots & \vdots \\ \mathbf{Q}_r, & x \ge \lambda_n t \end{cases} \tag{13}$$

3.3.2. The Riemann problem for nonlinear systems

For nonlinear systems such as the Euler equations, the wave structure of the solution is much more complicated and costly to compute — typically, iterative root-finding methods must be employed at each cell interface to determine the intermediate states $\mathbf{Q}^{*_i}$.

However, it is often possible to obtain good results by approximately solving the Riemann problem; through linearizations of the flux evaluated at carefully chosen states, we can obtain solutions that fit Eq. (13). Such approximate Riemann solvers must be used with care, as they can often produce non-physical solutions. We discuss the applicability of these solvers and how these undesirable conditions can be addressed in Sec. 3.4.1.

3.3.3. Upwinding flux splitting

The basic FVM Eq. (8) update scheme developed in Sec. 3.2 is not able to stably handle hyperbolic systems; we need to modify it to obey the principle of *upwinding*. We must take care to ensure that waves traveling in the positive direction use information from the *negative* direction.

Rather than use Eq. (8) to compute cell updates, we employ a scheme

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F^{n+}_{\chi_i^r} + F^{n-}_{\chi_i^l}\right) \tag{14}$$

where $F^{n-}_{\chi_i^r}$ is the part of the flux $F^n_{\chi_i^r}$ traveling in the negative direction and $F^{n+}_{\chi_i^r}$ the part traveling in the positive direction.

The waves W_i and speeds λ_i from the solution to a Riemann problem at χ_i^r is then

$$F^{n-}_{\chi_i^r} = \sum_{c=0}^{j} \lambda_c W_c \quad F^{n+}_{\chi_i^r} = \sum_{c=j}^{k} \lambda_c W_c \tag{15}$$

Where $\ldots < \lambda_j < 0 < \lambda_{j+1} < \ldots$; waves traveling with negative speeds are added to F^{n-} while those traveling with positive speed are added to F^{n+}.

3.3.4. Solution procedure

Given cell values $\mathbf{Q}^n$ for time t_n, a timestep is performed as follows to compute $\mathbf{Q}^{n+1}$:

1. For each interface between cells, compute the wavespeeds λ_i and fluxes F_i^n by solving the Riemann problem at that interface (described in Sec. 3.4.2)
2. Find the wavespeed with largest magnitude from $|\lambda_i|$ to compute timestep length Δt as described in Sec. 3.4.4.
3. For each cell i, advance to next time $\mathbf{Q}^{n+1}$ using the fluxes F^n at its neighboring interfaces χ_i^l, χ_i^r using Eq. (14).

For three-dimensional problems (see Sec. 3.4.4), we must compute three fluxes for each cell in the domain; solving the Riemann problems in step 1 becomes the computational bottleneck for non-trivial systems of equations. While expensive to obtain, carefully calculated fluxes are the key to handling discontinuous solutions on a coarse grid. Next, we describe what the Riemann problem is and how it can be used to compute flux between cells.

3.4. The Euler equations

We are interested in studying the motion of a compressible gas; the natural choice is the Euler system of equations. The simplification of Navier-Stokes that omits

viscous terms results in this nonlinear hyperbolic system of conservation laws. The omission of viscosity is a reasonable one to make for many physical problems in gas dynamics, just as the incompressible simplification of Navier-Stokes frequently used in graphics is reasonable for liquid simulation.

The Euler equations in conservation form (see Eq. (1)) are

$$\mathbf{q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \quad \mathbf{F}(\mathbf{q}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ (E+p)u \end{bmatrix}$$
$$\mathbf{G}(\mathbf{q}) = \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ (E+p)v \end{bmatrix}, \quad \mathbf{H}(\mathbf{q}) = \begin{bmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ (E+p)w \end{bmatrix} \tag{16}$$

Here ρ is density, u, v, and w the components of velocity, p the pressure, and E the total energy. An additional *equation of state* completes the system

$$E = \frac{p}{\gamma - 1} + \frac{\rho}{2}\left(u^2 + v^2 + w^2\right) \tag{17}$$

where γ is the adiabatic exponent of the fluid — typically 1.4 for air. It should be noted that for solutions to be physically valid, ρ, p, and E must all be strictly greater than zero.

3.4.1. Approximate Riemann solutions

As discussed in Sec. 3.3.2, computing the exact solution to the Riemann problem for nonlinear systems such as the Euler equations is prohibitively expensive for practical problems. Suitably approximated solutions to the Riemann problem are often able to achieve acceptable results for a fraction of the cost of solving them exactly.

We would like to apply the method for solving Riemann problems for linear systems presented in Sec. 3.3.1 to nonlinear problems; to this end we desire a matrix $\mathbf{A}$ such that $\mathbf{A}$ approximates $\mathbf{F}'(\mathbf{Q})$; here $\mathbf{F}'(\mathbf{Q})$ is the Jacobian of $\mathbf{F}$ as seen in the *quasilinear* form of the conservation law. This is simply the chain rule applied to (1): $\mathbf{Q}_t - \mathbf{F}(\mathbf{Q})_x = \mathbf{Q}_t - \mathbf{F}'(\mathbf{Q})\mathbf{Q}_x = 0$.

In a seminal paper, Roe [14] presented a simple method for approximating $\mathbf{F}'(\mathbf{Q})$ that preserves important conditions of the system, and it is this method that we have adapted for our solver. Roe's method uses a flux matrix $\mathbf{A}$ that is $\mathbf{F}'(\bar{\mathbf{Q}})$ evaluated at a specially chosen state $\bar{\mathbf{Q}}$ given $\mathbf{Q}_l$ and $\mathbf{Q}_r$ — this state has come to be known as the *Roe average state*.

Eigenvectors and eigenvalues of the flux Jacobian. The eigenvectors of the Jacobian $\mathbf{F}'(\mathbf{Q})$ give the waves necessary to compute the intermediate states as in Sec. 3.3.1, and its eigenvalues give the characteristic speeds λ_i with which these waves propagate. The eigenvalues of the flux Jacobian $\mathbf{F}'$ as computed from (16) are:

$$\lambda_{0\ldots4} = (u - c, u, u, u, u + c) \tag{18}$$

and the corresponding eigenvectors are:

$$\mathbf{r}_1 = \begin{bmatrix} 1 \\ u - c \\ v \\ w \\ H - uc \end{bmatrix} \quad \mathbf{r}_2 = \begin{bmatrix} 1 \\ u \\ v \\ w \\ \frac{1}{2}(u^2 + v^2 + w^2) \end{bmatrix}$$
$$\mathbf{r}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ v \end{bmatrix} \quad \mathbf{r}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ w \end{bmatrix} \quad \mathbf{r}_4 = \begin{bmatrix} 1 \\ u + c \\ v \\ w \\ H + uc \end{bmatrix} \tag{19}$$

Here $c = \sqrt{\frac{\gamma p}{\rho}}$ is the speed of sound and $H = \frac{E+p}{\rho}$ the total specific enthalpy. We have given only the eigenvalues and eigenvectors for $\mathbf{F}'$, but those for the Jacobians of the other flux functions $\mathbf{G}'$ and $\mathbf{H}'$ have similar structure.

Roe average state. Given two states $\mathbf{Q}_l = [\rho_l, u_l, v_l, w_l, E_l]$ and $\mathbf{Q}_r = [\rho_r, u_r, v_r, w_r, E_r]$, the Roe average is

$$\bar{\mathbf{Q}} = \left[\bar{\rho}, \bar{u}, \bar{v}, \bar{w}, \bar{H}\right]^{\mathrm{T}} \qquad \bar{\rho} = \tfrac{\rho_l + \rho_r}{2} \tag{20}$$
$$\bar{u} = \frac{\sqrt{\rho}_l u_l + \sqrt{\rho}_r u_r}{\sqrt{\rho}_l + \sqrt{\rho}_r} \quad \bar{v} = \tfrac{\sqrt{\rho}_l v_l + \sqrt{\rho}_r v_r}{\sqrt{\rho}_l + \sqrt{\rho}_r} \tag{21}$$
$$\bar{w} = \frac{\sqrt{\rho}_l w_l + \sqrt{\rho}_r w_r}{\sqrt{\rho}_l + \sqrt{\rho}_r} \quad \bar{H} = \tfrac{\frac{E_l + p_l}{\sqrt{\rho}_l} + \frac{E_r + p_r}{\sqrt{\rho}_r}}{\sqrt{\rho}_l + \sqrt{\rho}_r} \tag{22}$$

The specific variables shown here (in contrast to the conservative variables given in (16)) appear because they are precisely what is necessary to evaluate the eigenvalues Eq. (18) and eigenvectors Eq. (19) and obtain the waves and speeds for a given Riemann problem.

This average state has attractive properties when considering the structure of the Riemann problem; were we to choose a simple arithmetic average of the quantities at $\mathbf{Q}_l$ and $\mathbf{Q}_r$, the resulting eigenvectors may fail to be distinct and the solution would fail entirely. The criteria behind this particular choice of average are explained in detail in [14] and [16].

Enforcing physicality. Using the Roe average state (Eq. (20)) to approximately solve the Riemann problem is significantly faster then computing the exact solution to the Riemann problem, but the solver is known to generate nonphysical states for certain inputs $\mathbf{Q}_l$ and $\mathbf{Q}_r$. While the exact solution to the Riemann problem could be computed to obtain the physically valid intermediate state, this is unnecessary and overly expensive for visual simulation. When the approximate Riemann solver produces invalid states, we apply slight corrections to enforce physicality. We clamp ρ and p to be no less than 0.05 — in the case of p, this entails adjusting E according to Eq. (17).

For example, the fluid-rigid body simulations illustrated in Figs. 6, 9, 1, and 2 demonstrate plausible motion, and would not be possible using a simple approximate Riemann solver without these corrections.

3.4.2. Riemann solver for Euler equations

We have developed the theory of Riemann solvers for the Euler equations sufficiently to present the procedure for computing the Riemann solution at an interface given left and right states $\mathbf{Q}_l$ and $\mathbf{Q}_r$:

1. Compute Roe average $\bar{\mathbf{Q}}$ using Eq. (20)
2. Make $\bar{\mathbf{Q}}$ physically valid if needed, as per Sec. 3.4.1
3. Compute wavespeeds λ_i using Eq. (18)
4. Compute eigenvectors $\mathbf{r}_i$ using Eq. (19)
5. Project $\Delta\mathbf{Q}$ onto the eigenspace by computing the wave coefficients α_i and waves W_i using Eq. (12)
6. Compute left and right fluctuations $F^{n\pm}$ using Eq. (15)

3.4.3. Boundary conditions

We apply boundary conditions where needed through modified Riemann solvers; these do not solve for the flux at an interface due to two adjacent cells; we compute a 'ghost' intermediate state at the interface to determine these fluxes. In practice, we have found three types of boundary conditions useful:

Free-slip: This common boundary condition simply states that the component of flow normal to the interface is zero. We obtain this by modifying the Roe average Eq. (20) used in the cell to have zero velocity in the component normal to the boundary; thus $\bar{u}$ on a free-slip boundary normal to the x-direction is set to zero. Other components of the intermediate state $\bar{\mathbf{Q}}$ are simply treated as though $\mathbf{Q_l}$ were equal to $\mathbf{Q_r}$.

Velocity: This is a generalization of free-slip boundary conditions; rather than enforce zero velocity along an interface, some user-specified velocity is imposed as the intermediate component of velocity in the appropriate direction. Other components are treated as though the two adjacent cells were identical except for the energy E; given an imposed velocity $\bar{u}$ and the same component of velocity in the adjacent cell u_r, the velocity in the ghost cell is $u_l = 2\bar{u}(\bar{u} - u_r)$. Due to this difference in velocity, the energy in the ghost cell is not equal to its neighbor and is adjusted with Eq. (17).

Absorbing: It is often desirable to perform simulations where outgoing waves are simply absorbed rather than reflected; the computational domain behaves as if it were suspended in an infinite passive medium. At these interfaces, the fluxes in the Riemann problem are simply set to zero.

3.4.4. Dimensional splitting

The discussion so far has been limited to one dimension — our Eqs. (16) are three-dimensional, but the solution procedure in Sec. 3.3.4 performs updates in only a single dimension.

To solve three-dimensional problems, we perform *dimensional splitting*. To advance from time t_n to t_{n+1}, we make sub-step "passes" of a one-dimensional solver in each direction — first using the flux function $\mathbf{F}$ along x for all rows of constant y and z, then using $\mathbf{G}$ along y for all rows of constant x and z, and finally using $\mathbf{H}$ along z for all rows of constant x and y.

This approach allows us to apply the one-dimensional techniques previously described here in a straightforward manner; however, we must address how best to choose the timestep to take over the three passes.

Choosing a timestep. The timestep size Δt that we are able to take while advancing the solution with Eq. (8) is limited by the maximum characteristic speed $\lambda_{\max}$ from Eq. (18) in the solution we are updating, as per the Courant-Friedrichs-Lewy (CFL) condition [29]. For simulation in a single dimension, the procedure in Sec. 3.3.4 works perfectly — we compute the solution to all Riemann problems in the domain, which gives us the maximum characteristic speed, which we use to compute the timestep $\Delta t = \frac{\Delta x}{\lambda_{\max}}$. With dimensional splitting, we are not able to compute the maximum speed in dimension y prior to advancing the solution in x with some previously chosen Δt; the maximum speed in y depends on the results of the x-pass and is not generally equal to the $\lambda_{\max}$ from the x pass.

There are several ways to address this problem — for example, we could adopt a guess-and-check approach of estimating a timestep, advancing the solution with it, checking to see if it satisfies the CFL condition based on the maximum speed of the next level, and rewinding the whole computation if not, but this would be prohibitively expensive.

We take the very simple approach of always advancing the solution in a dimension with the largest timestep that satisfies the CFL condition in that dimension. This method clearly has effects on the solution; effectively, the grid is 'warped' over a timestep based on the ratios of maximum speeds in each dimension. However, we have found these effects to be negligible in the simulations we have run, even in cases where the flow (and therefore $\lambda_{\max}$) is highly biased along a single dimension (see for example Figs. 6, 7, and 9).

Our approach has an advantage over other methods and is particularly desirable for visual simulation; each dimension is advanced according to the chosen CFL number of the simulation. No dimension is forced to take a timestep at a low CFL number because of other, higher speed dimensions. This technique helps reduce the numerical artifacts that frequently plague visual simulations of natural phenomena.

3.5. Fluid-object interaction

Figure 1: Tower (without cap) blown apart by internal blast

Figure 2: Tower (with cap) blown apart by internal blast

We employ a method for bi-directional fluid-object coupling that is simple, stable, and efficient. At each timestep, solid objects are voxelized onto the grid and cells occupied by solids marked as such.

To capture the objects' effect on the fluid, we use the aforementioned free-slip modification to the Riemann solver along the boundary (in Sec. 3.4.3). This solver ensures that incoming waves are reflected off of solid bodies and enables effects like those seen in Figs. 1, 2, 6, 8, and 9; these demonstrate the effects of the solids in the scene on the flow.

The force exerted by the fluid on the objects is obtained by multiplying the pressures at each incident cell by the interface's normal direction and applying the resulting force to the object. This simple technique is responsible for the forces buffeting the objects in Figs. 1, 2, 6, and 9.

Any rigid body simulator is suitable for use with our method; we have used the *Bullet* collision and dynamics engine [30] because of its completeness and availability. Our voxelization is a simple custom tool based on triangle-grid intersections.

Considerable work [24, 13, 5] has been done to achieve stable fluid-solid interactions in the past, but these methods have focused on the interaction of rigid and deformable bodies with *incompressible* fluids. Stability problems frequently arise in such situations because of the differing needs of the rigid body dynamics and the fluid simulator; the implicit solver for incompressible fluid simulation generally takes large timesteps, which can result in a loosely-coupled, unstable simulation when rigid bodies are handled naïvely. Our method naturally takes many small, inexpensive timesteps to advance the solution; this allows tighter communication between the rigid body and fluid simulators and results in a more stable interaction.

4. Parallelization

The vast majority of the computation time in the algorithm described in Sec. 3.3.4, is spent in two kernels; the computation of solutions to the many Riemann problems across the grid (as described in Sec. 3.4.2) and the application of these Riemann solutions to the cells of the grid to advance to the next timestep (see Eq. (14)). We execute each of these kernels once per dimensional pass (as described in Sec. (3.4.4.))

The computation of Riemann solutions is essentially independent across all cell interfaces along the current dimension. Given the two cells adjacent to a given interface, we compute the fluxes and speeds that comprise the Riemann solution at that interface.

The update procedure is similarly data-parallel across each grid cell; to update a cell, we need only the global timestep being used for this dimensional pass (computed as per Sec. 3.4.4) and the Riemann solutions corresponding to the two interfaces shared with the cell's neighbors along the current dimension.

We therefore expect to achieve significant performance scaling from a parallelization of these kernels across the grid. As we shall see, there are a great number of factors to take into consideration when developing an effective parallel computation scheme.

4.1. Naïve parallelization

The two computation kernels described above are parallel across each interface and grid cell, respectively, but the number of these generally exceeds the number of processors available by several orders of magnitude, so it is reasonable to assign groups of these computations to each processor.

An obvious way of doing this is to group 'rows' of computation — for both kernels, we consider the computation

performed on the interfaces or cells along each row of the current pass as a single task. These tasks may then themselves be partitioned among the available processors; Fig. 3 depicts this scheme.

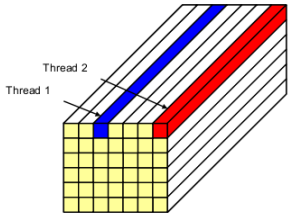


Figure 3: Computation of Riemann solutions and solution updates done in a pass are divided among threads

This scheme is exceedingly simple to implement atop an existing serial implementation but scales poorly, achieving about 8x scaling on 16 processors. Fig. 12 shows the scaling results for this scheme.

At first, the poor behavior of this approach may seem surprising, since it has many of the hallmarks of a good parallel algorithm. There is no explicit communication between threads, only synchronization barriers at the end of each kernel. The threads execute nearly identical code paths on equally-sized portions of the grid; we can therefore be confident that each thread performs a similar amount of work in each kernel. Additionally, the algorithm is compute-intensive — the Riemann solution kernel performs over 400 floating-point operations per interface, so concerns over bandwidth are mitigated.

The reason why this approach scales poorly becomes apparent when we consider the memory hierarchy of modern computer architecture and multi-core processor layout; after taking the memory layout into account, we can construct a superior parallelization scheme.

4.2. Hardware considerations

Modern processors — specifically, those found in commodity desktop and laptop computers — utilize a hierarchical memory layout, with several levels of cache between the processor and main memory. The latency of cache memory is typically an order of magnitude faster than that of main memory; multiple transactions with a given memory location can be greatly sped up if the contents of said memory can be kept in the cache.

Performance-minded implementations of important algorithms — such as the linear algebra operations found in the Automatically Tuned Linear Algebra Software (ATLAS) [31] package and the fast Fourier transforms found in the Fastest Fourier Transform in the West (FFTW) [32] library — are designed with CPU caches in mind, carefully blocking access patterns to maximize the effects of the cache's fast memory.

A serial implementation of our algorithm can be designed to traverse memory in a "cache-friendly" manner, but cannot be blocked in the same manner as some matrix operations. Ideally, the memory accesses during a timestep could be partitioned such that a portion of the grid is loaded into the cache and all operations necessary for that partition during the timestep would be performed before moving on to the next partition. The multi-pass nature of our algorithm requires that we traverse the grid multiple times — for each dimensional pass, once to compute the solutions to the Riemann problems and determine the maximum speed, and again to apply the updates to the grid.

However, an implementation on a parallel system will typically have much more cache available, albeit divided up among the various processors in the system. To effectively take advantage of the capabilities of a multi-core system it is therefore essential that we take the various caches available into account.

The cache structure found in multi-core computers is quite intricate and can vary greatly from system to system. For example, Intel's Pentium D processor featured two cores, each with a separate 1MB L2 cache, while Intel's Core 2 Duo processor's two cores share a single 4MB L2 cache. The system upon which we performed our parallelization benchmarks has four sockets, each with an Intel Xeon processor — these processors are in turn composed of four cores and two 2MB L2 caches, with each cache shared by two of the cores.

Let us consider how the naïve parallelization scheme presented in Sec. 4.1 behaves with a mind to memory access and cache behavior. For any single dimensional pass, each thread is assigned a disjoint portion of the grid to work with; assuming (for the moment) suitably aligned data, it is safe to say that no two threads will try to read or write the same location in memory.

However, each dimensional pass divides the grid up differently — for x, groups of rows of constant y and z, for y, groups rows of constant x and z, and for z groups of rows of constant y and z. This means that the portion of the grid assigned to each thread changes for each dimensional pass; in between each pass, all of the changes written by the pass's update kernel must be flushed out of all caches and exchanged. What initially looked like a moderately memory-intensive algorithm turns out to require tremendous amounts of bandwidth to satisfy the constantly-changing mapping of data to threads.

Cache lines and alignment. The above discussion of the effects of the system's caches on the naïve parallelization is itself simplified; to fully appreciate the subtleties of caches and how they affect a parallel program's performance, we must consider how caches are filled.

Caches always fetch and store contiguous groups of memory of fixed length in quanta known as *cache lines*. The size of a cache line varies with architecture; the Xeon processors in the 16-core machine we used for our parallelization benchmarks use 64-byte cache lines.

Every address in memory maps to exactly one cache

line and a transaction with an address will result in the cache line to which it belongs being read into the cache. This has the ramification that certain memory access patterns can be very inefficient; access with strides greater than a cache line, for example, can result in wasted bandwidth and cache use.

Parallel programs are subject to a more subtle issue due to cache lines: *false sharing*. Consider a situation in which a region of memory is partitioned among multiple threads, such as the partitioning of groups of rows in our naïve parallelization scheme. Depending on how the memory is partitioned and its alignment with respect to cache line boundaries, it is possible for multiple threads to be assigned the same cache line. If these threads are writing to these shared cache lines, a considerable amount of bandwidth and time is consumed as the cache line is read, written to and flushed by one thread after another.

False sharing can be prevented by ensuring that no two threads are reading and writing the same cache line; this is typically accomplished by ensuring that data structures are allocated along cache line boundaries and that shared portions of memory are padded appropriately.

Processor affinities. We have heretofore used the terms threads and processors interchangeably, assuming that each thread in the parallelization runs on a single processor for the entirety of its lifetime. In fact, operating systems are free to assign threads to any processor and *migrate* them to other processors during runtime.

Thread migration can result in a significant performance penalty for an algorithm designed to maximize the benefits of cache locality. Fortunately, most modern operating systems support the assignment of *affinity masks* to threads, which enumerate the set of processors the thread may execute on. Through this mechanism, we are able to request that each thread to run on a specific processor and therefore ameliorate the effects of thread migration.

Clearly, the naïve approach to parallelization described in Sec. 4.1 does not scale as well as we would like. Considering the effects of cache contention that occurs between dimensional passes and the likelihood of false sharing, the poor performance is understandable. With the limitations of this inferior scheme in mind, we will now describe a new parallelization scheme that performs much better.

4.3. Domain decomposition

Given a grid of dimension $l \times m \times n$ and a number of processors p, we split the grid into p rectilinear *tiles* using planes in x, y, and z. Note that we are presenting this algorithm in three dimensions, but is easily applied to two-dimensional problems.

The exact arrangement of the tiles depends on the factors of p, but we would like each tile to represent an equal amount of work along each dimension. For example, given a $64 \times 64 \times 64$ grid and 12 processors, we might decompose the problem into $2 \times 3 \times 2$ tiles, with 8 tiles of dimension $32 \times 21 \times 32$ and 4 tiles of dimension $32 \times 22 \times 32$ (see Fig. 4).

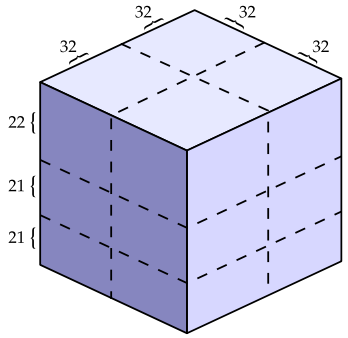


Figure 4: Decomposition of a 64^3 grid into 12 tiles: $2 \times 3 \times 2$

Within each tile, the solution is updated according to the serial algorithm described in Sec.3.3.4 except:

- Step 1 and step 3 are each computed within synchronization barriers across all threads.
- In each dimensional pass, step 2 computes Δt based on the largest speed found in *all* tiles and the resulting Δt is used in step 3 in each tile.
- The Riemann problems computed at interior boundaries — that is, those boundaries shared by adjacent tiles — must take into account the cells of the adjacent tiles. Boundaries that tiles share with the original grid are computed as described in Sec. 3.4.3.

To properly handle the computation of Riemann problems at interior boundaries, we need to make the values of the cells along the boundary shared with each adjacent tile available. For each tile, for each adjacent tile (with which it necessarily shares an internal boundary), we keep a buffer into which a copy of the necessary cells is written directly before the data is needed for the associated Riemann solution pass. Then, rather than computing a special simplified variant of the Riemann problem based on the boundary condition as in the standard algorithm, we use the values from the appropriate neighbor buffer.

During the neighbor-update step for a dimensional pass, each tile copies the necessary values from its down grid data to the appropriate buffers belonging to its neighbors. Note that it may be tempting to eliminate the need for these buffers by simply have a tile read the data directly from a appropriate neighbor tile's grid, but doing so risks false sharing behavior and wasted bandwidth.

Indeed, rather than share one global grid data structure as in the naïve scheme, we eliminate the possibility of false sharing by having each tile allocate its own grid, suitably aligned and padded so as to share no cache lines with other tiles' data. This slightly complicates the output of grid data, as each tile must carefully copy its own portion of the aggregate grid to the output location.

The results of this scheme are shown in Fig. 13; the scaling has improved slightly for small grid sizes, but is slightly *worse* than the naïve scheme for larger grid sizes.

One more optimization is necessary to achieve the scaling we desire. A decrease in scaling for large grid sizes suggests that the method is bandwidth limited; for these large grid sizes, each tile's portion of the grid no longer fits in cache, and the increased accesses to main memory begins to saturate the bus for large numbers of processors.

4.3.1. Reducing memory usage

To reduce the bandwidth requirements of the algorithm and improve scalability, we must reduce the working set associate with each tile. The original solution procedure as described in Sec. 3.3.4 computes the full Riemann solution at each interfaces in step 1, uses the computed speed to determine Δt in step 2, and saves the computed fluxes for the updates in step 3.

This memoization saves computation but consumes a considerable amount of bandwidth; the full Riemann solution for an interface of the 3-dimensional Euler equations contains 5 waves (which are vectors of 5 values) and 5 speeds, whereas a single cell of the grid has only a single vector of 5 values (see Eqns. (18) and (19)). Since there are roughly the same number of interfaces where these Riemann solutions are stored as there are actual cells, saving the solution to all of the Riemann solutions in a pass requires 6 times the storage of the grid alone.

We can modify our algorithm to instead compute just the maximum speeds at each interface in step 1 and compute the full Riemann solutions as they are needed to advance the solution in step 3. There is a net increase in actual computation, since we are computing a significant portion of the Riemann solution at each interface in step 1 just to determine the maximum speeds, but we expect the reduced bandwidth to greatly improve the scaling of the algorithm.

As demonstrated in Fig. 14, the scaling is now very close to linear for the larger problem sizes. The 32^3 grid does not scale as well because the relative overhead of copying data during the neighbor-update as compared to the work done computing the solution is larger for a smaller grid.

The dip in performance in the 32^3 grid for 14 threads is due to the factors being 2 and 7; the grid cannot be divided evenly among 7 tiles in one dimension, thus there are 6 tiles with a dimension along one axis of 4 while the remaining has a dimension of 9. This disparity in workload size is particularly exaggerated at the small grid size.

5. Results

We have implemented and tested our algorithm on several challenging scenarios. In this section, we first show some demonstrations of our algorithm, then describe our rendering methods, and finally discuss timing.

5.1. Applications

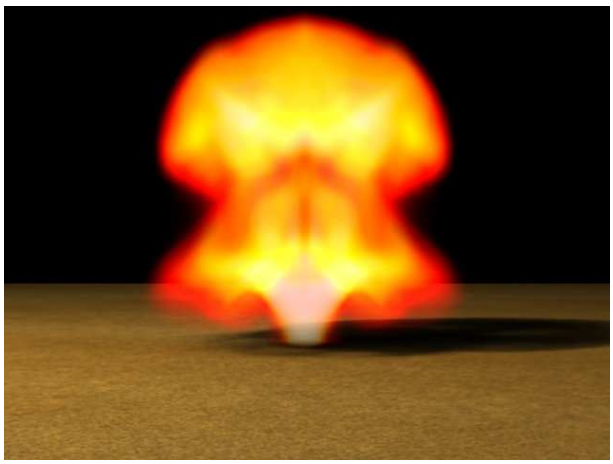


Figure 5: A mushroom cloud generated by our method

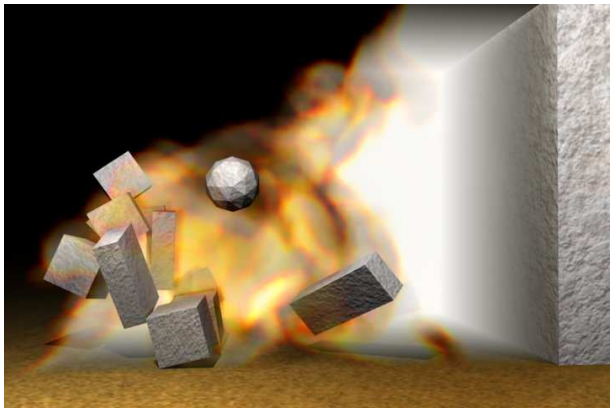
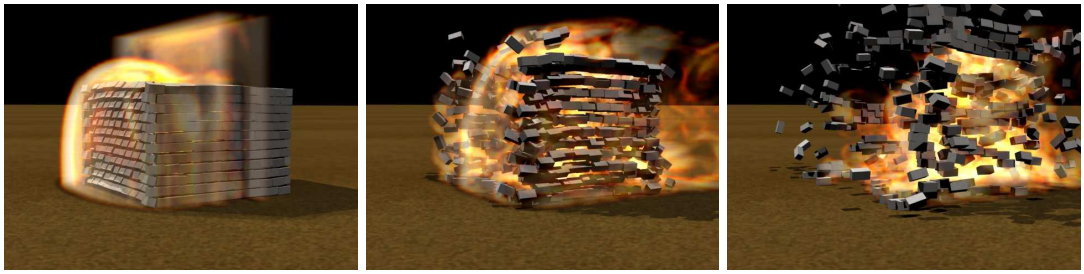


Figure 6: A stack of rigid bodies knocked over by a shock

We have constructed a number of scenarios that demonstrate the ability of our method to simulate visually interesting phenomena. The first segment of supplementary video is a two-dimensional simulation demonstrating vortex shedding — a traveling shock crosses a sharp obstacle and a powerful vortex forms in its wake. Further reflections of the shocks create new vortices which combine and travel around the domain.

Fig. 5 shows a mushroom cloud formed in the aftermath of a nuclear explosion; a low-density, high-temperature region left by the expanding shock is forced upwards by the pressure gradient caused by gravity; as it rises, the region expands and curls downward, forming a distinctive mushroom shape.

Fig. 6 demonstrates our method's ability to interact with moving boundary conditions; the stack of rigid bodies in this scene are bidirectionally coupled to the fluid. A traveling shock topples them, reflects off a nearby wall, and rebounds on the objects, throwing them away. The



(a) (b) (c)

Figure 9: Rigid body-fluid interaction

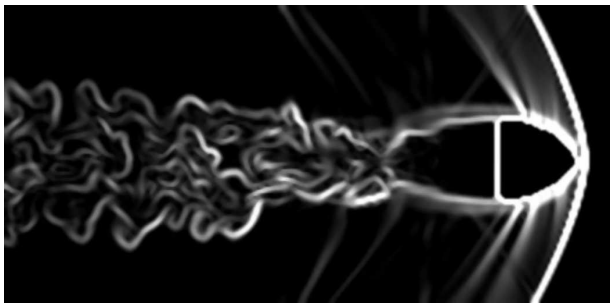


Figure 7: A bow shock and turbulence formed by the passage of a supersonic bullet

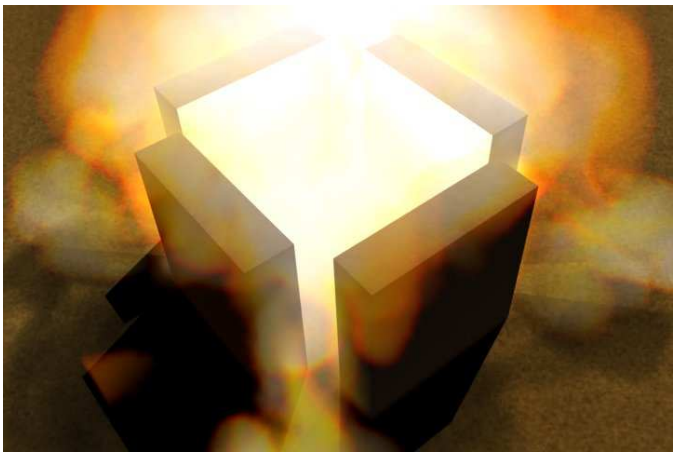


Figure 8: An explosion in a confined space

bodies' force upon the fluid creates vorticial patterns in the gas.

Fig. 7 shows a 2D slice of a 3D simulation of a projectile traveling faster than the speed of sound. The bow shock ahead of the body is typical of this type of rounded object and the rarefaction region behind the projectile creates a twisting trail of turbulence.

Fig. 1 and Fig. 2 are similar; in each, a cylindrical tower of 600 bricks is toppled by an explosion from within. Fig. 1 has no cap; the explosion forces nearly all of the air out of the cylinder as it bursts out of the top. The low-pressure area formed inside the cylinder causes it to collapse in upon itself while the force of the explosion venting from the top of the structure send bricks flying. Fig. 2 has a very heavy cap atop it; the explosive force cannot escape so easily and is partially reflected back into the structure, forcing a hole in the base and blowing out bricks near the top.

Fig. 8 shows an explosion occurring in an enclosed area; the force of the explosion forces air through the small openings in the chamber and creates high-density, turbulent tendrils.

Fig. 9 shows a series of frames from a simulation where a "house" made of 480 concrete bricks is struck by a powerful shock, causing the bricks to fly in all directions. The bricks shape and reflect the shock as it propagates through the scene.

Fig. 10 is a visual recreation of the first moments of the detonation of the first nuclear bomb 'Trinity'. The glossy "bubble" around the explosion is the expanding shock-front; the heat at the interface is such that light traveling through the region is dramatically refracted. Inside the shock, dust and flame are rising with a bright glow.

5.2. Rendering

Our 3D demonstrations were modeled in Blender [33] and rendered with the V-Ray raytracer [34]; the visualization of fluid effects in 3D were handled by our Monte Carlo volume raytracer plug-in for V-Ray. Atmospheric scattering was not used; these renders use ρ as advected by the fluid for the emissive and absorbing factors for the volume tracer, with color determined by a linear mapping of ρ into a blackbody colormap.

The 2D demonstrations were rendered with our simple custom 2D plotting tool; those using a monochrome colormap demonstrate our method's preservation of sharp shock features through a *schlieren* plot — namely, we plot $\sqrt{|\nabla\rho|}$. The term schlieren refers to a particular type of image formed by the passage of light through inhomogeneous media that causes shadows to appear in areas of high inhomogeneity; see the book by Settles [35] for more detail.

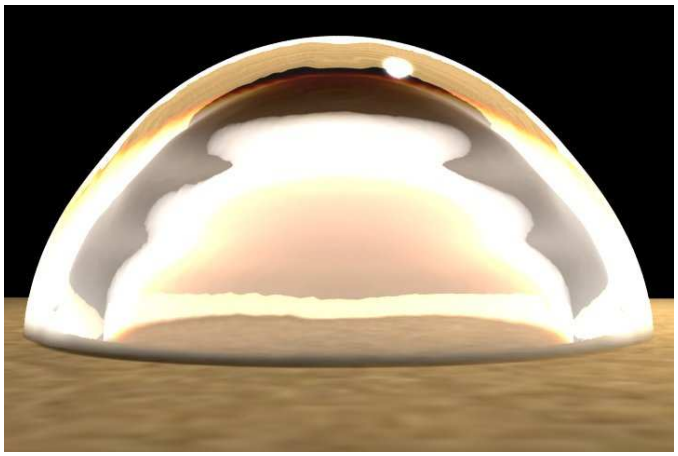


Figure 10: The initial moments of the "Trinity test" — the first atomic bomb

Table 1: Demonstrative timings of our method

Scene	resolution	sim. fps	avg. Δt	sim. time
Blast chamber	120×80×120	1.56	1.4e-4 s	16.25 min
Rigids w/ refl.	60×60×100	0.779	2.5e-4 s	29.93 min
Tower (top)	60×80×60	1.14	7.2e-4 s	30.21 min
Trinity	200×75×200	0.102	2.9e-5 s	32.88 min
Tower (no top)	60×100×60	0.939	6.8e-4 s	51.74 min
Mush. cloud	120×100×120	0.243	2.4e-2 s	57.74 min
House	100×100×100	0.310	4.6e-5 s	58.35 min
Projectile	250×100×100	0.191	1.0e-5 s	191.5 min

Timings showing grid resolution, simulation frames per second, average simulation timestep, and the total computation time needed for the entire simulation run.

5.3. Timings

We present performance data in Table. 1; these timings were collected on a 2GHz Core 2 laptop. Memory usage is linear in the number of grid cells — each demonstration fits within 500MB of memory. These timings are all for a single thread of computation; our parallelization results were discussed in Sec. 4.

Direct comparisons with previous works are difficult to produce because little or no timing information is available for these papers. Figure 2 in [18] shows a 2D slice of a 101^3 simulation of a shockwave interacting with a stationary wall; they reported a simulation time of 'overnight'. We reproduced this simulation with our method; for a 101^3 grid, we recorded a total simulation time of *15 minutes*. Conservatively estimating that our single core of our hardware is nearly 7 times faster and that 'overnight' is about 10 hours, the serial version of our method is at least 6 times faster than theirs at equivalent resolutions, and our simulation contains more visual detail. See Fig. 11 for a comparison.

To demonstrate the ability of our method to produce detailed results at coarse resolutions, we performed the same simulation on a 60^3 grid; this took less than *2 minutes* (roughly 45x faster) and the generated results exhibit more detail than the results computed on a 101^3 grid using

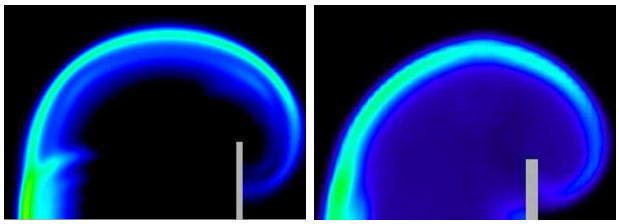


Figure 11: Left: Image from Yngve et al.'s paper; blast diffaction on a 101^3 grid. Right: Our method in similar scenario, 60^3 grid.

[18]. We have included these results in our supplementary video; the corresponding video from Yngve et al. can be found at [36].

5.4. Parallelization

Figs. 12, 13, 14 demonstrate the scaling for the various parallel schemes described in Sec. 4. These timings were collected on system with four Intel Xeon X7350 quad-core processors running at 2.93GHz; the system runs Microsoft Windows Server 2003 (64-bit) and has 16GB of physical memory.

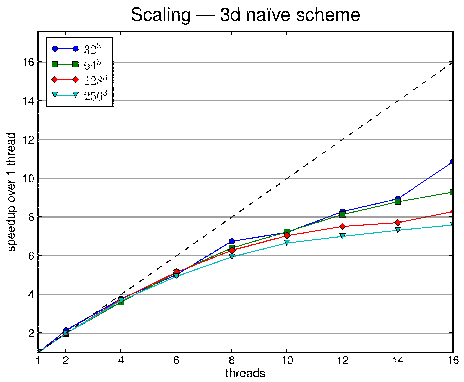


Figure 12: A naïve parallelization scheme scales poorly with the number of threads

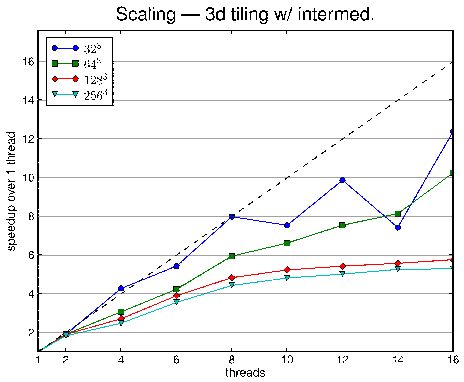


Figure 13: Scaling of the initial version of the tiled parallelization. Memoization of Riemann solutions leads to bandwidth saturation for large numbers of processors.

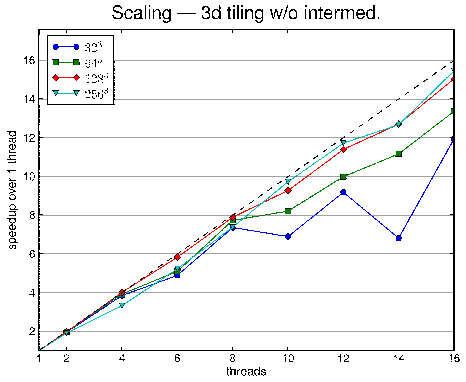


Figure 14: Scaling of the revised version of the tiled parallelization. The reduction in bandwidth requirements greatly improves scaling.

6. Conclusion

We have presented a method for efficient simulations of supersonic flows in compressible, inviscid fluids that is based on the finite volume method. We have demonstrated the ability of our method to capture the behavior of shocks and to handle complex, bidirectional object-shock interactions stably. We have also demonstrated an effective parallelization scheme based on architectural considerations that achieves near-linear scaling on modern multi-core architectures.

6.1. Limitations

Hyperbolic systems of equations (i.e. the compressible, inviscid Euler equations simulated here) are subject to the CFL condition as a requirement for convergence *and* stability. The unconditionally stable solvers popular for incompressible fluid dynamics are subject to the CFL condition for convergence, but not stability — indeed, the convention seems to take the CFL condition as a "guideline" and use CFL numbers upwards of 5.

Our technique performs well at simulating truly hyperbolic phenomena such as compressible, inviscid fluid dynamics, but cannot handle nearly incompressible phenomena (e.g. liquids) as efficiently as those simulations currently used in computer graphics. This fundamental limitation is due to the choice of equations — the actual propagation of acoustic waves so important to compressible fluids has a negligible effect on incompressible fluids.

6.2. Future work

There are a number of promising areas for future work. Many natural phenomena give rise to shocks — of particular interest to graphics are hydraulic jumps in the Saint-Venant (or shallow water) equations.

The tiled parallelization scheme we employ scales very well compared to a naïve parallel decomposition, but there is potential for further improvement with a nuanced investigation of further cache effects, operating system scaling, and processor layout. We also would like to investigate the extension of our method to new parallel architectures, such as Intel's Larrabee, IBM's Cell, next-generation graphics cards leveraging OpenCL and CUDA.

Acknowledgement: The authors would like to thank Yuri Dotsenko, Naga Govindaraju, Brandon Lloyd, Rick Molloy, and Avneesh Sud for helpful discussions on parallel algorithms.

This research is supported in part by the Army Research Office, National Science Foundation, U.S. Army RDECOM Intel Corporation, and Carolina Development.

References

[1] J. Stam, Stable fluids, in: A. Rockwood (Ed.), Siggraph 1999, Computer Graphics Proceedings, Addison Wesley Longman, Los Angeles, 1999, pp. 121–128.

[2] F. Losasso, F. Gibou, R. Fedkiw, Simulating water and smoke with an octree data structure, in: ACM SIGGRAPH '04, ACM Press, New York, NY, USA, 2004, pp. 457–462.

[3] B. Adams, M. Pauly, R. Keiser, L. J. Guibas, Adaptively sampled particle fluids, in: ACM SIGGRAPH '07, ACM, New York, NY, USA, 2007, p. 48.

[4] N. Chentanez, B. E. Feldman, F. Labelle, J. F. O'Brien, J. R. Shewchuk, Liquid simulation on lattice-based tetrahedral meshes, in: SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 219–228.

[5] C. Batty, F. Bertails, R. Bridson, A fast variational framework for accurate solid-fluid coupling, in: ACM SIGGRAPH '07, 2007.

[6] N. Foster, D. Metaxas, Realistic animation of liquids, Graph. Models Image Process. 58 (5) (1996) 471–483.

[7] N. Foster, R. Fedkiw, Practical animation of liquids, in: ACM SIGGRAPH '01, ACM Press, New York, NY, USA, 2001, pp. 23–30.

[8] R. Bridson, Fluid Simulation for Computer Graphics, AK Peters Ltd, 2008.

[9] B. E. Feldman, J. F. O'Brien, B. M. Klingner, Animating gases with hybrid meshes, in: ACM SIGGRAPH '05, ACM Press, New York, NY, USA, 2005, pp. 904–909.

[10] S. Elcott, Y. Tong, E. Kanso, P. Schröder, M. Desbrun, Stable, circulation-preserving, simplicial fluids, ACM Trans. Graph. 26 (1) (2007) 4.

[11] J. Wendt, W. Baxter, I. Oguz, M. Lin, Finite-volume flow simulations in arbitrary domains, Graphical Models 69 (1) (2007) 19–32.

[12] B. M. Klingner, B. E. Feldman, N. Chentanez, J. F. O'Brien, Fluid animation with dynamic meshes, in: ACM SIGGRAPH '06, ACM Press, New York, NY, USA, 2006, pp. 820–825.

[13] N. Chentanez, T. G. Goktekin, B. E. Feldman, J. F. O'Brien, Simultaneous coupling of fluids and deformable bodies, in: SCA '06: Proeedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2006.

[14] P. Roe, Approximate Riemann solvers, parameter vectors, and difference schemes, J Comput. Phys. (43) (1981) 357–372.

[15] B. van Leer, Towards the ultimate conservative difference scheme iv, J. Comp. Phys. (22) (1977) 276–299.

[16] R. J. Leveque, Finite Volume Methods for Hyperbolic Problems, Cambgridge University Press, New York, 2002.

[17] R. Fedkiw, G. Sapiro, C.-W. Shu, Shock capturing, level sets and PDE based methods in computer vision and image processing: A review on Osher's contribution, J. Comput. Phys. (185) (2003) 309–341.

[18] G. D. Yngve, J. F. O'Brien, J. K. Hodgins, Animating explosions, in: ACM SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000, pp. 29–36.
[19] J. Sewall, P. Mecklenburg, S. Mitran, M. Lin, Fast fluid simulation using residual distribution schemes, in: Eurographics Workshop on Natural Phenomena 2007, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 47–54.
[20] B. E. Feldman, J. F. O'Brien, O. Arikan, Animating suspended particle explosions, in: ACM SIGGRAPH '03, ACM, New York, NY, USA, 2003, pp. 708–715.
[21] A. Selle, N. Rasmussen, R. Fedkiw, A vortex particle method for smoke, water and explosions, in: ACM SIGGRAPH '05, 2005, pp. 910–914.
[22] M. Müller, D. Charypar, M. Gross, Particle-based fluid simulation for interactive applications, in: SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003, pp. 154–159.
[23] O. Genevaux, A. Habibi, J.-M. Dischler, Simulating fluid-solid interaction, in: Proc. Graphics Interface '03, 2003.
[24] M. Carlson, P. J. Mucha, G. Turk, Rigid fluid: animating the interplay between rigid bodies and fluid, ACM Trans. Graph. 23 (3) (2004) 377–384.
[25] E. Guenelman, A. Selle, F. Losasso, R. Fedkiw, Coupling water and smoke to thin deformable and rigid shells, in: ACM SIGGRAPH '05, ACM Press, New York, NY, USA, 2005, pp. 973–981.
[26] O. Mazarak, C. Martins, J. Amanatides, Animating exploding objects, in: Proc. Graphics Interface '99, AK Peters, Wellesley, MA, USA, 1999, pp. 211–218.
[27] M. Neff, F. Fiume, A visual model for blast waves and fracture, in: Proc. Graphics Interface '99, AK Peters, Wellesley, MA, USA, 1999, pp. 193–202.
[28] P. Lax, B. Wendroff, Systems of conservation laws, Comm. Pure Appl. Math. (13) (1960) 217–237.
[29] R. Courant, K. Friedrichs, H. Lewy, Über die partiellen differenzengleichungen der mathematischen physik, Mathematische Annalen 100 (1) (1928) 32–74.
[30] Bullet Physics Library, `http://www.bulletphysics.com/`.
[31] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing 27 (1–2) (2001) 3–35, also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).
[32] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231, special issue on "Program Generation, Optimization, and Platform Adaptation".
[33] Blender 2.45, `http://www.blender.org/`.
[34] V-Ray, `http://www.chaosgroup.com/en/2/vray.html`.
[35] G. S. Settles, Schlieren and shadowgraph techniques: Visualizing phenomena in transparent media, Springer-Verlag, 2001.
[36] G. D. Yngve, J. F. O'Brien, J. K. Hodgins, `http://www.cs.berkeley.edu/b-cam/Papers/Yngve-2000-AE/Stuff/wall_pressure.mpeg`.