Visually Believable Explosions in Real Time

Claude Martins
Electronic Arts Canada
4330 Sanderson Way
Burnaby, BC, Canada
cmartins@ea.com

John Buchanan
Electronic Arts Canada
4330 Sanderson Way
Burnaby, BC, Canada
juancho@ea.com

John Amanatides
Dept. of Computer Science
York University
4700 Keele Street
Toronto, ON, Canada
amana@cs.yorku.ca

Abstract

The paper presents a real-time physically based simulation of object damage and motion due to a blast wave impact. An improved connected voxel model is used to represent the objects. The paper also explores auxiliary visual effects caused by the blast wave that increase visual believability without being rigorously physically based or computationally expensive.

1. Introduction

Explosions, and their resultant blast waves, continue to be a popular phenomenon depicted in film, television, computer games, and other visual media. There are certain advantages to simulating the appearance and behavior of blast waves using computer graphics, such as safety, reproducibility, customizability, and interactivity.

Blast waves, however, follow complex fluid dynamics rules, and modeling them very accurately is computationally expensive. This is especially true for a game or a real-time application, where immediate visual feedback is paramount. Users expect instant gratification with unpredictable and chaotic blast effects. Meticulous physical accuracy is less desirable here than a fast simulation speed.

This paper takes the blast wave simulator and connected voxel model presented by Mazarak *et al* [MMA99] and extends its scope to real-time graphics. The pertinent enhancements and simplifications necessary for the improvement of computing time are discussed. The functionality of the model is also enhanced by the introduction of arbitrary voxel shapes. Furthermore, certain commonly expected visual cues are added to the simulation to improve the visual believability of the explosion at low computational cost.

2. Previous work

Previous efforts on explosion modeling in the field of computer graphics have traditionally focused more on the graphical representation of an explosion, rather than its effects on objects in the environment. Usually, the attributes and motion of an explosion are controlled explicitly by the user, or else the explosion follows a set of informal behavioral rules [Reeves83]. Other techniques concentrate on certain after-effects of an explosion, such as fire [COMM94] and smoke.

Interest has recently been rekindled in blast waves and their effects. In his Master's thesis, Bashforth employs volume cell subdivision to model shock front propagation [Bashforth98]. Neff and Fiume model the fracturing of a planar surface into polygonal fragments by a spherical blast wave [NF99]. Mazarak *et al* use connected voxels to model objects breaking into solid debris when hit by a spherical blast wave [MMA99]. Yngve *et al* model blast wave propagation using a combination of spatial voxelization and finite elements [YOH00]. Except for [MMA99], these approaches eschew simulation speed in favor of rigorous physical accuracy, which makes them inappropriate for use in real-time graphics.

Although papers discussing explosions and blast waves are somewhat rare in the field of computer graphics, it is important to note that this is not the case for the disciplines of physics and chemistry. There have been numerous works published in those areas that delve with great detail into the creation and detonation of explosive materials, the propagation of the generated shock fronts, and their effect on various materials.

Several computer methods for the modeling of blast waves have also been proposed and implemented [Baker73]. These models, however, are concerned primarily with predicting and duplicating the behavior of explosions and explosion effects down to the smallest detail. They require substantial computational power and time to handle the governing fluid dynamics equations.

In real-time computer graphics, replicating blast wave behavior in every detail is simply not feasible. Instead, we cull some of the overall relevant physical properties, and make certain simplifications and optimizations. In this way, it is possible to obtain a balance between accurate modeling and realistic visuals without sacrificing a great deal of computing time.

3. Blast wave theory

An explosion in air causes a *blast wave* to propagate outwards from the source at supersonic speed. Since the resurgence of recent interest in blast wave modeling in computer graphics, the fundamentals of blast wave theory have been adequately covered in several papers ([MMA99], [NF99], and [YOH00]).

Our blast wave model is a combination of simplified physical equations and experimental data. The air surrounding the explosion is assumed to be still and homogenous, and the explosion source is spherically symmetric. This results in an ideal blast wave that is itself perfectly symmetrical. The pressure profile generated by an ideal blast wave at a point at some fixed distance R removed from the center of the explosion is as shown in the following Figure 1 [Baker73].

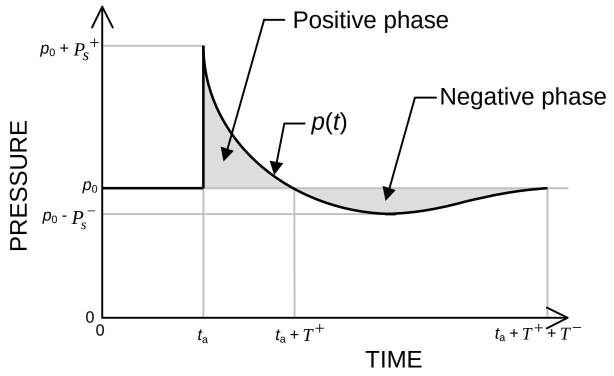

Figure 1. Pressure-time curve of an ideal blast wave

Before the shock front reaches the given point, the ambient pressure is p_0. At arrival time t_a, the pressure rises discontinuously to the peak value of $p_0 + P_s^+$. The quantity P_s^+ is called the *peak overpressure*. The pressure then decays to ambient in total time $t_a + T^+$, drops to a partial vacuum of value $p_0 - P_s^-$, and eventually returns to the ambient pressure p_0, in total time $t_a + T^+ + T^-$ [Baker73].

Our simulation models blast wave profiles using the modified Friedlander equation:

$$p(t) = p_0 + P_s^+ (1 - t / T^+)\ e^{-bt/T^+} \qquad (3\text{-}1)$$

Time is measured from time of arrival t_a. The blast wave parameters P_s^+, t_a, T^+, and b allow freedom to customize the pressure profile curve for any explosion, at various distances from the source.

The modified Friedlander equation improves over simpler formulae, which either have linear decay or fail to return to ambient pressure [Baker73]. More complex equations given by Brode [Brode95] and Dewey [Dewey64] generate blast wave profiles that are closer to experimental or theoretical models. For a real-time simulation, however, the increased accuracy does not justify the concomitant higher computational costs.

Our blast wave model employs Equation (3-1) to compute physically accurate pressure changes at any distance R from the source of the explosion. The blast wave parameters required by Equation (3-1) are obtained from experimental data for a reference explosion of one kilogram of TNT (trinitrotoluene) in a standard atmosphere [KG85]. The experimental data contains values for peak overpressure P_s^+, expected arrival time t_a, positive phase duration T^+, and the pressure decay coefficient b, measured at certain distances R_1, R_2, …, R_n, from the source. The corresponding parameters for any arbitrary distance R are obtained by linear interpolation.

Time and distance related parameters for explosions with different yields than one kilogram of TNT are derived from the reference explosion data by using an appropriate scaling factor as determined by the scaling laws [KG85]. This scaling factor is normally equal to the inverse of the cube root of the energy of the derived explosion:

$$Z = \frac{R}{W^{1/3}} \qquad (3\text{-}2)$$

where Z is the scaled distance, R is the actual distance from the center of the explosion in meters, and W is the mass of the explosive converted into an equivalent weight of TNT in kilograms.

Since blast waves move at supersonic velocities, they propagate in a non-linear fashion. Environmental interactions frequently cause pressure increases, irregular reflections, vortices, and other complex phenomena [Baker73]. These computationally intensive effects are difficult to simulate in real time, so we use a simple model that propagates blast waves outwards as if they were in the open air and not being affected by surrounding objects. As a result, object damage occurs that is not identical to a real-life explosion. Fracturing in our simulation is more accurate when there are no, or few, obstructions between the explosion source and a given object.

4. Modeling and animation

Rigid bodies in the environment are affected by blast waves in two major ways. Firstly, the blast wave creates tensile stress within the object that can cause it to fracture and break apart. Secondly, the blast wave pressure exerts forces on the object that can cause it to move and rotate. Therefore, a particular object model must be chosen that takes both of these effects into account.

4.1 Connected voxels

The objects in the simulation are modeled with connected voxels [MMA99]. An object is decomposed into volume elements, or *voxels*, that make up its volume. Adjacent voxels are connected to each other with inflexible *links* that keep the voxels firmly attached together.

The connected voxel approach offers several advantages over other models. Firstly, it is volumetric- rather than surface-based, so fracturing of an object's interior is possible. Secondly, the model is scaleable. If a more accurate simulation is desired, the voxel size can be reduced, allowing a finer representation. Likewise, if simulation speed is more important than precision, a larger voxel size can be used. Thirdly, the links connecting the voxels are infinitely stiff, unlike the spring-mass particle model [TPBF87]. This makes the object a true rigid body, and the simulation remains robust. Finally, surface association for the objects is not required when using connected voxels because each voxel already has a given shape assigned to it. This is an important benefit in a real-time explosion simulation, because accurately computing new surfaces for the dynamically created debris is non-trivial.

In the original implementation by Mazarak *et al* [MMA99], every voxel was identical and homogenous. A basic voxel cube shape resulted in objects that were blocky and unrealistic. Our simulation improves upon this noticeably, through the introduction of *arbitrarily shaped* voxels. The basic voxel cube shape can be scaled by any desired amount along any axis. As well, the voxel's vertices can be displaced by any amount in any arbitrary direction.

Each voxel can also have unique properties appropriate for the material it represents. This makes object modeling much more flexible, and permits the user to create voxel shapes that better suit his or her needs for a particular application.

Determination of visible surfaces is trivial in the connected voxel model. Any voxel face that does not have a link attached to it is visible and must be rendered. Rendering is thus linear on the number of boundary voxels.

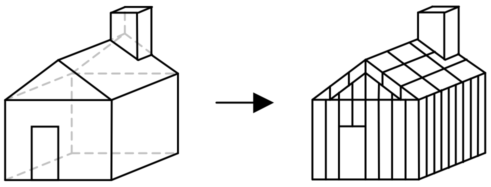
Figure 2. Flexible object modeling using arbitrary voxels.

Regardless of the voxel shape, the simulation assumes that any given voxel has a constant density, which simplifies the force and torque computations used in object animation.

4.2 Fracture simulation

When a blast wave hits an object, pressure differentials cause the object to weaken and fracture. We simulate this by weakening and breaking links and voxels in the object model. As the object breaks apart, new fragments may be created. Independent objects in the scene are represented by the connected components of the entire scene's connectivity graph, where the nodes are individual voxels and the arcs are the links between voxels [MMA99].

Every link has an associated *yield limit*, which is the maximum pressure that the link can withstand before breaking. A link is broken whenever the absolute value of the pressure at that link's midpoint exceeds its yield limit. Any links that encounter a blast wave have their yield limits weakened, making them vulnerable to subsequent explosions.

Additionally, links that are parallel to the direction of the wave are further weakened by an orientation factor, computed as a dot product of the wave's radius vector $\vec{R}$ and the link vector $\vec{l}$. The inclusion of this orientation factor allows the simulation to better mimic a real explosion, where tensile forces created by the blast wave inside an object are much stronger in the direction parallel to the direction of the wave than normal to it.

An enhancement can be made to the fracture model by adding a yield limit for the voxels themselves. Pressure from a blast wave is measured at the center of each voxel. If the absolute value of the pressure at a voxel's center exceeds its yield limit, that voxel is removed and replaced with a small particle system. Linear momentum of the original voxel is passed on to the individual particles in the replacement particle system. This simulates the destruction of object fragments into particulate dust if they undergo sufficient stress due to the blast wave. As with damaged links, voxels that survive a first blast wave have their yield limits weakened.

A significant benefit of replacing voxels with particle systems is that particles are easier to handle computationally, and are also generally quicker to render. Furthermore, allowing voxels to be destroyed helps circumvent certain problems such as prolonged interpenetration during collision.

Since the shock front pressure forces are very strong, objects in the environment tend to start tumbling everywhere once the blast wave hits them. This is unrealistic for buildings that are supposed to have foundations. For such fixed structures, the simulation tags voxels that are in contact with the ground as *foundation voxels*. An object having these foundation voxels in it is defined as "fixed", i.e., blast waves will affect voxel and link strength but will not impart any momentum to the structure. This allows debris to fly off the object and have momentum, but the original structure remains stationary and does not tumble around.

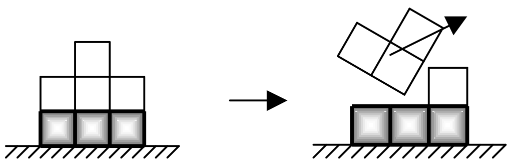

Figure 3. Foundation voxels (shown shaded) stabilize the rigid body.

Of course, blast waves can sometimes be strong enough to rip a building right off its foundations. So, a foundation yield limit is introduced for each foundation voxel. As with voxel yield limits, the blast wave pressure measured at the center of each voxel is used to diminish its foundation yield limit. Once a voxel's foundation yield limit drops to zero or below—assuming it hasn't been destroyed completely—its foundation voxel status is removed. If all of the foundation voxels in a structure revert to ordinary voxels in this way, that object is no longer "fixed". In effect, it has been blown off its foundations. Blast waves hitting the structure now impart momentum to it, as well as voxel and link damage.

We assign variable yield limits by perturbing some random value, generated within a certain range, around some mean value [TF88]:

$$\textit{yield\_limit} = \textit{mean\_yield} \pm \mathrm{rand}(\textit{var\_yield}) \qquad (4\text{-}1)$$

This allows the object to have a non-homogenous structure, with weaker and stronger sections. The continual deterioration of the object's links and voxels also simulates partial, persistent damage.

It is possible that a simple variance in object structure might be insufficient. For example, a game designer may want a specific building wall to be easily destroyed. He or she can then alter link and voxel properties explicitly to customize objects for a particular game and override the basic variance values assigned in object configuration.

4.3 Animation

The blast wave not only causes object fracture and fragmentation; it also affects their translation and rotation through the application of forces and torques. Computing the force at the voxel center [MMA99] is insufficient since one-voxel bodies would not experience torque. Furthermore, forces exerted on arbitrary voxel shapes would be inaccurate. Therefore, the forces exerted by the blast wave are evaluated at the vertices of each voxel.

The force at a given vertex is computed as a product of the blast wave pressure and the area of that voxel projected onto the surface of the shock front. Assuming a spherical blast wave, the force vector is congruent to the blast wave radius vector:

$$\vec{F} = p(t) \cdot A \cdot \frac{\vec{R}}{\left\|\vec{R}\right\|} \qquad (4\text{-}2)$$

where $p(t)$ is a function that returns the pressure value at a given voxel vertex at time t, A is the surface area of the voxel projected onto the spherical shock front, and R is a radius vector from the source of the explosion to the voxel vertex.

The projected surface area of a cubic voxel onto a spherical blast wave varies mainly on its size and very little depending on its orientation. Computing the actual projected surface area for each voxel is too time-consuming in a real-time simulation, so A is assumed to be constant, depending on the size of the voxel. Non-cubic voxels are assigned the projected surface area of a cubic voxel with a side equal to the original voxel's largest dimension. This approximation is imprecise, with the accuracy decreasing as the difference increases between the non-cubic voxel and the original cubic voxel.

The torque on the body is computed as the cross product of the relative position of a specific voxel vertex within the body, and the force experienced by that vertex:

$$\vec{T} = \vec{r} \times \vec{F} \qquad (4\text{-}3)$$

where $\vec{r}$ is the relative position of the voxel vertex to the center of mass of the body. The force and torque computation for a single voxel vertex is shown in Figure 4.

Noting that the body's boundary voxels approximate its surface implies that the force and torque vectors only need to be computed for the boundary voxels. These force and torque vectors are then summed up to obtain the resultant force and torque vectors for the entire body.

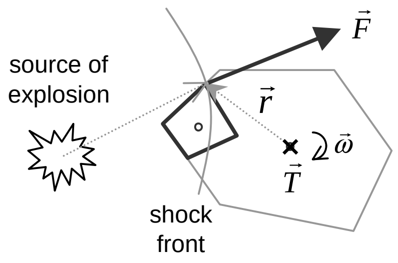

Figure 4. Force and torque computation.

$$\vec{F}_{body} = \sum \vec{F}_{boundary\_voxels} \quad (4\text{-}4)$$

$$\vec{T}_{body} = \sum \vec{T}_{boundary\_voxels} \quad (4\text{-}5)$$

The dynamic simulator then updates the body's position using the resultant force vector and the body's orientation using the resultant torque vector.

The linear and angular momenta of any newly created body fragments are computed as weighted portions of the linear and angular momenta of the original body:

$$P_{fragment} = \frac{m_{fragment}}{m_{original}} P_{original} \quad (4\text{-}6)$$

$$L_{fragment} = \frac{m_{fragment}}{m_{original}} L_{original} \quad (4\text{-}7)$$

where P is the linear momentum, L is the angular momentum, and m is the mass. This agrees with the law of conservation of momentum, and ensures accurate motion of the split fragments.

Using a discrete time step to simulate the continuous propagation of a shock front tends to lead to inaccuracy. It is possible for the shock front to "skip over" certain voxels completely from time t to next consecutive time $t + \Delta t$. Reducing the size of the time step will reduce this discretization error—at the cost of a slower-running simulation—but will not remove it completely. Voxel size and arrangement might still be such that the shock front skips over particular voxels. Using an adaptive time step size that ensured the shock front hit every voxel would eliminate any such error. However, the simulation would need a large number of tiny time steps whenever the shock front passes over voxels located in very close proximity, which is the case for any multi-voxel object.

We currently reduce this error by using *shock front precomputation*. Every time an explosion is introduced into the environment, the peak overpressure (the most significant part of the blast wave profile), arrival time, and resultant shock front force are computed for each voxel. These precomputed values are then stored in a chronologically sorted list for that voxel.

At each simulation time step, forces and torques are computed for a boundary voxel. In addition, that voxel's list of precomputed values is checked to determine if any of the precomputed shock front forces are set to occur. If the current simulation time is equal to the stored arrival time for that entry in the list, this implies that the shock front is precisely at that particular voxel's center. In this case, the precomputed values are used in place of the current force and torque computation. If the current simulation time is greater than the stored arrival time, this implies that the shock front has skipped over that voxel. So here, the precomputed values are added to the current force and torque computation. In both cases, the entry is removed from that voxel's list.

This treatment of precomputed shock front values is advantageous because it allows the simulation to maintain a constant time step while still taking into account the effect of the peak overpressure for each voxel. The method is not entirely accurate, however, since the stored arrival time is offset from the current simulation time by a small amount. The magnitude of the error is always smaller than the size of the time step:

$$| t_{current} - t_{arrival} | < \Delta t \quad (4\text{-}8)$$

In practice, this inaccuracy is negligible compared to the original discretization error.

It is important to note that only the blast wave's peak overpressure is precomputed. Pressure changes at other points in the pressure-time curve are computed as they occur during the simulation.

5. Collision detection

Once objects are in motion, there is a possibility that they will interpenetrate if appropriate checks for collision are not made.

5.1 Dynamic contact

Dynamic contact occurs when adjoining objects are moving together with some positive velocity. We limit collision tests to boundary voxels, since they approximate the object's surface. Additionally, a voxel will never collide with other voxels belonging to the *same* rigid body because, by definition, a rigid body's link structure coherently maintains the relative positions of all of its component voxels and prevents self-intersection. Hence, collision tests need only be done for voxels from separate bodies.

Associating bounding volumes to the objects can reduce the number of comparisons even further. If two bounding volumes are determined to be disjoint, then their contents must necessarily be non-intersecting, and no extra tests for those specific objects are needed. We use a two-level hierarchical scheme with spheres as the bounding volumes. At the higher level, a rigid body is assigned a bounding sphere with a radius equal to the distance between the center of mass and the voxel vertex furthest from the center of mass. Since a body rotates about its center of mass, this ensures that the bounding sphere covers all possible body orientations at a given position. At the lower level, each voxel is assigned a bounding sphere with radius equal to the distance between the voxel center and the voxel vertex furthest from the center, for a similar reason.

For a real-time simulation, unfortunately, very accurate collision detection is too time-consuming. Therefore, two bodies are considered to collide in the simulation when their voxel-level bounding spheres intersect. If the voxels are stationary or moving away from each other, no collision response is necessary since interpenetration will not occur. If the voxels are moving towards each other, a proper collision response is computed to prevent interpenetration of the bodies. The linear and angular momenta of the bodies is changed discontinuously ([BW97], [MC95]) to ensure that the collision occurs with little to no deformation. Each voxel has an associated *coefficient of restitution* that determines how "bouncy" a collision is. A coefficient of zero corresponds to a perfectly inelastic collision where all kinetic energy is lost, whereas a coefficient of one corresponds to a perfectly elastic collision where all kinetic energy is retained.

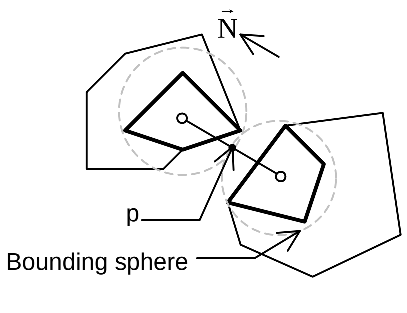

Figure 5. Dynamic contact collision.

Precise surface reconstruction is too time-consuming for real-time simulation, so the surfaces of the bodies are assumed to be smooth. Thus, the collision normal $\vec{N}$ is approximately parallel to the distance vector between the colliding voxels, and the collision point *p* is approximately equal to the midpoint of the distance vector [MMA99].

A blast wave can impart exceptionally high velocities to objects it hits, potentially causing some of them to interpenetrate noticeably in a single time step. Objects may even pass through each other completely. In principle, this problem can be solved by reducing the time step adaptively until the interpenetration is kept within a certain tolerance. However, this is not feasible for a real-time simulation because the time step may become prohibitively small.

One approach we use is the reduction of voxel strength by collision forces so that voxels that interpenetrate for too long are eventually destroyed. This method has an additional advantage because it approximates object damage due to collision. Another tactic is the creation of "dust"-like particle systems at collision points to mask interpenetration. This also adds visual information to the scene.

A difficulty with collision response algorithms is that the resolution of one contact could lead to the creation of new contacts. In theory, this situation could loop indefinitely and cause a major performance hit. In our implementation, all contacts are only resolved once in the current frame. Any new contacts created by collision resolution, as well as those that arise during the normal course of the simulation, are dealt with in the subsequent frame. Consequently, interpenetration is more pronounced but the frame rate is kept reasonable.

5.2 Static contact

Objects are defined to be at rest—i.e., in *static contact*—when they are in contact and are not moving with respect to each other. Finding static contact forces for a multi-body system is a complicated process, requiring efficient quadratic programming algorithms, such as those described in [Baraff94]. In order to facilitate static contact resolution, we introduce two limitations to collision.

Firstly, an object can only be at rest when it is in static contact with the ground, not with other objects. Objects that are adjacent and have zero relative velocity have no associated collision responses. This means that objects are kept from interpenetrating but tend to bounce, wobble, or slide when they fall atop one another with low velocities. Given that the blast wave from an explosion usually causes objects to move very fast, this is an acceptable restriction. Due to this limitation in our simulation, however, large debris piles eventually collapse towards the ground.

Secondly, static contact with the ground is resolved not with the computation of balanced repulsive forces, but through a combination of dynamic collision response and a heuristic—rather than physically based—implementation of friction that we call *pseudo-friction*.

When an object collides with the ground, the following assumptions are made to simplify calculations: the ground has infinite mass; the collision normal $\vec{N}$ is equal to the ground plane normal; and the collision point *p* is at the colliding voxel vertex.

If the object is moving into the ground, the simulator computes the proper dynamic collision response to prevent interpenetration. As mentioned above, collision forces diminish voxel strength—destroying the voxel and replacing it with a particle system if its strength drops to zero or below—and "dust"-like particle systems are spawned at collision points.

Simulating rigid body dynamics using the dynamic collision methods detailed above leads to the following difficulty. Objects tend to keep moving, spinning, and sliding around on the ground. It takes a long time for them to come to rest, if ever. In order to compel the objects to eventually come to rest, the simulation uses a sort of pseudo-friction.

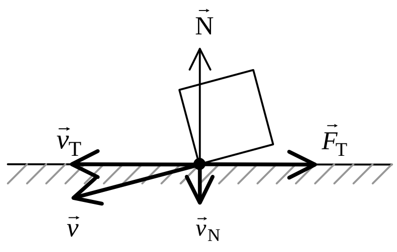

Figure 6. Ground collision.

An object in contact with the ground has an associated velocity $\vec{v}$ at the point of contact p. That vector $\vec{v}$ can be decomposed into a normal component $\vec{v}_N$ and a tangential component $\vec{v}_T$. If $\vec{v}_T$ is non-zero, the presence of tangential friction needs to be taken into account. This is done by the addition of a tangential force $\vec{F}_T$ to the object in the opposite direction of the tangential velocity $\vec{v}_T$. The magnitude of $\vec{F}_T$ is correlated to the magnitude of $\vec{v}_T$ and the mass of the object m; i.e., a heavier object moving faster experiences more friction than a lighter object moving slower.

Adding the above tangential friction prevents objects from sliding along the ground forever. However, objects still tend to oscillate. Subtracting a part of an object's linear and angular momenta whenever it comes into *resting contact* with the ground can minimize these oscillations. Resting contact is defined to occur whenever an object in contact with the ground has a normal velocity $\vec{v}_N$ of zero (within a certain threshold). Furthermore, if an object has three or more contact points with the ground, and either its linear or angular momentum approaches zero, it is clamped to zero. Using this implementation of pseudo-friction eliminates any small perturbations to a body, and thus also helps maintain the stability of an object already at rest.

We chose a lower limit of three contact points for the clamping of linear and angular momenta to zero because an object usually needs at least three ground contact points to remain at rest (e.g., a tripod). In the real world, however, an object whose center of mass lies outside the convex hull of its ground contact points—regardless of the number of contact points—is *unbalanced*, and will therefore topple over.

An object is in *balanced* rest only when its center of mass projected onto the ground plane lies within the convex hull of its ground contact points. Computing the convex hull of an object's ground contact points proves to be too computationally expensive to justify in a real-time simulation. Instead of the convex hull, an axis-aligned bounding box containing the ground contact points is computed. Pseudo-friction is only applied if the projected center of mass lies within this rectangular base. Obviously there are cases where this approximation is invalid, but it yields better results than if no "balancing" condition is used, and is much quicker than using an accurate convex hull.

6. Accelerated visual cues

In addition to object fracture, a spectator viewing a real explosion has numerous other visual *cues* that convince him or her that the detonation is authentic. A cue is something that adds *expected* or *perceived* visual information to the scene, and ultimately makes the animation more plausible. Any such cue need not be rigorously physically derived as long as it satisfies the viewer's expectations. A simulator that renders virtual explosions needs to mimic as many of these cues as possible in order to convince a viewer that the blast is believable. This is even more obvious in real time where it is easier to duplicate simple, expected explosion effects than it is to meticulously simulate unfamiliar phenomena.

Since this is a real-time simulation, all of the visual cues are generated concurrently with the blast wave computation and object rendering, rather than as a post-processing step after the animation is completed. More realistic methods may be chosen provided they are not too computationally intensive.

A primary visual cue is the explosion itself. Possibly influenced by films and television, a viewer expects to see a large fireball at the explosion source. A display like this is created in the simulation by spawning multiple flame-like particle systems at the explosion source. Adding a bright yellow-orange light to each explosion particle system also maintains the illusion of an intense fireball.

In real life, the density changes across the shock front cause the index of refraction to be modified. Light rays passing through these regions of different refraction indices are bent by varying amounts. The amount of refraction due to the blast wave is usually very minor, and can barely be seen amidst the sound and fury of the explosion unless a viewer knows exactly what to look for. An accurate representation of light refraction by the shock front can be obtained by using ray-traced images in the simulation. Unfortunately, ray tracing is too time-consuming to use in a real-time simulation.

Nevertheless, viewers still expect to see some visual indication of the shock front itself. Since the simulation assumes the blast wave is spherical, a viewer's expectations can be fulfilled in this case by rendering a semi-transparent sphere centered at the explosion source, and having a radius equal to the radius of the blast wave. Transparency is gradually increased as the radius increases.

Since the blast wave itself is almost invisible, its propagation is notable mainly by the effect it has on the environment. As the blast wave moves along the ground, it kicks up clouds of dust. We achieve this by adding dust-like particle systems at the intersection of the expanding blast wave sphere and the ground plane. The blast wave also knocks dust off of the objects in its path. To imitate this effect, we generate particle systems at the intersection of the blast wave and the voxels.

A visual indicator not limited to explosions is the dust thrown off when objects hit the ground or each other with sufficient force. Again, the simulation reproduces this by spawning dust-like particle systems at voxel collision points. The dust thrown up at collision points also helps to mask any interpenetration that occurs.

Explosions produce high temperature conditions that lead to nearby flammable objects getting set on fire. For the average viewer, there is an inextricable link—again, possibly reinforced by films and television—between explosions and fire. Currently, the simulation models the pressure changes across a shock front using physically derived methods. It is possible to emulate combustibility by using a simpler, heuristic approach.

As with the yield limit, each voxel has a *combustibility limit* that is assigned in a similar fashion. During an explosion, the blast wave pressure measured at each voxel's center is scaled by a user-definable parameter. This scaled pressure is then used to diminish the combustibility limit for that voxel. Once a voxel's combustibility drops below zero, it is "set on fire", i.e., it spawns fire- and smoke-like particle systems. Voxels that are on fire have their yield limits weakened continually as long as they are aflame. Links that are attached to burning voxels also have their yield limits weakened. This simulates the destructive nature of fire.

In the real world, fire propagates based on the presence of nearby flammable objects that are affected by elevated temperatures. Rather than use a proximity approach—which is much more computationally expensive—the simulation takes advantage of the existence of links and propagates heat along them instead. Non-burning voxels attached to burning voxels have their combustibility limits weakened in turn. Based on the observation that fire tends to spread upwards in a structure, an orientation factor is computed for the position of the neighboring non-burning voxel. Voxels located directly above the burning voxel are accorded a larger orientation factor than voxels located to the side or below. This orientation factor is then used to scale deterioration of neighboring voxels so that fire propagation and damage tends to progress upwards.

When objects break apart, the sides that were hidden in the original object usually have a different appearance than the exposed faces. Any voxel face not originally visible that subsequently appears due to object breakage is assigned a generic "voxel damage" texture. This effect can be improved by having specific "damage" textures for individual voxels. It can be generalized further by having appropriate "damage" textures painted on as a voxel diminishes in strength, or is affected by fire.

Lastly, it is self-evident that footage of real explosions is shot with real cameras. What is less obvious, though, is that they are as affected by the blast wave as other objects. Unless the camera is locked down completely (and sometimes even then), it will shake as the blast wave passes it. Adding a similar "wobble" to the simulated camera when it encounters the blast wave helps to bring the virtual scene even closer to its real-world counterpart.

7. Results

The simulation generates animations that are visually similar to real-world explosions, through the use of physically based methods and the presence of particular visual cues. Important aspects of the blast wave, such as the peak overpressure and the negative phase, are present in the simulation.

Without collision detection, the simulation speed is dependent on the computation of connected bodies in the scene. This has an upper bound of O(n), where n is the total number of voxels in the scene. Pair-wise testing for collision detection increases the simulation time to O(m^2), where m is the total number of boundary voxels in the scene. The addition of object-level bounding spheres reduces the initial number of tests to O(k^2), where k is the total number of objects in the scene.

The basic blast wave simulation—without graphical output—is very fast (see Figure 7). In fact, the frame rate with no rendering is consistently about two orders of magnitude faster than similar tests with rendering. This demonstrates that rendering each frame causes a large performance hit. The frame rate can be improved substantially using hardware graphics acceleration.

The implementation performs reasonably close to real time (see Figure 8). For small (<200) numbers of voxels, the performance is near 30 frames/sec. When the number of voxels increases to 500 and more, the performance drops below 20 frames/sec.

Apart from graphical output, the main bottleneck for the simulation itself is collision detection. As the number of independent objects in the scene increases—either due to a

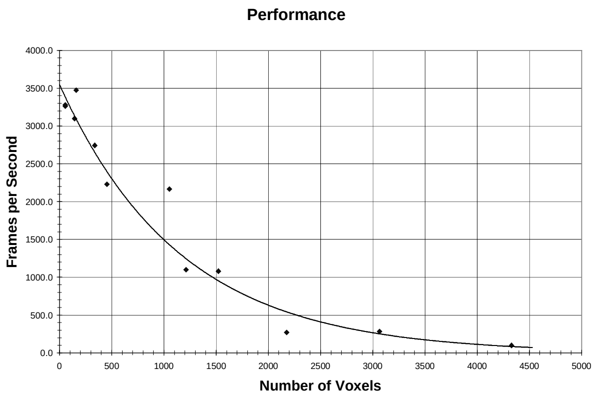

Figure 7. Performance without graphical output.

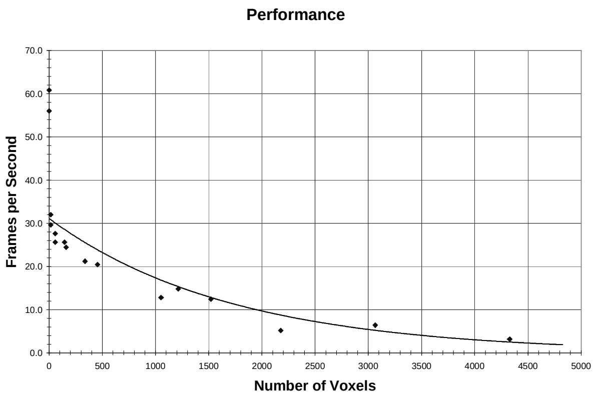

Figure 8. Performance with OpenGL rendering.

large initial set of objects or due to object fragmentation—the frame rate decreases, as more collision checks need to be made.

The simulation tests were performed on an Intel P3 900MHz machine with no hardware optimization for OpenGL.

8. Conclusions and future work

This paper has extended the connected-voxel model used in [MMA99] for the real-time animation of a blast wave impact on solid objects. Visual realism of the animation is achieved primarily through the volumetric representation of an exploding object, resulting in convincing solid debris. Arbitrary voxel shapes permit the creation of objects that are more complex using fewer voxels.

Several simplifications are required to achieve a simulation that runs in real time. The blast wave model assumes that the shock front remains perfectly spherical as it propagates. Using shock front precomputation minimizes the inaccuracy inherent in dynamic simulators using discrete time steps.

The collision response algorithms for both dynamic and static contact between objects are also somewhat imprecise. More accurate contact resolution would adversely affect the frame rate. Certain simplifications, such as the use of pseudo-friction in static contact resolution, are employed to maintain an acceptable simulation speed.

The addition of several key visual cues commonly associated with explosions improves the visual believability of the scene. Each cue is not rigorously physically derived or simulated in the interest of speed. It is sufficient to have a specific cue looking reasonably similar to its real-world counterpart without it being a totally accurate facsimile. Whereas our simulator works on a purely visual basis, it is noteworthy that an observer of an explosion also relies on auditory cues to provide authenticity. Adding immediate explosion sound effects to the simulation would enhance its believability considerably.

The user has a high level of control in most areas of the implementation. The design of an initial object can be customized to a large extent, from the way it looks (using arbitrary voxel shapes) to the way it behaves under a blast wave impact (using non-homogenous link and voxel properties). The level of accuracy in the simulation can also be chosen depending on the user's goals.

The main challenge would be to simulate more complex wave interactions with both objects and the environment in real time. Examples of this are reflection, refraction, Mach stem formation, and vortices [Baker73]. Yngve *et al* [YOH00] model these complex blast wave interactions fairly accurately, but running times for their simulation ranged from several hours to several days.

A related problem is the simulation of shock wave propagation within an object itself. The implementation can also be enhanced with more efficient collision detection to improve simulation speed, and more advanced object fracture modeling (such as [OH99], [SWB00]) to increase accuracy and visual believability.

9. References

[Baker73] Wilfred E. Baker, *Explosions in Air*, University of Texas Press, 1973.

[Baraff94] David Baraff, "Fast Contact Force Computation for Nonpenetrating Rigid Bodies", *SIGGRAPH 94*, pp. 23-34, 1994.

[BW97] David Baraff and Andrew Witkin, *Physically Based Modeling: Principles and Practice*, SIGGRAPH 97 Course Notes, 1997.

[Bashforth98] Byron Bashforth, *Physics-Based Explosion Modelling for Computer Graphics*, M.Sc. thesis, University of Saskatchewan, 1998.

[Brode95] H.L. Brode, "Numerical Solutions of Spherical Blast Waves", Jour. Appl. Phys., 26, 1995.

[COMM94] N. Chiba, S. Ohkawa, K. Muraoka, and M. Miura, "Two-dimensional Visual Simulation of Flames, Smoke and the Spread of Fire", *The Journal of Visualization and Computer Animation*, Vol. 5, No. 1, 1994.

[Dewey64] John M. Dewey, "The Air Velocity in Blast Waves from TNT Explosions", Proc. Roy. Soc., A, 1964.

[KG85] G. F. Kinney and F. J. Graham, *Explosive Shocks in Air*, 2nd Ed., Springer-Verlag, 1985.

[MMA99] Oleg Mazarak, Claude Martins, and John Amanatides, "Animating Exploding Objects", *Graphics Interface 99*, pp. 211-218, 1999.

[MC95] B. Mirtich, J. Canny, "Impulse-based Simulation of Rigid Bodies", *1995 Symposium on Interactive 3D Graphics*, pp. 181-188, 1995.

[NF99] Michael Neff and Eugene Fiume, "A Visual Model for Blast Waves and Fracture", *Graphics Interface 99*, pp. 193-202, 1999.

[OH99] James F. O'Brien and Jessicca K. Hodgins, "Graphical Modeling and Animation of Brittle Fracture", *SIGGRAPH 99*, pp. 137-146, 1999.

[Reeves83] William T. Reeves, "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects", *ACM Transactions on Graphics*, Vol. 2, No. 2, 1983.

[SWB00] Jeffrey Smith, Andrew Witkin, and David Baraff, "Fast and Controllable Simulation of the Shattering of Brittle Objects", *Graphics Interface 00*, pp. 27-34, 2000.

[TF88] D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture", *SIGGRAPH 88*, pp. 269-278, 1988.

[TPBF87] D. Terzopoulos, John Platt, Alan Barr, and K. Fleischer, "Elastically Deformable Models", *SIGGRAPH 87*, pp. 205-214, 1987.

[YOH00] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins, "Animating Explosions", *SIGGRAPH 00*, pp. 29-36, 2000.

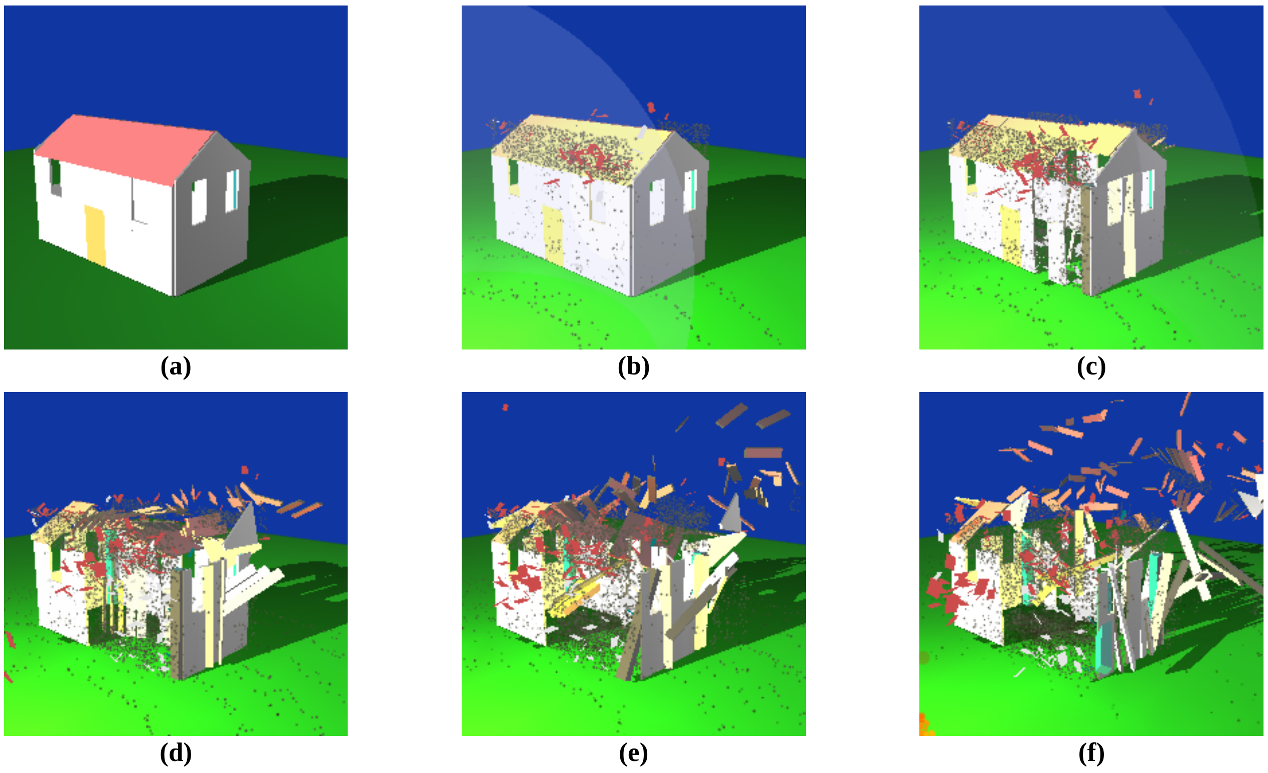

Figure 9. House destruction (single object of ~150 voxels).

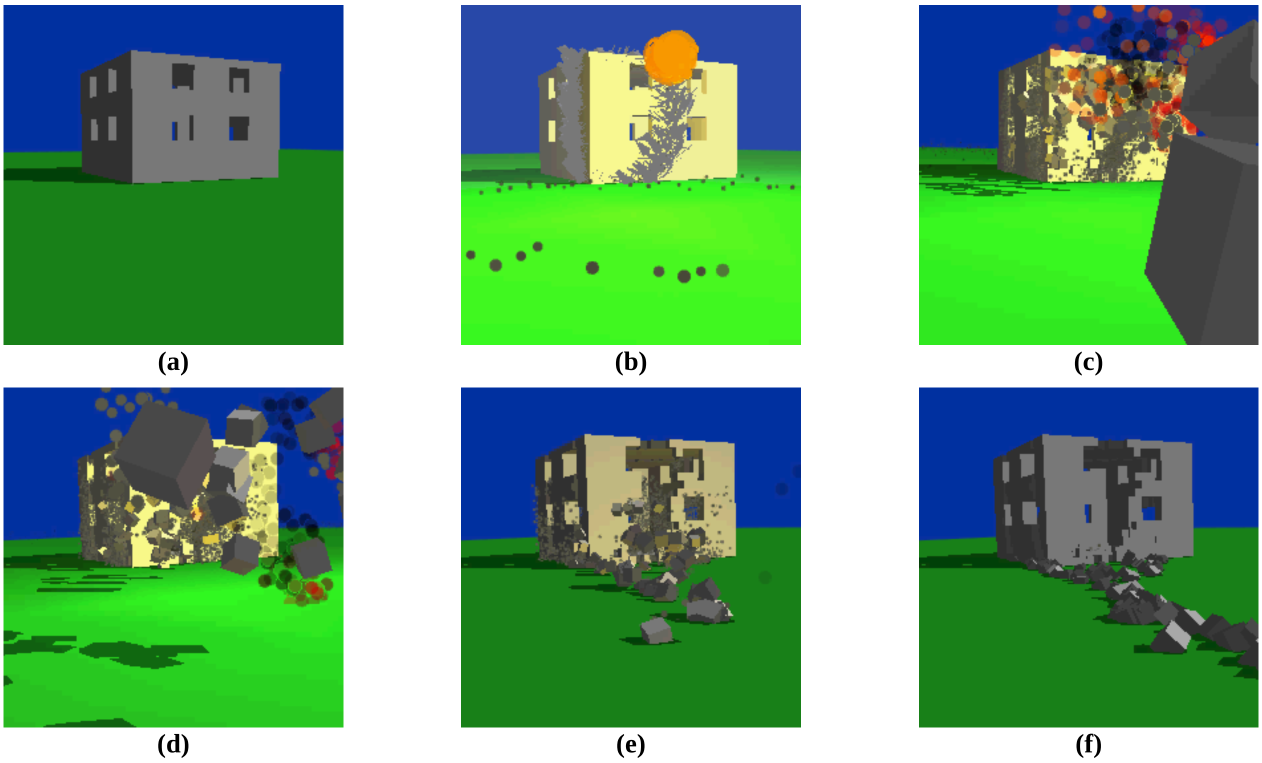

Figure 10. Building damage (single object of ~4300 voxels).

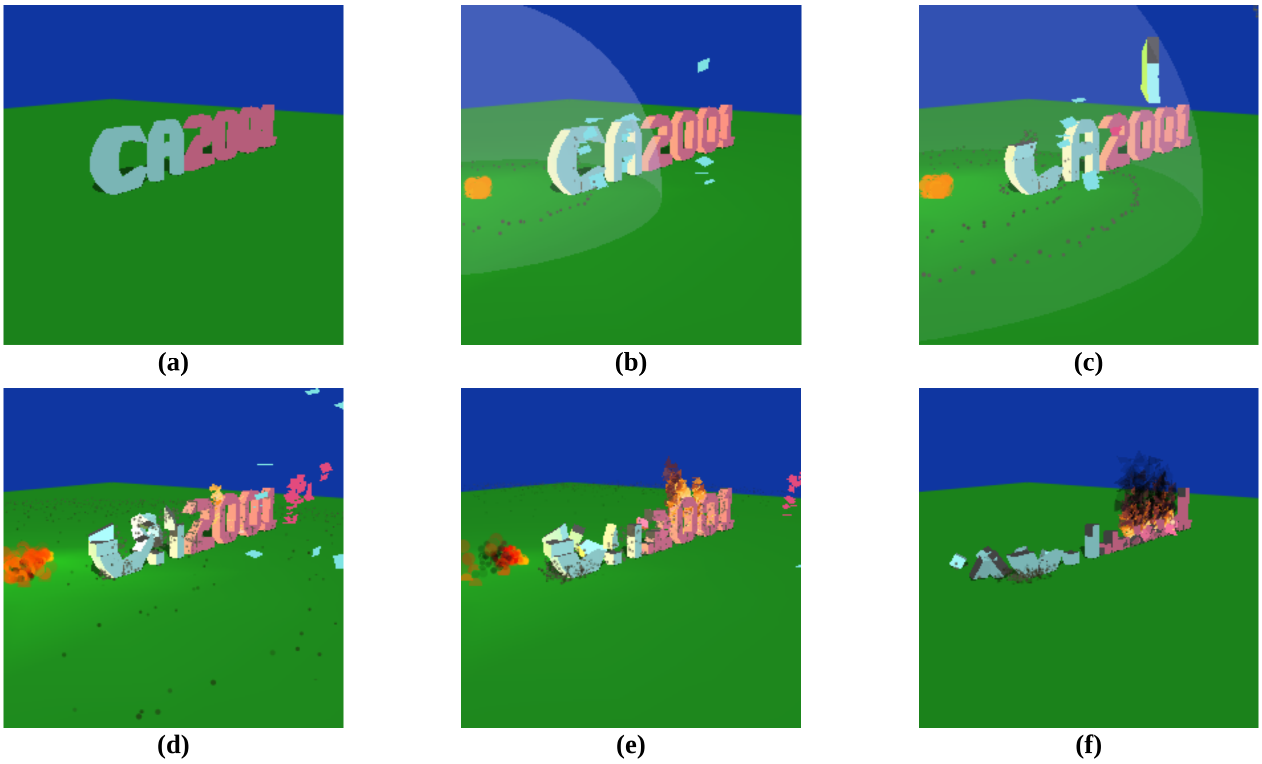

Figure 11. Logo destruction (multiple objects of ~70 voxels).
The "00" in "2001" was specified as more flammable than the other objects.