ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/262369351

Fast deformation of volume data using tetrahedral mesh rasterization

Conference Paper · July 2013

DOI: 10.1145/2485895.2485917

CITATIONS
19

READS
456

5 authors, including:

José Miguel Espadero
King Juan Carlos University
14 PUBLICATIONS **98** CITATIONS

SEE PROFILE

Alvaro Martín Pazmiño Pérez
Universidad Técnica de Babahoyo
26 PUBLICATIONS **217** CITATIONS

SEE PROFILE

Miguel A. Otaduy
King Juan Carlos University
193 PUBLICATIONS **5,389** CITATIONS

SEE PROFILE

All content following this page was uploaded by José Miguel Espadero on 13 December 2016.

The user has requested enhancement of the downloaded file.

Fast Deformation of Volume Data Using Tetrahedral Mesh Rasterization

Jorge Gascon Jose M. Espadero Alvaro G. Perez Rosell Torres Miguel A. Otaduy
URJC Madrid

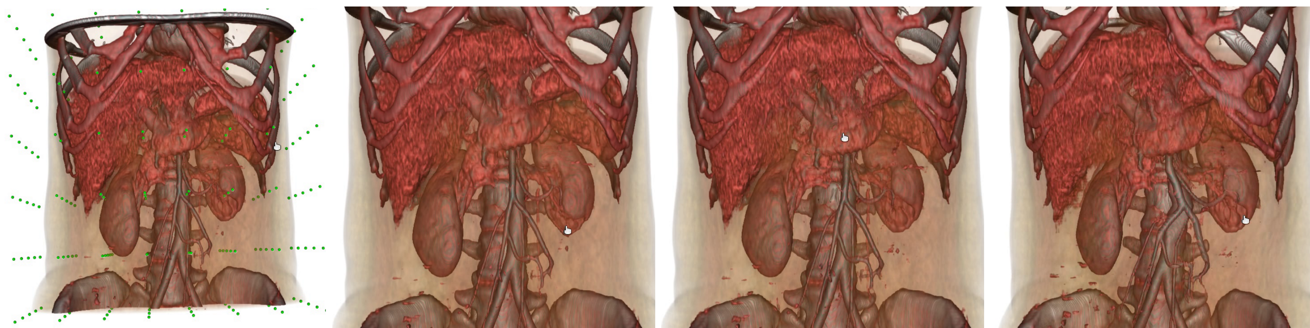

Figure 1: *On the left, a 3D medical image with the nodes of a tetrahedral mesh overlaid. The next four snapshots show, from left to right, interactive deformations of a kidney, the heart, and abdominal vessels. The* $256 \times 160 \times 122$ *volume is deformed at* 67*fps.*

Abstract

Many inherently deformable structures, such as human anatomy, are often represented using a regular volumetric discretization, e.g., in medical imaging. While deformation algorithms employ discretizations that deform themselves along with the material, visualization algorithms are optimized for regular undeformed discretizations. In this paper, we propose a method to transform high-resolution volume data embedded in a deformable tetrahedral mesh. We cast volume deformation as a problem of tetrahedral rasterization with 3D texture mapping. Then, the core of our solution to volume data deformation is a very fast algorithm for tetrahedral rasterization. We perform rasterization as a massively parallel operation on target voxels, and we minimize the number of voxels to be handled using a multi-resolution culling approach. Our method allows the deformation of volume data with over 20 million voxels at interactive rates.

CR Categories: I.3.3 [Computational Geometry and Object Modeling]: Physically based modeling—;

Keywords: Volume data deformation, tetrahedral rasterization

1 Introduction

Regular 3D grids are a popular way to store dense volumetric data. They are often used to capture and represent volumetric information of anatomy or other biological forms, notably in medical imaging [Sonka 2000]. Volume rendering offers a convenient way to illustrate in a single view the internal structures of solid volumetric objects stored in regular 3D grids [Rezk-Salama et al. 2008].

Elastic deformations, on the other hand, are typically solved by discretizing the deformation field on a Lagrangian mesh, i.e., a mesh that moves and deforms along with the material, as this enables trivial mass conservation. To deform and manipulate volume data, e.g., for medical planning applications, the data is typically segmented and meshed, and the deformed data is visualized using surface meshes [Shi and Malik 2000]. As a result, the visualization of the deformed data misses the full-resolution volumetric detail present in the original 3D grid.

Instead, we propose a method that takes as input a deformation field discretized on a tetrahedral mesh, and uses it to warp the original volume data into a new 3D grid. Our method is independent of the technique used to compute the deformation (See [Nealen et al. 2006] for a survey of deformation techniques). The data in the resulting grid can be visualized with standard volume rendering algorithms to reveal its full volumetric detail. We pose the problem of warping the volume data as tetrahedral mesh rasterization with 3D texture mapping, and the central contribution of our work is an extremely fast method for tetrahedral rasterization. As an example, the human torso data in Fig. 1, with 4.99M voxels ($256 \times 160 \times 122$), is deformed at 67fps. After a discussion of related work, in Section 3 we describe a massively parallel 3D image warping approach based on barycentric mappings. To further accelerate rasterization, we introduce a parallel culling algorithm described in Section 4. We conclude the paper with a discussion of performance and results.

2 Related Work

Rasterization of geometric primitives to a grid data structure is a largely studied problem, as it constitutes a key element of current GPU rendering algorithms [Fatahalian et al. 2009; Laine and Karras 2011]. They rasterize triangles into a 2D grid, and there are mainly two approaches to parallelize the process. One approach is to parallelize on a triangle basis. Each processor handles one triangle, computes an axis-aligned bounding box (AABB) around the triangle, and then processes internal pixels testing for inclusion in the triangle [Liu et al. 2010; Fatahalian et al. 2009]. The second approach is to parallelize on a tile basis. A first step assigns triangles to a tile of pixels, and then pixels are processed in parallel testing the list of triangles [Seiler et al. 2008; Eisenacher and Loop 2010].

Our approach for 3D rasterization of tetrahedra combines ideas from these two approaches. We parallelize at the voxel level, but

we produce candidate voxels based on the AABB of the rendered tetrahedron. In addition, we face different problems than traditional triangle rasterization algorithms. As opposed to triangle rasterization, in our setting the tetrahedral mesh constitutes a partition of space, hence only one tetrahedron covers each grid point. At the same time, the 3D AABB of a tetrahedron produces many more false candidate voxels than the false candidate pixels produced by the 2D AABB of a triangle.

3D rasterization of tetrahedra has also been studied, although the currently published algorithms work by traversing scan planes and scan lines [Rueda et al. 2004]. This approach is difficult to parallelize with effective load balancing, whereas our proposed method produces extremely uniform workload across processors.

Yet another related problem is the voxelization of triangle meshes. Current parallel approaches parallelize the voxelization on tiles, and construct an A-buffer per tile as a first culling approach [Schwarz and Seidel 2010; Pantaleoni 2011]. Surface voxelization, although connected to volume voxelization, also suffers different difficulties. Primitives occupy fewer voxels, but their AABBs produce many more false candidate voxels.

One of the problems that needs to be solved as part of our algorithm is a tetrahedron-cube intersection test. One possibility is to extend existing methods for triangle-square intersection [Akenine-Möller and Aila 2005]. Another possibility is to build on the general separating axis test for simple convex primitives [Gottschalk et al. 1996]. However, we exploit the fact that, in our problem, cubes are actually cells of a grid, and we design a faster algorithm that works in two steps: grid point classification followed by conservative tetrahedron-cube intersection test.

Our tetrahedral rasterization algorithm is intended as a method for volume data deformation. Other techniques have also been used for this purpose, such as deforming planes with semi-transparent textures that are rendered front to back [Nesme et al. 2010]. Instead, we propose a method that deforms the full volume data and allows the application of volume raycasting. More similar to our approach is the deformation method of Goksel and Salcudean [2009], who also map the deformation of a tetrahedral mesh using a texture mapping approach. However, their method to map deformed tetrahedra to voxels follows a scanline approach, and their interactive visualizations are limited to 2D images. Yet another possibility would be to apply volume raycasting on the deformed tetrahedral mesh [King et al. 2001; Georgii and Westermann 2006], but the resolution of the tetrahedral mesh is too low in our case, and rendering a high-resolution tetrahedral mesh would be very inefficient.

3 3D Grid Warping

The input data to our method is a regular 3D voxel grid G_0, where the voxels store a scalar field c_0 (which could be extended to vector or tensor fields). In addition, the method takes as input a tetrahedral mesh M_0, which may partially or completely embed the grid. Given a deformed tetrahedral mesh M_1, we wish to compute a deformed scalar field c_1 on an output 3D voxel grid G_1. We assume that the deformation field is linearly interpolated inside each tetrahedron.

We propose a massively parallel method to compute the deformed scalar field c_1, by rasterizing the deformed tetrahedra onto the output grid G_1, and assigning values of the scalar field as a texture mapping process. We trivially define the assignment of deformed scalar values through barycentric mappings inside each tetrahedron. Formally, given a point with barycentric coordinates $\mathbf{b}$ inside a tetrahedron T, and with undeformed (resp. deformed) position $\mathbf{x}_0$ (resp. $\mathbf{x}_1$), we define two barycentric mappings: $\beta_0 : \mathbf{x}_0 \rightarrow \mathbf{b}$ and $\beta_1 : \mathbf{x}_1 \rightarrow \mathbf{b}$. Given matrices $\mathbf{X}_0$ and $\mathbf{X}_1$ whose columns are formed respectively by the undeformed and deformed positions of the nodes of T, and a vector of ones $\mathbf{1}$, the mappings β_0 and β_1 are defined as the linear transformations

$$\mathbf{b} = \beta_0(\mathbf{x}_0) = \begin{pmatrix} \mathbf{X}_0 \\ \mathbf{1}^T \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{x}_0 \\ 1 \end{pmatrix} = \mathbf{B}_0\,\bar{\mathbf{x}}_0, \tag{1}$$

$$\mathbf{b} = \beta_1(\mathbf{x}_1) = \begin{pmatrix} \mathbf{X}_1 \\ \mathbf{1}^T \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{x}_1 \\ 1 \end{pmatrix} = \mathbf{B}_1\,\bar{\mathbf{x}}_1. \tag{2}$$

Here, $\bar{\mathbf{x}}$ represents a point $\mathbf{x}$ in homogeneous coordinates.

In practice, to assign the deformed scalar value of a voxel with position $\mathbf{x}_1$, we simply fetch the scalar value from the input point $\mathbf{x}_0$ with the same barycentric coordinates. This operation is formally defined as $c_1(\mathbf{x}_1) = c_0(\beta_0^{-1}(\beta_1(\mathbf{x}_1)))$. In our results, we have implemented the texture read operation $c_0(\mathbf{x}_0)$ as a trilinear interpolation of voxel values. Note also that $\mathbf{x}_0$ and $\mathbf{x}_1$ denote point positions in world coordinates, and they need to be transformed to and from device coordinates for texture access operations. This transformation accounts for deformations that change the overall size of the volume data, as well as anisotropic voxel spacing in the input data.

Next, we describe the massively parallel rasterization of the complete grid. We start with a basic algorithm, and in the next section we describe the addition of culling for improved efficiency. First, for each tetrahedron, we compute matrices of barycentric mappings $\mathbf{B}_0^{-1}$ and $\mathbf{B}_1$. We also compute an AABB for each deformed tetrahedron. Then, we process all voxels inside the AABB of each deformed tetrahedron, and compute their barycentric coordinates $\mathbf{b}$ following Eq. (2). For each voxel, we test if it lies inside its corresponding tetrahedron or not using the barycentric coordinates. If it does, then we compute and write the deformed scalar value.

The computation of barycentric mappings and the AABBs of tetrahedra are executed on the CPU. Processing the voxels, on the other hand, is remarkably amenable to GPU architectures. Our voxel rasterization algorithm is outlined in Algorithm 1. It barely suffers divergence, as the voxels that do not follow the main flow, i.e., those that lie outside the tetrahedron, are simply discarded. Texture lookups are not coalesced, but they enjoy high cache coherence.

Algorithm 1 GPU voxel rasterization algorithm

INPUT: *thread_id*, *block_id*, c_0
OUTPUT: c_1
corner = GetAABBCorner(*block_id*)
size = GetAABBSize(*block_id*)
$\mathbf{x}_1$ = *corner* + ComputePosInAABB(*thread_id*, *size*)
$\mathbf{B}_1$ = GetB1(*block_id*)
$\mathbf{b} = \mathbf{B}_1\,\bar{\mathbf{x}}_1$
if IsOutsideTet($\mathbf{b}$) **then**
  discard thread
end if
$\mathbf{B}_0^{-1}$ = GetB0Inv(*block_id*)
$\bar{\mathbf{x}}_0 = \mathbf{B}_0^{-1}\,\mathbf{b}$
c_1 = GetTrilinear(c_0, $\mathbf{x}_0$)

4 Hierarchical Culling

The volume of a tetrahedron is just $\frac{1}{6}$ of the volume of a prism defined by one of its corners and the three incident edges. This fraction of volume suggests that most of the voxels inside the AABB of a tetrahedron fall outside the tetrahedron itself. In fact, we found that, with the AABB-based rasterization described in the previous

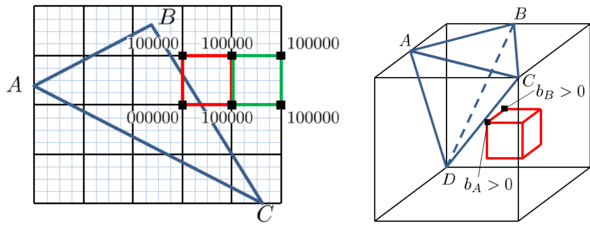

Figure 2: *Left: Examples of grid point masks for a triangle (A, B, C). The green cell can be culled because the barycentric coordinate of A is < 0 for its 4 vertices. Right: In 3D, our culling algorithm may produce false positives for cells close to an edge of a tetrahedron, such as the red cell in the figure.*

section, only 14.3% of the candidate voxels fall inside their corresponding tetrahedron. Next, we describe a hierarchical culling approach that reduces dramatically the voxels to be rasterized.

4.1 Grid Point Masks

For each tetrahedron in the deformed mesh M_1, we define a coarse grid with a spacing of N voxels. Each coarse cell encloses N^3 voxels of the output grid G_1, and if a cell does not intersect the tetrahedron, then its complete batch of enclosed voxels can be culled. Instead of testing for exact cell-tetrahedron intersection, we compute a spatial classification of coarse grid points, and then apply a conservative culling of coarse cells.

Based on barycentric coordinates $\mathbf{b} = \begin{pmatrix} b_A & b_B & b_C & b_D \end{pmatrix}^T$ for a tetrahedron (A, B, C, D), we define 8 half-spaces $b_A < 0$, $b_A > 1$, $b_B < 0$, $b_B > 1$, $b_C < 0$, $b_C > 1$, $b_D < 0$, and $b_D > 1$. Then, for each coarse grid point, we compute an 8-bit mask where each bit classifies the point w.r.t. one of the 8 half-spaces.

4.2 Cell Culling

Our culling algorithm is based on the following theorem. Given the 8 half-spaces of a tetrahedron as defined above, and the 8 vertices of a coarse cell, if there is at least one half-space such that all 8 vertices lie inside, then the tetrahedron and the cell do not intersect. As a corollary, all voxels inside the cell can be culled and do not need to be rasterized.

Based on this theorem and the readily available grid point masks, the culling of cells can be trivially executed as follows. For each cell, we perform a logical AND operation of the masks of its 8 vertices. The cell can be culled if the value of its mask is not 0. Fig. 2-left shows examples of point and cell mask computations.

In 2D, cell-triangle culling is exact if all 4 vertices of a cell are inside the AABB of the triangle. In 3D, cell-tetrahedron culling is conservative and may produce false positives for cells close to an edge of the tetrahedron, as shown in Fig. 2-right. However, as we show in our results, the number of false positives is small in practice, and we achieve a good trade-off w.r.t. culling cost.

Our algorithm would easily allow additional levels of hierarchical culling and an octree-based refinement strategy. However, our results indicate that, with just one level of hierarchical culling, rasterization of valid voxels becomes the bottleneck; therefore, more sophisticated culling would not yield additional speed-up.

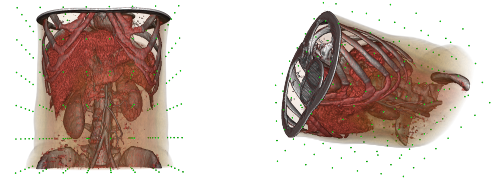

Figure 3: *Torso model for performance analysis. Left: in its initial configuration; Right: rotated* 45*deg around two orthogonal axes.*

4.3 Implementation Details

First, we process the coarse grid points of all tetrahedra and compute their masks. Subsequently, we process the cells of all tetrahedra and compute their masks too. Although these two procedures are highly amenable to GPU computation, we have found that a multi-core CPU implementation is fast enough and culling is not a bottleneck compared to rasterization, as we present in our results. After the computation of grid point masks and cell masks, we rasterize in parallel on the GPU all voxels of cells that cannot be culled. Each cell is treated as an AABB, and hence we follow the procedure already presented in Algorithm 1. In our tests, we found optimal performance by making the cell size the same as the CUDA block size. Once culling is executed, we construct on the CPU look-up tables that map each valid cell, through its corresponding CUDA *block_id*, to the AABB's size and location, and the barycentric mappings $\mathbf{B}_1$ and $\mathbf{B}_0^{-1}$.

5 Results and Evaluation

We have tested our fast rasterization algorithm on several volumetric deformation examples. All examples were executed on a 3.40GHz 8-processor Intel i7 CPU with 32GB of RAM, and a NVIDIA GeForce GTX 680 GPU with 2GB of RAM. Our parallel GPU rendering algorithm was coded using CUDA. We observed optimal processor occupancy with a CUDA block size of 512, and optimal balance between culling and performance trade-off with a cell size of $8 \times 8 \times 8 = 512$, i.e., equal to block size. For volume rendering, we have used VTK [Schroeder et al. 2004].

5.1 Performance Analysis

To evaluate the performance of our algorithm, we have designed a controlled deformation example, tested under various resolutions and settings. Fig. 3 shows two snapshots of a volumetric anatomical model, in its upright initial configuration (left), and rotated 45 degrees around two orthogonal axes (right). The model is meshed with an axis-aligned regular tetrahedral mesh, and the rotation creates a misalignment of axes and tetrahedral edges, increasing the volume of AABBs of tetrahedra. We have tested tetrahedral meshes ranging from 40 to 5000 tetrahedra, and volume data with resolutions ranging from $128 \times 128 \times 128$ to $512 \times 512 \times 512$.

First, we have evaluated the performance of our rasterization algorithm with no culling, as described in Section 3. Most of the time spent on rasterization is devoted to the computation of barycentric coordinates for voxels that fail the barycentric coordinate test. In fact, only 14.3% of the voxels processed in the GPU pass the barycentric test and are actually updated.

With our culling algorithm described in Section 4, on the other hand, 51% of the voxels processed in the GPU pass the barycentric test and are actually updated. The total rasterization time achieves

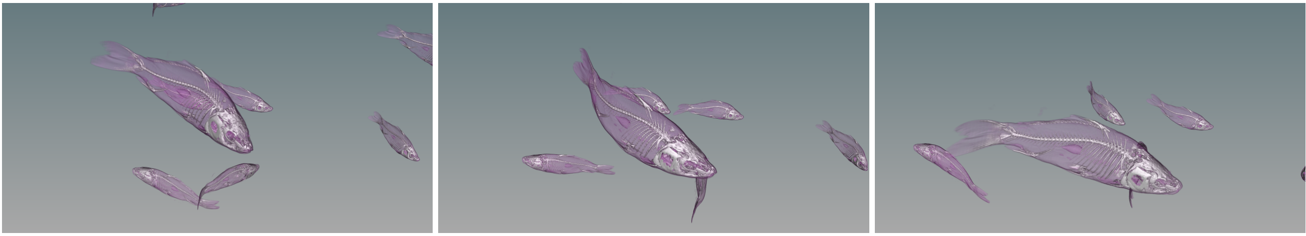

Figure 4: *Animation of swimming carps with raycasted volume visualization. Each carp is represented using a* $204 \times 202 \times 512$ *volume and deformed using a* 35*-tetrahedra mesh. Our rasterization algorithm runs at* 57.87*ms per carp.*

a speed-up of up to $1.5\times$. Fig. 5 shows graphs of total rasterization time per frame vs. volume data size, with and without culling. The speed-up becomes larger with larger datasets. For this comparison, we used a tetrahedral mesh with 1080 tetrahedra.

Our voxel batch culling takes only 7% of the total cost on average, and culls away 68% of the unnecessary voxels. As described in Section 4, we have used a parallel CPU implementation for the computation of grid point masks and cell masks. On an 8-core machine, it achieves up to $2.6\times$ speed-up over a single core implementation.

We have also evaluated the influence of the tetrahedral mesh resolution on performance, as shown in Fig. 6. The plot shows the total rasterization time per frame vs. tetrahedral mesh size, with and without culling, for the rotated configuration shown in Fig. 3. The data in the plot was collected for a volume with 233M voxels after the rotation. As expected, with culling, performance tends to decrease with denser tetrahedral meshes, as more tetrahedra need to be processed, and the cell resolution reduces the culling efficiency.

5.2 Simulation Examples

Our first two examples show the potential of our technique for animation purposes. Fig. 4 shows 8 carp models swimming, due to a scripted procedural deformation applied to their embedding tetrahedral meshes. Each carp is modeled using 35 tetrahedra, and the volume dataset consists of 21M voxels in the undeformed state. Each carp is rasterized in 57.87ms on average. Fig. 7 shows 6 oranges falling and rolling on a plane. We have modeled the deformation using a linear corotational finite element model [Müller and Gross 2004], and we have simulated frictional contact with the plane considering only collisions of the nodes of the tetrahedral mesh. Each orange is modeled using 160 tetrahedra, and the volume dataset consists of 6.3M voxels in the undeformed state. Each orange is rasterized in 15.64ms on average.

But, in addition to animation, our work is largely motivated by medical planning applications, with the possibility to provide an interactive volume manipulation and deformation tool. As a test example, we present the interactive deformation of the torso model in Fig. 1. Manipulating the original volume data is appealing for medical applications, as the full detail of the data is still available during the deformation, and the volume rendering settings can be dynamically tuned. The example uses a finite element model with 700 tetrahedra, and a volume dataset with 4.99M voxels. The full model is rasterized in 14.86ms on average. Note that even though the deformations may be localized, we rasterize the full volume to test the performance of our algorithm.

We have modeled the inhomogeneity of tissue properties using standard tables to convert opacity values into mechanical parameter values, and we compute nodal masses and per-element stiffness matrices by integrating per-voxel mechanical parameters. Nevertheless, the material properties and the actual deformations do not pretend to appear realistic; we simply demonstrate the performance of our method and the interaction possibilities.

6 Conclusion

We have presented an algorithm to efficiently deform volumetric data by rasterizing an embedding tetrahedral mesh. The simplicity of the method is key for its high performance, as it enables processing all target voxels in parallel with very simple operations and practically no divergence. To further accelerate rasterization, we apply efficient multi-core CPU culling as a first step.

Our method suffers some limitations, such as the existence of false positives during culling. However, these false positives do not hurt performance significantly. Another limitation is the smoothing introduced by trilinear interpolation of input data. More costly filtering approaches would produce higher quality results.

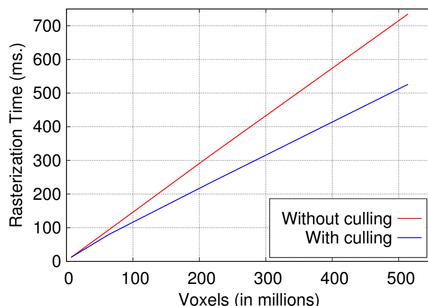

Figure 5: *Evaluation of performance (with and without culling) vs. volume resolution for the torso model in Fig. 3. We used a tetrahedral mesh with* 1080 *tetrahedra.*

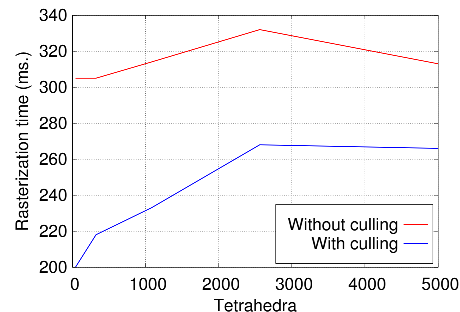

Figure 6: *Evaluation of performance (with and without culling) vs. number of tetrahedra for the torso model in Fig. 3. We used a volume with* 233*M voxels after the rotation.*

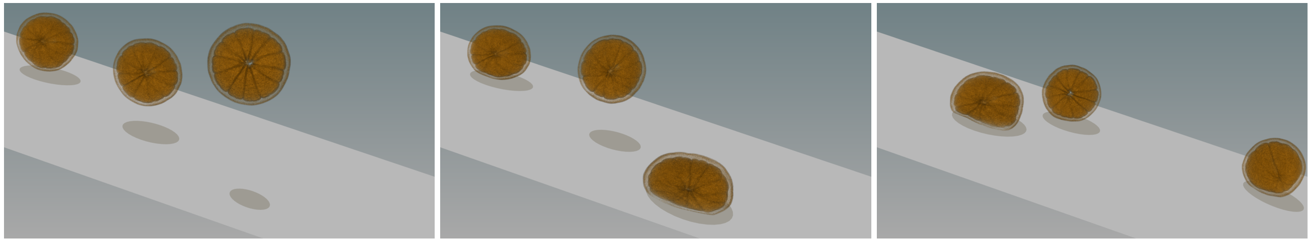

Figure 7: *Deformable oranges bounce and roll on a plane. Each orange is represented using a* $198 \times 199 \times 160$ *volume and deformed using a* 160*-tetrahedra mesh. Our rasterization algorithm runs at* 15.64*ms per orange on average.*

From an applied point of view, our method allows interactive editing, manipulation, and deformation of dense volume data. Its applicability could be extended by handling other types of mesh elements and basis functions, such as trilinear interpolation in hexahedra. This extension would require modifications to the mapping function and the culling algorithm.

Acknowledgements

We would like to thank the anonymous reviewers and the rest of the members of the GMRV team at URJC for their comments and help. The datasets for the carp and the orange were obtained from Stefan Roettger's volume library, and the human torso dataset was obtained from the OsiriX DICOM Viewer's site. This work was supported in part by the Spanish Ministry of Economy (grants IPT-2012-0401-300000 and TIN2012-35840) and the EU FEDER fund.

References

AKENINE-MÖLLER, T., AND AILA, T. 2005. Conservative and tiled rasterization using a modified triangle setup. *Journal of Graphics Tools 10*, 3, 1–8.

EISENACHER, C., AND LOOP, C. 2010. Data-parallel micropolygon rasterization. In *Proc. of Eurographics Short Papers*.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, 59–68.

GEORGII, J., AND WESTERMANN, R. 2006. A generic and scalable pipeline for gpu tetrahedral grid rendering. *Visualization and Computer Graphics, IEEE Transactions on 12*, 5, 1345–1352.

GOKSEL, O., AND SALCUDEAN, S. E. 2009. B-mode ultrasound image simulation in deformable 3-d medium. *IEEE Transactions on Medical Imaging 28*, 11, 1657–1669.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 171–180.

KING, D., WITTENBRINK, C., AND WOLTERS, H. 2001. An architecture for interactive tetrahedral volume rendering. In *Volume Graphics 2001*, K. Mueller and A. Kaufman, Eds., Eurographics. Springer Vienna, 163–180.

LAINE, S., AND KARRAS, T. 2011. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 79–88.

LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2010. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '10, 75–82.

MÜLLER, M., AND GROSS, M. 2004. Interactive virtual materials. In *Proceedings of Graphics Interface 2004*, 239–246.

NEALEN, A., MULLER, M., KEISER, R., BOXERMAN, E., AND CARLSON, M. 2006. Physically based deformable models in computer graphics. In *Computer Graphics Forum*, vol. 25, Wiley Online Library, 809–836.

NESME, M., FAURE, F., AND PAYAN, Y. 2010. Accurate interactive animation of deformable models at arbitrary resolution. *International Journal of Image and Graphics 10*, 2.

PANTALEONI, J. 2011. Voxelpipe: a programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 99–106.

REZK-SALAMA, C., HADWIGER, M., ROPINSKI, T., AND LJUNG, P., 2008. Advanced illumination techniques for gpu-based volume raycasting. ACM SIGGRAPH Asia Course Notes.

RUEDA, A. J., SEGURA, R. J., FEITO, F. R., DE MIRAS, J. R., AND OGÁYAR, C. 2004. Voxelization of solids using simplicial coverings. In *Proc. of WSCG*.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 2004. The Visualization Toolkit: An object-oriented approach to 3D graphics, 3rd edition. Tech. rep., Kitware Inc.

SCHWARZ, M., AND SEIDEL, H.-P. 2010. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph. 29*, 6, 179:1–179:10.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3, 18:1–18:15.

SHI, J., AND MALIK, J. 2000. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 22*, 8, 888–905.

SONKA, M. 2000. *Handbook of medical imaging: medical image processing and analysis*. SPIE-International Society for Optical Engineering.

View publication stats