Book Title
Book Editors
IOS Press, 2003

Exploiting Graphics Hardware for Haptic Authoring

Minho Kim [a,1], Sukitti Punak [a], Juan Cendan [b], Sergei Kurenov [b], and Jörg Peters [a]
[a] *Dept. CISE, University of Florida*
[b] *Dept. Surgery, University of Florida*

Abstract. Real-time, plausible visual and haptic feedback of deformable objects without shape artifacts is important in surgical simulation environments to avoid distracting the user. We propose to leverage highly parallel stream processing, available on the newest generation graphics cards, to increase the level of both visual and haptic fidelity. We implemented this as part of the University of Florida's haptic surgical authoring kit.

Keywords. virtual surgery training, haptic rendering, GPU shader, subdivision surface, etc.

1. Introduction

Shape artifacts, such as the transition between facets of an insufficiently refined model, can distract both a user's visual and haptic senses from a given task. However, modeling a large scenario to a level of refinement that hides such representation artifacts, requires considerable computational resources. Fortunately, recently, the graphics processing units (GPU) of graphics cards have become more powerful, offering highly parallel stream processing with access via *graphics shader programming*; moreover, novel data structures have been developed to shift the work of finely evaluating high quality surface representations, so-called subdivision surfaces, from the central processing unit (CPU) to the GPU, freeing the CPU for higher-level and user-input tasks. While the state-of-the art research addresses the need for real-time visual animation of high-quality representations, the higher frequency of haptic update has not, at this point, been satisfactorily addressed in the literature.

We propose to leverage highly parallel stream processing, available on the newest generation graphics cards, to increase the level of both visual and haptic fidelity by taking advantage of (i) the locality of haptic probing (Figure 4) and (ii) an improved communication channel (bus) between the GPU and CPU.

As a proof of concept, we implemented the visual and haptic improvement in the University of Florida's haptic surgical authoring kit. This haptic authoring kit is a low cost environment that allows the specialist surgeon to author haptic and multimedia training exercises. A key issue in such a haptic simulation is to determine the minimal level

[1]Correspondence to: SurfLab, Dept. CISE, CSE Bldg E325, University of Florida, Gainesville, FL 32611. Tel.: +1 352 392 1255; Fax: +1 352 392 1220; E-mail: {mhkim,jorg}@cise.ufl.edu.

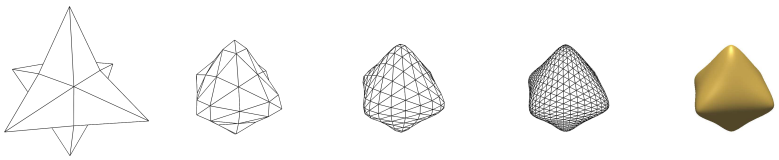

Figure 1. Surface mesh refinement (depth 0,1,2,3) and rendering of using Loop's Subdivision.

of visual and haptic fidelity required to make the exercise effective. We had found earlier that the consistent interplay between visual and haptic feedback is important to avoid distracting the author form the task. However, modelling the surfaces of organs, vessels and tissues as visually and haptically smooth surfaces results in an equally undesirable time-lag, especially when surfaces are allowed to deform.

Recently, many applications have been developed that leverage the parallel processing power of the modern GPUs. One of these applications is the run-time evaluation of subdivision surfaces (1) using programmable graphics hardwares, called *GPU shaders*. Since most of the human organs can be modeled compactly using subdivision surfaces, we adapt this approach using GPU shaders not only for visual rendering but also for haptic rendering for our authoring environment. By tracking the current position of the haptic device, we can localize the evaluation thus reducing rendering overhead for high-frequency rendering of deforming objects.

2. Background: Subdivision Algorithms, Spatial Data Structures and Shaders

Subdivision algorithms create smooth surface approximations by recursively refining the connectivity and smoothing the geometry of a polyhedral input mesh, known as the *control mesh* (see e.g. [1,2], 1). In all popular algorithms the position of a new mesh node is obtained as the weighted average of old nodes of a small submesh, whose graph is called *stencil*. One such refinement pattern splits each triangle into four and is called Loop subdivision (Figure 1).

Modern GPUs provide programmable parallel stream processing in the form of *vertex shaders* and *fragment shaders* [3]. Vertex shaders process attributes, such as positions, normals, and texture coordinates, of a single vertex without connectivity. The downstream fragment shaders process the rasterized data (i.e. attributes per pixel) and assign the resulting pixels. Fragment shaders are the key computation units for most GPU algorithms (as well as ours) because of their computation power and ability to read and write data by rendering to the framebuffer and copying to readable texture images.

Strategies and techniques for computation on GPUs are collected in [4]. A number of important algorithms have been modified to rebalance the workload between CPU and GPU and take advantage of parallel execution streams in programmable graphics hardwares: particle systems [5], collision detection [6], cellular automata [7], global illumination [8] and other numerical computations [9,10]. The algorithmic component on the GPU, called *shader*s, rely essentially on accessing regularly laid out data, typified by the 2D array, to minimize workflow branching and maximize parallelism. Irregular access typically requires interaction with the CPU.

3. Real-time subdivision surface reevaluation on the GPU

Most recently, researchers succeeded in mapping the irregular access structure characterizing general subdivision surface evaluation, to a representation on the GPU [11]. A locality-preserving data access keeps all irregularities strictly inside overlapping, independently refinable pieces of the mesh, called *fragment meshes* (Figure 2), allowing for parallel streams of work per fragment mesh and also per mesh node. The fragment meshes are encoded into one-dimensional array as a texture, called *patch texture*, fed into GPU pipeline in a parallel fashion, and subdivided *recursively* by fragment (pixel) shaders in an off-screen framebuffer. Since all topology (connectivity) information is lost in the one-dimensional array, a smart, pre-computed *stencil lookup table* is also sent to GPU as a texture. The GPU then re-evaluates a freely deforming surface at interactive frame-rates (20-30 frames-per second) for moderately-sized control meshes (ca 100 nodes); see also Table 2. Thanks to recursive subdivision, it can model *semi-smooth creases* (features that are sharp in the large scale, but rounded at the small scale) and *global boundaries* i.e. surface pieces.

There are several implementation challenges for this approach. One is to guarantee *water-tight boundaries*, i.e. no pixel-dropout artifacts between fragment meshes. Another is to avoid the *redundant data round-trip* of the final subdivision data that current generation graphics cards enforce: the evaluated values are stored in pixel formats in an off-screen framebuffer, the *pbuffer*, that cannot directly be rendered. In currently available standard GPU interfaces, it needs to be copied back to system memory and then returned to GPU pipeline again. This 'round-trip' of the refined data when it is the largest in size, slows the rendering considerably. The feature that enables us to use the data in pixel formats (stored in a texture of framebuffer) as vertex array is called *render-to-vertex-array* and there are several extensions for this purpose in OpenGL®, including the *PBO/VBO extension* and the *überbuffer (supperbuffer) extension*; the upcoming OpenGL® *FBO (framebuffer object) extension* is expected to replace and improve on both which should help our implementation.

3.1. GPU implementation of Loop subdivision

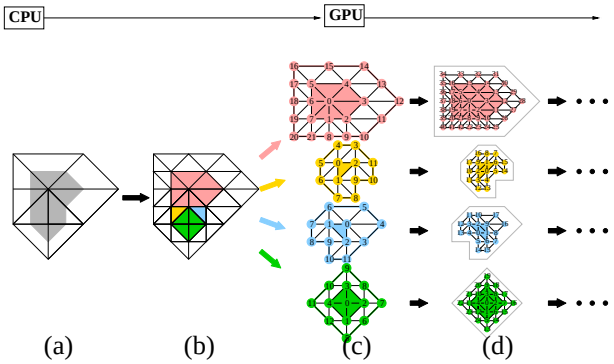

Figure 2. Workflow of GPU Loop subdivision kernel: (a) input control mesh (b) initial subdivision (c) fragment meshes and (d) fragment meshes subdivided in the GPU.

Most available mesh models consist of triangles and the natural subdivision scheme for triangulated data sets. Following the general approach of [11] we implemented Loop subdivision on the GPU solving two challenges specific to Loop subdivision. First, as Figure 2 shows, after one refinement on the CPU, we obtain *vertex fragment meshes* composed of two layers of triangles around each of the input control vertices. However, for each of the input facet, there is an additional center triangle that needs to separately treated as a *facet fragment mesh*. Its implementation, although the same over all facet fragment meshes, is nontrivial. Secondly, guaranteeing water-tight boundaries, especially where two facet and two vertex fragment meshes meet, despite rounding of intermediate results, requires a smart symmetric refinement.

4. Surgical Haptic Authoring

To address the constant need for practicing surgeons to quickly and conveniently refresh their knowledge of surgical procedures that they are not performing on a regular basis, the University of Florida surgical authoring kit provides a media rich and 'hands-on' channel for publishing procedures by a specialist surgeon (see Figure 3). A key point of this approach is that the specialist surgeon rather than a computer programmer will author the material to make the approach flexible and remove a level of indirection that slows publication and easily leads to wrong emphasis. Currently the kit's haptic component is based on the affordable Phantom® Omni™Haptic Device.

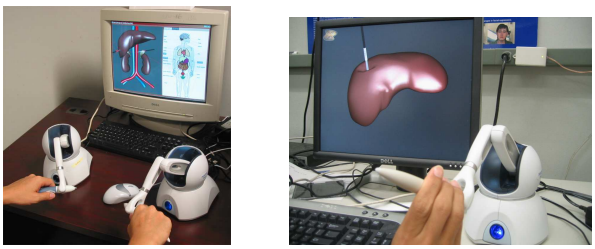

Figure 3. Screen shots from the University of Florida surgical authoring kit.

5. Haptic rendering exploiting the GPU subdivision kernel

The OpenHaptics™Toolkit is a popular SDK that enables developers to control several widespread haptic device lineups from SensAble Technologies, Inc.. One of its merits is that we can use the same OpenGL® geometry rendering modules for the haptic rendering. Since we already have the *finely subdivided* subdivision mesh calculated by GPU shaders for visual display, we should obviously use them again for haptic rendering.

We dramatically improve the haptic rendering performance, by bounding each fragment mesh to localize the haptic rendering of a deformable object. Since the geometry of the fragment meshes change as they are subdivided, such bounding boxes must enclose all the intermediate subdivided ones. Fortunately, in Loop subdivision, positions of all the vertices in the intermediate subdivided fragment meshes are convex combinations of

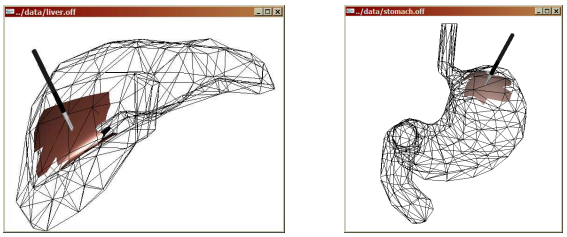

Figure 4. Visual display of the localized fragment meshes used for haptic rendering.

	CPU	GPU	system memory	video memory	shaders per patch	double buffering	buffer size	data round-trip removal
1	Pentium 4 (2.40GHz)	ATI Radeon 9700 Pro (Omega driver 2.5.97a)	1GB	128MB (AGP 4×)	2	no	2048×1024	none
2	Pentium 4 (3.00GHz)	nVidia GeForce 6800GT (driver 71.84)	1GB	256MB (AGP 8×)	1	yes	2048×256	PBO/VBO
3	Pentium M (1.60GHz)	nVidia GeForce 6200 (driver 70.87)	512MB	128MB (PCI Express 16×)	1	yes	2048×256	PBO/VBO
4	Pentium 4 (2.80GHz)	ATI Radeon X800 (Omega driver 2.5.97a)	1GB	256MB (AGP 8×)	1	no	2048×256	PBO/VBO

Table 1. Four hardware/shader configurations used for timings in Tables 2 and 3. All configurations use the pbuffer mechanism as off-screen framebuffer.

the initial ones and therefore stay inside of the convex hull of the initial fragment mesh. We enclose the convex hull by a bounding sphere to increase performance.

One of the bottlenecks when working with deformable objects are bounding box updates. However, the range of movement of the control vertices of deformable organs is limited, so that we can make the bounding spheres partially static, fixing its center position and adjusting only the radius. The radius depends on the deformation and the *speed of the haptic device*: if the device moves too quickly for the size of the bounding sphere, we may not detect the intersection of the device with the surface of the object but penetrate it to the interior, resulting in a well-known *falling-off* artifact. By calibrating the radius according to the speed of the haptic device, we can avoid this artifact.

Once the fragment meshes close to the haptic device are determined, only their GPU-subdivided meshes need to be handed to the haptic engine. Figure 4 visually illustrates the fragment meshes rendered for haptic feedback.

Results: Table 1 summarizes the four hardware/shader configurations tested for the visual feedback. Table 2 shows the resulting timings for purely visual feedback and Table 3 the frames per second for combined visual and local haptic rendering of real-time deforming of high-quality surfaces. The drop in performance with our current implementation can be as bad as 50% for small models on the older ATi9700, but is just 10-20% on the newer ATiX800. Considering that complete recomputation to depth 5 for each frame is beyond the need perceived by users, the numbers indicate that high-quality visual and local haptic rendering is now within reach of commodity PCs with high-end commodity graphics cards.

Acknowledgements This research was made possible in part by NSF Grants DMI-0400214 and CCF-0430891.

fps	liver		stomach		mechpart		venus	
	4	5	4	5	4	5	4	5
1	9.70	6.27	8.53	4.41	7.62	4.60	4.27	1.88
2	22.83	15.63	19.42	11.43	18.28	13.07	9.70	5.24
3	18.32	9.15	12.08	4.85	13.62	6.81	5.04	1.90
4	13.91	8.42	11.43	5.71	10.33	6.04	5.42	2.40

Table 2. Frames per second (fps) for visual rendering of the four configurations in Table 1; four data sets are refined to depth 4, respectively 5 (ca. 300K triangles for `liver`, 550K for `stomach`).

fps	liver		stomach	
	4	5	4	5
1	5.20	3.62	4.54	3.32
4	10.86	6.40	10.00	5.08

Table 3. Frames per second (fps) for visual rendering and haptic feedback on the two platforms/configurations 1 and 4 supporting the Omni and two surfaces relevant to our surgical authoring.

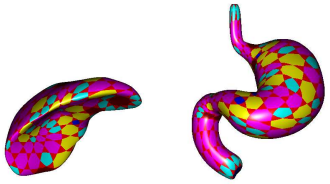

Figure 5.: Models `liver` and `stomach` shaded according to split into fragment meshes.

References

[1] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98 Conference Proceedings*, pages 85–94, 1998.

[2] Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers, 2002.

[3] Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH '01 Conference Proceedings*, pages 149–158, 2001.

[4] Mark Harris, David Luebke, Ian Buck, Naga Govindaraju, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell, and Cliff Woolley. GPGPU: General-purpose computation on graphics hardware. *Course notes 32 of SIGGRAPH 2004*, 2004.

[5] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Eurographics Symp Proc Gr Hardware*, 2004.

[6] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the Conference on Graphics Hardware*, pages 25–32. Eurographics Association, 2003.

[7] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the Conference on Graphics Hardware*, pages 92–101. Eurographics Association, 2003.

[8] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the Symposium on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[9] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03 Conference Proceedings*, pages 908–916, 2003.

[10] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH '03 Conference Proceedings*, pages 917–924, 2003.

[11] Le-Jeng Shiue, Ian Jones, and J. Peters. A realtime gpu subdivision kernel. In Marcus Gross, editor, *Siggraph 2005, Computer Graphics Proceedings*, Annual Conf Series, pages 1010–1015. ACM Press, 2005.