PRE-PROCESSING OF CAR GEOMETRY DATA FOR CRASH SIMULATION AND VISUALIZATION

N. Frisch, D. Rose, O. Sommer, T. Ertl

Visualization and Interactive Systems Group
Department of Computer Science
University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart, Germany
`http://wwwvis.informatik.uni-stuttgart.de`
{frisch, rose, sommer, ertl}@informatik.uni-stuttgart.de

ABSTRACT

In this paper we focus on a visualization tool for car crash simulations. By means of some examples we show how various data pre-processing features can facilitate the engineer's work. For example, parts can be assembled, replaced, welded, bonded, or deformed. Data pre-processing within the visualization tool means that some modifications can be done directly on the finite element mesh which is the input for the crash simulation. Some of the features are new within a crash visualization environment and some operations needed new algorithms to be developed: We present the generation of curved spotweld lines and adhesive bondings along flanges. Further we show how modern hardware features like textures and alpha blending can be employed for efficient visualization in the context of data pre-processing. The new features allow crash simulations in an early development phase, they also allow to test the impact of a potential improvement or to remove some shortcomings due to mesh generation out of CAD data. Thanks to these new features, the crash simulation engineer needs no longer to return the model data back to the CAD department for minor modification and re-meshing.

Keywords: visualization of car crash simulation, finite elements, pre-processing, CAD

1 INTRODUCTION

The increasing pace at which new industrial products are brought to the market requires appropriate software tools. Automotive companies aim to place new, enhanced car models on the market in short time, in order to be competitive. Software has a key role in car development; several software tools are involved. Regarding car body design, the passenger's safety must be considered. Crash tests are necessary and obligatory. Before performing a real crash test, hundreds of crash worthiness simulations are computed and analyzed. The number of real tests is reduced to a minimum, saving time and money.

In the car development process, the CAD data from the construction department are transformed into finite elements, a process called meshing. Meshing can be thought as a kind of tessellation. Most finite elements are quadrilaterals for numerical reasons. The mesh is the input for the finite element solver which computes the crash simulation. Mesh quality is a precondition for the correctness of the simulation result. Therefore, the mesh is verified and corrected if necessary. A prototype named crashViewer was developed at the University of Stuttgart within the BMBF[1] supported *AutoBench* project. The prototype can be used for improving finite element meshes as well as for visualizing the crash simulation input and output data, see also [1]. Visualizing output data in the so-called post-processing stage is necessary for analyzing the simulation results.

[1] German Ministry for Education and Research

The prototype is based on the OpenGL Optimizer [2] high-level graphics API from Silicon Graphics. It also has a CORBA and Java based interface for the software integration platform CAE-Bench as described in [3]. The integration platform facilitates the control and data exchange between the different applications involved in car body development, leading to a significant increase of productivity.

In the last years, a transition took place from meshing car body as a whole towards independent meshing of the car's components. This transition required special link elements to be introduced. Linking the car body components with special elements like spotwelds permits an independent meshing of each component. Otherwise, each component's mesh would need to match the neighbouring part's meshes at the contact areas. Changing, modifying or adding an assembly part would require to re-mesh the adjacent car body parts, too. In the past, this often entailed expensive post-processing in the mesh generation process, especially when the mesh data descended from inaccurate CAD models (see related work [4]). The new bonding elements reduce these shortcomings and therefore the development time.

Based on the crashViewer software, we implemented various new pre-processing features. These features are useful for making corrections and improvements to the simulation input deck. The main pre-processing features we will describe are the definition and modification of assembly part connections, like spotwelds and adhesive bondings. Further, we describe the mesh modification with the purpose of penetrations removal and the efficient visualization of potential flanges.

2 SPOTWELDS

2.1 Setting

Spotwelds are the prevalent link between car body components. Spotweld information can already be defined in the CAD data. However, often additional spotwelds need to be set and some spotwelds eventually need to be moved or deleted.

The visualization of spotwelds by means of small, scalable red cuboids has proven good in practice. Erroneous spotwelds, e.g. spotwelds with missing or inappropriately positioned assembly parts, are visualized with different color and/or geometry in function of the error type, see Figure 1. Since a car contains thousands of spotwelds, it

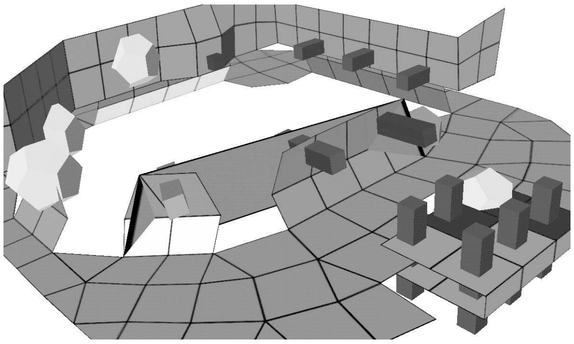

Figure 1: Various valid and erroneous spotwelds, visualized as cuboids and dodecahedra.

can be tedious to edit spotwelds one by one. For this reason, crashViewer provides the facility to define an entire spotweld line at once, by specifying the start and end point of the line with the mouse pointer. This generates a set of spotwelds along a straight line. The spotwelds are equidistantly positioned on this line. If the assembly part is curved, the straight line is projected onto the surface to find spotweld positions.

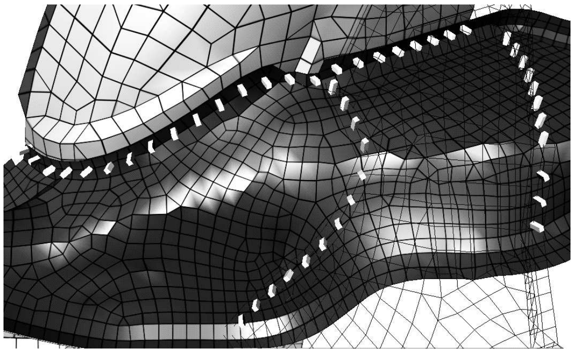

Figure 2: Three curved lines of spotwelds generated by our algorithm. One assembly part is rendered as wireframe.

2.2 Curved spotweld lines

Obviously, not all spotweld lines on a car will be straight, neither can each curved line be achieved by projection of a straight line onto a car component. Therefore, a feature for generating curved spotweld lines was implemented. A first idea was to achieve curved lines by means of spline curves. Three-dimensional spline-curves are hard to position exactly on the middle line of the flange. As spotwelds are usually positioned along flanges, a flange recognition algorithm was developed. Figure 2 shows an example application of our algorithm.

In Section 5 we describe an approach for fast visualization of potential flanges based on distances between components. The visualization gives the user a hint where flanges could be, but it is not an accurate flange detection. Flanges are special regions on plate components with the purpose of enhancing the connection between parts by increasing the contact area.

The flange detection algorithm for curved spotweld lines and bondings is done on a per-element basis. Each finite element is either a flange element or not, in function of the distance and the angle of this element with regard to the nearest element of the corresponding component. Distances are computed using the bounding box hierarchy described in Section 5. Finally, we observe that flanges have a considerable length but only a limited width.To achieve a curved spotweld line, the following steps are performed: first, the finite elements containing the start and end point of the desired line are determined. The next task is finding a shortest element path between those two. For each element on this path, the left and right flange borders are sought in order to find the mid-line. Finally, the spotweld positions on the mid-line are computed in function of the desired distance between spotwelds. The less trivial steps are finding the path and detecting the mid-line. For path finding we adapted an iterative algorithm from [5] originally designed for graph traversal. A special goal is to find a path even if the flange is interrupted by small gaps and corrugations (Figure 3). The gaps and corrugations can split the flange area and make it impossible finding a path consisting exclusively of flange elements. In this case, the path should lead over or beneath the obstacles. The algorithm performed good in our tests, with respect to result quality and computing time. Computing time is not noticeable by the user even for large flanges. The following pseudo code describes the path finding procedure:

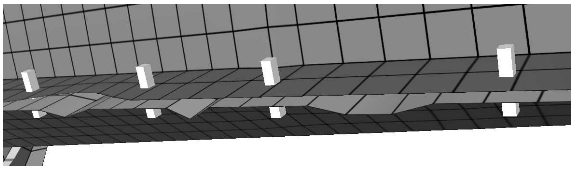

Figure 3: Irregularities like corrugations make flange detection difficult.

```
/*** method findPath ***/
list<Elements>
findPath(start, target) {
  Element start, target, element;
  fifoQueue<Elements> fifo;
  list<Elements> reserve, result;
  bool targetReached = false;
  fifo.add(start);
  while (not targetReached) {
    element = fifo.retrieve();
    for (all neighbour of element) {
      if (element == target) {
        targetReached = true;
        neighbour.previous = element;
        break;
      }
      if (neighbour.unvisited) {
        neighbour.markVisited();
        // remember the path we came
        neighbour.previous = element;
        if (neighbour.isFlange)
          fifo.add(neighbour);
        else
          reserve.add(neighbour);
      }
    }
    if (fifo.isempty)
      fifo.consume(reserve);
  }
  // collect saved elements
  for (element = target;
       element != start;
       element = element.previous)
    result.append(element);
  return result;
}
```

The algorithm above terminates in linear time O(N) in function of the number of elements N. Each element is treated at most once. The search is a breadth-first search of the finite element mesh, searching with increasing radius around the start element. The problem of the small gaps and corrugations is also solved. When the fifo queue runs empty, this means we have checked all connected flange elements without reaching the target. In this case we have to jump over non-flange elements in order to reach the target. Therefore, a list of the encountered non-flange elements is built. The search continues from each of these non-flange elements when all other possibilities are exhausted. The shortest path we get in this case therefore also contains non-flange elements. These will be properly treated later when determining the mid-line.

The algorithm just distinguishes between flange and non-flange elements. A possible alternative would be to rate the elements in function of their flange properties. Then, a recursive algorithm could find all possible paths rated in function of their length and flange element quality. This needs exponential time, and seems not being necessary regarding the good results with the iterative algorithm. Once a shortest or best path has been found, it is necessary to compute the flange's mid-line in order to place spotwelds on it (see Figure 4). Some algorithms for finding mid-

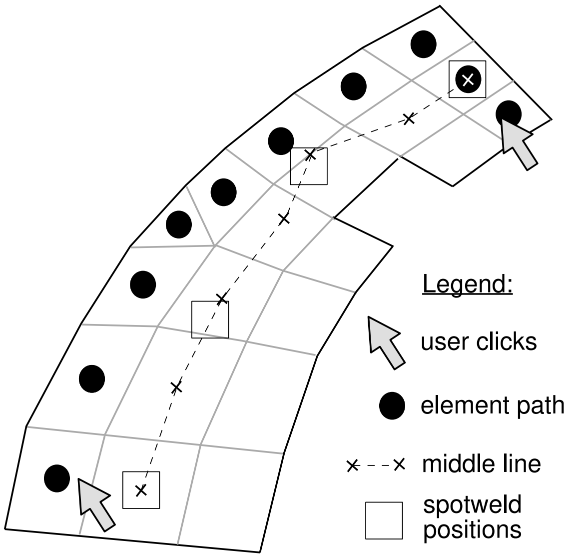

Figure 4: Path and mid-line example

lines of polygons can be found in the literature, see [6] and the references there. However, none of them seems to be perfect for our needs, either because it handles just bitmapped, or just planar polygons, and they require the conversion of the finite element mesh into a proper polygon.

Instead, we developed another, more straightforward approach: For each element of the previously found path, search the left and right flange border closest to the current element. Then add the midpoint between the left and right border as a new point of the polygon line that will be the medial axis - or at least a sufficiently close approximation.

It is not enough to find the two nearest flange borders. The distance has to be measured on the flange surface, which is not necessarily a plane. Furthermore, the nearest two borders do not always define the mid-line, for example at flange corners. And last but not least, parts of the path may run cross to the flange and mid-line direction. Hence, for each flange element we search for the border in all directions, i.e. left and right, forward and backward, add the border distance of the opposite directions and find the minimum: `Min(left + right, forward + backward)`.

Since finite elements on flanges are mostly aligned along flange direction, the minimum above is the flange width, the maximum would be the flange length. Problems arise at triangular elements and at flange corners containing triangular elements, as these elements introduce irregularities in the mesh. As these cases are not frequent, we can skip those regions and interpolate the mid-line. Also we skip non-flange elements of the previously found path, since the mid-line there is undefined. Interpolating such undefined areas with a straight line gave good results in practice.

3 ADHESIVE BONDING

Another new bonding technique is the usage of adhesives. In contrast to spotweld lines adhesive bondings are surface links which entails a completely different way of representation of the bonding agent. However we kept the user interaction as simple as with spotweld lines.

3.1 Visualization of Adhesive Strips

The problem of all contact types is that they are hardly visible from outside since they naturally are comparatively small and are placed between two or more assembly parts. There are three different ways to solve this problem:

Transparent components:
- ⊕ bonding agent easy to see
- ⊕ counterpart can be seen
- ⊖ confusing with lots of bondings due to many transparent components (substantial argument against this solution)

Partially transparent components:
- ⊕ only in vicinity of bonding agent transparent
- ⊖ parameter mapping interferes with transparency

Illustrate bonding by thickened representation:
- ⊕ easy locating the bonding agent because it sticks out of the bonded component's surface
- ⊖ to see the bonding surface the representation needs to be transparent, solution see below

All solutions have in common that they use transparency in any form. In doing so the problem is a shortcoming in the high level graphics API we are using: The render action does not sort transparent objects back to front. A workaround for this problem will be shown below. We have chosen the latter possibility: It minimizes difficulties with this deficiency and it yields the best clarity when visualizing many bonding structures.

There are two reasonable ways to thicken adhesive bondings:
1: Constant thickness equal to the maximum allowed distance of the assembly parts.
2: Variable thickness so that the adhesive representation barely sticks out of the surfaces of the two bonded components.

The advantage of the second way is that we might as well represent the adhesive bond by textures on the assembly parts stuck together, so we need no extra geometry. On the other hand we cannot use 1D parameter textures (Section 5) anymore, since few workstations support multi-texturing. The first solution has the benefit that the engineer gets a feedback about the distance of the assembled parts which finally led to the decision in favor of the first possibility. There the thickening is achieved by creating two additional surfaces shifted along the averaged node normal vectors.

To improve the conspicuousness of the bonding layer we use a checkerboard look alike texture alternating opaque white and full transparency. Through the transparent parts the joined components can easily be seen. Unfortunately we cannot use the alpha test feature of OpenGL [7] to avoid drawing into the depth buffer as this may result in a performance problem on some machines. Therefore, we need a workaround for the mentioned Cosmo3D transparency problem. We simply append all bonding related shapes at the end of the scene graph, even after the spotweld representations. Thus, it is guaranteed that spotwelds are visible in combination with adhesive bondings, which happens very often.

Figure 5 shows that we closed the outside of the adhesive strip so we get a better impression of the boundaries of the bonding surface which is the topic of the following section.

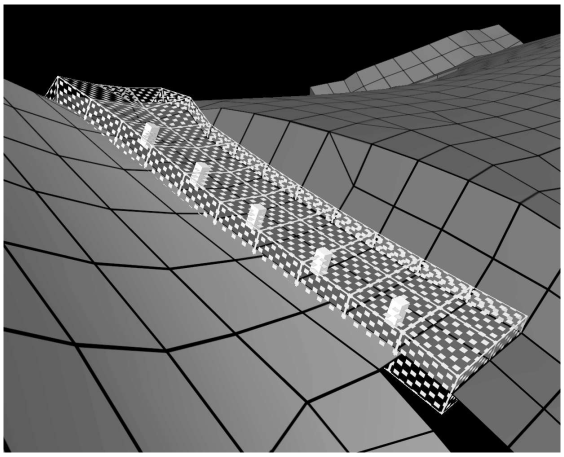

Figure 5: Example of an adhesive bonding in combination with spotwelds

3.2 Detecting Boundaries

The search for the boundaries is based on a simple algorithm: For each edge we count the number of conjoint finite elements. In the internal crashViewer data structure we have already given the neighbourhood relations of the elements and we also know the nodes which build up a finite element. For each adhesive type element we examine its edges. All edges with two common finite elements are within the material, all edges which belong to only one element lie at an outer boundary (Figure 6). Three or more common el-

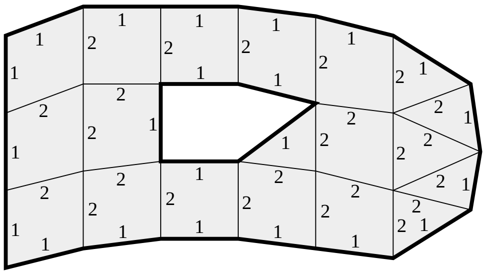

Figure 6: Finding the boundary edges

ements for one edge are also possible, for example at T-joints (Figure 7). We call this an inner boundary and treat it like outer boundaries of all three (or more) elements. This is both a simple as obvious solution since we can achieve exactly the same when using independent bondings. This is valid since it produces the same numerical results. Now, we have a list of pairs of node IDs,

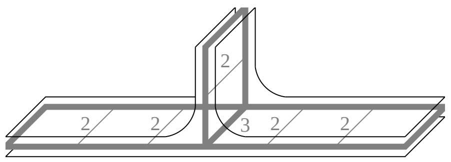

Figure 7: Special boundaries, e.g. T-joint

a stack of edges. We must merge them to get a continuous polygon line, the boundary. Of course there can be more than one boundary line. At adhesive strips with a hole for example, we get two boundary lines, an outer one and another one for the hole (Figure 6).

One can think of the node ID pairs as domino stones. We start with a random stone and add matching stones to the ends of the forming domino queue. If we cannot find matching stones anymore this boundary line is complete. Because we have non-manifold surfaces, the two ends of the queue must match also, we get a closed line stroke. If we used up all stones (no edges are left) the job is done. Otherwise we start another boundary queue with one of the unused 'stones'.

For speed-up purposes we may keep the node ID pairs sorted in two lists (one for each side) and do a binary search on them. The algorithm is fast without this optimization (10 seconds for about 500 different materials consisting of nearly 200,000 elements on an RS10k at 250 MHz). It would be a nice feature to reduce the order of $O(n^2)$ to $O(n \cdot \log_2 n)$ (where n is the number of boundary edges).

3.3 Creation of Adhesive Strips

When creating adhesive strips we use the same techniques as for spotweld lines. Both, user interaction and internal algorithms are similar or inherited. The engineer picks one assembly part and defines the length of the bonding on the second component with just three mouse clicks overall. The initial width of the adhesive strip is determined by a scalable maximum distance from the prior mentioned mid-line of the flange. All elements within that area meeting the flange criteria are used to build up the mid-surface of the actual bonding agent. Therefore, we project the eligible nodes of the first on the corresponding elements of the second component. The averaged coordinates of original and projected nodes define the mid-surface.

One problem is left: the shape of the adhesive strip around the start and end point. We need to find a rule to get a straight termination. Thus, we take the vector $\vec{p}$ built from the first two coordinates of the mid-line as initial clue (Figure 8). Then we project all four (or three) nodes of the

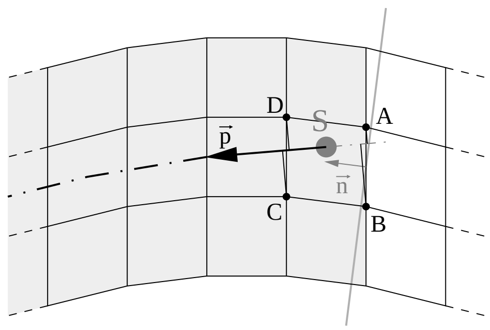

Figure 8: Calculation of the limiting planes

nearest element onto this vector. The two nodes which lie farthest from the mid-line (these have the smallest dot-products) define the terminating edge for this element. To obtain a limiting plane for the whole adhesive strip we simply take the midpoint of this edge and the center of the element (average of all four or three node coordinates) to define the normal vector $\vec{n}$ of this limiting plane. In the same manner we specify the termination of the other end of the bonding strip.

4 INITIAL PENETRATION HANDLING

By means of the meshing step parametric surfaces of the CAD data will be transformed into a discretized finite element mesh. Since the whole car body model consists of hundreds of independently meshed car body parts, this process may include '*initial penetrations/perforations*'. Figure 9 points out the initial penetrations as points, where one discretized surface is closer to another surface than the specified material thickness (left and right circle). Areas where elements intersect each other are called perforations (mid circle). Initial penetrations will cause initial forces in the simulation task, and this will falsify the simulation results. Therefore, it is important to detect and remove initial penetrations during the preprocessing of the simulation input data deck.

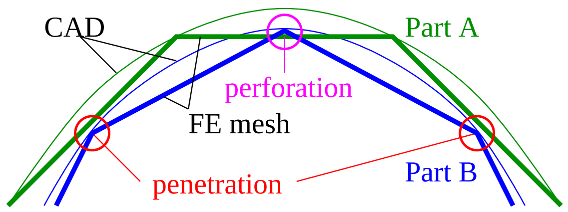

Figure 9: The assembly of independently meshed car body parts may cause '*initial penetrations*' which will influence the simulation results in an undesirable way. Even perforations (mid) could occur where the finite element meshes interpenetrate each other.

In order to detect those vertices which are positioned too close to an element of another car component, the minimal distance of each vertex to each finite element has to be calculated. This task can only be efficiently solved by using some kind of hierarchical substructuring.

Gottschalk et al. [9] presented bounding volume hierarchy algorithms which have been developed to enable real-time collision detection. Their approach compares the effectiveness of different bounding objects and introduces a fast overlap test for oriented bounding boxes. Their results have shown that object oriented bounding boxes perform better for collision detection than axis-aligned boxes because they need quite less interference tests.

Since we have to calculate point-to-polygon distances, which is cheaper than polygon-to-polygon tests, we use an axis-aligned bounding box hierarchy for the detection of initial penetrations. This requires less time for the bounding volume tree generation and saves any transformations of point coordinates during testing.

First of all we specify the maximum distance of interest which should be at least as thick as the maximal car component thickness. In the initialization phase this value is stored as the current minimal distance. During the test of one vertex with another sub-mesh, first the distance between the vertex and the bounding volume is computed. Only if it is smaller than the currently stored minimal distance, the children of the bounding object will be tested next. A child can be a set of more bounding volume instances or one or more finite elements, if the bounding object was a leaf node in the hierarchy.

During the distance calculation this approach eliminates nearly all car components except the direct neighbours at the top level of the bounding volume hierarchy. Just a small number of tests are applied until the point is tested on a per element basis. There the minimal distance is calculated by the slightly modified algorithm proposed by Campagna [8] which considers each projection case and computes values only if they are needed for that particular case. For example, the computation of the minimal point-to-polygon distances for a car model with more than 600 components consisting of about 200,000 elements/nodes takes 17 seconds on an SGI R12k at 300MHz. Afterwards, the values are mapped into coordinates of a one-dimensional texture that is used for distance visualization [1].

After detecting all initial penetrations the engineer can mark car parts as '(un-)modifiable' before the removal algorithm is started which moves each node of the modifiable meshes along the calculated force vector in a number of iterations unless the initial force is larger than null. The selection of modifiable car parts is very important for the replacement of individual components by variants — here, the node coordinates of the variants should be adopted while the rest of the car body model stays fixed.

Before detecting and removing any initial penetrations all initial perforations have to be eliminated. For the detection of initial perforations, oriented bounding boxes perform better than axis-aligned ones because the task is similar to a collision test. Until now the elimination is done manually. In future work the nodes of the perforating mesh could be projected on the correct side of the corresponding element. Then, there exists an initial penetration instead of a perforation. In a second pass this penetration could be removed as already described above.

5 FLANGE VISUALIZATION

During the assembly of a simulation input deck it is important to properly define the constraints between car parts, for example, with spotwelds or adhesive bondings. Generally, such contacts are placed at flanges. Since the simulation models become more and more complex it is helpful for the engineer while connecting adjacent components to restrict the visualization of the car body to those flanges. In crashViewer this can be done interactively without generating new geometry in the underlying scene graph API.

After the minimal point-to-closest-element distance has been computed for each mesh node, a previously specified distance range is mapped into the texture coordinate range $[0.0, 1.0]$. If these coordinates are used together with a one-dimensional (RGB)α-texture map and the alpha test is employed, the visibility of geometry is influenced in correspondence to the mapped parameters. Additionally, modern graphics hardware supports efficient transfer functions by means of texture color lookup tables. So if we map the distance values into indices of such a table, the visibility of geometry can be controlled interactively by modifying the transfer function of the α-channel. Now, it depends on the texture environment settings: GL_DECAL restricts the colored distance visualization to those areas where the entry of the *alpha*-channel pass the alpha test (Figure 10, mid). GL_MODULATE allows the restriction of geometry rendering to those regions. (Figure 10, right)

If the user has found a satisfying threshold and wants to restrict the rendering to the corresponding regions, crashViewer determines which nodes of the finite element mesh fulfill the specified range limitation. Then an indexed geometry is generated that includes all elements which reference at least one of those nodes. The indexed geometry is used to share the coordinate set with the original scene graph in order to minimize memory allocation.

6 CONCLUSIONS

We introduced a set of techniques which reduces the workflow paths in the car development process. The switch to independently meshed car

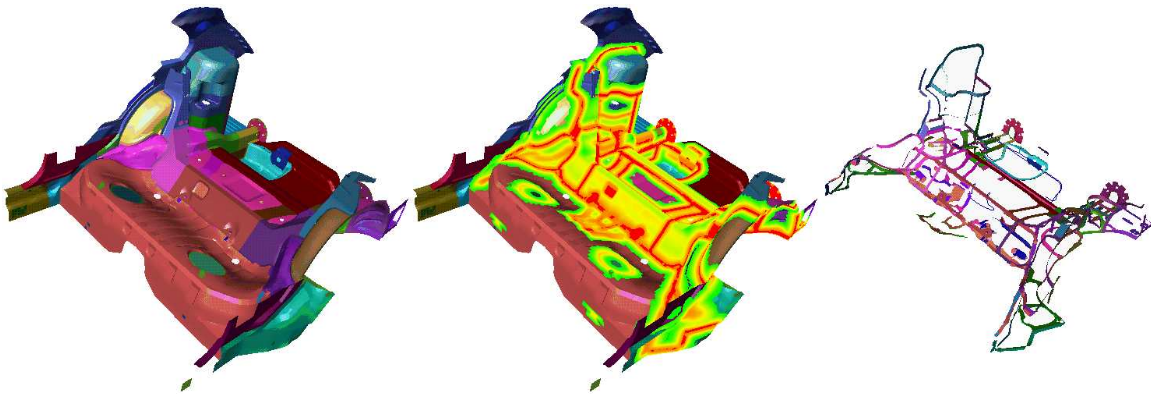

Figure 10: These images show parts of the back compartment of a car. The illustration in the middle visualizes the minimum distance from each node to the closest surface of another car body part up to 50 mm. On the right the same values are mapped to hide all geometry where this distance is more than 2 mm using the texture subsystem and the alpha test. The rendered geometry show potential flanges.

body parts required efficient algorithms for the interactive definition, modification, and deletion of assembly part connections like spotwelds and adhesive bondings. The presented visualization of such constraints and the rendering restriction to potential flanges supports the engineer in the preprocessing step. Furthermore, the algorithms for the detection and the controlled removal of initial penetrations allow the testing of multiple component variants. The described tools have been developed in cooperation with the BMW Group and some of them are in productive use at the crash simulation department.

REFERENCES

[1] O. Sommer, T. Ertl: *Geometry and Rendering Optimizations for the Interactive Visualization of Crash-Worthiness Simulations*, in Proc. of SPIE: Visual Data Exploration and Analysis VII, vol.3960, pp. 124-134, January 2000.

[2] Silicon Graphics, Inc.: *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization*, at `http://www.sgi.com/software/optimizer/tech_info.html`

[3] N.Frisch, T.Ertl: *Embedding Visualization Software into a Simulation Environment*, in Proceedings of the Spring Conference on Computer Graphics, pp. 105-113, Bratislava, 2000

[4] G.Barequet, S.Kumar: *Repairing CAD Models*, in *IEEE Visualization '97 Conference Proceedings*, pages 363–370, IEEE Computer Society Press

[5] R. Sedgewick: *Algorithms in C++, Parts 1-4*, Addison-Wesley 1998.

[6] F. Chin, J. Snoeyink, and C. A. Wang: *Finding the Medial Axis of a Simple Polygon in Linear Time*, Proc. 6th Ann. Int. Symp. Algorithms and Computation (ISAAC 95), Lecture Notes in Computer Science 1004, pp. 382-391, 1995.

[7] R. Kempf, Ch. Frazier: *OpenGL Reference Manual*, second ed., Addison-Wesley, 1998.

[8] Swen Campagna: *Polygonreduktion zur effizienten Speicherung, Übertragung und Darstellung komplexer polygonaler Modelle*, PhD thesis, University of Erlangen-Nuremberg, Germany, 1998.

[9] Stefan Gottschalk, Ming Lin, and Dinesh Manocha: *OBB-Tree: A hierarchical structure for rapid interference detection*, in Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996, held in New Orleans, Louisiana, 04-09 August 1996.