Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2009)
E. Grinspun and J. Hodgins (Editors)

Real-Time Deformation and Fracture in a Game Environment

Eric G. Parker
Pixelux Entertainment

James F. O'Brien
University of California, Berkeley

Abstract

This paper describes a simulation system that has been developed to model the deformation and fracture of solid objects in a real-time gaming context. Based around a corotational tetrahedral finite element method, this system has been constructed from components published in the graphics and computational physics literatures. The goal of this paper is to describe how these components can be combined to produce an engine that is robust to unpredictable user interactions, fast enough to model reasonable scenarios at real-time speeds, suitable for use in the design of a game level, and with appropriate controls allowing content creators to match artistic direction. Details concerning parallel implementation, solver design, rendering method, and other aspects of the simulation are elucidated with the intent of providing a guide to others wishing to implement similar systems. Examples from in-game scenes captured on the Xbox 360, PS3, and PC platforms are included.

Keywords: Physics engine, game physics, simulation, deformation, fracture, real-time physics, Star Wars: The Force Unleashed.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Animation; Simulation and Modeling [I.6.8]: Types of Simulation—Games; Computing Milieux [K.8.1]: Personal Computing Games—Simulation.

Figure 1: *A character in the game* Star Wars: The Force Unleashed *directs a surge of energy at a closed gate, blasting it open. The motion of the gate and resulting debris was simulated with a real-time finite element system running on an Xbox 360.*

1. Introduction

Techniques for physically based animation have progressed significantly over the last twenty years. Real-time simulations on low-cost machines can now generate results that would have previously taken many hours of computation on high-end workstations. Offline simulations of complex phenomena involving fluids, solids, and their interaction can produce animations that many viewers find indistinguishable from reality. These advances are due in part to the continued march of Moore's law, but they are also a direct result of specialized simulation algorithms developed in the graphics community that have been designed specifically for stability and visual fidelity.

This paper describes a system we have developed for simulating solid deformation and fracture in the real-time setting of a commercial video game such as the one shown in Figure 1. This setting presents a number of strict criteria that must be satisfied before a simulation method can be successfully integrated. Simulation performance must be efficient and predictable so that simulated game content can be reliably integrated into a game level without causing sporadic drops in frame rate or stutters in control response. The simulation cannot blow up or crash because even infrequent crashes can quickly sour a user's experience. These properties should apply regardless how the user interacts with the system and one should never assume that the user will behave in a predictable or even reasonable fashion. Memory usage is an additional concern, particularly for console games. Mechanisms for easy authoring and artistic control should also be provided and should be accessible to typical level designers.

The core of this system is a tetrahedral finite element method that incorporates ideas from several simulation techniques previously published in the computer graphics and computational physics literatures. This combination was selected to provide a solution that is fast, robust to inconsistent input, easily separable, amenable to parallelization, and capable of producing the dynamic visual appearance of deformation and fracture.

SCA 2009, New Orleans, LA, August 1–2, 2009.

Our intended contribution is to describe in detail the design of this system, including aspects of implementation that were specific to making it work robustly in a game context. These details include parallel implementation of the solver and other simulation components, collision handling designed to prevent undesirable vibratory behavior, partitioning methods used to break large problems into a set of smaller ones, and algorithmic changes to minimize wasted CPU time due to memory latency. We also describe several graphical "tricks" we use to improve the overall appearance of the simulated materials and compensate for limitations on the simulation imposed by the real-time requirement.

Some of the techniques we discuss may already be in use internally at a few research labs and companies that specialize in real-time simulation. However, these important implementation details are often omitted from academic publications. Our goal is to provide implementation details and design rationale that will be of benefit to other researchers and practitioners developing similar interactive real-time simulation systems. Stated more colloquially, we want to share some of the inside secrets for achieving robust interactive real-time simulation.

2. Background

The use of simulation-based methods to animate deformable objects has been an active area of graphics research for over twenty years. Key concepts in the area were originally introduced by [TPBF87] and other contemporaneous work. The survey article in [GM97] details much of the early work on deformable modeling, while [NMK*05] surveys recent approaches, including several targeting real-time performance.

In the context of finite element methods, [MDM*02] describe a method for decomposing deformation into separate rigid-body and strain components. The decomposition allowed linear analysis of the deformation independent of rotation, and thus permitted the use of fast solution methods in the context of large deformation. However, their node-centric decomposition produced undesirable ghost forces. Further work in [MG04] and [EKS03] improved upon the method by using an element-based approach. [ITF04] robustly deals with flat and inverted elements by diagonalizing the 3×3 deformation matrix using SVD and this approach was adopted for real-time use in [CAR*09].

In the finite element literature, this basic notion of separating out rotation is known as a *corotational formulation* (*e.g.* [Fel07]). Work by [BH79] first introduced the term corotational in 1979. Initially, the idea was used to remove an overall rigid body motion from the entire domain. However, [NOR91] described an element by element approach that acts as a filter around an existing finite element library, effectively extending the domain of application for the library into the large deformation regime.

Collision detection for deformable objects remains an active area of research. The survey paper [TKZ*04] covers recent techniques including bounding-volume hierarchies, distance fields, stochastic methods, and spatial subdivision.

The related problem of collision response and persistent contacts for deformable bodies is another active research area. Contact methods can be classified into two main categories: penalty and constraint. Penalty methods (*e.g.* [MW88]) produce a contact force proportional to some characteristic of the contact, often penetration depth. Errors or discontinuities in determining a good contact depth and normal can lead to high-frequency bouncing (*i.e.* "chatter"), non-conservative rebound, and other difficulties. (See discussion in [HFS*01].) An additional problem is choosing the magnitude for the penalty stiffness constant such that it minimizes overlap while keeping the system stable and well conditioned (*e.g.* [Fel07]). However, penalty methods remain popular due to their ease of implementation and compatibility with nearly any type of solver.

Constraint methods attempt to prevent any overlap by solving for the exact force required to prevent interpenetration. This topic has been explored thoroughly in the context of rigid bodies [Bar89, Bar94, GBF03, KEP05, Erl07]. For soft bodies, [BW92] provides a nice overview of the geometry of flexible body collision and contact. Constraint-based techniques for handling collisions with deformable bodies can be found in [BW98,JP99,BFA02,HSO03,SBT07, HVT08, OTSG09].

Destruction is an important feature that fits well into the settings of many computer games. Both [OH99] and [OBH02] describe a method for simulating fracture propagation based on a stress map derived using the finite element method. Work presented in [ITF04] uses an alternative element replication technique (see [MBF04]) that aims to reduce the impact of sliver elements. In a real-time setting, [SWB00] describe a method using breakable constraints across tetrahedral faces, and [MMDJ01] performs a quasi-static finite element analysis to determine how object should break during an impact. Similar work in [BHTF07] also computes fracture based on a quasi-static analysis and extends the idea to include ductile deformation. For interactive simulations, [MG04] compute the maximal tensile stress for each tetrahedra, and then separate the mesh at one of the vertices (randomly) of tetrahedra which exceed a threshold.

Although methods based on quasi-static analysis, such as [SWB00, MMDJ01, BHTF07], have the potential to be cheaper than a fully dynamic deformation simulation, we found that fully dynamic fracture propagation produces more energetic, vibrant, and realistic looking results. Prior to breaking, the material deforms and stores elastic energy. When fracture occurs, the new degrees of freedom allow the material to move so that the stored elastic energy is converted to kinetic. This released energy dramatically changes the resulting fracture patterns and motion of the material. In contrast, methods that ignore stored elastic energy, such as quasi-static finite elements or rigid bodies joined by constraints, produce deadened motions where the objects look like they simply fell apart.

The use of physical simulation in video games follows a long tradition. The game *Space Wars*, considered by many

to be the first video game, required players to account for the gravity of a central sun and to keep track of their ship's momentum. Other early games including Pong, Asteroids, and Lunar Lander all include physical simulation as a key aspect of the game play. Unfortunately, the academic literature on physics in computer games is somewhat sparse. Two books that specifically cover game physics are [Ebe04] and [ESHD05]. The text by Eberly contains chapters on rigid and soft bodies, collision detection, shader programming, ODE integration, linear complementarity, and quaternions. The text by Erlben and coauthors covers the finite element method, collision detection, kinematics, rigid bodies, and contains a basic math primer. [MSJT08] provides a tutorial covering soft bodies, rigid bodies, and fluids. Erwin Coumans maintains a forum as part of his Bullet physics library [Cou08].

Although not targeted specifically at real-time applications, the study in [HGS*07] presents a detailed analysis of parallelized simulation code running on a simulated multicore architecture. Many of the observations from this study concerning bottlenecks and memory access patterns are also applicable on a smaller scale to parallelized real-time simulation code.

3. Finite Element Formulation

We use a finite element method to model simulated materials that the user can interact with. The materials can deform and fracture, and can be assigned a wide range of material properties. The objective is not to model materials in a predictive fashion, but instead to give content designers the freedom to specify virtual materials with properties that match technical direction for a given setting.

The finite element method we use employs tetrahedral elements with linear basis functions so that the strain field is piecewise constant over the mesh. This type of element has found widespread use in the graphics literature, for example [OH99, MDM*02, MG04, ITF04, BWHT07, CAR*09], and is described in detail in most finite element texts, for example [CMPW01].

Each element is defined by four nodes with reference positions $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, $\boldsymbol{u}_3$, and $\boldsymbol{u}_4$. Let $\boldsymbol{D}_u$ be a 3×3 matrix with columns $\boldsymbol{u}_2 - \boldsymbol{u}_1$, $\boldsymbol{u}_3 - \boldsymbol{u}_1$, and $\boldsymbol{u}_4 - \boldsymbol{u}_1$, the element basis matrix is then $\boldsymbol{\beta} = \boldsymbol{D}_u^{-1}$. Each node also has its current position and velocity in world space denoted respectively by $\boldsymbol{x}_i$ and $\boldsymbol{v}_i$. The matrices $\boldsymbol{D}_x$ and $\boldsymbol{D}_v$ are defined as $\boldsymbol{D}_u$ but with the $\boldsymbol{x}_i$ or $\boldsymbol{v}_i$ in place of the $\boldsymbol{u}_i$. The deformation gradient $\boldsymbol{F}$ and its time derivative $\boldsymbol{G}$ are then given by

$$\boldsymbol{F} = \frac{\partial x}{\partial u} = \boldsymbol{D}_x \boldsymbol{\beta} \quad \text{and} \quad \boldsymbol{G} = \frac{\partial v}{\partial u} = \boldsymbol{D}_v \boldsymbol{\beta} \quad . \tag{1}$$

Cauchy's infinitesimal strain tensor, $\boldsymbol{\varepsilon} = 1/2(\boldsymbol{F} + \boldsymbol{F}^\mathsf{T}) - \boldsymbol{I}$ is cheap to compute directly from $\boldsymbol{F}$, scales linearly with deformation, and its Jacobian with respect to the $\boldsymbol{x}_i$ is constant. All of these properties are desirable, but unfortunately $\boldsymbol{\varepsilon}$ is not invariant with respect to rotation, and so creates objectionable artifacts when applied to large deformations. To compensate, we use a corotational method that factors out the rotation on a per-element basis. Corotational methods are often discussed in finite element texts (*e.g* [CMPW01]), and variations of the concept have been used in the graphics literature, [MDM*02, EKS03, MG04, ITF04]. Like [EKS03] and [MG04] we perform a polar decomposition of $\boldsymbol{F}$ into $\boldsymbol{F} = \boldsymbol{Q}\boldsymbol{A}$ where $\boldsymbol{Q}$ is orthonormal and $\boldsymbol{A}$ is symmetric.

Elements with degenerate or inverted world configurations would normally cause the simulation to "blow up" which would be unacceptable in a game setting. In [ITF04] they use a nonlinear strain metric and fully diagonalize to explicitly correct the offending singular values of $\boldsymbol{F}$. We have found a cheaper, albeit less accurate, solution that works well in practice is to detect when a tetrahedron has fallen to below 6% of its reference volume, and in those cases switch to a QR decomposition of $\boldsymbol{F}$. With this decomposition, $\boldsymbol{Q}$ is still orthonormal, and now guaranteed to have positive determinant, but $\boldsymbol{A}$ is upper triangular rather than symmetric. This switch prevents inverted elements from disrupting the entire simulation, but QR decomposition produces a coordinate-biased rotation and it will change discontinuously from the $\boldsymbol{Q}$ produced by the polar decomposition. However, inverted elements typically only occur during violent motions which mask any induced artifacts. The merits of QR decomposition for handling element inversion have also been noted by [NPF05].

Once we have $\boldsymbol{Q}$ computed, we can factor it out of the deformation gradient by replacing $\boldsymbol{F}$ with $\tilde{\boldsymbol{F}} = \boldsymbol{Q}^\mathsf{T}\boldsymbol{F}$ and the corotational strain is given by $\tilde{\boldsymbol{\varepsilon}} = 1/2(\tilde{\boldsymbol{F}} + \tilde{\boldsymbol{F}}^\mathsf{T}) - \boldsymbol{I}$. Using a linear isotropic constitutive model, the element stress is given by $\boldsymbol{\sigma} = \lambda \mathrm{Tr}(\tilde{\boldsymbol{\varepsilon}})\boldsymbol{I} + 2\mu\tilde{\boldsymbol{\varepsilon}}$.

The elastic force exerted by the element on node i is $\boldsymbol{f}_i = \boldsymbol{Q}\boldsymbol{\sigma}\boldsymbol{n}_i$, where $\boldsymbol{n}_i$ is the area-weighted outward normal for the face opposite node i expressed in reference coordinates. The Jacobian of $\boldsymbol{f}_i$ with respect to the position of node j is the 3×3 matrix given by

$$\boldsymbol{J}_{ij} = -\boldsymbol{Q}(\lambda \boldsymbol{n}_i \boldsymbol{n}_j^\mathsf{T} + \mu(\boldsymbol{n}_i \cdot \boldsymbol{n}_j)\boldsymbol{I} + \mu \boldsymbol{n}_j \boldsymbol{n}_i^\mathsf{T})\boldsymbol{Q}^\mathsf{T} \; . \tag{2}$$

The core of (2), consisting of inner and outer normal products, does not change as the mesh deforms and can be cached for each element. If the individual 12×12 stiffness matrix for a single element were required it would be assembled from the 16 3×3 $\boldsymbol{J}_{ij}$ blocks for all node pairs. Note that because $\boldsymbol{J}_{ij} = \boldsymbol{J}_{ji}^\mathsf{T}$, the element stiffness matrices are symmetric.

Following [OBH02] and [MG04], we use an additive plasticity model that separates the total strain into separate elastic and plastic components so that we have $\tilde{\boldsymbol{\varepsilon}} = \boldsymbol{\varepsilon}^P + \boldsymbol{\varepsilon}^E$. The total strain, $\tilde{\boldsymbol{\varepsilon}}$, is computed as described above, and $\boldsymbol{\varepsilon}^E$ takes the place of $\tilde{\boldsymbol{\varepsilon}}$ in force computation. Initially $\boldsymbol{\varepsilon}^P = 0$. If $||\boldsymbol{\varepsilon}^E||$ (Frobenius norm) exceeds a material yield threshold, y, then we update $\boldsymbol{\varepsilon}^P$ according to

$$\boldsymbol{\varepsilon}^P := \boldsymbol{\varepsilon}^P + c\frac{||\boldsymbol{\varepsilon}^E|| - y}{||\boldsymbol{\varepsilon}^E||}\boldsymbol{\varepsilon}^E \tag{3}$$

where c is a material parameter that controls creep rate. If

$||\boldsymbol{\varepsilon}^P||$ exceeds the material's maximum plastic strain, n, we clamp the plastic strain by setting $||\boldsymbol{\varepsilon}^P|| := \boldsymbol{\varepsilon}^P n/||\boldsymbol{\varepsilon}^P||$. As pointed out by [ITF04], this additive plasticity model is only reasonable for small strains. However we have found that it works well with the corotational formulation we use and does not add significant cost to the simulation.

We follow the now common practice of divorcing the graphical representation of an object from the mesh used to simulate it. For each object we construct a tetrahedral mesh that encloses the majority of the object's graphical representation. Each vertex of the graphical model is assigned to the tetrahedron that encloses it (or that it is closest to) and we store the barycentric coordinates of the vertex relative to that tetrahedron. As the mesh nodes move, the vertex positions are updated using the barycentric weights.

In modern video games, the graphical mesh may consist of many thousands or even millions of triangles. Typically these meshes are stored within the GPU's memory and are not readily available for the CPU to operate on. We perform deformation within the GPU using shader programs. The graphical mesh is stored using the vertex barycentric coordinates within the owning tetrahedron instead of the usual Cartesian coordinates. Historically, mesh deformation on GPUs is done for character skinning using dedicated bone matrices. The number of available hardware matrices is typically limited to 256 or less. To support larger simulation meshes, we instead encode the deformed simulation mesh nodal positions into a texture map stored in the GPU's main memory. Similar to [Mic05], the vertex positions are computed from blended texture references in a vertex shader. Recent GPUs support texture fetches from vertex shaders, but for older GPUs we must perform deformation on the CPU.

One difficulty often encountered during mesh generation is the presence of a few badly shaped elements. These elements are characterized by $\boldsymbol{\beta}$ matrices with large singular values. To facilitate content authoring we adopt the approach from [IO06] that modifies the matrices to clamp the magnitude of their singular values. This approach, combined with the simulation's tolerance of inverted elements, allows level designers to work without excessive concerns regarding mesh quality.

4. Matrix Assembly and Time Integration

With a real-time system, time integration presents several opportunities for difficulties. The bulk of the simulation's compute time is spent integrating the system forward, so inefficiencies here can easily destroy overall performance. Time integration is also where stability issues arise that can cause the simulation to blow up, oscillate in a physically unrealistic way, or otherwise produce unacceptable results. For these reasons we have focused a significant amount of attention on tuning and parallelizing the code responsible for time integration and related tasks.

Multithreaded code offers significant advantages over serial implementations, even when running on a single processor. Many operations are applied independently to each element or node and can be trivially parallelized with performance gains nearly linear in the number of available processors. Additionally, on modern architectures memory access can be a more significant bottleneck than CPU contention. With CPU features such as hyperthreading, one thread can utilize the CPU while another is stalled waiting for a memory fetch. Modern CPUs often include some form of vector instructions such as SSE or AltiVec. Ideally the compiler would make optimal use of these vector instructions, but results obtained in practice leave something to be desired. Explicitly coding key parts of the code to take advantage of vector instruction can result in large gains, particularly if the memory layout of one's data structures is adjusted to facilitate loads to the vector registers.

To facilitate parallelism our meshes are broken up into *islands* that persist across multiple timesteps. Individual islands may be in one of three states: *live*, *asleep*, or *kinematic*. The islands are formed at runtime in a greedy fashion based on nodal adjacency. Nodes within an island are partitioned using METIS [KK99]. This improves cache locality for live islands (which are always maximal), and provides boundaries for breaking up kinematic and sleeping islands based on factors such as available memory and target island size.

Islands are initially asleep and are not simulated. They are awoken if they are struck by another object or if a force above some threshold is applied to one of their nodes. Islands whose nodes are below a velocity threshold for multiple timesteps are put to sleep. Kinematic islands follow an imposed or scripted behavior and are not simulated. Islands are dissolved if fracture occurs within the island, or if some of it's nodes change state. Groups of islands that are in contact form a *composite-island*. Composite-islands are not persistent since contacts are largely a transient phenomenon. Each composite-island is a single subsystem that can be integrated separately.

Our simulations use a linearized backward Euler integrator. As pointed out by [BW98], this semi-implicit integrator can behave stably even when given very stiff materials. Nonlinear strain metrics can reduce these stability benefits, but the corotational linear strain works well with this integration scheme. Many researchers have pointed out that backward Euler can exhibit excessive damping. However, we found this behavior acceptable because the scenarios being modeled in our game context are violent situations where external forces are actively driving the simulation. In these contexts, the damping behavior is actually somewhat desirable because we don't want objects to continue bouncing around for a prolonged period once the interaction has stopped, and the good stability characteristics are mandatory.

The differential equation that describes each island to be integrated is of the form

$$\boldsymbol{M}\boldsymbol{a} + \boldsymbol{C}\boldsymbol{v} + \boldsymbol{K}(\boldsymbol{x} - \boldsymbol{u}) = \boldsymbol{f} \tag{4}$$

where $\boldsymbol{x}$, $\boldsymbol{v}$, and $\boldsymbol{a}$ are the concatenated vectors of positions,

velocities, and accelerations for all the nodes within the island, $\boldsymbol{f}$ is the vector of external forces acting on the nodes, and $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{K}$ are respectively the mass, damping, and stiffness matrices. The mass matrix is lumped with diagonal entries corresponding to the mass of each node. We use Raleigh damping where $\boldsymbol{C}$ is a linear combination of $\boldsymbol{M}$ and $\boldsymbol{K}$.

The stiffness matrix, $\boldsymbol{K}$, has a sparse structure where nodes that belong to the same tetrahedron will correspond to a 3×3 block of non-zero values in $\boldsymbol{K}$ that are the Jacobian of the force on one node with respect to the position of the other node. Since the edge between two nodes generally belongs to multiple tetrahedra, the system Jacobian entry is the sum of the per-element Jacobian entries given by (2).

We store $\boldsymbol{K}$ as an array of *noderows* where each noderow holds the three consecutive rows in $\boldsymbol{K}$ corresponding to a given node. The noderows are stored as row-compressed lists [Pis84] with each entry being a 3×3 block of floats. On machines that support vector instructions, these blocks are padded to facilitate fast loads of their rows into the vector registers. Although $\boldsymbol{K}$ is symmetric, we store the full matrix redundantly as doing so resulted in code that was both simpler and faster. This structure is optimized for multiplying $\boldsymbol{K}$ by a vector. Because the matrix is symmetric, there is no need to support fast multiplication by the matrix's transpose.

Assembling $\boldsymbol{K}$ can take a substantial amount of time so we parallelize the computation. First the $\boldsymbol{J}_{ij}$ blocks are computed for each element. These computations are independent so groups of elements can be dispatched to separate threads. Second, groups of noderows are assigned to threads and each noderow is assembled independently of the others.

Once we have $\boldsymbol{K}$ we can integrate (4) forward over Δt to obtain new positions and velocities, $\boldsymbol{x}^+$ and $\boldsymbol{v}^+$. Let $\boldsymbol{a}^+$ be the acceleration at the end of the timestep, then we have $\boldsymbol{v}^+ = \boldsymbol{v} + \Delta t \boldsymbol{a}^+$ and $\boldsymbol{x}^+ = \boldsymbol{x} + \Delta t \boldsymbol{v}^+$. Substituting into (4) and rearranging gives us

$$\left(\boldsymbol{M} + \Delta t \boldsymbol{C} + \Delta t^2 \boldsymbol{K}\right) \boldsymbol{v}^+ = \Delta t \boldsymbol{f} + \boldsymbol{M}\boldsymbol{v} - \Delta t \boldsymbol{K} (\boldsymbol{x} - \boldsymbol{u}) \tag{5}$$

which can then be solved for $\boldsymbol{v}^+$, allowing us to compute $\boldsymbol{x}^+$. We solve (5) using the conjugate gradient method, which is well described in [She94]. The main expense of the method comes from matrix-vector multiplications involving $\boldsymbol{K}$ and $\boldsymbol{K}^\mathsf{T}$, for which our sparse data structure well suited.

Generally we will have multiple live islands at any given time. These islands can be computed independently. It is also common to have a small number of islands that are significantly larger than the others. We use the heuristic that an island is considered "large" if contains at least sixty nodes and the number of nodes in the island is more than one quarter the total number of nodes in all live islands.

We use two complementary strategies to parallelize computing the solution to (5). Regular islands will be each solved with a serial implementation of the conjugate gradient method, but multiple regular islands will be dispatched simultaneously to different threads. The large islands will be solved using a parallelized implementation of the conjugate gradient method. The parallelized implementation distributes the row-vector multiplies over multiple threads. Both the serial and parallel implementations take advantage of the block structure of the noderows to compute all three row-vector multiplies for a given node simultaneously.

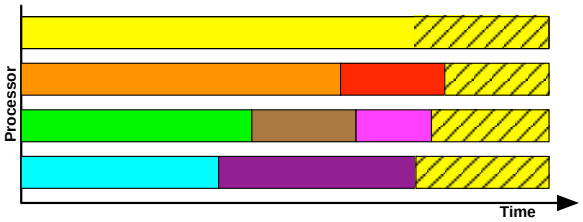

Figure 2: *Pending jobs are scheduled on available threads in order of expected compute time. When a thread becomes available and no jobs remain to be scheduled, the free thread is assigned to assist with any in-progress large jobs and that large job switches to a parallel solver.*

Islands are dispatched to threads in a greedy fashion where larger islands are dispatched first. When a thread becomes available, the next largest pending island is assigned to it. If no islands are waiting for assignment, then any idle threads are assigned to assist in parallel solving of any large islands that have not yet completed. This procedure is illustrated in Figure 2.

Conjugate gradient iterations for a given island stop once a relative error of 0.001 has been reached. We also limit the number of iterations to a predefined maximum. We have found that it is convenient to allow the content designer direct access to set the maximum number of iterations as if it were another material parameter.

5. Fracture

The fracture algorithm we use is a simplified version of the algorithm that appears in [OH99] and [OBH02]. The main difference is that we have disabled splitting individual tetrahedra. Although the remeshing process is quite cheap and our solver is not adversely effected by the quality of the split elements, the difficulty is that splitting elements causes the number of tetrahedra in a scene to grow in a way that content designers cannot easily plan for. In an environment where maintaining a specified framerate is mandatory, allowing the number of tetrahedra in a scene to grow in an unpredictable fashion is not desirable. Other algorithms that replicate tetrahedra rather than splitting them, for example [MBF04], would raise similar objections.

The fracture algorithm does not necessarily run at every timestep and it does not run at all for nodes that are not part of materials that have been tagged as breakable. When it is run, we gather the forces on each node to compute the separation tensor as described in [OH99]. This process is easily parallelizable on a per-node basis.

Once the 3×3 separation tensors are available for the nodes, we dispatch multiple threads to compute their eigenvalues. When we find a node with a positive eigenvalue that exceeds the material's toughness parameter, we set the split plane to be the plane normal to the corresponding eigenvector passing through the node, and we replicate the node. We then reassign the tetrahedra that where attached to the original node to either the original or new node depending on whether the tetrahedron is predominantly on the positive or negative side of the split plane. This procedure is identical to [OH99] except that we skip the step of splitting tetrahedra that straddled the plane.

Ideally this process would proceed from largest eigenvalue to smallest and include the residual propagation and updating the separation at nodes in the star of elements effected by a previous fracture as done in [OH99]. However, to facilitate efficient parallelization, we examine nodes in a random order, we do not perform residual propagation, and we do not update the separation of nearby nodes. We also set an arbitrary limit on the number of fracture events that we will allow in any given timestep.

Unfortunately, splitting only on the existing tetrahedral mesh boundaries tends to produce unappealing looking fracture surfaces. Rather than looking as if a material like stone or wood has been torn apart, the materials look as if they were constructed from artificial blocks with triangular faces. To alleviate this problem we extend the method we described previously of embedding a higher resolution graphical model in the mesh. When artists generate the graphical representations that will be embedded in the finite element mesh for rendering, they can also designate how the dynamically created fracture surfaces should appear.

The surface appearance is specified by breaking the embedded geometry up into many pieces that are significantly smaller than the tetrahedral elements. We call these pieces *splinters*. The splinters are rendered as embedded surfaces, but we also create an association between each splinter and a single tetrahedron that contains the splinter's centroid. During fracture when we create new boundary faces in the mesh (the fracture surfaces) we test to see if any splinter vertices belong to a tetrahedron that is no longer connected to the tetrahedron that contains the splinter's centroid. If this condition exists, we de-associate the vertex from the disconnected tetrahedron and re-associate it with the nearest connected one. (See example in Figure 3.)

The splinters allow artistic control over the appearance of the object as it breaks. This control is particularly important for materials like wood that should fracture according to the material's underlying anisotropic structure. Splinters will also mask the appearance of individual tetrahedra so that the fracture surfaces look more appropriate for the material being simulated. For some types of materials, such as brick or wood, we have built tools that take an object and automatically subdivide it in to appropriately shaped splinters. For other materials, an artist may elect to manually divide it using standard modeling software.

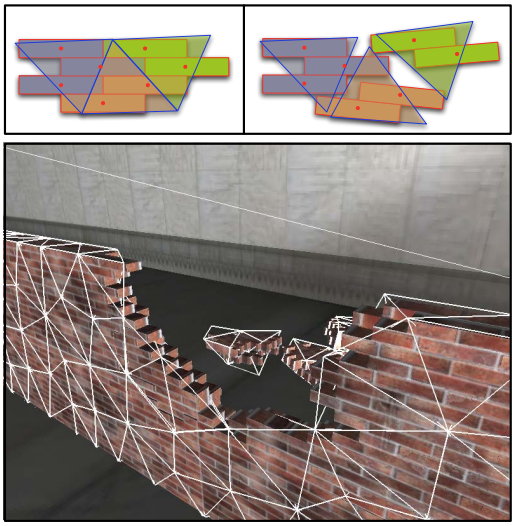

Figure 3: *The apparent geometric detail of the simulated objects is enhanced by embedding higher-resolution, textured polygonal surfaces in a coarse tetrahedral finite element mesh. To preserve the appearance of high detail during fracture, discrete units of geometry called splinters are kept intact.*

This splinter approach superficially resembles scoring the object into static pieces that will break apart in a predetermined way. However, the key difference is that the splinter approach still allows fractures to propagate through the mesh based on computed dynamic behavior. The splinters only serve to add realistic detail that is finer than the resolution of the simulation mesh.

Fracture is suppressed if it would create a connected component with fewer than three face-connected tetrahedra. We do so to prevent unattractive fragments that are recognizable as single tetrahedra. If a fracture would leave two components only connected by a single edge, creating a floppy hinge joint, or single node, creating a ball joint, we also separate the edge or node. For some materials like concrete, we create graphical particle effects distributed over the new fracture surfaces.

6. Collision Detection and Response

Collision detection and response for unstructured finite element methods can easily consume more time than all other simulation components combined. Additionally, algorithms for accurate collision resolution can impose timestep limitations that are substantially more restrictive than those otherwise imposed by the integrator. To prevent unacceptable slowdown of our simulation system we have focused on implementing a collision subsystem that sacrifices accuracy in favor of speed and stability.

To justify this trade off we observe that game contexts requiring the type of simulation we are developing are not typically contexts where subtle collision errors are likely to be

Figure 4: *A wooden panel being smashed. The first four images show frames from a sequence where a wooden panel is being battered to pieces. The inset image on the far right shows the concept art that motivated the design of the material.*

noticed. Furthermore in situations involving many dynamic collisions (for example the collapse of a large structure) we expect users to be more forgiving of inaccurate collision response than they would be of a drop in framerate or simulation crash. We also believe that some types of inaccuracies are more acceptable than others. Incorrect friction or slightly penetrating objects are unlikely to be noticed, while objects that chatter on the ground are likely to be objectionable.

Collision detection starts with an initial broad-phase application of the "sweep and prune" algorithm over the island bounding boxes. (See [Bar92] and [CLMP95] for detailed descriptions of the algorithm.) This algorithm maintains a persistent list of contacts that is incrementally updated each frame. This property is particularly useful in keeping self-collision efficient as it avoids rediscovering at each timestep numerous false collisions that occur due to mesh adjacencies.

Once the broad-phase has located potential collisions, a narrow-phase tests individual triangle pairs for intersection. Only the exterior triangle faces of the tetrahedral mesh are tested. The triangle-pair tests are computed in parallel using a vectorized version of the OpCode library [Ter01].

When two intersecting tetrahedra are located, we follow the approach in [O'B00] and compute the polyhedron defined by their overlap. The separating force will be applied at the polyhedron's center of mass, $\boldsymbol{c}$. The direction of the force, $\boldsymbol{r}$, is given by the sum of area-weighted normals for the faces contributed by either one of the tetrahedra.

Our initial approach was to add an implicit penalty force to the system with magnitude proportional to the volume of the overlap. We did this by linearizing the penalty force with the direction fixed and approximating the change in volume with respect to node position using only the terms due to motion in the force direction. This force was represented as a temporary element involving the eight nodes and it contributed 64 3×3 blocks of non-zeros to $\boldsymbol{K}$. (This type of element is often referred to as a "mortar" element in the finite element literature [CMPW01].) This approach worked quite well *nearly* all the time. Unfortunately, situations would very rarely arise where objects would align in some very particular configurations that resulted in disturbing jitter. For an offline system this infrequent annoyance would not be an issue. However, in a game setting the simulation must run continuously while unsupervised and even infrequent jitter was determined to be objectionable.

To eliminate this problem, we chose to implement collision response using only damping forces. When two tetrahedra are found to overlap each other we compute the overlap volume, force direction, and force application point as described above. We then compute barycentric weights for the position $\boldsymbol{c}$ with respect to each of the two colliding tetrahedra, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$. Using the node velocities of the two tetrahedra, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, we compute the relative approach speed, scaled by the magnitude of $\boldsymbol{r}$, as $s = \boldsymbol{r} \cdot (\boldsymbol{b}_1^\mathsf{T}\boldsymbol{v}_1 - \boldsymbol{b}_2^\mathsf{T}\boldsymbol{v}_2)$ and we set the response force to be proportional to s in the direction $\hat{\boldsymbol{r}}$, where $\hat{}$ denotes normalization. The force is applied at $\boldsymbol{c}$ so that the force is distributed to the nodes of the two tetrahedra according to the barycentric weights of $\boldsymbol{c}$. If we treat $\boldsymbol{c}$ as a constant, this force is linear in the node velocities and can be treated implicitly by including an extra 64 3×3 blocks of non-zeros in the damping matrix. The block between nodes i and j is $\phi\hat{\boldsymbol{r}}\boldsymbol{r}^\mathsf{T} b_i b_j$ where b_i and b_j are the relevant barycentric weights with appropriate sign, and ϕ is the penalty coefficient. These terms preserve the symmetry of the damping matrix. In practice this force behaves well with the implicit integrator and ϕ can be set quite high without inducing stability problems.

We use a fixed-sized timestep for predictable real-time performance, with the resulting drawback that fast-moving or small objects could potentially pass through each other or find themselves in a deeply embedded configuration. Damping forces alone cannot correct existing overlap between objects, so it is important to detect contact as soon as possible. An approximate form of continuous collision detection is employed to help reduce this overlap error. Objects that move more than $1/6$ their bounding box diagonal per timestep are considered "fast". The positions of fast objects are linearly interpolated using a discrete number of substeps, chosen such that the object moves less than $1/6$ of its bounding box diagonal during each sub-step. Collision detection is performed after each sub-step, and the time of first contact for the object is saved. Fast objects for which collisions were detected are arbitrarily moved to their interpolated location where the first collision was detected. This procedure is clearly only approximate, but it does help to reduce the most egregious overlap and tunneling errors.

Friction is implemented with a hybrid system in which dynamic friction is handled implicitly while static friction is

Figure 5: *Images captured from glass explosion scene. These images are from a scene where an explosive force ruptures a glass detention unit. The walls of the enclosure are formed by glass panes held in metal frames.*

computed explicitly. The justification for this model is that static friction is only needed when the simulation is nearing equilibrium, and large fluctuations in frictional forces that would cause instability are less likely to occur. Our dynamic friction model acts solely as a damping force and is computed as part of the implicit solve, so it is inherently stable and cannot lead to jitter.

Similar to the contact normal force used to prevent interpenetration, we compute the dynamic force proportional to the tangential velocity $\boldsymbol{w} = (\boldsymbol{b}_1^{\mathsf{T}}\boldsymbol{v}_1 - \boldsymbol{b}_2^{\mathsf{T}}\boldsymbol{v}_2) - s \cdot \boldsymbol{r}$. In the Coulomb friction model, the tangential force *should* be proportional to the normal force $F_f <= \mu F_n$. However, we separate the two and approximate the dynamic friction force magnitude using a heuristic because linking them would lead to a non-symmetric K requiring a more sophisticated, and likely slower, solution method. This simplification leads to the same set of 64 3×3 blocks of non-zeros as for the normal force, making it very simple to modify the normal force computation to account for dynamic friction.

The normal force heuristic for dynamic friction is computed as $F_n = M_{island} \times G \times \mu \times r_y \times \alpha$ where M_{island} is the mass of the composite-island, G is the force of gravity, μ is the Coulomb coefficient of friction, α is the fraction of volume for this contact with respect to the total volume of all contacts in the composite-island, and r_y is the y-component of the contact normal direction as described above. This heuristic approximates the equilibrium normal force magnitude for objects under the influence of gravity.

We compute an imprecise static friction using a software-based PID controller [Sel01]. Static friction is enabled once the magnitude of the tangential velocity drops below a threshold value. The PID constants were derived from an optimization procedure on a large number of objects falling down a hill using adaptive simulated annealing [Ing96]. Static friction is reset once contact is broken, or the velocity once again rises above the threshold value.

7. Other Implementation Issues

Rigid-body dynamics are a commonly used simulation method in many current computer games. Although finite element simulation could replace rigid-body simulation in most applications, game designers are already familiar with rigid-body tools and so new simulation methods must be able to interoperate with rigid-bodies. Most current rigid body systems use an impulse-based solver requiring multiple iterations to converge. Interfacing with this type of solver requires computing the deformation response to each impulse for each iteration. For a rigid body this response can be computed analytically [Ebe04], but for finite element systems it requires a full solve.

Instead, we use a force-based approach where the contact reaction force on a proxy of the rigid body is computed, and then applied to the rigid body system as an explicit force. The rigid body system is then stepped and the process repeats. This is a type of interleaved hybrid simulation system [BW97] that requires multiple frames to converge towards a reasonable solution. It is limited to small stacks, or contact with soft objects where the resulting contact errors are typically not objectionable. While not ideal, this method was sufficient for the small numbers of interactions with rigid bodies that we have encountered.

With the finite element method it is easy to modify the system to account for kinematically driven animations such as elevators, sliding doors, or walking characters. In each of these cases, some or all of the nodes of the mesh will have prescribed velocities which should not be altered by the solver. Modifying the system to account for the motion of these nodes follows a cookbook procedure. Equation 5 reduces to a single linear matrix equation where $\boldsymbol{v}$ are the velocities to be solved for, and $\boldsymbol{b}$ is a column vector of constants: $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{b}$. Each row of this system represents an equation that must be satisfied. Prescribed values of $\boldsymbol{v}$ produce a constant value on the left hand side of these equations. This value may be moved to the right hand side by simply subtracting it from both sides. We then just strike the rows and columns corresponding to prescribed degrees of freedom to return the system to nonsingular status.

8. Results and Discussion

The system we have described has been implemented as a separate physics engine that can be integrated with other game components. It is currently being sold commercially by Pixelux Entertainment under the trade name DMM and has been successfully used in the Xbox 360 and the PS3 versions of the video game *Star Wars: The Force Unleashed* published by LucasArts. The results presented here were generated using both our own standalone test software and

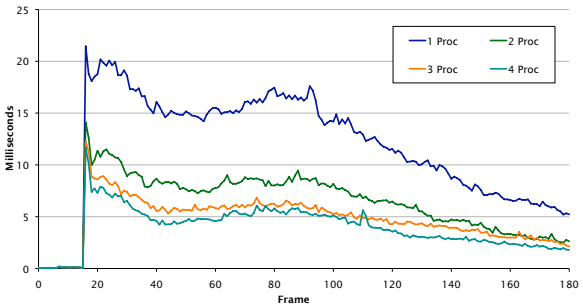

Figure 6: *Timing results for glass explosion. This plot shows the simulation time required to model the scenario shown in Figure 5 using a variable number of processors on a PC with Intel 2.4 GHz Core 2 Q6600, 667 MHz DDR2. The simulation involved approximately 2400 tetrahedra.*

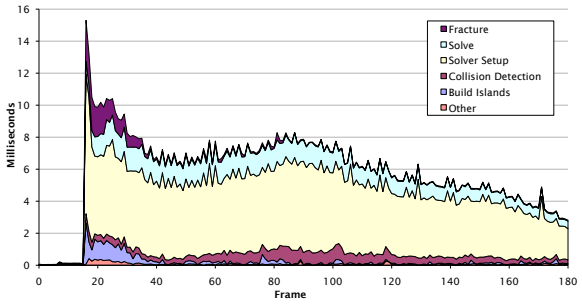

Figure 7: *Breakdown of timing results for glass explosion. This stacked plot shows a breakdown of time spent in each major phase of the simulation algorithm. The data were gathered from an additional run of the four-processor scenario plotted in Figure 6.*

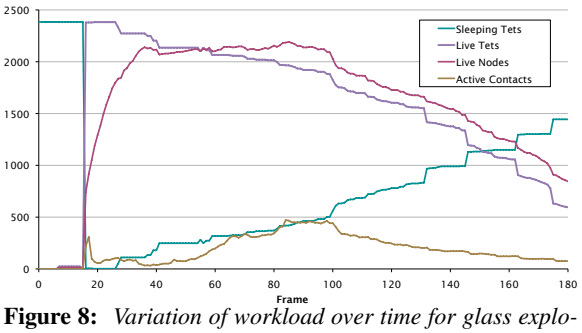

Figure 8: *Variation of workload over time for glass explosion. This plot shows the numbers of sleeping tetrahedra, live tetrahedra, live nodes, and active contact points varying as the simulation is run. The data were gathered from the same run that was used for Figure 7.*

the release version of *Star Wars: The Force Unleashed*. Figures 1 and 5 show two representative scenes from the game. Additional examples are included in the companion video to this paper.

The development of our physics engine involved a substantial amount of close interaction with the game's development team. This process helped inform the design and selection of many of the algorithms described above. For example, although element splitting is computationally cheap and the stability issues with split elements are easily dealt with, splitting was avoided because it causes the number of tetrahedra to change dynamically and complicates level balancing. Instead we used the splinter technique which avoids dynamic element creation and also provides an avenue for artistic control. Feedback from the content artists indicated that they liked the splinter approach and the control it affords, and did not find the required authoring to be burdensome. Figure 4 shows images from a test program used to preview materials. A wood material is shown along with the concept artist's sketch of what the desired material should look like.

In the published game, objects fade away once the game engine has determined that they are no longer needed. This decision was made by the game's development team not because of limitations in the simulation, but because they did not want the levels to become cluttered with debris that would interfere with the kinematic motion of the player and other characters.

Our system has been implemented to run on several architectures including standard PCs, the Xbox 360, and the PS3. Unfortunately, contractual restrictions preclude disclosure of detailed timing information for the Xbox 360 and PS3 implementations. However, we can provide information measured from our PC implementation. The plot in Figure 6 shows our system's performance for the scene depicted in Figure 5. (This scene and several others are included in the video.) As one would expect, we do not achieve perfect linear speedup scaling with the number of processors, in part because not all parts of our code have been parallelized and also because of issues with memory coherence on the Q6600. However we see that the parallel versions of the code realize substantial gains over the serial version. The total time for the case with four processors is broken down in Figure 7 to show how much time was spent in each part of the simulation code. For reference, Figure 8 shows how the number of active tetrahedra, nodes, and contact regions varies over the course of the simulation run.

Although we designed this system for real-time simulation, there is no reason why it cannot be used to simulate the behavior of larger systems that push it beyond the limits of what it can do in real-time. Figures 9 and 10 are examples of such scenes. They demonstrate the type of real-time behaviors that one could expect in the near future.

The example shown in Figure 11 demonstrates an effect that could not be modeled with a quasi-static method. As the tough wooden pole is twisted it stores a large amount of elastic energy. Once stress in the relatively brittle material reaches its fracture threshold, the released energy causes a cascade of fractures resulting in a violent mechanical explosion and flinging high-speed debris outward.

The current design decisions represent what we felt was a "sweet spot" in the continuum of possible choices. This

Figure 9: *Trestle collapsed by a hurled boulder. This example demonstrates a complex heterogeneous structure made from different materials. This result was generated by applying our system to a scenario that exceeds its real-time capacity and it took a couple minutes to compute.*

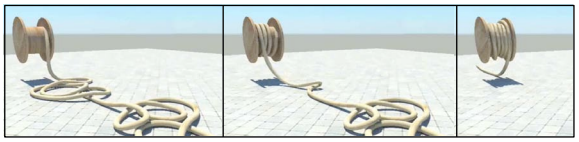

Figure 10: *A rope being wound onto a spool. To resolve this scenario with no visible penetration our collision handler required a smaller integration step that only allowed interactive, not real-time, performance.*

sweet spot changes as the capacity of available hardware evolves. For example, our tests with a multigrid solver show that it under-performs our current solver for the size of mesh that we can currently model in real-time. For larger systems this relation reverses and the multigrid solver will be superior. Other options that might prove to be useful for different types and sizes of system include fill-reducing direct linear solvers, non-linear iterations to converge the corotational element formulation, and element splitting.

Acknowledgments

The authors would like to thank their colleagues at Pixelux Entertainment and U.C. Berkeley for their help and support, and *The Force Unleashed* team at LucasArts for giving us the opportunity to work with them. Jonathan Shewchuk and Nuttapong Chentanez provided valuable feedback and editorial suggestions. The video for this paper was edited by Sebastian Burke, and Vik Sohal helped with video capture. Eric Larsen, Karl Hillesland, Mitch Bunnell, Vik Sohal, and Dave McCooey at Pixelux Entertainment worked on the development of the DMM physics engine.

References

[Bar89] BARAFF D.: Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proceedings of ACM SIGGRAPH 1989* (1989), pp. 223–232.

[Bar92] BARAFF D.: *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Computer Science Department, Cornell University, 1992.

[Bar94] BARAFF D.: Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of ACM SIGGRAPH 1994* (1994).

[BFA02] BRIDSON R., FEDKIW R., ANDERSON J.: Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of ACM SIGGRAPH 2002* (2002).

[BH79] BELYTSCHKO T., HSIEH B.: Application of higher order corotational stretch theories to nonlinear finite element analysis. *Computers & Structures 11* (1979), 175–182.

[BHTF07] BAO Z., HONG J.-M., TERAN J., FEDKIW R.: Fracturing rigid materials. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 370–378.

[BW92] BARAFF D., WITKIN A.: Dyanmic simulation of non-penetrating flexible bodies. In *Proceedings of ACM SIGGRAPH 1992* (1992), pp. 303–308.

[BW97] BARAFF D., WITKIN A.: *Partitioned Dynamics*. Tech. rep., Robotics Institute, Carnegie Mellon University, 1997. Technical Report CMU-RI-TR-97-33.

[BW98] BARAFF D., WITKIN A.: Large steps in cloth simulation. In *Proceedings of ACM SIGGRAPH 1998* (1998), pp. 43–54.

[BWHT07] BARGTEIL A. W., WOJTAN C., HODGINS J. K., TURK G.: A finite element method for animating large viscoplastic flow. In *Proceedings of ACM SIGGRAPH 2007* (Aug. 2007), pp. 291–294.

[CAR*09] CHENTANEZ N., ALTEROVITZ R., RITCHIE D., CHO L., HAUSER K. K., GOLDBERG K., SHEWCHUK J. R., O'BRIEN J. F.: Interactive simulation of surgical needle insertion and steering. In *Proceedings of ACM SIGGRAPH 2009* (Aug. 2009).

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M. K.: I-COLLIDE: An interactive and exact collision detection system for large scale environments. In *Proceedings of the ACM Symposium on Interactive 3D Graphics* (Apr. 1995), pp. 189–196.

[CMPW01] COOK R. D., MALKUS D. S., PLESHA M. E., WITT R. J.: *Concepts and Applications of Finite Element Analysis*, fourth ed. John Wiley & Sons, New York, 2001.

[Cou08] COUMANS E.: Bullet physics. www.bulletphysics.com, 2008.

[Ebe04] EBERLY D. H.: *Game Physics* . Morgan Kaufmann, 2004.

[EKS03] ETZMUβ O., KECKEISEN M., STRAβER W.: A fast finite element solution for cloth modelling. In *Proceedings of the Pacific Conference on Computer Graphics and Applications* (2003), p. 244.

[Erl07] ERLEBEN K.: Velocity-based shock propagation for multibody dynamics animation. In *ACM Trans. on Graphics* (2007), vol. 26.

[ESHD05] ERLEBEN K., SPORRING J., HENRIKSEN K., DOHLMAN H.: *Physics based animation*. Charles River Media, 2005.

[Fel07] FELIPPA C.: Introduction to finite element methods. www.colorado.edu/engineering/cas/courses.d/NFEM.d, 2007. Course notes published as webpages.

[GBF03] GUENDELMAN E., BRIDSON R., FEDKIW R.: Nonconvex rigid bodies with stacking. In *Proceedings of ACM SIGGRAPH 2003* (2003).

[GM97] GIBSON S. F. F., MIRTICH B.: *A Survey of Deformable Modeling in Computer Graphics*. Tech. rep., Mitsubishi Electric Research Laboratories, 1997.

[HFS*01] HIROTA G., FISHER S., STATE A., LEE C., FUCHS H.: An implicit finite element method for elastic solids in contact. In *Proc. of Computer Animation* (2001), pp. 136–146.

[HGS*07] HUGHES C. J., GRZESZCZUK R., SIFAKIS E., KIM D., KUMAR S., SELLE A. P., CHHUGANI J., HOLLIMAN M.,

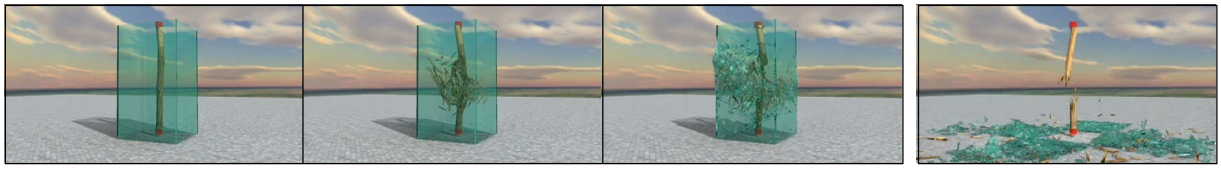

Figure 11: *A mechanical explosion. In this example, a thick wooden pole is twisted in opposite directions from the top and bottom. The stress builds in the pole until it shatters violently, flinging high-speed debris into the surrounding glass panes.*

CHEN Y.-K.: Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors. *ACM SIGARCH Computer Architecture News 35*, 2 (2007), 220–231.

[HSO03] HAUSER K. K., SHEN C., O'BRIEN J. F.: Interactive deformation using modal analysis with constraints. In *Graphics Interface* (June 2003), pp. 247–256.

[HVT08] HARMON D., VOUGA E., TAMSTORF R.: Robust treatment of simultaneous collisions. In *Proceedings of ACM SIGGRAPH 2008* (2008).

[Ing96] INGBER L.: Adaptive simulated annealing (asa): Lessons learned. *Control and Cybernetics 25* (1996), 33–54.

[IO06] IBEN H. N., O'BRIEN J. F.: Generating surface crack patterns. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Sept. 2006), pp. 177–185.

[ITF04] IRVING G., TERAN J., FEDKIW R.: Invertible finite elements for robust simulation of large deformation. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (July 2004), pp. 131–140.

[JP99] JAMES D. L., PAI D. K.: Artdefo: accurate real time deformable objects. In *Proceedings of ACM SIGGRAPH 1999* (1999), pp. 65–72.

[KEP05] KAUFMAN D. M., EDMUNDS T., PAI D. K.: Fast frictional dynamics for rigid bodies. In *Proceedings of ACM SIGGRAPH 2005* (2005).

[KK99] KARYPIS G., KUMAR V.: A fast and high quality multilevel scheme for partitioning irregular graphs. *SIA J. Sci. Comput.* (1999), 359–392.

[MBF04] MOLINO N., BAO Z., FEDKIW R.: A virtual node algorithm for changing mesh topology during simulation. *ACM Trans. Graph. 23*, 3 (2004), 385–392.

[MDM*02] MÜLLER M., DORSEY J., MCMILLAN L., JAGNOW R., CUTLER B.: Stable real-time deformations. In *ACM SIGGRAPH Symposium on Computer Animation* (July 2002), pp. 49–54.

[MG04] MÜLLER M., GROSS M.: Interactive virtual materials. In *Proceedings of Graphics Interface 2004* (2004), pp. 239–246.

[Mic05] MICROSOFT: Direct x 10 developer documentation/texture skinning. msdn.microsoft.com/en-us/library, 2005.

[MMDJ01] MÜLLER M., MCMILLAN L., DORSEY J., JAGNOW R.: Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic workshop on Computer animation and simulation* (2001), pp. 113–124.

[MSJT08] MÜLLER M., STAM J., JAMES D., THÜREY N.: Real time physics: class notes. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes* (2008), pp. 1–90.

[MW88] MOORE M., WILHELMS J.: Collision detection and response for computer animationr3. In *Proceedings of SIGGRAPH 1988* (1988), pp. 289–298.

[NMK*05] NEALEN A., MÜLLER M., KEISER R., BOXERMAN E., CARLSON M.: *Physically based deformable models in computer graphics*. Tech. rep., Eurogrphics 2005 state of the art report, 2005.

[NOR91] NOUR-OMID B., RANKIN C. C.: Finite rotation analysis and consistent linearization using projectors. *Comp. Methds. Appl. Mech. Engrg. 93* (1991), 353–384.

[NPF05] NESME M., PAYAN Y., FAURE F.: Efficient, physically plausible finite elements. In *Eurographics 2005, short papers* (Trinity College, Dublin, Irlande, Aug. 2005), Dingliana J., Ganovelli F., (Eds.).

[O'B00] O'BRIEN J. F.: *Graphical Modeling and Animation of Fracture*. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[OBH02] O'BRIEN J. F., BARGTEIL A. W., HODGINS J. K.: Graphical modeling and animation of ductile fracture. In *Proceedings of ACM SIGGRAPH 2002* (Aug. 2002), pp. 291–294.

[OH99] O'BRIEN J. F., HODGINS J. K.: Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH 1999* (Aug. 1999), pp. 137–146.

[OTSG09] OTADUY M. A., TAMSTORF R., STEINEMANN D., GROSS M.: Implicit contact handling for deformable objects. *Computer Graphics Forum (Proc. of Eurographics) 28*, 2 (apr 2009).

[Pis84] PISSANETZKY S.: *Sparse Matrix Technology*. Academic Press, London, 1984.

[SBT07] SPILLMANN J., BECKER M., TESCHNER M.: Non-iterative computatino of contact forces for deformable objects. *Journal of WSCG 15*, 1–3 (2007), 33–40.

[Sel01] SELLERS D.: An overview of proportional plus integral plus derivative control and suggestions for its successful application and implementation. In *Proceedings for the 2001 International Conference on Enhanced Building Operations* (2001).

[She94] SHEWCHUK J. R.: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. Rep. CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Mar. 1994.

[SWB00] SMITH J., WITKIN A., BARAFF D.: Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Interface* (may 2000), 27–34.

[Ter01] TERDIMAN P.: Memory-optimized bounding-volume hierarchies. www.codercorner.com/Opcode.pdf, 2001.

[TKZ*04] TESCHNER M., KIMMERLE S., ZACHMANN G., HEIDELBERGER B., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNETAT-THALMANN N., , STRASSER W.: *Collision detection for deformable objects*. Tech. rep., Eurographics Association, 2004.

[TPBF87] TERZOPOULOS D., PLATT J., BARR A., FLEISCHER K.: Elastically deformable models. In *Proceedings of ACM SIGGRAPH 1987* (jul 1987), vol. 21, pp. 205–214.