Pixel-Exact Rendering of Spacetime Finite Element Solutions

Yuan Zhou* Michael Garland* Robert Haber†

Center for Process Simulation and Design
University of Illinois at Urbana–Champaign

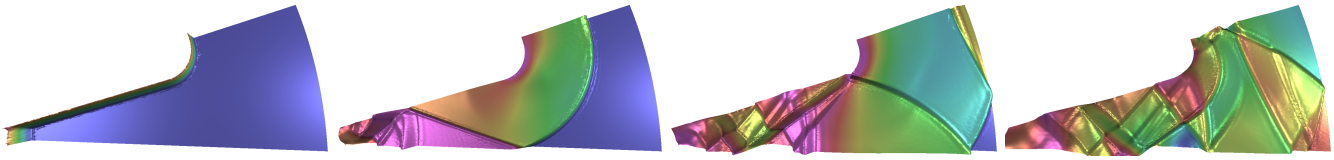


Figure 1: Shock waves propagating through a partial 2-D cross-section of a solid rocket booster.

ABSTRACT

Computational simulation of time-varying physical processes is of fundamental importance for many scientific and engineering applications. Most frequently, time-varying simulations are performed over multiple spatial grids at discrete points in time. In this paper, we investigate a new approach to time-varying simulation: spacetime discontinuous Galerkin finite element methods. The result of this simulation method is a simplicial tessellation of spacetime with per-element polynomial solutions for physical quantities such as strain, stress, and velocity. To provide accurate visualizations of the resulting solutions, we have developed a method for per-pixel evaluation of solution data on the GPU. We demonstrate the importance of per-pixel rendering versus simple linear interpolation for producing high quality visualizations. We also show that our system can accommodate reasonably large datasets — spacetime meshes containing up to 6 million tetrahedra are not uncommon in this domain.

CR Categories: I.6.7 [Computing Methodologies]: Simulation and Modeling—Simulation Support Systems; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: pixel-exact rendering, pixel shaders, spacetime finite elements

1 INTRODUCTION

Providing techniques for displaying time-varying data produced by computational simulation of physical phenomena is a key problem in the visualization area. Simulations across a very broad range of applications — from fluid dynamics to quantum mechanics and elastodynamics — are frequently performed via finite element methods. One particularly natural approach to visualizing the result has always been to animate a series of constant-time snapshots of the solution data.

Most visualization systems in use today render finite element solutions using piecewise linear representations; height fields and color fields are particularly common. However, the actual solutions produced by finite element methods are frequently higher order functions. Rendering these solutions with piecewise linear approximations can seriously misrepresent the result of the simulation.

We demonstrate that the capabilities of modern programmable GPUs can support a substantial improvement in the visual fidelity of displayed solutions. We utilize custom pixel shaders to evaluate solution polynomials on a per-pixel basis. Combining this with adaptive subdivision of the height field, we are able to provide pixel-exact renderings of the finite element solution.

We investigate these visualization issues in the context of spacetime discontinuous Galerkin (SDG) methods. Unlike traditional finite element methods, SDG methods represent the solution within each element independently. While not guaranteeing continuity between adjacent elements, this has several advantages from the standpoint of efficient local computation and parallelization. From our standpoint, this is also advantageous because it means that each element can be rendered in isolation from the others. This is a natural fit for graphics hardware, which generally disallows non-local data access, say between elements or between adjacent nodes.

2 RELATED WORK

Our focus is on the visualization of spacetime finite element solutions for time-varying problems. In this paper, we concentrate specifically on problem areas involving 2-dimensional spatial domains. Our 3-dimensional spacetime is covered by a tetrahedral mesh. However, it is important to note that it is not a simplicial complex as we allow non-conforming tessellations.

There has obviously been a great deal of prior work on rendering of 3-D volume data. There have also been several methods proposed for rendering spacetime volumes. Here we discuss the most salient examples of this prior work.

Spacetime Rendering There are multiple possible fundamental approaches to visualizing spacetime volumes. For our purposes here, most standard visualization techniques can be applied to our 3-D spacetimes. The more general setting of 4-D spacetime requires somewhat more generalized techniques [24].

Arguably the most common approach to visualizing spacetime data is by time slicing. Given that one dimension of the spacetime volume is temporal, it is extremely natural to extract and animate multiple spatial cross-sections of the spacetime. Vis5D [10] provides a good example of a system for time-varying visualization that makes extensive use of temporal slicing. Woodring *et al.* [29] extend this notion of slicing for direct rendering of 4-D spacetime volumes.

*Dept. of Computer Science, {yuanzhou,garland}@uiuc.edu
†Dept. of Theoretical and Applied Mechanics, r-haber@uiuc.edu

One natural approach to rendering both spacetime and spatial cross-sections is direct rendering via splatting. Splatting of 3-D spacetime volumes can be implemented directly with traditional splatting methods [28]. It can also be nicely generalized to 4-D spacetime volumes [17] and even to general n-D hypervolumes [2]. Splatting also fits quite nicely within the framework of traditional texture mapping hardware.

Another common approach to spacetime rendering is via isosurfacing. The most popular method for extracting isosurfaces from regular grids is Marching Cubes [12]. Similar algorithms have been developed for irregular tetrahedral grids [22]. Marching methods of this sort have been generalized to higher dimensions [4] although the necessary lookup tables can become quite large [3].

GPU-Assisted Rendering One of the most important recent developments in graphics hardware is the evolution from fixed-function pipelines to programmable GPUs. This has provided many opportunities for implementing more advanced rendering algorithms directly in hardware. Of particular importance to us is the fairly recent move to full floating point support within the graphics pipeline.

In the past, there has been substantial work on designing custom hardware for volume rendering. More recently, techniques have been developed to efficiently perform such rendering tasks on standard PC hardware [11, 9]. Guthe *et al.* [8] and Weiler *et al.* [27] have both demonstrated GPU-based techniques for rendering tetrahedral volumes.

Programmable GPU features have also been used to evaluate higher-order geometry elements. Vlachos *et al.* [26] transparently convert triangles with per-corner normal data to piecewise-polynomial patches. Losasso *et al.* [13] evaluate bi-cubic B-splines to produce smooth surfaces that are C^2 almost everywhere, except at certain C^1 cut vertices.

There are now multiple systems that have been developed for providing higher-level language constructs for GPU programming. Both Proudfoot *et al.* [19] and Mark *et al.* [15] describe C-like shader languages. In contrast, McCool *et al.* [16] use a metaprogramming paradigm to embed shader programs directly in C++ code. We rely on the Cg system [15] for developing our vertex and fragment programs.

3 Finite Element Simulation

The visualization system we describe in this paper is a part of a larger project aimed at developing new methods for simulating time varying physical processes. Before describing our rendering techniques, we first outline the finite element problem domain in which they are being used.

3.1 Spacetime DG Method

The standard finite element approach to simulating physical processes over time is semi-discrete. The spatial domain is discretized with a fixed mesh, inducing a discretized set of differential equations that are solved by a time-marching integration scheme. Usually, a uniform time-step is used across the entire spatial domain, thus effectively computing a solution over a fixed mesh at several constant points in time.

Spacetime discontinuous Galerkin methods [23, 6, 14, 30, 31, 18] are a relatively new class of finite element methods that comprise an interesting alternative to semi-discrete methods. Unlike conventional finite element models, SDG methods work with meshes covering the entire spacetime analysis domain. The SDG algorithm weakly enforces the governing equations over each spacetime element, eliminating the need for a separate time integration procedure. Another distinguishing feature of SDG methods are their use of discontinuous basis functions with support on individual elements, rather than the usual continuous bases. This approach eliminates artificial coupling between adjacent elements when the mesh satisfies certain causality constraints.

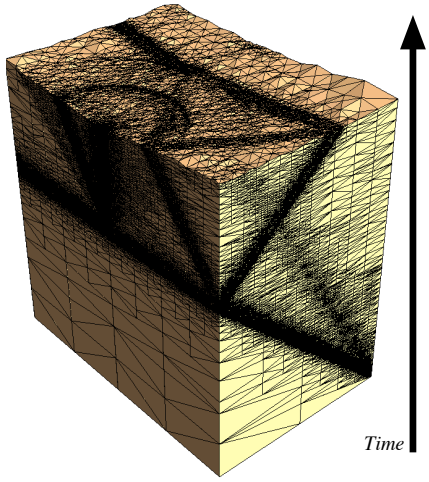


Figure 2: Spacetime mesh for a crack-tip wave scattering problem. Spacetime tetrahedra are formed by repeatedly lifting vertices of a planar space mesh forward in time. Local refinement occurs along propagating shock fronts.

3.2 Spacetime Meshing

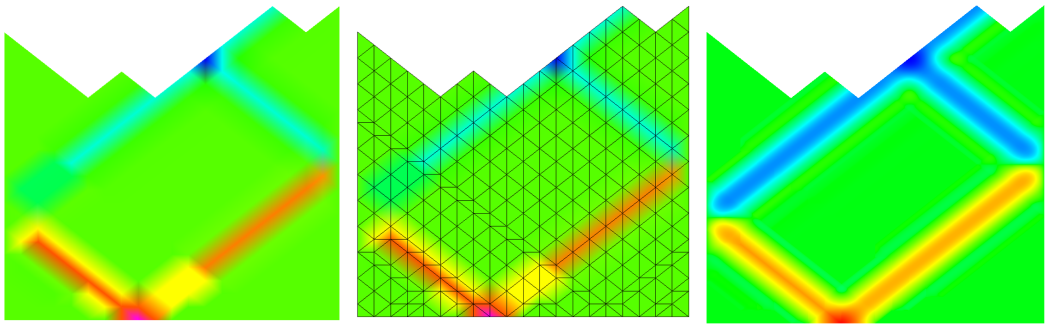
The spacetime meshes used in the underlying solution system are constructed using the *Tent Pitcher* algorithm [25, 7]. It is the first algorithm to build graded spacetime meshes over arbitrary simplicially meshed spatial domains. Unlike most traditional approaches, the Tent Pitcher algorithm does not impose a fixed global time step on the mesh, or even a local time step on small regions of the mesh. Rather, it produces a fully unstructured simplicial spacetime mesh, where the duration of each spacetime element depends on the local feature size and quality of the underlying space mesh.

Given a triangular mesh of some planar domain, Tent Pitcher meshes the target spacetime domain using an advancing front algorithm. Elements are added to the evolving mesh in small patches by moving a vertex of the front forward in time. The amount by which a vertex may be lifted into the future is limited by local causality constraints. The SDG solution is computed locally within each new patch as soon as it is created. The mesh can also be adaptively refined or coarsened in response to *a posteriori* error estimates computed by the numerical code [1]. This adaptation generates non-conforming spacetime meshes; two adjacent spacetime elements may not share a common face. Figure 2 shows an example of a spacetime mesh built by this system.

4 Rendering System

We have developed a visualization system designed to display the results of a spacetime discontinuous Galerkin system, such as the one outlined in the previous section. From the standpoint of the renderer, these spacetime DG solutions have the following important properties:

- We are presented with a (potentially nonconforming) simplicial decomposition of spacetime.



(a) Per vertex shading (b) Underlying spacetime mesh (c) Pixel-exact shading

Figure 3: A simple 1-D linear elastodynamic finite element solution over a 2-D spacetime. The solution is piecewise cubic within each spacetime triangle, and per-vertex shading (a) produces a very poor representation of the actual solution (c).

- The solution within each element is given independently, and these solutions are represented with higher-order basis functions.

Currently, we are only working with problems that involve 1-D and 2-D spatial domains. Therefore, the resulting spacetimes are either 2-D triangulations or 3-D tetrahedralizations, respectively. Figure 2 shows an example of the kind of spacetime mesh our visualization system is designed to process. This is a simulation of a crack-tip wave scattering problem (see §5 for more details). Shock waves propagating through the medium are clearly visible from the resulting mesh refinement. The algorithm used for performing this refinement (and coarsening) produces non-conforming spacetime elements whenever it adapts the mesh density.

Our goal is to visualize the simulation as a time-varying process. We do this by constructing multiple constant-time slices through the spacetime mesh. Each time-slice represents the state of the spatial domain at a constant point in time. We render each time-slice and animate the result.

We assume that the user wishes to display one or more scalar fields computed from the underlying solution. For any given visualization, we restrict the possible number of scalar fields to 2, mapping one to height and one to color. Each scalar field can be described using a polynomial on a per-element basis. We aim to produce the most accurate possible rendering of these higher-order scalar fields. To do so, we take advantage of the ability of modern GPU hardware to evaluate fairly complex functions on a per-pixel basis.

To illustrate the importance of per-pixel rendering, consider the example shown in Figure 3. This is a very simple linear elastodynamic system over a 1-D space domain — the entire triangulated spacetime is shown. This is a simulation of a displacement propagating through a rigid bar fixed at one end. The initial displacement at the center of the bar travels with constant wavespeed towards the two ends of the bar. The wavespeed is a constant depending on the material of the bar. The wave reflects out of phase from the the fixed end of the bar and travels to infinity past the free end. The difference between computing the color field on a per-vertex basis and a per-pixel basis is striking.

4.1 Slicing Spacetime

The spacetime mesh we are given consists of a set of vertices V and a set of tetrahedral elements T. Each vertex $v_i = (x_i, y_i, t_i)$ is a point in the 3-D spacetime, with two spatial coordinates (x_i, y_i) and a time value t_i. A given tetrahedron is a quadruple of indices $\tau = (i, j, k, l)$ referencing the vertices that are its corners. We do not assume that the spacetime mesh is a simplicial complex. Our only assumption is that each tetrahedron is non-degenerate (i.e., its volume in spacetime is non-zero).

To render the state of the simulation at some time t_i, we must find the intersection of the spacetime mesh with the plane $t = t_i$. In particular, we want to find the set of all spacetime elements that intersect this plane. Once we have found this set of tetrahedra, we cut each one with the plane. This produces a set of polygons (either triangles or quadrilaterals) that all exist at a common instant in time. Because these polygons are produced by slicing tetrahedra, they vary considerably in size. Indeed, as the time plane moves forward in time, the mesh edges move as the plane cuts the tetrahedra at different points. This results in significant temporal aliasing artifacts when using per-vertex rendering.

Because a single dataset covers the entire spacetime extent of a simulation, the meshes that we work with can grow quite large. For example, the spacetime mesh shown in Figure 2 has roughly 11 million tetrahedral elements. It is therefore necessary to organize the data so that it can be accessed efficiently. Fortunately, our access pattern makes this quite straightforward.

To create an animation of the time-varying solution, we begin with a time-slice at time $t = 0$. After rendering each frame, we advance the time plane into the future by some small increment Δt. For each tetrahedron, we can easily compute its minimum time value — the time t of its lowest vertex. We then sort the tetrahedra based on this minimal time value. This makes it particularly easy to index the entire spacetime efficiently.

4.2 Displaying Scalar Fields

Once we have constructed a time-slice mesh, we need to render the appropriate scalar fields at that point in time. As mentioned previously, we consider the case where the user wishes to draw two independent scalar fields, one which we map to height and the other which we map to color. In general, these scalar fields might be any aribitrary functions.

For the particular examples given in this paper, the underlying solution is a displacement field represented in each element by a cubic polynomial. The two scalar fields of interest to us are: (1) *velocity magnitude*, which we map to height, and (2) *strain energy density*, which we map to color using a log scale. Both are derived analytically from the underlying displacement field. We represent

our polynomials using a complete cubic basis, requiring 20 coefficients per element. The finite element solver computes its solution polynomials for each element in a local coordinate system, requiring that we store an additional 6 transformation coefficients per element.

Our system is designed to move all scalar computation onto the GPU. The task of the host processor is to manage the overall spacetime dataset and to construct time-slices. What is sent to the graphics hardware is a planar mesh with per-polygon polynomials. The work in the GPU is shared between a vertex and a fragment program, which compute the height and color fields, respectively.

4.2.1 Elastodynamics Setting

The examples we present are all elastodynamic problems for which the finite element solver is computing a spacetime *displacement* field. The solution assigns a 2-D displacement vector $\mathbf{u}$ to every point (x,y,t) in spacetime. Within each spacetime element, the displacement field is represented with a complete cubic polynomial basis that contains 20 monomial functions:

$$\mathbf{u}(x,y,t) = \sum_{\alpha=1}^{20} \mathbf{c}_\alpha m^\alpha(x,y,t)$$

Here the 2-D coefficient vectors $\mathbf{c}_\alpha$ are the finite element solution data and m^α ranges over the cubic monomials $(1,x,y,t,x^2,xy,xt,y^2,yt,t^2,x^3,x^2y,x^2t,xy^2,xyt,xt^2,y^3,y^2t,yt^2,t^3)$. For the convenience of the solver, these solutions are represented in a spacetime coordinate system local to the current element. Given a spacetime point $\hat{\mathbf{p}}$ described in the global coordinate system, the transformation to local coordinate system of an element is given by

$$\mathbf{p} = \begin{bmatrix} 1/w_x & 0 & 0 \\ 0 & 1/w_y & 0 \\ 0 & 0 & 1/w_t \end{bmatrix} (\hat{\mathbf{p}} - \hat{\mathbf{c}})$$

where $\mathbf{c}$ is the center of the element (in the global coordinate frame) and w_x, w_y, w_t are the extents of the element along the x,y,t axes.

The spacetime *velocity* field $\mathbf{v}$ is the time derivative of the displacement field,

$$\mathbf{v}(x,y,t) = \sum_{\alpha=1}^{20} \mathbf{c}_\alpha \dot{m}^\alpha(x,y,t)$$

Note that only 10 of the functions $\dot{m}^\alpha$ are non-zero, and thus only 10 of the coefficient vectors $\mathbf{c}_\alpha$ are relevant to this computation.

The *strain energy density* U is the spacetime scalar field given by

$$U(x,y,t) = \frac{1}{2}\varepsilon(x,y,t) : \mathbf{C}(x,y)\varepsilon(x,y,t)$$

in which $\varepsilon = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)$ is the *strain tensor* and $\mathbf{C}$ is the fourth-order *elasticity tensor* that maps the strain tensor into the stress tensor. In our system, this is mapped to the color field on a log scale.

4.2.2 Vertex Program: Height Field Evaluation

The time-slice mesh constructed by the application is planar; each vertex simply encodes its position in the 2-D spatial domain $(x,y,0)$. The task of the vertex program is to compute the magnitude of the velocity field $z = \|\mathbf{v}(x,y,t)\|$ at the given corner of the current polygon. and to displace the corner

To evaluate the velocity magnitude, the vertex program requires a total of 26 scalar parameters: 6 coefficients for the global-to-local transformation and 2 scalars for each of the 10 relevant solution coefficients $\mathbf{c}_\alpha$. As these parameters vary on a per-polygon basis, they are passed to the vertex program via texture registers. The current time t is a global constant that only changes on a per-frame basis.

Note that, by evaluating the height only at the vertices of the mesh, we are constructing a piecewise-linear approximation of the true height field. For a piecewise-cubic displacement field, the true velocity magnitude field would be piecewise-quadratic. In principle, we could compute per-pixel heights using a root finding in the pixel shader. This would be similar in spirit to GPU-based ray tracing [20, 5]. However, full per-pixel height evaluation yields a very small increment in quality versus simpler methods, and further stresses the already busy pixel shaders. Therefore, we have decided against this approach.

Rather than evaluating height on a per-pixel basis, we simply perform polygonal subdivision on the host processor. Any polygon which is deemed too large is quadrisected. This can be performed recursively if necessary. Highest fidelity results are achieved by quadrisecting based on the projected screen size of the portion of the height field represented by the polygon in question. However, except in extreme circumstances, we have found simple fixed subdivision based on an area threshold to be preferable. It avoids the substantial increase in CPU load required by the screen-space size estimates. Fixed subdivision patterns are also more amenable to hardware acceleration, using features such as render-to-vertex-array. Figure 4 illustrates the effect of subdivision.

Having finished its computation of the velocity magnitude, the vertex program performs two tasks. First, it displaces the current vertex to its proper position: $(x,y,0) \rightarrow (x,y,z)$, where $z = \|\mathbf{v}(x,y,t)\|$. Second, it uses texture registers to pass its 26 parameters plus the position (x,y,z) to the fragment program. For most current GPU architectures, 8 texture registers are available for data transfer to the fragment program. We use 1 register for transferring geometry, leaving 7 for parameter transfer. Notice that this allows us to transfer 28, rather than just 26, parameters to the fragment program. We take advantage of this otherwise unused bandwidth by passing an extra 2 coeffcients through the vertex program that it would not otherwise require; this data is then passed through to the fragment program.

4.2.3 Fragment Program: Color & Lighting

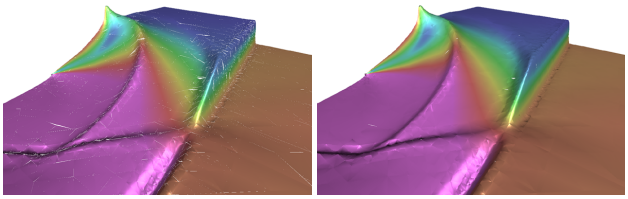
The task of the fragment program is twofold: to compute a color field and to compute pixel-exact lighting of the height field. The color field is computed by evaluating the strain energy density U at each pixel. Similarly, we light the surface by evaluating the normal of the height field at each pixel, and then use a standard Phong illumination model.

The color ramp used in generating the color field is simply a 1-D texture. This is provided by the user. The pixel shader converts the strain energy density U into a texture coordinate s using a log scale mapping:

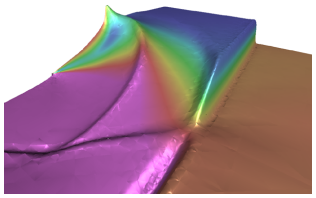
$$s = \frac{\log(U+1)}{\log(U_{\max}+1)}$$

The texture coordinate s is clamped to the range $[0,1]$ and used to lookup a color value in the ramp texture. The value $U_{\max}$ can either be computed as the maximum over the field or can be provided by the user (to exercise greater control over the color distribution).

The pixel shader requires 46 coefficients: 20 each for the x and y spatial derivatives of the displacement field and 6 for the local transformation. Of these 46, 28 are known or used by the vertex program and are passed by it to the pixel shader. The remaining 18 coefficients are passed to the pixel shader by the CPU in a texture rectangle. We use the NVIDIA `GL_TEXTURE_RECTANGLE_NV` extension to create these textures because of two important characteristics. First, it allows texel coefficients to be arbitrary floating point scalars, rather than limiting them to the range $[0,1]$. This is essential as it allows us to preserve the precision of the solution



(a) Raw polygons (b) With edge overdraw



(c) And subdivision as well

Figure 4: Drawing time-sliced polygons alone leads to noticeable cracks. Using edge overdraw plus polygon quadrisection eliminates these problems.

data. Second, it provides for exact texel addressing and does not perform any interpolation of neighboring texels, which would obviously produce spurious results in our setting.

For each frame, we must build a texture rectangle containing the relevant coefficient data. Each element is allocated a horizontal span of 6 texels, whose *rgb* values are used to store the required 18 coefficients. These 6-texel spans are packed into a texture rectangle such that they are never broken across rows. The maximum defined resolution of a texture rectangle is 4096×4096, thus we can pack $\llcorner 4096/18 \lrcorner = 227$ elements per row. Each texture rectangle can thus accommodate the data for a total of $4096 \times 227 = 929,792$ elements. Extremely large datasets might therefore require more than one texture per frame. However, as typical datasets currently have on the order of $50,000$ elements per frame, this upper limit is not at all constricting. The CPU packs element coefficients into the texture rectangle in the order in which the polygons will be drawn, thus the fragment shaders will access the texture in (approximately) scanline order.

To light the surface, we use a standard Phong illumination model. The diffuse and specular reflectances are simply scalar multiples of the color computed above. The pixel shader already has access to the coefficients necessary to compute the spatial derivatives of the height field function, and thus its normal. One slight problem arises when the velocity magnitude is 0 — the spatial derivatives of the height field will be undefined. However, it is clear that geometrically the height field is flat, and that its normal is simply $(0,0,1)$. It is also important to note that we do not need to perform any interpolation of normals over the polygon. At each pixel, we compute an *exact* normal vector directly from the underlying height field polynomial.

4.3 Discontinuity Antialiasing

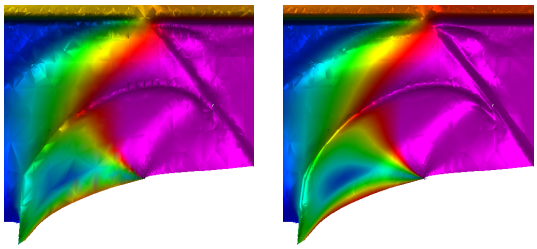
Recall that the solutions we are drawing are represented independently within each element. These solutions are not required to be fully continuous across element boundaries. Therefore, even for solutions with very tight convergence bounds, we can wind up computing subtly different scalar values along shared edges. Unless the solution has a fairly large error, this is generally not easily noticed in the color field. However, it can lead to very obvious artifacts in the height field. Specifically, even small height discrepancies can lead to aliasing during polygon rasterization that causes small cracks to appear in the height field (see Figure 4).

Our solution to this problem is to overdraw all edges shared between polygons. While this obviously increases the per-frame rendering time, it removes what would otherwise be very distracting aliasing artifacts. This edge overdraw approach is similar to the antialiasing approach adopted by Sander *et al.* [21]. However, our problem is somewhat easier. They need to blend lines smoothly with the underlying polygons to antialias discontinuity edges (e.g., silhouettes). We do not require blending, as we are only trying to fill gaps rather than blend discontinuities.

5 Results

In this section, we demonstrate some visualization results from our system on selected elastodynamic problems. All rendering was performed on a standard PC with a 2.4 GHz Pentium 4 processor, 1 GB of RAM, and an NVIDIA GeForce FX 5800 Ultra graphics card. On this hardware configuration, our renderer generally achieves interactive rates of roughly 10 frames per second on spacetimes in the range of 3–5 million tetrahedra.

Our first example — as seen in Figures 4, 5, and 6 — models crack-tip wave scattering within an elastic solid subjected to shock loading. A stationary crack embedded in a rectangular plate subjected to a spatially uniform tensile traction on its top and bottom edges. As shown in Figure 6, the crack covers the left half of the bottom boundary, with the crack tip in the center of the bottom edge. These solutions actually cover only the upper right-hand quadrant of the symmetric domain.



(a) Per-vertex color (b) Per-pixel color

Figure 5: Per-pixel color computation is clearly much more faithful to the underlying solution than per-vertex color computation.

In Figure 5 we see a comparison over a small portion of the solution between (a) per-vertex and (b) per-pixel computation. In both cases, we are using per-vertex height computations without subdivision. For the per-vertex color case, we compute colors only at the vertices and linearly interpolate them over the triangle. Note that we are using a darker color ramp to highlight the color differences. As with the much simpler example shown in Figure 3, we see that the per-pixel rendering provides a far better view of the actual solution being computed. Note in particular the substantial color distortion on the lower-left spike in the per-vertex rendering.

Figure 6 shows a sequence of constant time snapshots of the solution to the crack-tip scattering problem. The total spacetime mesh contains approximately 25 million tetrahedra, and there are roughly

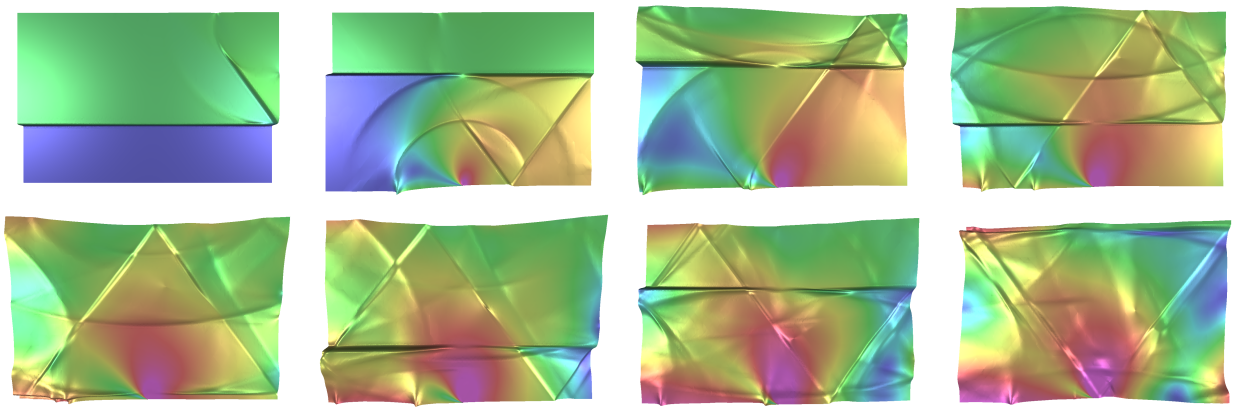


Figure 6: Time sequence showing a shock wave approaching a crack, which lies along the lower edge of the quadrant shown. The shock scatters off the crack tip located in the middle of the quadrant.

20,000 polygons per time-slice. The initial wave enters the domain from the top, reflects off the crack-tip at the bottom, and then continues to reflect back and forth across the domain. Both the wave fronts and color field are very well-resolved by the per-pixel rendering. It is important to remember that essentially all shading artifacts in this picture are a result of the structure of the solution. This is another important practical benefit gained from pixel-exact rendering. With inexact per-vertex color computations, it would be unclear whether visual artifacts were a result of the rendering or the solution. Here, we know that we are faithfully rendering the solution. Therefore, artifacts such as shading discontinuities are indicative of actual normal discontinuities in the field being computed. This makes our per-pixel rendering approach much more useful as a diagnostic tool, for assessing the quality of the computed solution, than a per-vertex rendering system would be.

In Figure 7 we examine wave scattering in a representative volume element for a fiber-reinforced composite material with stiff fibers embedded in a more flexible matrix. The fiber sections appear as circular inclusions in the model. As the shock wave passes through the medium, the inclusions begin to debond from the surrounding material. As before, our rendering system is able to resolve the complex wave and stress patterns quite well.

Figure 8 demonstrates the impact of our pixel-exact rendering in this example. The per-vertex rendering has many more color discontinuities than the per-pixel rendering. More importantly, we can see that the overall structure of the stress field appears substantially different. Specifically, compare the red regions around the central inclusion and the stress fields along the upper boundary. The stress patterns differ markedly in the per-vertex and per-pixel renderings.

Figure 9 shows our final example solution. Here we are seeing a single sector of a 2-D cross section of a solid rocket booster. Shock waves are propagating through the solid rocket fuel from the left, which points towards the center of the rocket where combustion has begun. This simulation produces a fairly complex wave pattern in the height field and an equally complex strain energy density field that is mapped to color. This complexity is quite nicely resolved — and at interactive rates — by our per-pixel rendering system. This data set contains a total of 4.7 million tetrahedra, with roughly 30,000 polygons per time-slice.

6 Conclusion

In this paper, we have outlined an approach for pixel-exact rendering of spacetime finite element solutions. The system we have described uses modern programmable GPU features to offload a sizeable portion of the visualization task onto the graphics hardware. This frees the CPU to devote all its resources to data management and user interaction. We have shown that computing lighting and color fields from higher-order polynomials is both possible and produces far greater visual fidelity than per-vertex rendering. We have also explored a fairly new application domain for visualization: spacetime discontinuous Galerkin finite element methods.

We believe that this work can be extended in a number of promising directions. One important area for future work is to address the problem of extremely large spacetimes. At the moment, our system assumes that the entire spacetime can be kept in main memory, whether through direct I/O or through memory mapping. However, long running simulations may produce far more data than can fit in a 32-bit virtual memory space. Out-of-core data management techniques will clearly become necessary. Taking advantage of near-term hardware advances, particularly the ability to render into vertex arrays, should make achieving highly accurate height fields much easier. It would also be very interesting to explore alternative spacetime rendering modalities. We have restricted our attention to time-slicing. Various direct pixel-exact renderings of spacetime might also provide useful information.

7 Acknowledgements

This research was funded in part by the National Science Foundation under an ITR grant DMR-0121695. We thank the numerous people involved in the CPSD spacetime discontinuous Galerkin project for producing the simulation results visualized here. We would like to specifically thank Shuo-Heng Chung for the 3-D spacetime image (Fig. 2), Reza Abedi and Morgan Hawker for providing us with their simulation data, and Christopher Wojtan for his work on the 2-D spacetime rendering (Fig. 3).

References

[1] Reza Abedi, Shuo-Heng Chung, Jeff Erickson, Yong Fan, Michael Garland, Damrong Guoy, Robert Haber, John M. Sullivan, Shripad Thite, and Yuan Zhou. Spacetime meshing with adaptive refinement and coarsening. In *Proc. 20th Annual ACM Symposium on Computational Geometry*, pages 300–309, June 2004.

[2] C. Bajaj, V. Pascucci, G. Rabbiolo, and D. Schikore. Hypervolume visualization: A challenge in simplicity. In *1998 Volume Visualization Symposium*, pages 95–102, October 1998.

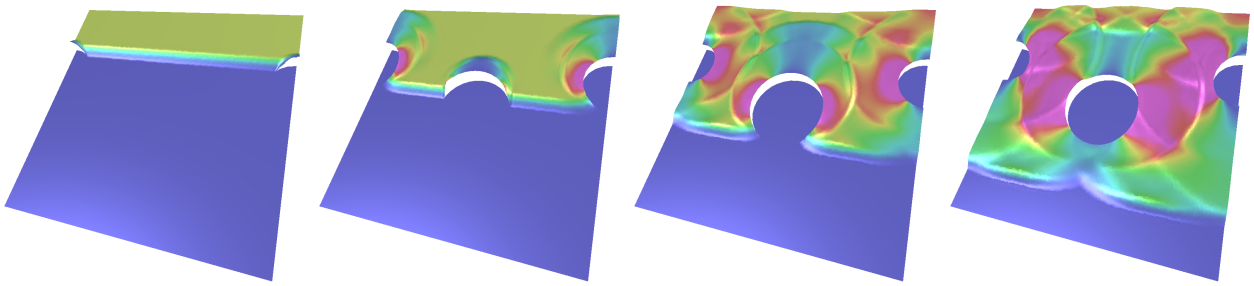
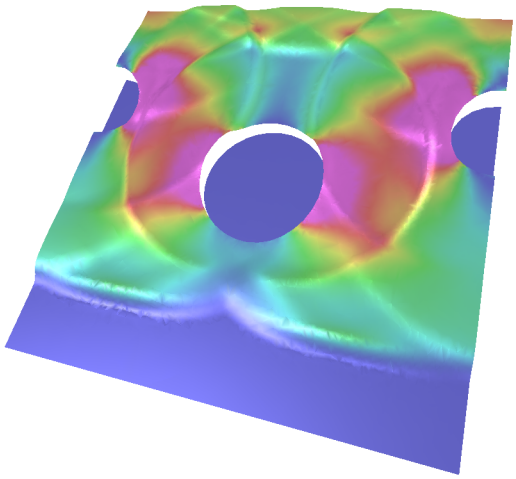
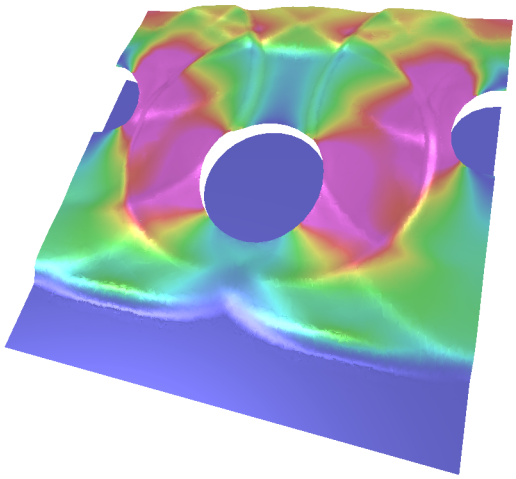


Figure 7: Shock propagating through a medium with circular inclusions. The shock causes the inclusions to debond from the surrounding medium.



(a) Per-vertex color



(b) Per-pixel color

Figure 8: Per-pixel vs. per-vertex comparison for the last time step shown in Figure 7.

[3] David C. Banks and Stephen Linton. Counting cases in Marching Cubes: Toward a generic algorithm for producing substitopes. In *Proceedings of IEEE Visualization 2003*, pages 51–58, October 2003.

[4] P. Bhaniramka, R. Wenger, and Roger Crawfis. Isosurfacing in higher dimensions. In *IEEE Visualization 2000*, pages 267–273, October 2000.

[5] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Graphics Hardware 2002*, pages 37–46, September 2002.

[6] B. Cockburn and P. A. Gremaud. Error estimates for finite element methods for hyperbolic conservation laws. *SIAM J. Num. Anal.*, 33:522–554, 1996.

[7] Jeff Erickson, Damrong Guoy, John M. Sullivan, and Alper Üngör. Building space-time meshes over arbitrary spatial domains. In *Proc. 11th Int. Meshing Roundtable*, pages 391–402, 2002.

[8] Stefan Guthe, Stefan Röttger, Andreas Schieber, Wolfgang Straßer, and Thomas Ertl. High-quality unstructured volume rendering on the pc platform. In *Graphics Hardware 2002*, pages 119–126, September 2002.

[9] Markus Hadwiger, Christoph Berger, and Helwig Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 301–308, October 2003.

[10] Bill Hibbard and Dave Santek. The VIS-5D system for easy interactive visualization. In *Proceedings of the IEEE Conference on Visualization*, pages 28–35, October 1990.

[11] J. Krüger and R. Westermann. Acceleration techniques for gpu-base volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 287–292, October 2003.

[12] William E. Lorenson and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):163–169, July 1987.

[13] Frank Losasso, Hugues Hoppe, Scott Schaefer, and Joe Warren. Smooth geometry images. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 138–145, 2003.

[14] R. B. Lowrie, P. L. Roe, and B. van Leer. Space-time methods for hyperbolic conservation laws. In *Barriers and Challenges in Computational Fluid Dynamics*, volume 6 of *ICASE/LaRC Interdisciplinary Series in Science and Engineering*, pages 79–98. Kluwer, 1998.

[15] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003. Proceedings of SIGGRAPH 2003.

[16] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Graphics Hardware 2002*, pages 57–68, September 2002.

[17] Neophytos Neophytou and Klaus Mueller. Space-time points: 4D splatting on efficient grids. In *Proceedings of Symposium on Volume Visualization*, pages 97–106. ACM SIGGRAPH, 2002.

[18] Jayandran Palaniappan, Robert B. Haber, and Robert L. Jerrard. A spacetime discontinuous galerkin method for scalar conservation laws. *Comp. Methods Appl. Mechs. Engng.*, 2004. in press.

[19] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Han-

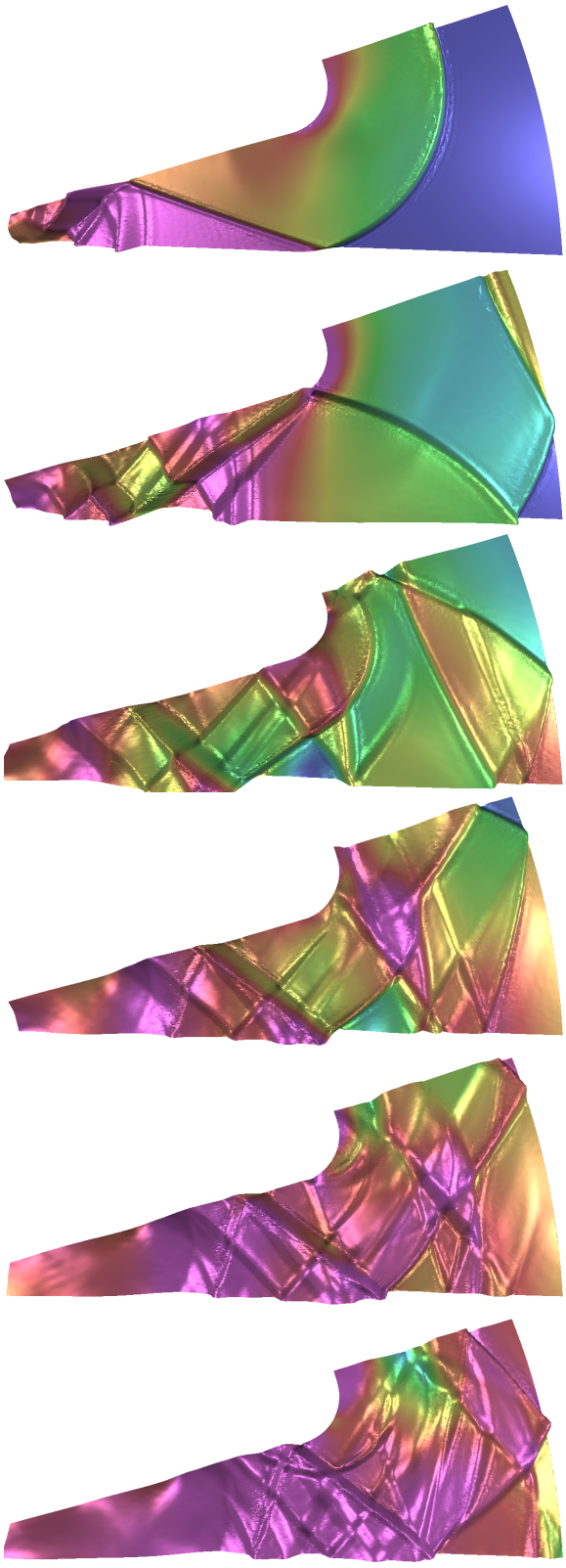


Figure 9: Time sequence of shock wave propagation in a solid rocket booster. Note the complexity of both the wave and color patterns.

rahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001.

[20] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[21] Pedro V. Sander, Hugues Hoppe, John Snyder, and Steven J. Gortler. Discontinuity edge overdraw. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 167–174, March 2001.

[22] Han-Wei Shen and Christopher R. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Proceedings of IEEE Visualization 1995*, pages 143–150, October 1995.

[23] L. L. Thompson. *Design and Analysis of Space-Time and Galerkin Least-Squares Finite Element Methods for Fluid-Structure Interaction in Exterior Domains*. PhD thesis, Stanford University, 1994.

[24] M. Tory, N. Röber, T. Möller, A. Center, and M. S. Atkins. 4D space-time techniques: A medical imaging case study. In *Proceedings of IEEE Visualization 2001*, pages 473–476, October 2001.

[25] Alper Üngör and Alla Sheffer. Pitching tents in space-time: Mesh generation for discontinuous Galerkin method. *Int. J. Foundations of Computer Science*, 13(2):201–221, 2002.

[26] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved pn triangles. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 159–166, March 2001.

[27] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, October 2003.

[28] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):367–376, August 1990.

[29] Jonathan Woodring, Chaoli Wang, and Han-Wei Shen. High dimensional direct rendering of time-varying volumetric data. In *Proceedings of IEEE Visualization 2003*, pages 417–424, October 2003.

[30] L. Yin, A. Acharya, N. Sobh, R.B. Haber, and D. A. Tortorelli. A space-time discontinuous Galerkin method for elastodynamic analysis. In B. Cockburn, G. Karniadakis, and C. Shu, editors, *Lecture Notes in Computational Science and Engineering*, volume 11, pages 459–464. Springer, 2000.

[31] Lin Yin. *A Spacetime Discontinuous Galerkin Finite-Element Method for Elastodynamic Analysis*. Ph. D. thesis, Department of Theoretical & Applied Mechanics, University of Illinois, Urbana, IL, 2002.