LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-209351

Material Interface Reconstruction in VisIt

J. S. Meredith

February 1, 2005

NECDC2004
Livermore, CA, United States
October 4, 2004 through October 8, 2004

Proceedings from the NECDC 2004 UCRL-CONF-209351

Material Interface Reconstruction in VisIt (U)

J.S. Meredith

Lawrence Livermore National Laboratory, Livermore, California 94550

In this paper, we first survey a variety of approaches to material interface reconstruction and their applicability to visualization, and we investigate the details of the current reconstruction algorithm in the VisIt scientific analysis and visualization tool. We then provide a novel implementation of the original VisIt algorithm that makes use of a wide range of the finite element zoo during reconstruction. This approach results in dramatic improvements in quality and performance without sacrificing the strengths of the VisIt algorithm as it relates to visualization. (U)

Introduction

Background

Computer simulations of physical phenomena have a need to support materials, i.e. discrete regions of space with different physical properties. For example, a simulation of tidal waves needs to partition space into water and air, and a simulation of an automobile accident must model glass, metal, and rubber. In these simulations, space is often divided into a computational mesh of cells. However, to maintain accuracy, the material regions will not necessarily conform to the cells of the mesh. (See Fig 1.) Sometimes, this is because the mesh is static and the materials move; sometimes it is because the mesh cannot deform to handle the twisting and bending of materials without tangling; and sometimes it is simply because the materials must be modeled at a higher resolution than the mesh to maintain accuracy. The result is that very often, a single cell may be mixed: the cell is composed of pieces of two or more materials. Cells that contain only one material are called clean.

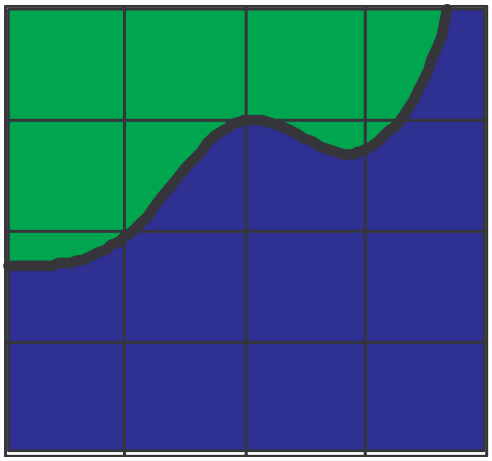

Fig. 1. Material boundaries will not in general conform to the cells of the mesh.

Since the materials can now be of greater resolution than the mesh, the question of how to keep track of this material information in mixed cells becomes a problem. The typical method has been to store in each cell just the material volume fractions (VFs), or the percentage of the cell filled with each material. Unfortunately, the shape of the interface is not typically stored. (See Fig. 2.) A difficulty in visualizing results from these simulations thus involves determining what the material interfaces looked like based purely on what volume fractions are stored for each material in a cell. Methods of reconstruction and tracking of these mixed material interfaces have been researched since the 1960s. They are important for visualization, but they are also important for physical simulations themselves since an inaccurate reconstruction can give inaccurate results.

1.0	1.0	1.0	0.6 0.4
1.0	0.5 0.5	0.2 0.8	0.1 0.9
0.2 0.8	1.0	1.0	1.0
1.0	1.0	1.0	1.0

Fig. 2. Volume fractions for the boundaries shown in Fig. 1.

0.0	0.0	0.0
0.5	0.5	0.5
1.0	1.0	1.0

1.0	1.0	1.0
0.5	0.5	0.5
0.0	0.0	0.0

Fig 3: Example volume fractions for water (left) and air (right)

To help explore what it means to have a good reconstruction, let us imagine a very simple example: a 3x3 cell mesh where exactly the bottom half of the mesh is water and the top half is air. (See Fig. 3) This means the top three cells are completely air, the bottom three cells are completely water, and the middle three cells contain half water and half air. These three middle cells are mixed, and it is the reconstruction method that must determine which portion of each of these cells is air and which is water. In this simple example, it makes both intuitive and physically meaningful sense that the center cell is "correctly" reconstructed by assigning water to the bottom half of the cell and air to the top half of the cell. Other examples, such as the one of Figures 1 and 2 are not quite as obvious; "correctness" should be probably be defined not only in terms of accurately recreating an interface which covers the exactly quantity of a mixed cell, but that it puts the materials in a natural position within the cell.

Survey of Previous Methods

One of the first methods for material interface reconstruction (MIR) is a method that uses tracking particles to define the interface (Amsden, 1966). Noh and Woodward (1976) created the simple line interface calculation (SLIC), which determines the material interface using these volume fractions. SLIC is a piecewise constant method which aligns the material interface with one of the major coordinate axes. Also originating at this time were similar piecewise constant/stair-stepped methods such as the volume-of-fluid (VOF) method (Nichols and Hirt, 1975). An improvement to these methods came with piecewise linear interface calculations (PLIC) such as the one from Youngs (1992).

When performing reconstruction on a cell, most of these methods examine a window around the cell for the contextual information needed to determine the shape of the interface in that cell. To continue with our earlier example, see Fig. 4, where a cell of interest is outlined in red, and a window around it is outlined in blue.

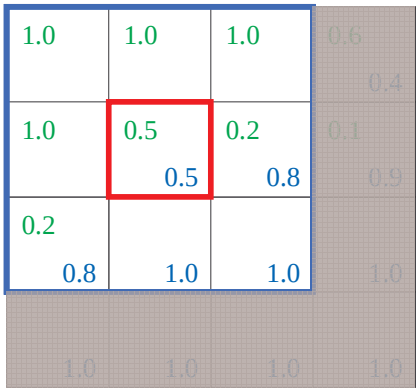

Fig. 4. A cell of interest (red) and its window of surrounding cells.

Examining the PLIC Algorithm

Since the PLIC algorithm is commonly used, we will examine it in more detail. First, select a material to reconstruct. This step by itself is often a source of error because a "correct" reconstruction requires, for reasons that will become clear, that we follow the materials in order from the outside in. In this example we choose the green material, and its volume fractions for the cell of interest and surrounding window are shown in Fig. 5.

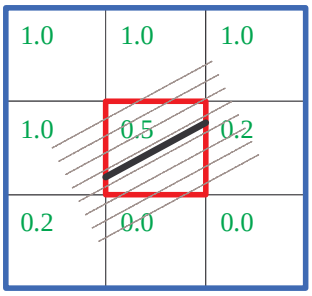

Fig. 5. Slope and intersection for PLIC

Next, calculate the slope of the interface using a window around the cell of interest (for example, using the gradient of volume fractions), and then calculate the intercept of that interface line such that it intersects the exact volume fraction in the cell of interest. Fig. 5 also shows this slope and intersection position. Once this is done, fill in the chosen material as in Fig. 6.

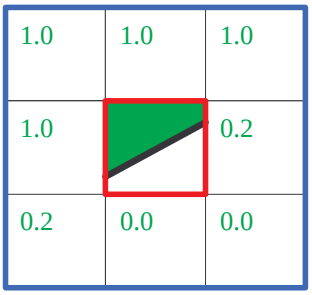

Fig. 6. End of first material pass in PLIC.

This leaves empty space in the cell, and the process is repeated for each material using the unallocated space in that cell. Finally, when there is only one material remaining, its volume fraction by definition must be exactly the same as the remaining amount of empty space, so we simply fill the remaining space with that remaining material.

This process is repeated for all cells: choose a material, find the slope, intersect the cell, fill with that material, and repeat for all materials. When done, our example looks a lot like Fig. 7. As a final note, be aware that this algorithm is not limited to structured 2D grids.

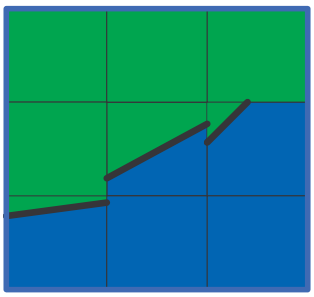

Fig. 7. Final result of PLIC.

Material Interface Reconstruction for Visualization

One must note that the methods shown above, are concerned not just with the interface reconstruction, but also with the ability to track materials using this reconstruction. In other words, these algorithms are not designed to look correct, but to perform correctly within a CFD or

hydrodynamics code. This makes visualization of material interfaces using these methods unappealing. Their primary drawback is that the interface is never designed to be continuous across cells. In two dimensions, visualization would clearly show the stair stepping and discontinuities inherent in these algorithms, and Fig. 7 shows this clearly. In three dimensions, a lack of continuity means a lack of connectivity, and makes most geometric algorithms on the reconstructed mesh unnaturally difficult. In fact, it is impossible using only a single linear interface to obtain an interface that is guaranteed to be both accurate and continuous across cell boundaries, because it can only approximate what should in many cases be a curved line.

One approach that guarantees continuity is called the Isosurface method. It generates a continuous variable based on volume fractions, and finds interfaces where a material volume fractions is exactly equal to the value 0.5. Using this method, it is possible to force many visualization codes that do not natively understand materials to do a plot of these material interfaces. However, this method has some drawbacks. First, since it does not understand natively what a material is, it cannot respect clean cells, and it will often draw interfaces through them. Second, it will leave "gaps" where three or materials approach each other. Third, it cannot guarantee that it will reconstruct the correct amount of material in a cell as specified by the volume fractions used as input to the method. And fourth, it is a surface based approach and cannot be used to perform reconstruction before doing further visualization operations like slicing. Some of these consequences of these will be shown later.

Fig. 8. All volume fractions for all materials.

The VisIt scientific visualization code of Lawrence Livermore National Laboratory uses a different method to reconstruct and visualize mixed material meshes. The basic premise is similar to the isosurface algorithm, except that it natively understands what a material is, and it finds surfaces where the material volume fractions are equal *to each other*.

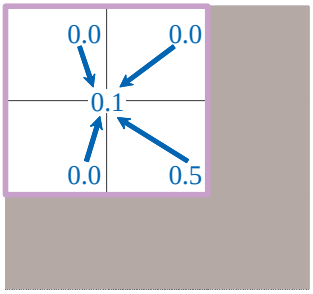

Fig. 9. Averaging cell-centered VFs to the nodes.

The outline of this algorithm is the following. First, collect the volume fractions for all materials in the cell of interest and the window around it (see Fig. 8). Second, create a linearly interpolated variable over the mesh using the volume fractions for each material. This is done in this case by averaging the volume fractions of each material in all four cells surrounding a mesh node to that node (see Fig. 9), then repeating for all nodes in the cell of interest for all materials (see Fig. 10).

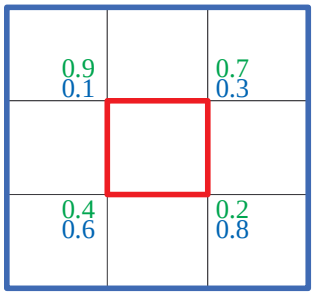

Fig. 10. Final nodal VFs for the cell of interest.

Next, using an interpolation function (such as bilinear interpolation), evaluate the function for each material at every point within the cell. Where the value for the blue material is

greater, it is determined that the point contains our blue material, and where green is greater, it is determined that the point contains our green material.

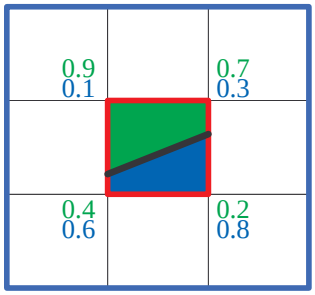

Fig. 11. Final result in the cell of interest.

For this example, the blue material has the greatest value in the lower-right portion of the cell, and the green material has the greatest value in the upper-left portion of the cell, leading to results in the cell of interest that look like Fig. 11. Note that for a linear interface within a cell, the evaluation need not actually happen at every point in the cell; the actual implementation is much more efficient for linear interfaces because it only calculates the intersections along the cell edges.

This example shows only two materials, but the extension to three materials is actually very straightforward in that there was no assumption in the mathematics about the number of materials. Where the real difference lies is in the comment in the previous paragraph concerning an efficient implementation – it actually calculates intersections between pairs of materials. To accomplish this, it handles multiple materials by visiting them in stages, and the output of each stage will be the input to the next. For example, examine Fig. 12 where we show the full reconstruction process one stage at a time. In Fig. 12a we have the square input cell, and are visiting the green material, and we have trivially determined that of all visited materials and the new material, the green one is greatest at all four corners and thus the output from the first stage is at Fig. 12b. In Fig. 12b, we decided that when visiting the blue material, the blue material was greater than the green material at the bottom two corners, and as output from this stage we get Fig. 12c. Now we have *two* intermediate cells as input to this third stage where we examine the red material. We decide for the blue input cell that red is now greatest along the right edge, and we decide for the green input cell that red is also greatest along the right edge. This gives us as final output Fig. 12d. Note that while we visited the materials in some order, unlike the PLIC algorithm the particular order we choose will have almost no impact on the final result.

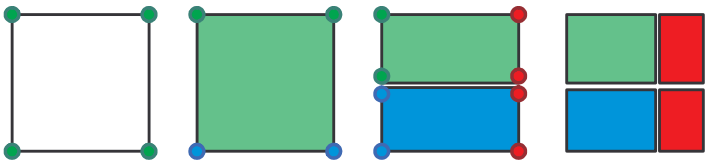

Fig. 12. The input cell is at (a). Intermediate stages in the reconstruction at (b) and (c). Final output at (d).

This approach has another distinct advantage, which is that it is guaranteed to generate continuous interfaces from one mixed cell to the next (see Fig. 13); the values obtained at a node by averaging the neighboring four cell are the same no matter which cell of the four we are considering. This means the interpolant generates the same values at any edge no matter which of the two cells along the edge we are considering. Note that there is what appears to be a

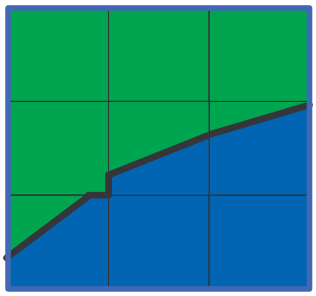

Fig. 13. Final result of the visualization reconstruction.

discontinuity in Fig. 13, but in fact this is merely showing the ability for this method to respect clean cells, unlike the Isosurface method. Specifically, this need not break connectivity; simply reconstruct this clean cell as if it were mixed but before completing reconstruction on that cell, force all output geometry to be the correct (clean) material. This generates reconstructed geometry with good connectivity that still respects clean cells.

In addition, this approach, like the isosurface approach, generates boundaries which look subjectively "good" – the water material in some mixed cell will be near the water in neighboring cells, and the air material in a mixed cell will be near the air in neighboring cells. In general, this technique will not be exactly accurate; as mentioned above, with so few degrees of freedom it may be impossible to ensure accuracy if it ensures continuity anyway. However, it produces results more suitable for scientific visualization.

Reconstruction for Visualization in Three Dimensions

Like the PLIC algorithm, the VisIt algorithm intuitively extends to three dimensions. However, like the isosurface algorithm, the earliest implementations of this algorithm in three dimensions had the disadvantage of reconstructing only surface geometry instead of a pure volumetric reconstruction; this produced some of the same aberrant results as the isosurface algorithm. More recently, a volumetric method was created to solve these problems, and it is this latest iteration that has been the most successful in terms of generating the best results for visualization.

However, the cost of a full volumetric reconstruction is expensive in terms of code size and complexity, memory usage, output cell and polygon count, and execution time. The reasons for this are many. For one, there are potentially many material in a single mesh, and unlike the isosurface algorithm, the materials cannot be reconstructed in isolation from the other materials – this forces the algorithm to keep more in memory at a time.

In addition, a surface reconstruction can output a single polygon per material per zone mixed, but a volumetric construction in general cannot output a single cell per material per mixed zone. Imagine slicing a cube in half at an angle such that the face between the two halves is a pentagon or hexagon. Since the primitive cell types have only three or four sided faces, each half must by definition be composed of more than one cell. To phrase this differently, the difficulty is that a volumetric reconstruction almost necessitates working with arbitrary polyhedra. This is not particularly difficult in two dimensions, but to remain feasible in three dimensions, a different approach must be taken.

A common approach in spatial algorithms like this relies on splitting cells into tetrahedra, because of the simplicity of the tetrahedron and the small number of ways one can split a tetrahedron. In other words, step one in a volumetric reconstruction is to take the input mesh and convert every cell into tetrahedra, then perform reconstruction on the tetrahedral mesh.

There wind up being two nontrivial ways to split a tetrahedron: the first generates a tetrahedron and a wedge as output; the second generates two wedges as output. Recall, however, that as we visit other materials in the cell we must take these output shapes and feed them back as input. Since we cannot operate on a wedge, we must convert that wedge into three new tetrahedra, so a single cut on an a tetrahedron will wind up with four or six – not two – shapes as input to the next stage.

Furthermore, simply breaking up a nontrivial mesh into tetrahedra without breaking connectivity is not an easy task and can be counterproductive. One major problem with tetrahedralization is that if one is not very careful about how they tetrahedralize every one of the input cells, then they might break the connectivity of the mesh before even beginning the material interface reconstruction. Specifically, there are 2^6=64 different ways to tetrahedralize any individual hexahedron. If for any cell, you choose a tetrahedralization that does not correctly match with its neighbors, you have broken connectivity *before* reconstruction, and no intelligent choice of reconstruction algorithm is going to fix it.

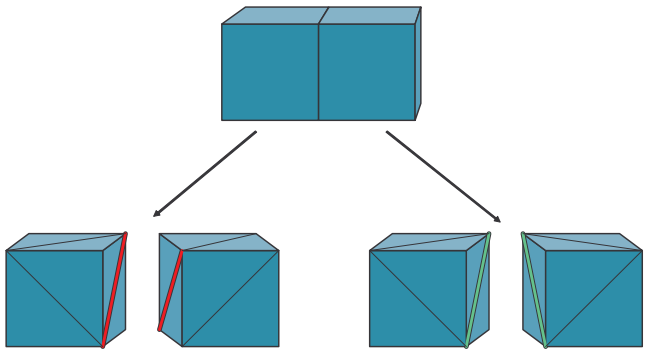

Fig. 14. Top: two hexahedral cells with an adjoining face. Bottom left: a tetrahedralization for these two cells that results in a break in connectivity. Bottom right: a good tetrahedralization that leaves connectivity intact.

This problem is shown in more detail in Fig. 14. The two cells at the top of the figure share a quadrilateral face. The bottom left and bottom right images show these two cells under two different sets of tetrahedralizations. The bottom left image shows a poor matching of tetrahedralizations. What happened in this case is that when the hex was split into tets, the quadrilateral was thus split into triangles (i.e. the faces of the tets), and the two cells did not agree on how to split that quadrilateral. This causes an immediate break in face connectivity, disrupting most further visualization operations. The image on the bottom right, however, shows a good matching of tetrahedralizations, because both pairs of triangles of that matching face have all their nodes the same, and thus connectivity is not broken.

The most obvious solution appears at first glance to be global: pick a tetrahedralization for one cell, then ensure that for your neighbors you pick one that matches. Luckily, there is a local solution you can guarantee is correct without having to look at your neighbors, under the commonplace assumption that the node indexing is unique. It is as simple as this: always split any quadrilateral face along the diagonal starting at the lowest numbered node. For a hexahedron, for example, we pick from one of two possible splits for each of the six quadrilateral faces, giving us exactly one of the 64 possible tetrahedralizations. Since the node numbers for any given quadrilateral face

are going to match no matter which of the two cells share that face, both cells will locally make the same decision and ensure good connectivity remains. See Fig. 15 for an example.

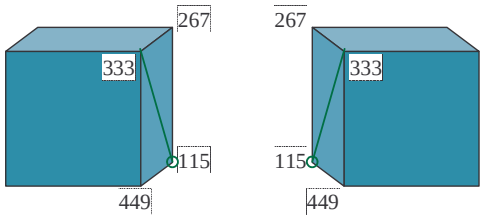

Fig. 15. Example indexing for a shared face its implicit subdivision.

While we now have a local solution that we can implement, there remain some significant hurdles to doing a full volumetric reconstruction in three dimensions using tetrahedralization. The first problem is that the logic described above to determine the appropriate tetrahedralization is not trivial to implement, either in terms of code complexity or final speed. A more inherent problem is that even the common hexahedral cell will be split into a minimum of at least five, and usually six, tetrahedra. Thus, interface reconstruction now must operate on a data set with almost six times the number of cells as the original, with all the associated downsides such as memory usage, time to do the reconstruction calculation, and the number of polygons in the final surface.

One other downside to doing tetrahedralization before the reconstruction is a visual artifact I will call interpolation imprinting. Fig. 16 explains the situation in two dimensions for simplicity. The square on the left of the figure shows a 2D cell that will have reconstruction performed. Since according to our visualization setup the blue material is greatest in the lower-left corner and the green material is greatest at the other corners, we know that our reconstruction algorithm finds edge material intersections and in this case will find a blue/green intersection along the left edge and along the bottom edge.

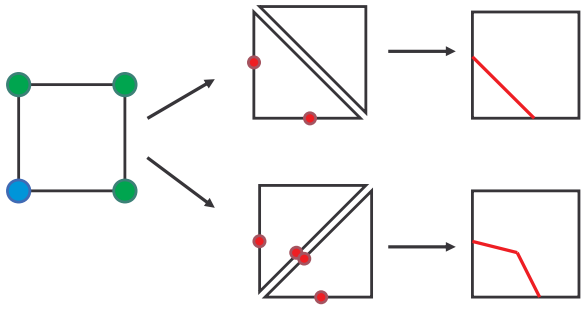

Fig. 16. Interpolation imprinting: an arbitrary choice of cell splitting before reconstruction can cause unintended wobbles in the reconstructed interface.

However, we first need to perform a tetrahedralization, and the 2D analogue would be to split this square into triangles. The top path of Fig 16 shows one way it might happen, and the bottom shows another way, and both of these are equally likely to occur based on our node-index scheme. Recall that we are only doing edge material intersections, and note that on the bottom path we have created a *new* edge – the diagonal – along which to interpolate material values and along which we must by definition have another intersection. This upper-right value in the original square cell was not used in calculating the original edge intersections and in general cannot create an intersection which causes a smooth surface in the final reconstruction; the bent red line in the bottom path is an example of what often results.

The Zoo Based Algorithm

Can we create an algorithm that will perform a volumetric reconstruction while avoiding both the need to work with arbitrary polyhedra and the need to split the mesh into tetrahedra? Can we use it to avoid the interpolation imprinting artifact entirely? And furthermore, we would like one further ideal: is it possible to require that we split cells into the geometric minimum number of output shapes? For example, if we take a cube and slice it in half along its axis, can we create an algorithm that will be guaranteed to return exactly two hexahedra as output shapes? The answer to all these is yes.

First in Two Dimensions

Let us return to the two-dimensional world, assume we have created an algorithm which gives us the ideal results just described, and examine how this works. We make the common assumption that 2D cells are only triangles or quadrilaterals.

In fact, we will start at the simplest possibility geometrically as the input to a stage in our new reconstruction algorithm: triangles. Fig. 17 shows the full range of ways in which we may split a triangle (which, by symmetry, is only two different ways). The first is the trivial split: a triangle is the input shape, we found no intersections, and have the full input triangle as the output. The second is the nontrivial split: a triangle is the input shape, one (or symmetrically two) nodes were of a different material, and the output is one quadrilateral and one triangle. For this second case, suppose we now want to visit a *third* material in a subsequent stage in our reconstruction. At this point, we had better understand how to handle quadrilaterals as input to our reconstruction algorithm, or else we have reached an impasse.

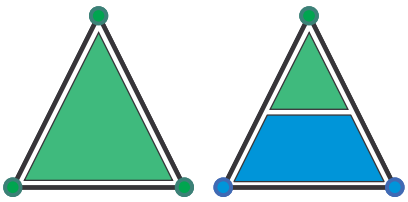

Fig. 17. All clipping cases for triangles.

So let us examine what might happen if we take quadrilaterals as input to a reconstruction stage. Fig. 18 shows all the possibilities here as well, which by symmetry is only four different ways. The first is the trivial split, with a quad as input and a quad as output. The last is the on-axis split, with a quad as input and with two squished quads as output. The other two are the difficult ones, as one has a pentagon as output and the other has a hexagon.

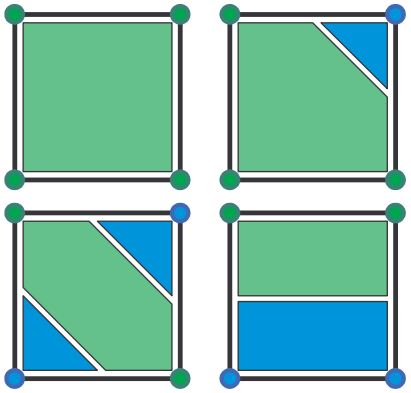

Fig. 18. All clipping cases for quads.

The difficulty here may be obvious: we might have a five- or six-sided polygon as input to our next stage, and splitting those again might create up to nine-sided polygons as input to the next stage, and so on. One solution is to handle arbitrary polygons. The other is to split those shapes into triangles and deal with those. This is, in fact, simply a rephrasing of the

reasons in three dimensions that led us to tetrahedralization – we know what to do with triangles/tetrahedra, and we don't want the complexity of arbitrary polygons/polyhedra. More specifically, if we want to avoid the arbitrary polygons, we need to have *closure* over the input and output shape types, and the easiest way to do that is by having the output of any splitting operation be triangles.

However, one fact might be more clear now that we have investigated the problem in some detail in the two-dimensional setting: we don't have to split the pentagon and hexagon into *triangles*. In fact, our hexagon can be handled by splitting it into two quadrilaterals, and our pentagon can be handled by splitting it into one triangle and one quadrilateral. By doing this, we now no longer have up to 6 total output shapes by splitting one input quad, we only have up to 4. And we have simultaneously limited the output shape types to only triangles and quadrilaterals, which we know how to handle. In other words, we have cut down on the quantity of cells we need to handle while maintaining closure of the input/output shape types.

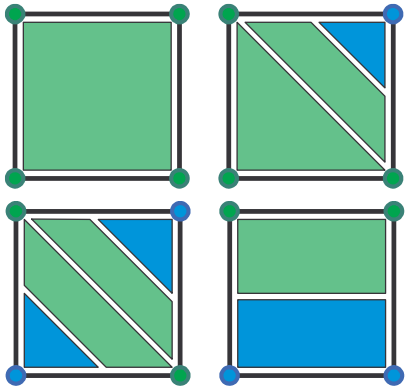

Fig. 19. Clipping cases for quads with triangle/quad closure.

There is one open issue before claiming the two-dimensional case is solved: there are three different ways to split a hexagon into two quadrilaterals, and five different ways to split a pentagon into a quad and a triangle – which of these ways should we subdivide? While any method of making this decision would suffice, one way turns out to be better than the others because it is unambiguous: if you have to split the corner off a quadrilateral, force another split along the parallel diagonal of that split. This is shown in Fig. 19.

Thus, the combination of the splitting cases in Figures 17 and 19, when symmetry is added, result in a complete closed solution for two dimensional clipping without resorting to arbitrary polygons or triangulating all input cells.

The Zoo Extension into Three Dimensions

Let us now move back to three dimensions. However, instead of showing the full solution, I will show two examples. First, we have one ideal hexahedron splitting shown in Fig. 20, where the split is cuts the hex into two new squished hexes.

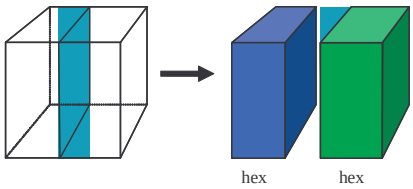

Fig. 20. One ideal clip for a hex.

We have another ideal hexahedron splitting shown in Fig. 21. This one cuts one edge off the hexahedron, and the shape cut off with the edge is a wedge, or triangular prism. This is exactly one of the shapes in the finite element zoo. However, note that when we make this cut, the top and bottom face of the original hex are now cut into one triangle and one pentagon. This is a real

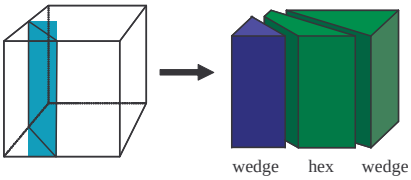

Fig. 21. Another ideal clip.

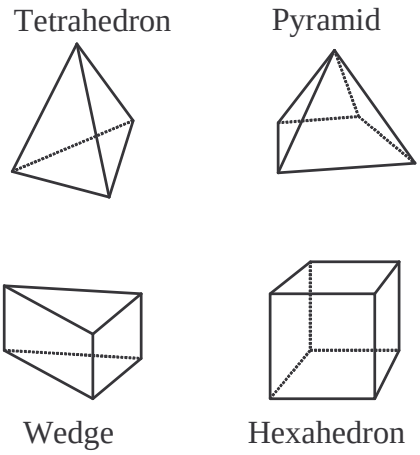

Fig. 22. The three-dimensional shapes in the finite element zoo.

difficulty, because while we would like every time we split an input shape to result in only two output shapes, it is not possible here because none of the shapes in our finite element zoo (see Fig. 22) has a pentagon as a face. Thus we must create a total of at least three shapes as the output for Fig. 21. By making full use of the cells at our disposal in the finite element zoo, we have done exactly that as shown in Fig. 21. Thus in this case we have also created the geometric minimum number of output shapes without using arbitrary polyhedra.

The fact that all of the 3D shapes in the finite element zoo have only triangles and quadrilateral as faces is actually critical to the 3D solution. Apart from the simplicity of explanation, this was another reason to demonstrate the concept in two dimensions: we simply apply the solution for 2D polygons to the faces of three dimensional cells. For example, this is what we did to the top and bottom faces of Fig. 21 – we cut along the corner, and effected another split along the parallel diagonal. This guarantees that we will never have anything but a triangle or quad as face for an output shape.

This also highlights the danger of ambiguity in which way to split the pentagons or hexagons that might result in 2D: we are performing the same operation on the faces of 3D cells, and these faces must be split the same way no matter which of the two cells that share a face we are operating on.

With the 2D solution now applied to the faces of 3D cells, can we always split an input cell cleanly into output shapes from the zoo? In other words, it was a necessary condition, but was it sufficient to guarantee closure in the zoo?

The answer is actually “No”, but it turns out not to matter. In some cases, you cannot group the newly created faces into meaningful output shapes without adding extra edges along some of these faces. This is, of course, not allowed, because we have no way to guarantee that our neighboring cell will add the same edges. Instead, we add a new point on the interior of the output shape and create tetrahedra and pyramids from this new point to the triangles and quads we have guaranteed as output shapes. This sounds like many more shapes than we would like, but remember the following facts. First, the tets and pyramids we output are part of the zoo, so we have thus ensured closure. Second, it is a last resort reserved for hard cases only; most of the cases can be solved with fewer shapes. Third, remember that we are comparing the number of output shapes against the method where we tetrahedralize the cell *before* attempting to cut it.

In short, applying the lessons from 2D results in the geometric minimum number of output shapes and always results in a complete closed solution for clipping 3D cells without resorting to arbitrary polyhedra.

Qualitative Results

In Fig. 23, we see the value of a continuous interface. The image on the left was generated with a PLIC algorithm, and the discontinuities not only look bad but would cause difficulties when performing other visualization or analysis operations on this reconstruction. The image on the right was generated with the continuous VisIt algorithm and has none of those problems.

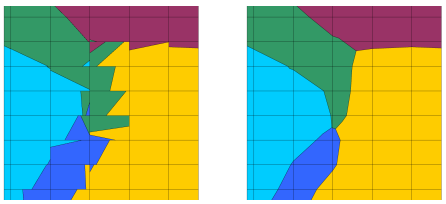

Fig. 23. PLIC vs. Zoo-based

In Fig. 24, the left image was generated with the isosurface algorithm. Notice that where three materials meet, a large void has appeared. The VisIt algorithm ensures there are no gaps between materials.

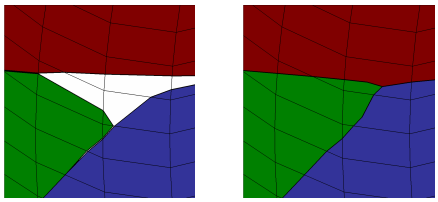

Fig. 24. Isosurface vs. Zoo-based

In Fig. 25, we see one advantage of a volumetric reconstruction. In the left image, we needed to slice the volume fractions before we could perform reconstruction, because it requires a volumetric algorithm to be able to reconstruct before slicing. The reason for the poor material boundaries is that slicing volume fractions is not physically meaningful; the materials are not evenly distributed throughout the cell. The image on the right was generated with the VisIt algorithm performing reconstruction before the slice.

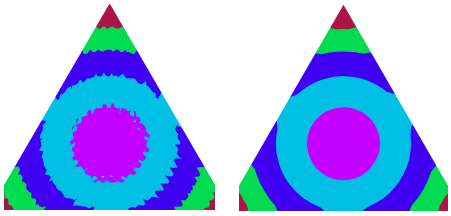

Fig. 25. Surface vs Volumetric Reconstruction

In Figs. 26 and 27, we see the advantages of the zoo-based approach to clipping cells. The left image in Fig. 26 split every mixed cell before reconstruction just as a tetrahedralization approach would. In fact, this is an optimistic picture because many tetrahedralization approaches would also split every clean cell. However, the zoo-based approach has no need to split clean cells except when necessary to guarantee connectivity, and even when splitting mixed cells does it as few times as possible. In Fig. 27, we see how using fewer unnecessary splits in the zoo-based approach results in fewer interpolation artifacts.

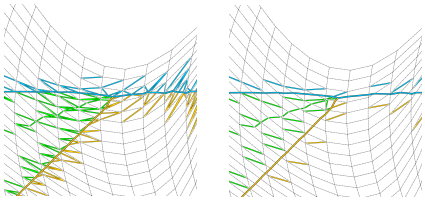

Fig. 26. Tetrahedralizing vs. Zoo-based subdivision

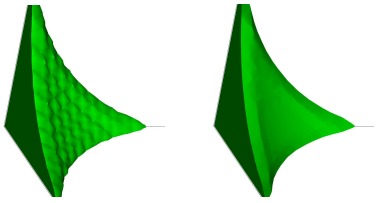

Fig. 27. Tetrahedralizing vs. Zoo-based interpolation

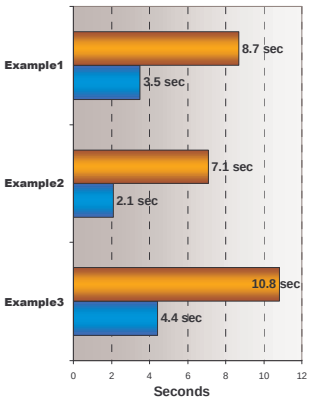

Fig. 28. Execution time.

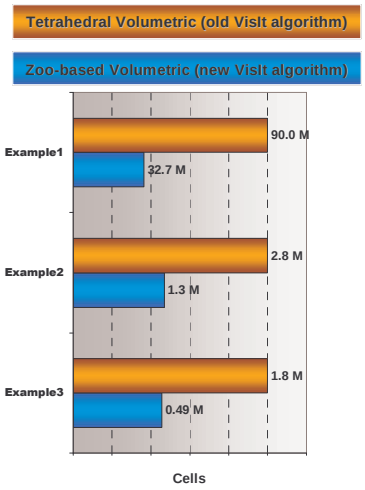

Fig. 29. Output cell count.

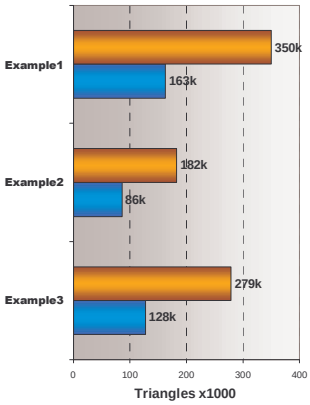

Fig. 30. Output triangle count.

Quantitative Results

In Figures 28 - 30, we compare the performance of the predecessor reconstruction algorithm in VisIt (orange), which was a volumetric tetrahedralization method, to the new algorithm in VisIt (blue), which is the finite element zoo based approach described in this paper. One important point to note is that the tetrahedralization algorithm was highly optimized over a period of two years, and among other improvements it is able to avoid tetrahedralization on all but the mixed cells, and it can avoid re-tetrahedralizing split tets if they are not used as input to another stage in the same cell.

These examples are all real-world problems. Example 1 has 512 domains, 20 million cells, 30 materials, and was run in parallel on 64 processors. Example 2 has 300,000 cells, 17 materials, and was run on 1 processor. Example 3 has 420,000 cells, 30 materials, and 3 domains, and was run on 1 processor.

In Fig. 28, we see that on average, execution time has decreased by about 3x. In Fig. 29 (the scales are normalized to the old algorithm), we see that the output cell count has also decreased by a factor of about 3x. In Fig. 30, we see that the number of output triangles has decreased by a factor of just over 2x in each case.

Conclusions

We have first provided some background information on material interface reconstruction and how it has been implemented in the past. In addition, we have discussed in detail the previous algorithm used by the VisIt visualization code and explained its strengths and weaknesses.

Finally, we presented a new algorithm that greatly improves upon the previous ones in terms of realism, quality, and speed. The resulting algorithm are runs several times faster

than the previous volumetric reconstruction in VisIt, uses one third the memory, avoids floating point numeric hashing entirely, always retains perfect logical connectivity, cuts the volumetric cell count down by a factor of three and thus speeds up subsequent operations on the reconstructed dataset by the same factor, results in about half the polygons, improving rendering speed by a factor of two and resulting in smoother, more accurate output with fewer interpolation artifacts.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

Amsden, A. A. “The Particle-in Cell Method for the Calculation of the Dynamics of Compressible Fluids”, *Los Alamos Scientific Laboratory Report LA-3466*. (1966).

Nichols, B. D., and Hirt, C. W. “Methods for Calculating Multi-Dimensional, Transient Free Surface Flows Past Bodies”, *First International Conference on Numerical Ship Hydrodynamics, Gaithersburg, MD.* (1975).

Noh, W. F., and Woodward, P. “SLIC (Simple Line Interface Calculation)”, *Lecture Notes in Physics*, 59, Springer Verlag. (1976).

Parker, B. J., and Youngs, D. L. “Two and Three Dimensional Eulerian Simulation of Fluid Flow with Material Interfaces”, *Third Zababakhin Scientific Talks, Kyshtim, USSR.* (1992).