Collision-Driven Volumetric Deformation on the GPU

G. Aldrich[1,2], D. Pinskiy[1], B. Hamann[2]

[1] Walt Disney Animation Studios, [2] University of California Davis

Abstract

We present a novel parallel algorithm to animate the deformation of a soft body in response to collision. Our algorithm incorporates elements of physically-based methods, and at the same time, it allows artistic control of general deformation behavior. Our solver has important benefits for practical use, such as evaluation of animation frames in an arbitrary order, effective approximation of volume preservation, and stability under large deformations. To deform production-complexity data at interactive rates, we leverage the computational power of modern GPUs. In our solution we propose a specialized, CUDA-based framework for the efficient access and update of mesh structures.

Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Geometric algorithms, languages, and systems, I.3.7 [Computer Graphics]: Animation, I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1. Introduction

In a virtual world, believable physical interactions involving contact between objects are essential for animation. This drives the need for an animation tool that produces realistic-looking deformation on soft bodies as they collide with hard objects. This deformation has two contradictory requirements - it should both be physically-based and offer user control. Additionally, the tool has to run at interactive rates, and the resulting deformation should be completely time-independent (i.e. the deformation should be calculated for each animation frame individually and independent of any other frame). This gives animators the flexibility to access animation frames in an arbitrary order. It also allows the use of this tool for secondary deformations, where its input geometry is the output of an art-directable deformer, animated over time.

To address these animation needs, we draw on several existing approaches, but synthesize them to obtain a high performance, parallel algorithm for collision-driven deformation. The interactive rate and time-independent requirements prohibit us from running a precise physics simulation. Instead, we designed a deformer which mimics the physical properties of an object while still allowing a CG artist to achieve a particular look for the deformation. We treat objects as solids instead of shells in order to achieve volumetric effects. The user controls the global deformation of the surface by positioning the soft object and colliders in the scene. In each animation frame, we resolve collisions between the undeformed soft body and colliders, repositioning intersecting vertices to eliminate penetrations. The global surface properties are then recovered by locally restoring volume and minimizing the material stretch.

We also offer a GPU framework for a CUDA-based implementation of our algorithm. Our GPU implementation is specifically designed to provide efficient access to the local mesh topology. This efficient memory access is critical, as without it, memory transactions would be a major bottleneck in our deformation.

2. Related Work

The Finite Element Method (FEM) is considered to be one of the most physically correct methods to calculate deformation of colliding solids. The quasistatic variant of FEM particularly appeals to us as it simplifies the model by removing the time integration (e.g. [TSIF05]). However, this technique requires solving complex partial differential equations, making it prohibitively expensive. Our approach is closer to mass-spring models, which gain efficiency due to their simplicity. Jakobsen simplifies mass-spring calculations by limiting strain of the springs [Jak01]. His model is very efficient, but it is not directly applicable to our case as it requires time integration and does not explicitly restore volume. Mollino et al. offer a quasistatic-like approach for a spring model, where they implicitly simulate physics by solving the optimization problem to arrive at the mesh with the desired properties [MBTF03]. Although our approach is inspired by this work, their goal is different from ours. Their method is designed to produce the volume mesh, with optimum aspect ratio for edges, whereas we need to output a surface mesh with desired physical properties.

Alternatively, collision response on a solid can be achieved by manually sculpting the appropriate shape using modeling deformers, which are proven to be fast and art-directable. Both our method and volumetric modeling deformers (e.g. lattice and cage-based deformers [JSW05]) are driven by displacements of control cage vertices, which implicitly drive deformation of the solid, embedded into the

© The Eurographics Association 2011

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

control cage. However, the classic volumetric deformers require the user to directly manipulate control volume vertices. In contrast, our method automates this deformation, as we apply rules to move the volume vertices in accordance with the physical properties of the solid.

Due to the computationally intensive nature of physically-based deformation there has been an effort to offload these tasks onto the GPU. The data structure in [RMS08] is the most similar to ours. They offer an efficient mechanism to access indices of vertices connected to a given vertex. Their data structure is designed for the uniform grid, and is based on static-sized arrays. However, we need to handle a volume mesh that has arbitrary high and low valences. Using arrays of varying size, we optimize our data structure both for speed and space.

3. Algorithm

We start the discussion of our algorithm with a problem definition. The input soft solid to be deformed is given in the form of a polygonal mesh ***M*** that approximates the soft solid's surface boundary (shell). The material property of the solid is provided as a stiffness coefficient *s*. The colliding hard objects are also given as polygonal meshes that bound their volumes. Our algorithm should deform ***M***, producing the soft object's reaction to collision. The resulting deformation should have a physically plausible appearance that is defined by (1) approximating volume preservation and (2) enforcing the desired stiffness. In addition, no topological changes to ***M*** are allowed, and therefore no mesh simplification or surface resampling can be performed. Also our calculation for a particular animation frame should be independent from previously computed frames. Consequently, if no collision occurred in the current animation frame, the soft object should remain undeformed. That is different from dynamic solutions where the surface might carry a deformation produced by the collisions in previous frames.

3.1. Overview

The key part of our deformation algorithm is building and utilizing a helper volume mesh ***H*** that embeds ***M*** and serves as its solid representation. Our deformation is performed iteratively, and each iteration is comprised of four steps. First, we deform the volume mesh ***H*** to resolve collisions of the input soft solid with the hard objects. By doing so, ***H***'s physical properties can be violated. To overcome this, in the next step we restore volume in ***H***. Then we enforce the prescribed stiffness on the stretched portion of ***H***. Finally, we end our iteration with mapping displacements in ***H*** onto ***M***'s vertices. In this step we treat ***H*** as a lattice around ***M***, and by restoring the desired physical properties of ***H***, we implicitly restore the physical properties of ***M***.

We iterate over these four steps until the steady state is reached (i.e. ***M***'s vertex displacements become below the user's specified threshold value). To ensure that our solution remains stable under large deformations, we limit the vertex movement in any single iteration (usually half of the average edge length).

By starting with the collision step, we let the collision conceptually drive the rest of the deformation. If there is no collision, naturally ***H***'s physical properties are not violated. As a result, ***H***'s vertices are not displaced during the steps that restore physical properties, and no deformation is introduced to ***M***. Enforcing the desired stiffness after the volume preservation, we give priority to user's control of the material stiffness over volume preservation.

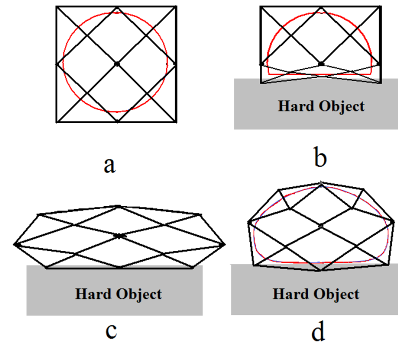


Figure 1: *A 2D example of the deformation steps: a) during the preprocessing the helper mesh (in black) is constructed based on vertex positions of the input surface mesh (in red); b) after the collision detection step, surface displacements are mapped to the helper mesh; c) area (volume in 3D) is restored in the helper mesh; d) the desired stretchiness is enforced in the helper mesh and its final displacement mapped back onto the surface.*

3.2. Preprocessing Step

We build ***H*** in the preprocessing step, employing a type of volume mesh very similar to one described in [LS07]. We direct the reader to the above paper for the details while we provide the key features of this volumetric meshing. The volume mesh is constructed based on an octree-like structure that sorts vertices of ***M***. Naturally we obtain smaller cells near the surface boundary where we need more precision and large octree cells inside the volume where there are less details. For simplicity, a tetrahedral volumetric mesh structure in ***H*** is enforced. This is done by inserting an additional volume vertex at the center of each octree cell, connecting this center vertex with the cell's corner vertices and newly inserted center points of the neighboring octree cells. To simplify the tessellation of neighboring cells, the neighboring leaves cannot vary in depth by more than one. The resulting volume mesh has an important property that we will use later – each vertex is connected to at least three tetrahedra. The main difference between their and our volume meshing is how the octree cells that are partially outside the surface ***M*** are handled. In the original method, the volume meshing ends by cutting such cells to the surface, while we skip this step to avoid possible numerical instabilities associated with tessellation of the resulting intersections. Figure 1a shows the result of the preprocessing step for an analogous 2D case where the complexity of ***M*** requires only one quadtree split.

Since ***H***'s octree cells will be used as lattice-controlled volumes, we also compute lattice coordinates for ***M***'s

© The Eurographics Association 2011

vertices in the preprocessing step. Later, as under usual lattice deformations, the displacements of ***H***'s vertices can be interpolated to reposition the vertices of ***M*** in the 3D world space using the lattice coordinates (e.g. [JSW05]). Reverse interpolation can be applied so that displacements of ***M***'s vertices would define new locations for ***H***'s vertices.

3.3. Iterations

We begin each iteration with resolving collision between the input soft object and the hard colliding objects. First we push ***M***'s vertices toward the closest colliding surface. Then we deform ***H*** by mapping the displacements of ***M***'s vertices onto ***H***. This is accomplished using an approximation of reverse interpolation based on the lattice coordinates of the surface vertices (see Figure 1b).

Instead of solving for volume globally, we estimate it by locally conserving volume at each vertex of ***H***. We do this by repositioning each vertex such that tetrahedra, connected to this vertex, regain their original volumes. Consider a tetrahedron, with vertices A, B, C, D. Its current signed volume can be calculated as

$$V = \frac{(D-A)\bullet Base}{6} \qquad (1)$$

where $Base = (B-D)\times(C-D)$.

If we substitute the current signed volume with the initial signed volume V_{init}, we can solve (1) for vertex A to find its displacement to arrive at the original volume. In matrix form, this equation can be written as following:

$$\left[Base_X \; Base_y \; Base_z\right]\begin{bmatrix}A_x\\A_y\\A_z\end{bmatrix} = \left[D\bullet Base - 6\,V_{init}\right] \quad (2)$$

In this equation, we have three unknowns (A_x, A_y, A_z). By the volume mesh construction, we guarantee more than three connected tetrahedra to any vertex, and therefore we have an overdetermined system:

$$\begin{bmatrix}Base_x^0 \; Base_y^0 \; Base_z^0\\ Base_x^1 \; Base_y^1 \; Base_z^1\\ \ldots \\ Base_x^N \; Base_y^N \; Base_z^N\end{bmatrix}\begin{bmatrix}A_x\\A_y\\A_z\end{bmatrix} = \begin{bmatrix}D^0\bullet Base^0 - 6\,V_{init}^0\\ D^1\bullet Base^1 - 6\,V_{init}^1\\ \ldots \\ D^N\bullet Base^N - 6\,V_{init}^N\end{bmatrix} \quad (3)$$

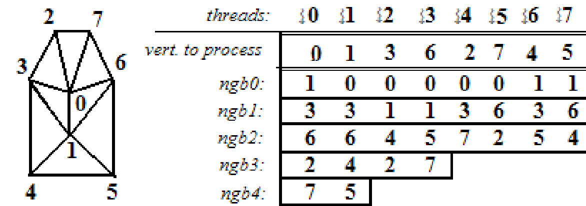
We solve the above system of equations using least squares and update each volume vertex by moving it towards the newly computed position. The end result of this iteration step for the 2D case is illustrated in Figure 1c.

It is possible that while the volume is preserved, the shape is stretched. As the next step of the iteration, we prescribe the desired stiffness of the object by limiting the change in edge lengths in our volume mesh. This can be thought of as the forces acting on the individual fibers in an object. For each volume vertex P_0, we examine the set of connected edges $P_0 P_1$, $P_0 P_2$, …, $P_0 P_n$ with original edge length e_1, e_2, …, e_n and calculate displaced position P'_0 based on stiffness coefficient s ($s\in[0,1]$), using the following equation:

$$P'_0 = (1-s)\,P_0 + s\sum_{i=1}^{n}\left(\frac{e_i}{n\left\|\overrightarrow{P_0\,P_i}\right\|}\overrightarrow{P_0\,P_i} + P_i\right) \quad (4)$$

If collision inverts tetrahedra, (4) alone would not necessary resolve this issue, but the signed volume restoration would.

Finally to map the deformation back from the volume mesh ***H*** onto the surface mesh ***M***, we interpolate surface points from the volume mesh (Figure 1d). As the volume and edge length are enforced, some vertices of ***M*** may end up colliding with the hard object. In this case, we need to continue iterating even if we have arrived at the static state.



threads:	0	1	2	3	4	5	6	7
vert. to process	0	1	3	6	2	7	4	5
ngb0:	1	0	0	0	0	0	1	1
ngb1:	3	3	1	1	3	6	3	6
ngb2:	6	6	4	5	7	2	5	4
ngb3:	2	4	2	7				
ngb4:	7	5						

Figure 2: *GPU data-structure for a simple mesh. The rows represent arrays and the columns represent thread's memory access.*

4. GPU Implementation

Modern GPUs are massively parallel, and therefore they have a great potential for computational performance. To benefit from running on the GPU, a method should involve a significant amount of arithmetic computations and have data-level parallelism. Both the volume restoration and the stiffness preservation steps are not only computation-heavy but also inherently parallel since during each of the steps, we can compute new vertex positions independently. That makes them perfect candidates to be implemented on the GPU. However, for a successful GPU implementation, we need to provide effective memory access, especially given that both steps are very memory intensive.

When we restore volume and preserve edge lengths for a particular vertex, we need fast access to its surrounding primitives such as tetrahedra and edges. To store the actual data (e.g. vertex positions, initial volumes of tetrahedral, edge lengths), we choose the texture memory because of its powerful caching mechanism. Thus, in both iteration steps, we need efficient memory access to indices to retrieve data associated with primitives that surround a given vertex. We address this need by coalescing memory access.

In CUDA, the memory coalescing allows 16 threads to access 16 locations in global memory in just one or two transactions. One of the requirements for coalescing is that all threads must access words sequentially. To facilitate this, we provide our own special data structure that stores information about the primitives connected to each vertex.

A simple example of the data structure is shown in Figure 2. This structure is designed to provide access to vertex indices connected to a particular vertex. (In general any local primitive can be accessed using this data structure.) Our data structure is based on an array of pointers to several arrays, which reside in global memory. Each thread is

© The Eurographics Association 2011

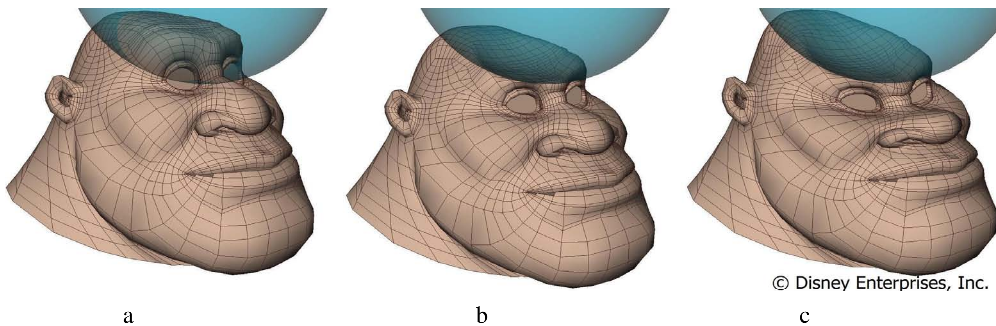


Figure 3: *Deformations with various stiffness: a) the original shape, b) stiff material, c) soft material.*

conceptually associated with a column index in each edge array, e.g. thread t that processes a vertex V_t reads the index for V_t's first neighbor at *ngb0[t]*, then reads the second neighbor at *ngb1[t]*, and so forth. Since during each of these reads data is accessed sequentially across threads, coalescing can be implemented. It is essential that *ngb* arrays contain vertex indices, not actual vertex data that needs to be updated during each iteration step. Otherwise, since the same vertices appear in neighboring lists of different vertices, we would have multiple writes per single vertex update. In addition, we would still have to store the vertex indices in order to update the array structure with new data, and that would defeat the purpose. We sort V_t's by their valence in the descending order, and consequently we can vary *ngb* arrays in size rather than allocate them the same length and fill the empty spots with *NULL*. In addition, by keeping our table sorted by valance, we limit the threads' divergence.

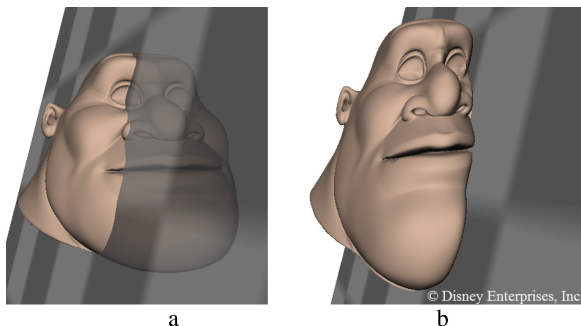


Figure 4*: Deformation caused by collsion that consumes more than 60% of volume: a) original shape, b) shape after deformation.*

5. Results

To recover from a significant collision, we need to run a large number of iterations. For example, in the case shown in Figure 3, we had to run about 120 iterations on the head model, which has 2400 vertices. To recover from collision that consumed more than 60% of the volume, we had to run over 250 iterations (Figure 4). In the CPU implementation, the bottleneck of these iterations is, by far, in restoring volume and enforcing stiffness. However, by moving steps to GPU and implementing the discussed memory data structure, this bottleneck was removed. Our GPU implementation of these steps has a 7x speedup over our multithreaded CPU implementation running on 4 cores. (We used an NVIDIA GTX 460 and 2.67 GHz Intel Core I5 respectively to benchmark our code.) The frame-rate for the whole algorithm applied to the model is 20 fps though this number is dependent on the choice of the collision library.

Figure 3 also provides an interesting observation regarding soft and stiff deformations in our model. When the material is soft, the deformation is local since we allow edges to stretch by large amounts to accommodate the volume pushed from the hard collider. If we mimic a stiff material and limit how much edges can stretch, than the whole object has to absorb the pushed volume and the deformation becomes more global.

6. Conclusion

We have presented a novel approach to produce collision-driven deformation that combines properties of physics-based approaches (e.g. retaining volume and stiffness) with time-independence of animation deformers. We also offer a framework for its efficient implementation on the GPU.

References

[Jak01] Jakobsen T.: Advanced character physics. http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml, 2001.

[JSW05] Ju T., Schaefer S., Warren J.: Mean value coordinates for closed triangular meshes. *ACM Transaction on Graphics. 24(3)* (2005*)*, pp. 561-566.

[LS07] Labelle F., Shewchuk J. R.: Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. *ACM Transaction on Graphics. 26(3)* (July 2007*)*, pp. 57-67.

[MBTF03] Molino N, Bridson R, Teran J, Fedkiw R.: A crystalline, red green strategy for meshing highly deformable objects with tetrahedra, In *Proc. of 12th International Meshing Roundtable* (2003), pp 103-114.

[RMS08] Rasmusson A., Mosegaard J., Sorensen T. S.: Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA. In *Proc of 4th International Symposium Biomedical Simulation,* (2008), Vol 5104/2008, pp. 49-58.

[TSIF05] Teran J., Sifakis E., Irving G., Fedkiw R,: "Robust Quasistatic Finite Elements and Flesh Simulation", In *Proc. of ACM/Eurographics Symposium on Computer Animation,* (2005), pp. 181-190.

© The Eurographics Association 2011