Interaktive Visualisierung von Strukturmechaniksimulationen

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Ove Sommer

aus Kiel

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. G. Greiner
Tag der mündlichen Prüfung:	15. September 2003

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2003

Abstract

This thesis presents visualization techniques and interaction concepts that have been developed for the pre- and post-processing of structural mechanics. This work was done in cooperation with the crash simulation department of the *BMW Group*.

The automotive industry's main goal is ensuring its share of a global market, which is becoming ever more competitive and dynamic. On the one hand, product quality has to be increased in relation to fuel consumption, weight, and passive safety. On the other hand the development process needs to be streamlined in order to reduce development costs and time to market. These objectives are hoped to be achieved by making extensive use of virtual prototyping.

The vehicle development process has completely changed during the past two decades. In the early eighties finite element analysis found its way into the simulation of structural mechanics in the German automotive industry. Initially developed for military purposes, in 1983 a group of engineers of several automotive companies started feasibility studies based on finite element models containing less than 5 000 beam elements. Seven years later productive results of finite element analysis in structural mechanics started to influence car body development. In the second half of the nineties important project decisions were made during the early phase of the development process based on a deeper insight into crash behavior provided by crash-worthiness simulation results.

Today, numerical simulation is an indispensable part of the computer aided product development chain in the automotive industry. After the design of a new car model is completed, a digital model is constructed by means of computer aided design (CAD). In a meshing step the created parametric surfaces defined by spline curves are discretized. The results describe the geometry of the car body by finite elements, for crash simulation mainly three- or four-sided shells but also beams and volumetric elements. The finite element model is completed, for example, with material properties, contact information, and boundary conditions. This is done in a preprocessing step. Once the simulation input data is available, the whole model is handed over to the simulation process which takes about one or two days using massive parallel processing. The deformation is recorded in a large result file. Commonly, the first 120 milliseconds of a crash are simulated and each two-thousandth time step is stored. A time step of less than one microsecond leads to a result file containing more than sixty snapshots. Typically, physical data such as acceleration, velocity, displacement, and forces per nodes or thickness, stress, strain, and other energies per element are recorded.

During post-processing the information is analyzed in order to evaluate crash-worthiness. The feedback provided for the construction department closes the loop when all goals set for the car body structure have been attained.

The virtual car body development in the pre- and post-processing of crash simulation divides into two main phases: (1) In the concept phase, new ideas are discussed and evaluated in order to reach a priority objective like minimizing car body weight. Only few people per car project are involved for a period of 12 to 18 months in this first stage. (2) After the basic decisions have been made regarding new concepts, a second phase follows which compromises all other fields to refine those concepts with respect to other objectives coming from the area of stability, dynamic or NVH. This phase requires many engineers from different departments for a period of more than 30 months to optimize the product. The later undesirable properties are revealed the higher the cost will be for the then necessary changes.

Until the late nineties transitions between adjacent car body parts were modeled as two finite element meshes that share common nodes at the border. Each time a car body part was replaced by an optimized variant in order to enhance the crash behavior either the whole car model had to be re-meshed or border nodes had to be adapted manually. Therefore, preparing such variant models was a very time consuming task. After simulation codes like PAM-CRASH were able to simulate car body connections like for example spotwelds, finite element models became more like their counterparts made of steel. Now, a flange is constructed, along which adjacent car body parts can be connected without sharing common nodes. This allows for an independent meshing of car body parts and therefore for more variant computations. However, independent meshing lead to mesh penetration or even perforation in flange areas when inhomogeneous meshes are assembled. Thus, a new task arose for preprocessing where those mesh errors have to be removed in order to avoid error-prone simulation results.

On the post-processing side, large time-dependent data sets require dedicated visualization methods that help the engineers to interpret the huge amount of data at interactive frame rates. Therefore, finite element meshes need to be prepared for an optimized processing in the graphics pipeline. Typical types of data that have to be visualized are scalar values, vectors, tensors or any combination thereof. In 1997, when most of the commercial visualization tools provided by solver companies as add-on to the simulation software could no longer provide state-of-the-art graphics, a demand arose for new visualization techniques available for productive use. Besides, the growing throughput caused by an increasing number of car body variants and the acceleration of computation hardware allowing more optimization cycles, made it necessary to automate as much processing steps as possible. The growing outsourcing of development tasks to suppliers and the number of merges in the automotive industry, which entail cooperation of corresponding departments at different sites, require a client-server-solution for cooperative work. Finally, the simulation community is concerned about the validity of their results compared to real crash-tests. It is important to detect and minimize the sources of scattering in the results originated by the simulation model or process. This is also a precondition for stochastic simulation,

which is used in order to optimize the crash behavior by varying input parameters.

The goal of this thesis is to provide solutions to some of the issues just stated. First of all, the applicability of different scene graph APIs is evaluated for large time-dependent data sets. APIs such as Performer or Cosmo3D / OpenGL Optimizer have been developed to take advantage of multiprocessing. Those APIs can perform model optimization during scene graph creation and benefit from multiprocessing using frustum culling and occlusion culling while traversing the scene graph to increase frame and interaction rates. Because of the large time-dependent databases and the limited memory of the workstations an efficient scene graph design is very important in order to handle the complex data interdependencies and to achieve high rendering speed. While five years ago the models consisted of about 250 000 finite elements with nearly the same number of nodes, today the size of the models has almost quadrupled. Since the element topology does not change in crash simulation, the connectivity of the finite elements needs to be stored just once. An index set representing the topology is shared across the sub-graphs of all time steps. A Gouraud shaded surface requires to do the edge detection on the state where geometry is deformed most, which is in general the last one. In order to minimize memory consumption, the index set is used for both coordinates and normals. Therefore, coordinates at vertices with multiple normals need to be added once per normal. Other scene graph nodes, e.g. the one specifying the appearance, can also be shared. Handling triangular elements as degenerate quadrilaterals allows to represent a mesh of three- and four-sided shell elements in one scene graph node. The prototype application named *crashViewer*, which was implemented to evaluate the methods developed in this thesis, uses Cosmo3D / OpenGL Optimizer for historical reasons: this bundle had been presented as predecessor of the 1997 announced Fahrenheit project, which was aborted two years later. Nevertheless, the proposed scene graph design allows to visualize 60 time steps of a model containing half a million elements and the same number of nodes with a memory consumption of 360 MB (flat-shaded) or 970 MB (Gouraud-shaded), provided that a crease angle of 20 degree leads to 40% more normals than vertices.

One basic requirement for an interactive visualization application is that the frame rate does not fall below an acceptable minimum threshold. What "acceptable" means depends on data and experience with other tools. Generally speaking, visualization data should be rendered as fast as possible. For this reason, one aim of this work is to point out methods to optimize the finite element meshes' rendering acceleration. Two approaches are examined: (1) concatenation of adjacent elements to reduce data redundancy during geometry processing and (2) mesh simplification to remove information that does not significantly influence the shape of a car body part. Although OpenGL Optimizer provides a tri-stripper (opTriStripper) that is able to convert any polygonal mesh into strips of triangles, it is not applicable in this field because the original mesh structure should still be visible in wireframe mode. Hence, a quadrilateral stripper was developed, which analyses the mesh structure and generates many parallel bands of maximum length. As a matter of fact, quad-strips are not as versatile as tri-strips because each turn costs two extra vertices. However, compared to opTriStripper's reduction to 63.5% the proposed bandification algorithm reduces the

number of referenced vertices to 54% of an unstripped representation averaged over 3 274 car body parts. Depending on the availability of vertex arrays the bandification leads to a rendering speed-up factor of about 4.5 without and 1.7 with vertex arrays in comparison to the unstripped geometry.

In order to achieve even higher frame rates during camera interaction, a two-stage level-of-detail concept is developed. In addition to a fine level displaying the original mesh resolution, which is essential for the visualization in pre- and post-processing of structural mechanics simulation, a second level with coarse triangles is used as intermediate model for camera movement. Each time the user modifies the view, a previously simplified mesh is rendered until the camera parameters are no longer changed. Then the finer level of detail is displayed. An simplification algorithm was implemented which uses the one-sided Hausdorff distance as an error measure and which is compared to the **S**uccessive **R**elaxation **A**lgorithm provided by OpenGL Optimizer as opSRASimplify. Aside from the interface opSRASimplify, which turns out to be unsuitable for getting an optimal decimated mesh with respect to a predetermined error tolerance, the resulting mesh quality is not as high as with the new HECSimplify simplifier. Breaking the error criterion can be avoided by defining an appropriate cost function for opSRASimplify, which causes less triangles to be removed compared to HECSimplify. If the decimation target is specified to achieve the same level of reduction, then the resulting triangle mesh contains gaps and the model appears distorted. Different car body models are reduced to 9-18% of the original number of triangles applying HECSimplify. This leads to a rendering speed-up factor of between 3.4 and 6.7. There is a trade-off between rendering speed and memory consumption. HECSimplify only applies half-edge collapses to the polygonal mesh. These operations just modify the topology, not the vertex coordinates. Therefore, the original and the reduced mesh are able to refer to the same set of coordinates. On the one hand, coordinate set sharing requires less memory, on the other hand the speed-up factor of a model that could be reduced to 9% of the original triangles is far away from 11.

Both modules, the quad-stripper and the simplifier, are embedded into the Cosmo3D / OpenGL Optimizer framework by providing corresponding action objects and new scene graph nodes. Furthermore, the scene graph API was extended by several other new objects to overcome the restrictions with line picking or to provide new functionality. For example, the csClipGroup node enables the user to control a freely movable clip-plane, which affects only the underlying scene graph. This provides better insight into heavily deformed car body structures.

Another form of clipping that is developed within this thesis uses one-dimensional RGBα texture maps in order to hide geometry that should not be displayed under certain conditions. This visualization method can be used, for example, to mask out those model regions that do not correspond to a given critical value range. In combination with distance values to adjacent car body parts the rendering can be restricted to potential flange regions of the model. For that purpose, first of all the parameter has to be transformed into a texture coordinate with respect to the specified parameter range. Then, a color scale is defined as a texture map. Using the GL_DECAL environment the color coding can be limited to

regions of certain value ranges. Using the same mechanism in the GL_MODULATE environment with the alpha-test enabled allows for clipping the geometry where the corresponding values map into texture regions with an alpha component set to be fully transparent. This technique facilitates the accentuation of critical structures, because the user is able to interactively modify the texture map or the texture lookup table, if an index texture is used to control the visualization.

As stated above, the most important change in finite element modeling for crash simulation was the introduction of independently meshed car body parts. Since the assembly of such inhomogeneous meshes may include perforations or penetrations, for example, caused by a shifted discretization along curved flanges, there was a growing demand for an interactive method to detect and remove this kind of mesh errors. Perforating regions can be detected by applying collision detection to the finite element model. Efficient collision detection as proposed by Gottschalk et al. [31] requires hierarchical sub-structuring of the car body model. Consequently, in this study, each car body part is subdivided by a bounding volume tree (BVT). Different bounding volume types are tested: spheres, axis-aligned (AABB), and object-oriented bounding boxes (OBB). For perforation detection, where computation-intensive element-element-intersectiontests are necessary at the lowest BVT level, OBBs turn out to be most efficient because they are very tight positioned around the element structure. Hence, downward traversals in the BVT can be terminated early because any one of 15 separating axes that defines a plane disjoining both volumes can be found. In order to detect penetrating nodes the minimum node-element-distances need to be computed. The BVT traversal algorithm is adapted appropriately. Children of a BVT node are visited only if their distance falls below a specified maximum distance of interest. Penetrating node-element-pairs are collected in a list for subsequent visualization. Using an interface to the original PAM-CRASH algorithm for penetration removal it is possible to provide the desired interactive mechanism that allows the engineers even to restrict mesh modification to selected car body parts. This is a big advantage over starting the simulation until initial forces move penetrating nodes apart from penetrated elements and restoring the computed mesh modification in the input data deck. First, the interactive method gives direct feedback and the engineers do not have to switch tools. Second, during the optimization of the car body structure by replacing single car body parts by variants this procedure enables the engineers to keep everything but the variant part fixed. Thus, only the nodes of the incoming part are aligned to its neighborhood and the confined modification makes it easier to compare the results of two simulations runs.

What is more, the bounding volume hierarchy provides beneficial effects on many other tasks in pre- and post-processing of crash-worthiness simulation. It can be used to detect and follow flanges automatically or to spot flange regions that have only been inadequately connected. A basic task is the verification of connecting elements. For example, a spotweld must not exceed a maximum distance to those parts it should connect. Otherwise, a pending spotweld may suspend the simulation run delaying the development process and raising costs.

In this study, several criteria for the validation of connecting elements are elaborated.

Erroneous spotweld elements are emphasized by different colors and/or geometry. The engineers are successively guided to each problematic connection. Without this feature it would not be possible to find these model errors in such a short time. Furthermore, a method is developed that enables the computation engineers to effectively add missing connection information by means of spotwelds to the input model. Thus, it becomes even practicable to start with crash-worthiness simulations before detailed connection data is available from CAD data.

One aim in virtual vehicle development is to combine the results of several areas in numerical simulation. In order to map the real development chain closely onto the virtual one, material properties influencing preliminary steps like forming have to be considered. For example, a deep-drawn blank sheet has a lower thickness in areas of high curvature than in other areas. As long as the material of a car body in crash simulation is assumed to be constant the before mentioned manufacturing influences cannot be properly represented. A hardware-based method is developed in this thesis for the efficient mapping of any type of data between incompatible meshes that are geometrically congruent. It utilizes the transformation and interpolation capability of the graphics subsystem. Element identifiers of one mesh are color-coded. For each element of the other mesh the view matrix is set up appropriately and the visible part of the first mesh is rendered. The colors that represent the element IDs can be read back into main memory. After the correlation between elements is finished in graphics hardware, the values are finally transfered in software.

The post-processing of crash-worthiness simulation results necessitates the handling of large data sets. Since a binary result file may contain 2 GB of data but an engineer's workstation often is limited to 1 GB of main memory these boundary conditions need to be considered while designing data structures and algorithms of visualization software for this application area. On the other hand, interactivity and high rendering performance is a precondition to obtain acceptance by the user. The required tool should provide interaction mechanisms that assist the user in exploring and navigating through the data. Mainly, it should help to interpret the data by making the invisible visible. Besides an effective scene graph design, in this study, the internal data structures of the developed prototype application have been implemented with memory consumption in mind. Parameter transfer in post-processing is done state by state very fast by pointer-based data structures. The extensive use of texture mapping enhances the rendering performance. Visualization techniques are proposed that use textures for the direct mapping of scalar values onto the car body geometry, for the animated display of vector data, and for the visual discretization of the finite element mesh in the form of a wireframe texture map. All these approaches spare the transformation stage of the graphics pipeline additional processing of vertex-based data. For example, the traditional display method for shaded geometry with visible element borders is two-pass rendering, which halves the frame rate. The application of a black-bordered luminance texture, which is white inside, onto each geometric primitive balances the load between geometry and texture unit. Also, the encoding of a vector's direction by applying an animated texture onto a line reduces geometry load and leaves the underlying structure mostly visible in contrast to conventional vector visualization with arrows.

Force flux visualization, first presented by Kuschfeldt et al. [44] gives an overview over which components of the car body model absorb or transfer forces. It is necessary to detect and to understand the force progression within the car body structure. For example, the longitudinal structures within the front part of a car body play an important role for increasing the ability of the body to absorb forces in a frontal crash. Force flux visualization enables the engineers to design car components with an optimal crash behavior. This technique was made available for interactive daily use in crash simulation analysis. Providing a dedicated interaction mechanism, the prototype application allows to interactively define a trace-line along which the force flux can be visualized. For each section plane positioned in small intervals perpendicular along the trace-line the simulated node forces are accumulated. The resampling is accelerated by utilization of the bounding volume hierarchy. Each section force sum is then represented by color and radius of one ring of a tube around the trace-line. The dynamic trace-line definition aligns the force tube to the deforming structure of the analyzed car body part, for example, a longitudinal mounting. The specified trace-lines can be stored in order to precompute force tubes off line. This can be done by another prototype application in batch processing after simulation has finished. During a visualization session the precomputed values can be directly converted into time dependent force tube representations. The decoupling of time consuming computation and the interactive visualization further accelerates the analysis of crash-worthiness simulation.

Starting multiple simulation runs with the same input data deck will produce different results. The scattering in results has to be minimized in order to be able to evaluate the influence of structure modifications. This work presents a method to detect and visualize instabilities of the simulation. The above stated texture-based visualization points out sources of instability and helps the engineers to determine if a branching is caused by the model structure or if it was originated by the solver. The pros and cons of different measurement functions are discussed.

Furthermore, a CORBA-based synchronization of multiple viewers displaying different data sets is presented. This allows to analyze the simulation results of one run in direct comparison to those of other runs. It is very useful to view the differences in crash behavior of multiple car body models on one workstation. Moreover, this functionality can be used in combination with a telephone call to supersede a meeting between a computation engineer and his external supplier. The visualization is done locally on each client. Providing that data and software is available at each participating client, the only data that have to be transferred during a cooperative session are the events triggered at the steering master-client and propagated to one or more slave-clients. A master token decides which participant is able to send generated events to the other instances. This mechanism avoids conflicting camera control when multiple users try to modify their view at the same time.

Another approach describes how an image-based client-server model can be used in this context. After a frame has been rendered on the server, it is encoded to reduce the amount of data. The encoded image stream is transferred to any client that is able to decode and display. There are less requirements for the client but the connection needs to provide a certain bandwidth. This scenario can also be used for remote visualization. The prototype

crashViewer can be connected to a Java applet running inside a web-browser.

Finally, a method for standardized analysis of crash-worthiness simulation is presented. A batch-processing prototype application has been developed to generate digital movies using a predetermined camera path. The contributions of this thesis aim at further acceleration of the virtual vehicle development process, for example, by introducing new interaction mechanisms, making extensive use of hierarchical data structures, using hardware-accelerated visualization techniques, and providing solutions for process automation.

Danksagung

Die vorliegende Arbeit wurde in den Jahren 1997 bis 1999 am Lehrstuhl für Graphische Datenverarbeitung der Friedrich-Alexander-Universität Erlangen-Nürnberg und im Anschluss daran bis 2001 in der Abteilung für Visualisierung und Interaktive Systeme des Instituts für Informatik an der Universität Stuttgart durchgeführt. Ich möchte mich bei all denen bedanken, die mir bei der Durchführung der Arbeit behilflich waren.

Mein besonderer Dank geht an meinen Doktorvater Prof. Dr. Thomas Ertl, der mir dieses interessante Thema zur Verfügung gestellt hat, den Fortlauf der Arbeit stets durch fruchtbare Anregungen und Diskussionen förderte und mir auch den notwendigen Freiraum zur Ausgestaltung der Arbeit einräumte. Darüber hinaus möchte ich ihm für seinen Beitrag zu der angenehmen Atmosphäre in seiner Arbeitsgruppe danken.

Für die Übernahme des Zweitgutachtens bin ich Prof. Dr. Günther Greiner dankbar.

Als entscheidendem Mitinitiator dieser Arbeit gilt mein Dank Dr. Michael Holzner, der das Thema anfangs bei BMW betreute. In diesem Zusammenhang möchte ich mich bei der Firma BMW für die Gewährung eines Doktorandenstipendiums bedanken. Viel Dank gebührt auch allen Mitarbeitern der Karosserieentwicklung bei EK-21, die durch zahlreiche Gespräche und wertvolles Feedback zum Gelingen dieser Arbeit wesentlich beigetragen haben. Hier möchte ich insbesondere Dr. Sven Kuschfeldt, Dr. Christoph Lübbing und Horst-Uwe Mader hervorheben, die nicht nur durch ihre Fachkenntnis, sondern auch durch ihre unkomplizierte Art stets wertvolle und angenehme Gesprächspartner waren.

Für die aus den Ergebnissen ihrer Diplomarbeiten hervorgegangenen Beiträge danke ich Jan Kraheberger, Horst Hadler, Christian Ernst und Manfred Weiler.

Nicht versäumen will ich, mich bei den Mitarbeitern der beiden Arbeitsgruppen in Erlangen und Stuttgart zu bedanken. Hier möchte ich Dr. Klaus Engel, Prof. Dr. Leif „P.“ Kobbelt und Prof. Dr. Rüdiger Westermann für die Zusammenarbeit bei verschiedenen Projekten danken. Für vielseitige Anregungen und das nette Raumklima danke ich meinen langjährigen Büroinsassen Dr. Peter Hastreiter und Dr. Martin Schulz.

Es freut mich, dass die Ergebnisse dieser Arbeit von weiteren Doktoranden aufgegriffen wurden und den entstandenen Prototypen durch Resultate aus den jeweiligen Forschungsgebieten bereichert haben; diesbezüglich danke ich Katrin Bidmon, Norbert Frisch und Dirc Rose für ihre Geduld im Umgang mit dem entwickelten Sourcecode.

Den engagierten Mitarbeitern, die sich neben ihren Forschungsaufgaben auch der Aufrechterhaltung des Rechnerbetriebs verpflichtet sahen, um der gesamten Abteilung ein

reibungsloses Arbeiten zu ermöglichen, sei an dieser Stelle auch gedankt, für die Zeit in Stuttgart waren das vor allem Matthias Hopf und Marcelo E. Magallon. Weitere Mitarbeiter, denen ich wegen ihrer Unterstützung in verschiedenen Bereichen danke, sind Sabine Iserhardt-Bauer und Dr. Martin Kraus.

Als helfende Hände bei organisatorischen Angelegenheiten möchte ich mich bei den beiden Sekretärinnen Maria Baroti und Ulrike Ritzmann für ihre Unterstützung bedanken.

Für die wertvolle Grundkonfiguration vom für mich unverzichtbaren Emacs und die schnelle Hilfe bei Fragen im Zusammenhang mit der SGI-Plattform möchte ich mich bei meinem Freund Alexander Dietz bedanken.

Dr. Susanne Schüle danke ich für die Korrektur von dem englischen Abstract.

Mein ganz besonders herzlicher Dank gilt meiner Frau Vera, die mich besonders in den letzten beiden Jahren sehr untertützt hat, indem sie viele Wochenenden mit unseren drei Söhnen weitestgehend ohne meine Mithilfe verbracht hat, um mir das Zusammenschreiben zu ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Problemstellung

Der Computer ist heute eines der wichtigsten Hilfsmittel beim Entwurf und der Entwicklung von Fahrzeugen. Zunächst wurde er von Konstrukteuren für das Computer Aided Design (CAD) von virtuellen Fahrzeugmodellen eingesetzt. Inzwischen ist er in vielen anderen Bereichen der Fahrzeugentwicklung unentbehrlich geworden: der Entwicklungszyklus von Automobilen ist durch die Computer-gestützte numerische Simulation substanziell verkürzt worden. An virtuellen Prototypen gewonnene Ergebnisse können zunächst optimiert und anschließend am realen Prototypen validiert werden. Untersuchungen der Fahrdynamik, des Crash-Verhaltens, der Innenraumakustik und der Außenhautumströmung sind nur einige Anwendungsfelder, in denen Computer-basierte Technologien zur Senkung der Entwicklungskosten beitragen.

Nach den Vorgaben aus dem Design wird ein Konstruktionsmodell erstellt; die erzeugten CAD-Daten beschreiben die Fahrzeugkomponenten anhand parametrischer Flächen und zusätzlicher Materialinformationen. Um die Daten in numerischen Simulationen verwenden zu können, müssen die Flächenbeschreibungen zunächst diskretisiert werden. Für die Strukturmechaniksimulation wird das CAD-Modell daher in ein Finite-Element-Modell umgewandelt, das dann zum größten Teil aus drei- und viereckigen Schalenelementen besteht. Das Finite-Element-Modell wird in einem Vorverarbeitungsschritt aufbereitet und mit zusätzlichen Daten ergänzt, bevor es dem Simulationsprogramm als Eingabedaten übergeben wird. Nach der meist sehr zeitintensiven Simulation, die für Gesamtfahrzeugmodelle mehrere Tage in Anspruch nehmen kann, liegen als Ergebnis große Datenmengen vor. Diese können aufgrund ihres Umfangs und ihrer Komplexität nur mit ausgereiften Visualisierungswerkzeugen ausgewertet werden. Die Erkenntnisse aus der Simulationsanalyse fließen an den Konstrukteur zurück. So schließt sich der Zyklus, der den virtuellen Prototypen solange iterativ verbessert, bis alle Zielgrößen erreicht werden.

Bereits Anfang der achtziger Jahre wurden Strukturanalysen mit Hilfe numerischer Simulation anhand einfacher Balkenmodelle durchgeführt. Die Modellkomplexität geht mit der Leistungssteigerung der Hardware und der Weiterentwicklung der Simulationssoftware einher: die Modellgröße hat sich seitdem alle drei Jahre mehr als verdoppelt. Der Notwen-

digkeit, dem Entwicklungsingenieur entsprechende Visualisierungswerkzeuge zur Verfügung zu stellen, wurde bisher von den Herstellern der Simulationssoftware mit eigenen Lösungen begegnet. Anfangs entstanden einfache Darstellungsanwendungen, die das Fahrzeug als Gittermodell zeichneten, ohne einen räumlichen Eindruck zu vermitteln; die Applikationen wurden weiterentwickelt, entsprechen jedoch heute in der Regel nicht mehr dem, was im Bereich der Softwareentwicklung und vor allem der Visualisierung Stand der Technik ist.

Der Fortschritt durch wissenschaftliche Forschung in der Computergraphik und die Weiterentwicklung der Hardware, insbesondere der Graphiksubsysteme, lässt die Lücke zwischen dem, was in der Visualisierung möglich ist, und dem, was in kommerziellen Produkten zur Datenanalyse angeboten wird, immer größer klaffen. Zudem ist die Wissenschaft im Bereich der angewandten Informatik stets bemüht, Einsatzgebiete zu identifizieren, in denen neu entwickelte Methoden evaluiert und verbessert werden können. Eine enge Zusammenarbeit zwischen Ingenieuren und Wissenschaftlern erscheint daher als sehr vielversprechend und zwingend notwendig.

Die vorliegende Arbeit ist im Rahmen einer engen Kooperation mit der Berechnungsabteilung der *BMW Group* entstanden. Sie hat zum Ziel, den bestehenden Fahrzeugentwicklungsprozess im Umfeld der Strukturmechaniksimulation zu analysieren und durch Adaption neuer Methoden der Computergraphik aus anderen Bereichen sowie durch Entwicklung neuer Visualisierungstechniken und Interaktionsmechanismen zu beschleunigen. Eine Evaluation der eingesetzten Konzepte soll anhand einer prototypischen Applikation vorgenommen werden.

Bisher wurde in der Visualisierung im Bereich der Strukturmechaniksimulation auf Basis von vierseitigen Schalenelementen nur wenig geforscht. Für die Analyse von Crash-Simulationsergebnissen müssen große, zeitabhängige Datensätze effizient verarbeitet werden. Dabei soll die Netzstruktur des Finite-Element-Modells erhalten bleiben, ohne dass dabei auf hohe Interaktionsraten verzichtet werden muss. Eine schnelle Datenaufbereitung spielt dabei eine ebenso wichtige Rolle, wie die Navigation durch das virtuelle dreidimensionale Fahrzeugmodell mit Hilfe der an einem Standardarbeitsplatz vorhandenen Eingabegeräte.

Die Weiterentwicklung der Simulationscodes hat es ermöglicht, Teilstrukturen mit Randbedingungen zu versehen, so dass nun Berechnungen an Teilmodellen vorgenommen werden können. In der Karosserieberechnung werden anstatt homogen vernetzter Gesamtmodelle seitdem unabhängig voneinander vernetzte Bauteile zu virtuellen Fahrzeugmodellen zusammengesetzt. Der Netzanschluss benachbarter Bauteile wird nun nicht mehr über das aufwändige Abgleichen und Nutzen gemeinsamer Randknoten hergestellt; stattdessen überlappen die Bauteilnetze in Flanschbereichen, wo sie durch neu entwickelte Verbindungselemente aneinander gebunden werden. Dies hat unter anderem den Vorteil, dass einzelne Bauteile durch Varianten ausgetauscht werden können, ohne dass die Umgebung neu vernetzt werden muss.

Wichtige Entscheidungen müssen bereits in der frühen Phase eines Fahrzeugprojektes aufgrund der durch numerische Simulation gewonnenen Erkenntnisse getroffen werden. Das setzt voraus, dass die bis dahin verfügbaren Konstruktionsdaten in rechenbare Si-

mulationsmodelle umgesetzt werden können. Durch die unabhängige Vernetzung einzelner Bauteile und den unterschiedlichen Konstruktionsstand der verschiedenen Fahrzeugkomponenten kommt es nach der Zusammenführung häufig zu Berührungen und Durchdringungen der Bauteilnetze im diskretisierten Finite-Element-Modell. Diese müssen zunächst detektiert und beseitigt werden, da sie ansonsten die Simulationsergebnisse verfälschen würden. Darüber hinaus müssen die Bauteilnetze miteinander durch Verbindungselemente verbunden werden. Da die Konstruktionsdaten in der frühen Phase jedoch keine vollständigen Verbindungsdaten beinhalten, muss der Berechnungsingenieur den Datensatz mit entsprechender Information aufbereiten. Die Vorverarbeitung von Eingabedaten für den Simulationsprozess nimmt angesichts steigender Variantenrechnungen und einer halbwegs automatisierten Standardauswertung der Simulationsergebnisse gegenüber der Nachbearbeitung einen immer höheren Stellenwert ein.

Derartige Ergänzungen der Simulationsmodelle mussten bisher mit Hilfe eines Text-Editors direkt an den Eingabedateien vorgenommen werden. Netzfehler und fehlende Anbindungen zwischen Bauteilen konnten lediglich durch Anrechnen des Modells entdeckt werden. Dazu wurde der Simulationsprozess gestartet und nach einiger Zeit wieder abgebrochen. Durch die Analyse der bis dahin berechneten Zwischenergebnisse werden derartige Unzulänglichkeiten des Eingabemodells sichtbar. Vorweggenommene Konsistenzprüfungen machen zeitaufwändige Anrechnungen überflüssig.

Weiteres Prozessoptimierungspotenzial liegt in der Integration und Angleichung verschiedener Werkzeuge. Eine enge Kopplung des Simulationsprozesses an die Vor- und Nachbearbeitung der Daten durch ein und dieselbe Applikation, die sowohl Ein- als auch Ausgabedaten verarbeiten kann, schafft die Grundlage für eine schnelle Iteration im Optimierungsprozess und trägt zur angestrebten Reduzierung der vom Ingenieur zu bedienenden Vielzahl von Applikationen bei.

In Zeiten fortschreitender Globalisierung und Fusionierung, aber auch durch zunehmendes Outsourcing der Teilmodellerstellung an darauf spezialisierte Dienstleistungsunternehmen steigt der Kommunikationsbedarf über räumliche Grenzen hinweg zusammenarbeitender Entwicklungsteams. Kostenintensive Besprechungen, zu denen sich alle Beteiligten an einem Ort zusammenfinden müssen, können nur durch Verbesserung der bereits vorhandenen Fernkommunikationsinfrastruktur reduziert werden. Da auf die gemeinsame Betrachtung der Modelldaten als Diskussionsgrundlage bei Besprechungen nicht verzichtet werden kann, bietet sich eine netzwerkbasierte Kopplung entfernter Arbeitsplätze an, um kleinere Besprechungstermine durch Telefonate mit zeitgleicher kooperativer Visualisierungssitzung am Arbeitsplatzrechner ersetzen zu können.

Ein weiterer Aspekt befasst sich mit der Absicherung der Zuverlässigkeit von Simulationsergebnissen. Um eine Aussage darüber machen zu können, welche Modifikation der Fahrzeugstruktur das simulierte Verhalten positiv beeinflusst hat, müssen zunächst alle Einflussfaktoren, die auf die Simulation wirken, analysiert werden. Dies geschieht in Stabilitätsanalysen, in denen anhand gleicher Eingabedaten die Streuung der Simulationsergebnisse gemessen und die Ursache dafür erforscht wird. Durch konstruktive Maßnahmen soll in Ursprungsbereichen maximaler Streuung das Modell dahingehend modifiziert werden,

dass gleiche Eingabedaten zu annähernd gleichen Simulationsresultaten führen.

Ziel dieser Arbeit ist, das Pre- und Postprocessing von Strukturmechaniksimulation im Fahrzeugentwicklungsprozess bezüglich neuer Funktionalität und Leistungsfähigkeit durch die Einbeziehung neuer Graphiktechnologien sowie durch die Entwicklung neuer Visualisierungsalgorithmen signifikant voranzubringen.

1.2 Beiträge dieser Arbeit

Da die Arbeit in enger Zusammenarbeit mit der Karosserieberechnungsabteilung der *BMW Group* entstand und die Forschungsergebnisse direkt von den Ingenieuren an alltäglichen Problemstellungen eingesetzt werden sollten, mussten zunächst Voraussetzungen für eine erhöhte Akzeptanz bei den Anwendern geschaffen werden. Dazu gehören vor allem kurze Ladezeiten der Daten, eine intuitiv zu bedienende Benutzerschnittstelle sowie komfortable Funktionalität, die es ermöglicht, die Aufgaben schneller zu lösen als mit anderen Applikationen. Die Analyse der zu verarbeitenden zeitabhängigen Simulationsdaten und der zum Einlesen zur Verfügung gestellten Bibliothek führte zu einer geeigneten internen Datenstruktur, die eine effiziente Datenaufbereitung erlaubt und damit zu geringeren Start-up-Zeiten führt als bei vielen kommerziell verfügbaren Visualisierungswerkzeugen. Ferner resultiert aus der Evaluation verschiedener Szenengraphbibliotheken unter Berücksichtigung der Rechnerplattform im Anwenderumfeld die Entscheidung zu Cosmo3D / OpenGL Optimizer. Durch die Datenstrukturen und Funktionalitäten der Bibliothek bleibt der Ressourcenbedarf im Zusammenspiel mit einem optimierten Szenengraph-Design im Rahmen dessen, was auf einem Standard-Arbeitsplatzrechner zur Verfügung steht.

Zur Beschleunigung der Bildsynthese werden bereits bekannte Verfahren zur Optimierung polygonaler Modelle für die Weiterverarbeitung in der OpenGL Pipeline mit adaptierten Algorithmen verglichen, die unter Berücksichtigung der zugrunde liegenden Daten Quadrilateralstreifen maximaler Länge bilden. Zusätzlich wird der Einsatz verschiedener Detailstufen im CAE-Umfeld untersucht und eine Lösung zur deutlichen Steigerung der Bildwiederholrate während der Navigation durch das Finite-Element-Modell präsentiert.

Durch die Entwicklung und Evaluation Textur-basierter Visualisierungsverfahren werden deren Vorzüge anhand verschiedener Beispiele aus dem Berechnungsumfeld verdeutlicht. Daraus lässt sich die Notwendigkeit ableiten, Standard-Arbeitsplatzrechner in Zukunft mit entsprechender Graphik-Hardware auszustatten, um von den Möglichkeiten moderner Visualisierungsalgorithmen profitieren zu können.

Im Rahmen der vorliegenden Arbeit wurde die Möglichkeit geschaffen, nach dem interaktiven Zusammenführen der Modellkomponenten aus verschiedenen Datenquellen auftretende Netzfehler zu visualisieren und selektiv zu beheben. Dazu kommen auf hierarchischen Datenstrukturen basierende Algorithmen zum Einsatz. Verbunden mit Methoden zur Darstellung und interaktiven Modifikation von Verbindungselementen sowie der Detektion fehlerhafter Schweißpunktdaten wird die Grundlage geschaffen, um Finite-Element-Modelle für die Crash-Simulation effizient aufzubereiten und die Modellerstellung für Variantenrech-

nungen stark zu vereinfachen. Speziell auf die Bedürfnisse der Berechnungsingenieure zugeschnittene Interaktions- und Navigationsmechanismen sowie frei bewegliche Clip-Objekte erleichtern den Umgang mit den Modelldaten.

Durch die Entwicklung dreidimensionaler Selektionsobjekte und eine effiziente Schnittkraftberechnung steht die Kraftflussvisualisierung mit Hilfe dynamischer Kraftflussröhren nun auch als interaktives Analysewerkzeug im Postprocessing zur Verfügung. Die Berechnung der notwendigen Größen im Batch-Betrieb und die anschließende Zwischenspeicherung in einem eigenen Dateiformat ermöglicht die Standardauswertung von festgelegten Kraftflussverläufen im Anschluss an die Simulation. Dies trägt weiterhin zur Beschleunigung und Automatisierung der Nachverarbeitung bei. Mit der Entwicklung eines Bild- beziehungsweise Filmgenerators konnte mit Ergebnissen dieser Arbeit zur Entwicklung eines Integrationswerkzeuges für die Ablaufsteuerung und das Datenmanagement in der Karosserieberechnung beigetragen werden.

Es werden zwei Lösungen für Szenarien einer kooperativen Sitzung mit mehreren Rechnern präsentiert. Die Ergebnisse zeigen auf, wie zeitaufwändige Treffen zwischen Ingenieuren durch Telefonate mit gleichzeitiger kooperativer Visualisierungssitzung ersetzt werden können. Das vorgestellte Verfahren zur Stabilitätsanalyse von Simulationsprozessen hilft, Ursprünge von Instabilitäten aufzudecken und die Aussagekraft der Simulationsergebnisse verbesserter Modelle zu erhöhen.

Durch diese Arbeit ist eine prototypische Visualisierungsplattform für die Vor- und Nachbereitung von Strukturmechanikdaten entstanden. Das objektorientierte Softwaredesign des Prototypen erlaubt die Integration weiterer Datenformate sowie die Implementierung neuer Algorithmen zu deren Evaluation im produktiven Einsatz in der Karosserieberechnung, aber auch in anderen CAE-Bereichen.

1.3 Gliederung der Arbeit

Im Folgenden wird ein Überblick über den Aufbau dieser Arbeit gegeben, woraus der Zusammenhang der Kapitel untereinander hervorgeht.

Kapitel 2 motiviert die Entwicklung einer in die Simulation integrierten Vor- und Nachverarbeitungsapplikation. Zu Beginn führt das Kapitel in das breite Feld der digitalen Fahrzeugentwicklung ein. Das Umfeld der Crash-Simulation wird detaillierter betrachtet, wodurch ein Fundament für ein besseres Verständnis der darauffolgenden Kapitel geschaffen wird.

Kapitel 3 gibt einen Überblick zu den Grundlagen der interaktiven Computergraphik und der Visualisierung. Darüber hinaus werden die der Crash-Simulation zugrunde liegenden Daten beschrieben, indem zunächst die Strukturen, aus denen sich ein Gesamtfahrzeugmodell im Allgemeinen zusammensetzt, erläutert werden und anschließend auf die Datenformate der Simulationseingaben beziehungsweise der Simulationsergebnisse eingegangen wird, um die breite Spanne der zu verarbeitenden Daten zu beleuchten.

Kapitel 4 präsentiert ein effizientes Szenengraph-Design für zeitabhängige Finite-Element-Modelle mit invarianter Topologie. Dazu werden die notwendigen Grundlagen der verwendeten Graphikbibliotheken vermittelt und Erweiterungsmöglichkeiten diskutiert.

Kapitel 5 erläutert die Architektur des im Rahmen dieser Arbeit entstandenen Prototypen und gibt einen Überblick über entwickelte Interaktionsmechanismen zur effizienten Datenanalyse.

Kapitel 6 diskutiert verschiedene Verfahren zur Darstellungsbeschleunigung. Während sich die Quadrilateralstreifengenerierung und die Simplifizierung mit der Optimierung der modellierten Geometrie für eine effiziente Verarbeitung in der Graphik-Hardware auseinandersetzt, zeigt der letzte Teil dieses Kapitels, wie durch den Einsatz von Texturen zusätzliche Geometrieverarbeitung überflüssig wird.

Kapitel 7 stellt spezielle Funktionalitäten für die Vorverarbeitung von Eingabemodellen vor. Außer der in verschiedenen Bereichen eingesetzten Distanzvisualisierung wird das interaktive Modifizieren von Schweißpunktdaten und die Parameterübertragung zwischen inkompatiblen Gittern erläutert.

Kapitel 8 veranschaulicht Konzepte für die Nachverarbeitung von Simulationsergebnissen. Es werden Techniken zur Visualisierung skalarer und vektorieller Größen präsentiert. Darüber hinaus werden die interaktive Kraftflussvisualisierung und die Darstellung von Instabilitäten in Simulationsergebnissen betrachtet. Die CORBA-basierte Erweiterung zum gemeinschaftlichen Arbeiten räumlich getrennter Anwender, sowie die Batch-basierte Bild- und Film-Generierung von Strukturmechanikdaten schließen die Vorstellung der im Rahmen dieser Arbeit neu entwickelten Methoden ab.

Kapitel 9 stellt die Ergebnisse dieser Arbeit in einen Kontext. An Beispielen aus dem produktiven Entwicklungsprozess werden die erzielten Fortschritte verdeutlicht und die Akzeptanz bei den Anwendern kritisch beleuchtet. Abschließend wird ein Überblick über weiterführende Arbeiten gegeben, die auf den vorliegenden Ergebnissen basieren.

1.4 Betreute Diplomarbeiten und Studienprojekte

- „Interaktive Aufbereitung von vorvernetzten Bauteilgeometrien für die Fahrzeugberechnung“, Jan Kraheberger, 1998 [40]
- „Evaluierung und Implementierung verschiedener Optimierungsverfahren für die effiziente Visualisierung komplexer Fahrzeugmodelle“, Horst Hadler, 1998 [32]
- „Medizinische Visualisierung im WWW mittels 3D-Texturen“, Christian Ernst, 1999 [16]
- „Evaluierung und Einsatz von OpenGL Volumizer zur Volumenvisualisierung auf strukturierten und unstrukturierten Gittern“, Manfred Weiler, 1999 [72]
- „OpenManip — Manipulatoren für Cosmo3D“, Michael Braitmaier, Michael Haiss, Markus Knauß, Siegfried Langauf, Stefan Opferkuch, Gunnar Stein, Philip Stolz, Mai 2000 – April 2001 [8]

Kapitel 2

Fahrzeugentwicklung und Strukturmechaniksimulation

Als Carl Benz 1886 in Mannheim der Öffentlichkeit mit dem Patent-Motorwagen das erste Automobil der Welt vorstellte, spielten Begriffe wie Design, Produktivität oder gar Passive Sicherheit noch keine Rolle. Inzwischen scheinen Kraftfahrzeuge ein unverzichtbarer Bestandteil unseres alltäglichen Lebens zu sein: während 1957 schon eine Million Pkw in Deutschland hergestellt wurden, waren es in den letzten drei Jahren stets über fünf Millionen jährlich. Über 43 Millionen Pkw waren im Jahr 2000 in Deutschland zugelassen. Aus diesen Zahlen lässt sich ableiten, wie wichtig es für Automobilhersteller ist, die Bedürfnisse des schnelllebigen expandierenden Marktes in immer kürzeren Produktentwicklungszyklen zu befriedigen. Grundvoraussetzung für eine derartig schnelle Weiterentwicklung ist der massive Einsatz digitaler Hilfsmittel über alle Entwicklungsphasen.

Die nachfolgenden Abschnitte geben einen Überblick über die digitale Fahrzeugentwicklung und führen in die Einsatzgebiete der Strukturmechaniksimulation, insbesondere in das Umfeld der Crash-Berechnung ein.

2.1 Digitale Fahrzeugentwicklung

Der Begriff *Digitale Fahrzeugentwicklung* deckt die gesamte Produktentstehungskette vom Computer Aided Design (CAD), über das Computer Aided Engineering (CAE) bis hin zum Computer Aided Manufacturing (CAM) ab. Die Ideen der Designer werden von den Konstrukteuren in Form von parametrischen Flächenbeschreibungen als CAD-Daten den Berechnungsingenieuren zur Weiterverarbeitung zur Verfügung gestellt. Letztere transformieren die mathematische Beschreibung des virtuellen Fahrzeugmodells in ein diskretisiertes Finite-Element-Modell und untersuchen die dynamischen und statischen Modelleigenschaften mit Mitteln der Finite-Element-Methode. Parallel dazu werden weitere Untersuchungen zum Beispiel zur Fahrdynamik, zur Fahrzeugaußenhautumströmung oder zur Ergonomie und Akustik in der Fahrgastzelle angestellt. Darüber hinaus wird auch die

Produktion der Bauteile, zum Beispiel der Umformprozess im Rahmen von Tiefziehsimulationen, und der Einbau von Fahrzeugkomponenten beim so genannten *Digital Mockup* am Computer simuliert.

Heutzutage tragen die numerischen CAE-Verfahren in allen Bereichen der Fahrzeugentwicklung dazu bei, dass zwischen dem Designentwurf und dem Beginn der Serienproduktion nur etwa viereinhalb Jahre vergehen. Dieser Zeitraum setzt sich aus der Konzeptentwicklung und der Serienentwicklung zusammen. Während in der Konzeptphase wenige Ingenieure involviert sind und dort fundamentale Entscheidungen beispielsweise in der Karosserieberechnung über Trägerverläufe und Türen- oder Stirnwandauslegung treffen, befassen sich in der Serienentwicklung sämtliche Fachabteilungen damit, das Fahrzeugmodell zur Serienreife zu entwickeln. Die in der Fahrzeugentwicklung zum Einsatz kommenden Finite-Element-Methoden lassen sich in vier Problemklassen kategorisieren [19] (Abbildung 2.1).

	Linear	**Nichtlinear**
Statik	• kleine Verschiebungen (Translationen, Rotationen) • kleine Dehnungen • linearer Spannungs-Dehnungszusammenhang • konstante Randbedingungen Beispiele: - Steifigkeit (Rohbau, Türen, Klappen, ...) - Festigkeit (Federbeinaufnahmen, Wagenheberkonsolen, ...) - Betriebsfestigkeit (Dauerfestigkeit stark beanspruchter Bauteile)	• große Verschiebungen (Translationen, Rotationen) • große Dehnungen • nichtlinearer Spannungs-Dehnungszusammenhang • veränderliche Randbedingungen (Geometrie, Topologie) Beispiele: - quasistatische Versuche mit großen Verformungen - Beulen (zum Beispiel von Beplankungsteilen) - Kontaktproblematiken bei Türbelastungen - Bewertung plastischer Verformungen bei Missbrauchslastfällen
Dynamik	• kleine Verschiebungen (Translationen, Rotationen) • kleine Dehnungen • linearer Spannungs-Dehnungszusammenhang • veränderliche Randbedingungen (zeitabhängige Anregungen) • Masseneinfluss • Dämpfungseinfluss Beispiele: - Eigenschwingung des Fahrzeugs oder einzelner Bauteile - Antworten auf bestimmte Anregungen (Akustik, Lenkradzittern)	• große Verschiebungen, Geschwindigkeiten • große Dehnungen • nichtlinearer Spannungs-Dehnungszusammenhang • Dehnratenabhängigkeit • veränderliche Randbedingungen (zeitlich, geometrisch, topologisch) • Masseneinfluss • Dämpfungseinfluss Beispiele: - Crash-Simulation (Front, Heck, Seite, Dach, ...) - Schwingungsuntersuchungen mit nichtlinearen Teilstrukturen (Federn, Dämpfer, Lager, ...)

Abbildung 2.1: Diese Tabelle charakterisiert vier Problemklassen der Finite-Element-Methode und listet jeweils Beispiele von typischen Lastfällen auf.

Da es nicht nur schwieriger sondern auch wesentlich kostenintensiver ist, Änderungen in einer späteren Phase der Fahrzeugentwicklung vorzunehmen, gewinnt unter anderem die Karosserieberechnung in der frühen Entwicklungsphase immer mehr an Bedeutung. Die Strukturmechaniksimulation erlaubt, das Schwingungs- und Crash-Verhalten schon vor

dem Bau des ersten Prototypen zu untersuchen, und hilft damit, Fehlentscheidungen aufzudecken und zu korrigieren, bevor weitere Folgekosten in der Serienentwicklung entstehen. Um bereits in dieser frühen Phase zu rechenbaren Modellen zu kommen, werden lediglich die Strukturen, auf die sich die Entwicklung zunächst konzentriert, neu konstruiert, vernetzt und schließlich durch Fahrzeugkomponenten schon existierender Modelle zu einem Gesamtfahrzeugmodell vervollständigt. Daraus ergeben sich bereits in der Konzeptphase Modellgrößen von 300 000 Finite-Elementen und mehr.

Parallel zur rasant steigenden Rechenleistung der Simulationssysteme durch verbesserte Hardware und massive Ausnutzung von Parallelisierung werden die Simulationsmodelle immer komplexer. Während vor zwei Jahrzehnten noch Machbarkeitsstudien mit Balkenmodellen, bestehend aus 3 000 Elementen, durchgeführt wurden, reicht die Modellkomplexität für Crash-Simulationen nun an die Millionengrenze heran. Dadurch steigen in gleichem Maße die Anforderungen an die Software-Werkzeuge, mit denen die Modelle für die Simulation aufbereitet werden, und an die Visualisierungssysteme, die zur Analyse der Simulationsergebnisse benötigt werden.

2.2 Pre- und Postprocessing der Crash-Simulation

Einen Bereich in der Fahrzeugentwicklung stellt die Karosserieentwicklung dar, die sich zum einen mit der Schwingungsanalyse, also der linearen Simulation, und zum anderen mit der nicht-linearen Crash-Simulation auseinandersetzt. Da die vorliegende Arbeit in enger Kooperation mit der Crash-Berechnungsabteilung der *BMW Group* entstanden ist, stehen die weiteren Erläuterungen in engem Zusammenhang mit diesem Umfeld.

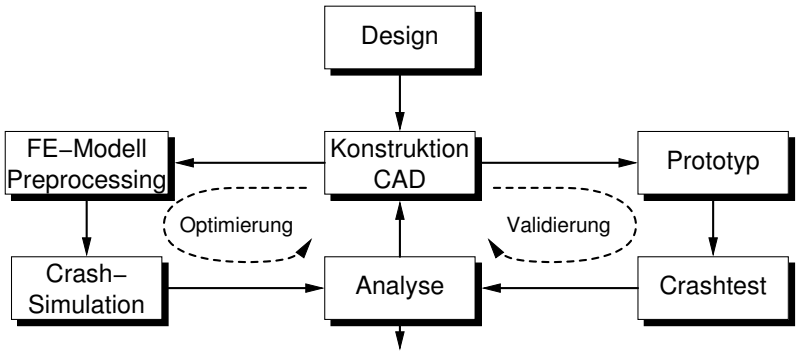


Abbildung 2.2: Die Strukturmechaniksimulation, hier am Beispiel der Crash-Simulation, wird bereits in der frühen Entwicklungsphase als Bestandteil der virtuellen Fahrzeugentwicklung eingesetzt und dient der Konzeptionierung der Karosseriestrukturen sowie in späteren Phasen der Optimierung des Crash-Verhaltens. Dagegen wird der reale Crash-Versuch wegen des zeit- und kostenintensiven Prototypenbau nur noch zur Absicherung der Simulationsergebnisse eingesetzt.

Aus der Sicht eines Berechnungsingenieurs stellt sich der Produktentwicklungszyklus während der frühen Phase eines neuen Fahrzeugmodells wie in Abbildung 2.2, links dar. Als Ausgangspunkt greift der Berechnungsingenieur auf die CAD-Daten der Konstruktionsabteilung zu und wandelt diese mit Vernetzungswerkzeugen wie zum Beispiel *ANSA* [4], *CATIA* [12], *HyperMesh* [2] oder *MEDINA* [67] in Finite-Element-Netze um. Dies geschieht im Allgemeinen Bauteil-basiert in Abhängigkeit der zugeordneten Materialbeschreibungen, die vorgeben, ob eine Substruktur aus Balken-, Schalen- oder Volumenelementen (siehe auch Abschnitt 3.4.1) besteht. Umfangreiche Datenmanagementsysteme erlauben, die Fahrzeugmodelle modular abzuspeichern, hierarchisch zu verwalten und die Substrukturen erst zur Modellerstellung wieder zusammenzuführen. Dieser Vorverarbeitungsschritt, in dem ein Gesamtfahrzeugmodell aus einzelnen Komponenten zusammengesetzt wird, nennt sich *Assembly*; dazu gehört auch das Vervollständigen des Datensatzes durch die Definition von Kontakten, Verbindungselementen und Randbedingungen sowie die Positionierung von Barrieren und Dummies.

Nachdem der Datensatz des Fahrzeugmodells vervollständigt und auf Fehlmodellierung hin untersucht wurde, wird er dem Simulationsprogramm, auch *Solver* genannt, als Eingabedaten gegeben. Während bis 1999 ein Gesamtfahrzeugmodell bestehend aus 500 000 Elementen auf einer SMP[1]-Architektur mit 6 CPU's in 5–6 Tagen berechnet worden wäre, benötigt eine MPP[2]-Architektur auf 16 CPU's nur noch 1–2 Tage für die gleiche Aufgabe. Durch das Aufprägen von entsprechenden Randbedingungen können Teilmodelle separat berechnet werden, was den virtuellen Entwicklungszyklus weiter beschleunigt.

Die steigende Leistungsfähigkeit der Simulationshard- und -software führt dazu, dass die Modelle noch präziser ausmodelliert werden, um genauere Aussagen über das Strukturverhalten zu erlangen. Dadurch steigt die Modellkomplexität stetig an. Außerdem können im Vergleich zu vergangenen Jahren mehr Berechnungen durchgeführt werden. Damit sind die Berechnungsingenieure in der Lage, die Strukturen an Schwachstellen im Fahrzeugmodell zu optimieren, indem sie einzelne Bauteile gegen Varianten austauschen. Als *Variante* werden Bauteile bezeichnet, an deren Konstruktion oder Vernetzung etwas geändert wurde und die das ursprüngliche Bauteil im Fahrzeugmodell ersetzen. Eine wichtige Voraussetzung für die effiziente Durchführung von Variantenrechnungen ist der modulare Aufbau des Gesamtfahrzeugmodells durch unabhängig voneinander vernetzte Bauteile. Bis Ende der neunziger Jahre waren aneinander grenzende Bauteilnetze in der Crash-Simulation dadurch miteinander verbunden, dass sie sich gemeinsame Randknoten teilten (Abbildung 2.3, oben). Wenn ein Bauteil durch eine Variante im Fahrzeugmodell ersetzt wurde, mussten die Randknoten der Variante in einem aufwändigen Verfahren mit dem umgebenden Finite-Element-Netz in Einklang gebracht oder das gesamte Modell neu vernetzt werden. Dieser Vorverarbeitungsschritt entfällt inzwischen, da die Simulationsprogramme nun die Auswirkungen verschiedener netzunabhängiger Verbindungselemente abbilden können. Die Bauteilnetze besitzen Flansche oder überdecken sich in den Bereichen, in denen sie miteinander verbunden werden sollen. Das bringt zum einen das Simulationsmodell der Realität näher

[1] SMP – shared memory processing
[2] MPP – massive parallel processing

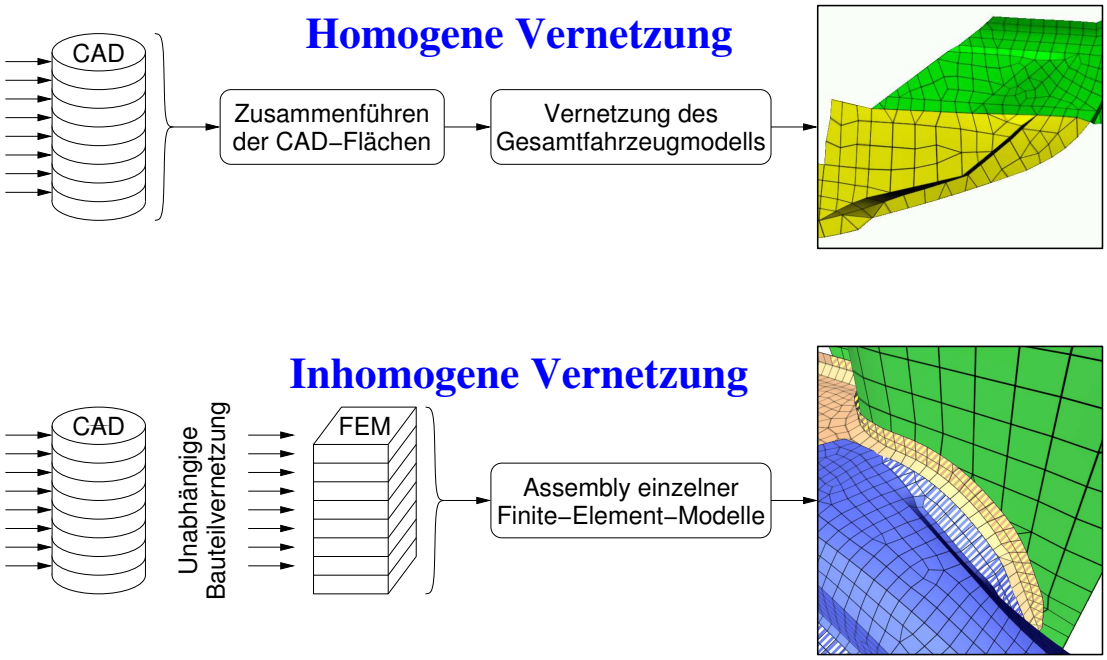


Abbildung 2.3: Bisher wurden aneinandergrenzende Bauteile im Finite-Element-Modell durch homogene Vernetzung, also korrespondierende Randknoten miteinander verbunden. Die Weiterentwicklung der Simulationscodes erlaubt inzwischen eine inhomogene Vernetzung, bei denen die unabhängig voneinander vernetzten Bauteile in Flanschbereichen (schraffiert) durch punkt-, linien- oder flächenförmige Verbindungselemente aneinandergekoppelt werden.

und hat zum anderen den Vorteil, den Modellaufbau zu beschleunigen und insbesondere den Austausch von Bauteilen wesentlich zu vereinfachen.

Durch das Zusammenführen unabhängig voneinander vernetzter Bauteile mit nichtkorrespondierenden Netzknoten kommt es besonders in gekrümmten Flanschbereichen, in denen die Bauteilnetze mit einem geringen Toleranzbereich sehr nahe aneinander liegen müssen, zu Netzberührungen (Penetrationen) oder gar -durchdringungen (Perforationen). Wie der Abbildung 2.4 zu entnehmen ist, müssen die Finite-Elemente im Flanschbereich benachbarter Bauteile mindestens die Hälfte der aufsummierten Blechdicken auseinanderliegen, um sich nicht zu berühren, wenn das Schalenelement die Mittelebene des Bleches darstellt. Ein weiterer, häufig auftretender Grund für derartige Netzfehler sind unterschiedliche Modellstände zusammengeführter Daten. Die Folge initialer Perforationen und Penetrationen im Eingabemodell der Crash-Simulation sind initial auftretende Kräfte, die die Simulationsergebnisse verfälschen. Daher gilt es, derartige Netzfehler zu detektieren und zu beseitigen, bevor das Finite-Element-Modell der Simulation übergeben wird.

Nachdem das Finite-Element-Netz bereinigt wurde, müssen bisher unverbundene Bauteile an angrenzenden Bauteilen des Gesamtfahrzeugmodells durch Verbindungselemente

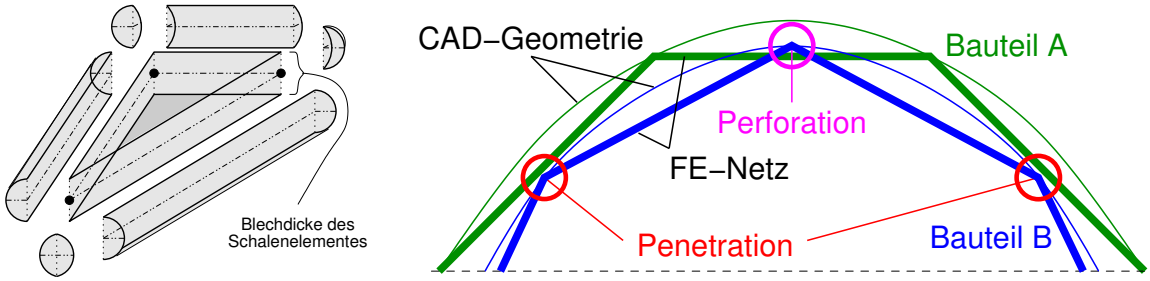


Abbildung 2.4: Das Bild links veranschaulicht, dass ein Schalenelement in der Simulation als voluminöser Körper behandelt wird, da die Blechstärke zu berücksichtigen ist. Bei der Zusammenführung unabhängig voneinander vernetzter Bauteile kann es daher, wie rechts skizziert, in Flanschbereichen zur Penetration (Berührung) oder gar zur Perforation (Durchdringung) kommen.

(zum Beispiel Schweißpunkte und -nähte, Klebeschichten, Nieten oder Schrauben) angebunden werden. Besonders in der frühen Phase eines Fahrzeugprojektes wird dieser Arbeitsschritt vom Berechnungsingenieur getätigt, da das Konstruktionsmodell noch keine Informationen zu Verbindungselementen enthält. Das Definieren von Verbindungselementen war vor Beginn dieser Arbeit lediglich durch das Editieren des Eingabedatensatzes mit dem Texteditor oder bei Schweißpunkten durch Angabe der Koordinaten in einem Dialogfenster möglich. Eine effiziente Bearbeitung der Modelldaten im Vorverarbeitungsschritt erforderte in diesem Bereich eine leicht zu handhabende, intuitive Möglichkeit, diese Daten interaktiv zu definieren.

Als weiterer Arbeitsschritt gehört das Einbringen nicht ausmodellierter Bauteile durch Verteilung ihrer Masse auf benachbarte Substrukturen des Fahrzeugmodells, der so genannte *Massentrimm*, zu den Vorverarbeitungsaufgaben in der Strukturmechaniksimulation. Nicht zuletzt müssen die Bauteilnetze auch bestimmten Qualitätsansprüchen genügen, daher spielt die Modellvalidierung eine wichtige Rolle. Zum Beispiel müssen die Innenwinkel der Finite-Elemente in einem gewissen Toleranzbereich liegen und eine mögliche Torsion vierseitiger Schalenelemente sollte einen vorgegebenen Grenzwert nicht überschreiten. Des Weiteren gilt es, fehlerhafte Verbindungselemente zu eliminieren, da diese unter Umständen die Simulationsberechnung zum Abbruch bringen können.

Abschließend wird direkt im Datensatz spezifiziert, wie viele Zeitschritte über welchen Zeitraum simuliert und welche Parameter berechnet und in den Ergebnisdaten abgelegt werden sollen. Zu den standardmäßig ausgewerteten Größen gehören

- Verschiebungsvektoren der Netzknoten, deren Geschwindigkeit und Beschleunigung
- minimale/maximale plastische Dehnung der Schalenelemente
- Dicke der Schalenelemente

Je nach Lastfall werden zusätzlich noch Knotenkräfte, Knotenmomente und Elementenergien ausgewertet.

Die Berechnungsingenieure sind bemüht, nachzuweisen, dass die Aussagekraft ihrer Simulationsergebnisse der von realen Crashtests entspricht. Dazu wird untersucht, wie groß die Streuung der Ergebnisse bei den verwendeten Simulationsmodellen beziehungsweise dem Simulationsprogramm ist, indem die Ergebnisse mehrerer Simulationsläufe mit den gleichen Eingabedaten miteinander verglichen werden. Durch stochastische Simulation, in denen bestimmte Parameter der Eingabedaten leicht modifiziert werden, wird überprüft, ob die Ergebnisse innerhalb eines tolerierbaren Streubereichs bleiben. Gleichzeitig können die Erkenntnisse dazu verwendet werden, das Fahrzeugmodell zu optimieren. Dank einer integrierten Vor- und Nachverarbeitung von Simulationsdaten durch direkte Anbindung entsprechender Applikationen an den Solver wird es dem Berechnungsingenieur in der Crash-Simulation voraussichtlich schon in naher Zukunft möglich sein, die Optimierung der Fahrzeugstruktur zu steuern und somit den Entwicklungsprozess stark zu beschleunigen.

Kapitel 3

Visualisierung im CAE-Umfeld

Die wissenschaftliche Visualisierung großer Datenmengen hat zum Ziel, aussagekräftige Bilder zu generieren, die dem Betrachter eine Analyse der Daten ermöglichen. Die dafür notwendigen Arbeitsschritte umfassen die Datenaufbereitung, die Wahl einer geeigneten Datenrepräsentation und eine effiziente Bildsynthese. Dieses Kapitel gibt einen Überblick zu den Basistechniken, die in der interaktiven Computergraphik und Visualisierung insbesondere im Umfeld der digitalen Fahrzeugentwicklung zum Einsatz kommen. Es schafft die Grundlagen für die weiteren Kapitel, in denen die im Rahmen dieser Arbeit entwickelten und eingesetzten Algorithmen erläutert werden. Der folgende Abschnitt gibt einen Überblick über die notwendigen Verarbeitungsschritte auf dem Weg von den Rohdaten bis zur Darstellung eines Bildes.

3.1 Visualisierungs-Pipeline

Die erste Stufe der in Abbildung 3.1 dargestellten Visualisierungs-Pipeline repräsentiert die Datenquellen, die die Rohdaten liefern. Die Rohdaten werden im Allgemeinen mit spezieller Hardware (CT-/MR-Scanner, Teleskop, Mikroskop etc.) analog gemessen und anschließend digitalisiert oder sie stammen von komplexen Modellbeschreibungen beziehungsweise aus Ergebnissen einer Simulation. Die inhomogenen Rohdaten sind in der Regel mehrdimensional und können skalare Größen, Vektoren, Matrizen und Tensoren enthalten. Den meisten Werten ist eine Position und bei zeitabhängigen Daten ebenfalls ein Zeitpunkt zugeordnet. Die verschiedenen Arten eines Gitters, auf dem die Daten vorliegen, lassen sich nach unterschiedlichen Kriterien klassifizieren. Sofern das Gitter, auf dem die Daten vorliegen, nicht für die Weiterverarbeitung geeignet erscheint, werden die Daten in einem Rekonstruktionsschritt zum Beispiel durch ein geeignetes Interpolationsverfahren auf das gewählte Gitter transformiert. Der Rekonstruktionsschritt ist fester Bestandteil der Visualisierungs-Pipeline für Messdaten und üblicherweise in das Messgerät integriert; unter Umständen kann er aber auch auf Simulationsdaten angewendet werden, wenn der Gittertyp gewechselt werden muss, um spezielle Algorithmen ausnutzen zu können. Durch Filterung werden

die Rohdaten dann auf die für die Weiterverarbeitung wesentlichen Visualisierungsdaten reduziert.

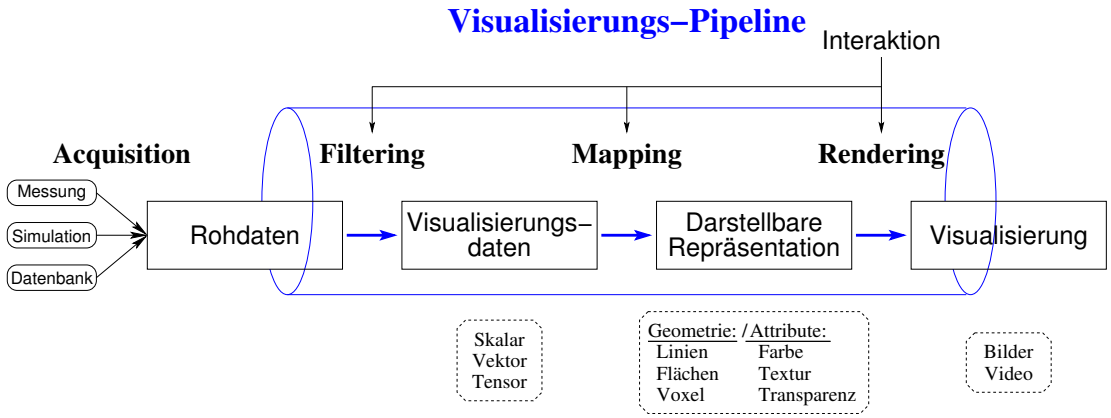


Abbildung 3.1: Die dargestellte Visualisierungs-Pipeline zeigt die verschiedenen Stufen der Datenverarbeitung bei der Visualisierung von den Rohdaten bis hin zum erzeugten Bild.

Im nächsten Verarbeitungsschritt, dem *Mapping*, wird eine geometrische Repräsentation für die zugrundeliegenden Daten gewählt. Dies können je nach Datentyp graphische Primitive wie Linien, Flächen oder Volumenelemente sein. Zusätzlich können die Primitive noch mit Attributen wie Farbe, Textur oder Transparenz versehen werden, um weitere Informationen in der Darstellung unterzubringen. Es kommen auch Glyphen zum Einsatz: das sind Hilfsgeometrien, die je nach Komplexität durch ihren geometrischen Aufbau und das Anwenden verschiedener Attribute für die Visualisierung mehrdimensionaler Daten genutzt werden. Als Zwischenergebnis entsteht eine darstellbare Repräsentation der Daten.

Der letzte Schritt in der Visualisierungs-Pipeline setzt sich mit der effizienten Bildsynthese (engl.: *Rendering*[1]) auseinander. Dazu werden aus der dreidimensionalen Szenenbeschreibung unter Anwendung von 3D-Graphikbibliotheken zweidimensionale Bilder generiert; dies geschieht, teilweise durch Ausnutzung spezieller Graphik-Hardware, stets mit dem Ziel, hohe Bildwiederholraten zu erreichen, die es dem Betrachter erlauben, durch interaktives Navigieren durch die Daten zusätzliche Informationen zu erhalten.

Für große Datensätze, wie sie aus Strömungs- oder Strukturmechaniksimulationen in der Fahrzeugentwicklung resultieren, müssen in allen Stufen der Visualisierungs-Pipeline Optimierungsschritte vorgenommen werden, um für die stetig steigende Komplexität der Daten hohe Interaktionsraten zu gewährleisten. Dazu gehören

- Wahl eines geeigneten Gitters: Reguläre kartesische Gitter ermöglichen zwar eine einfache Zellfindung und Partikelverfolgung durch implizite Nachbarschaftsinformation,

[1] *Rendering* ist eine allgemeine Bezeichnung für den gesamten Verarbeitungsprozess von einem dreidimensionalen Objekt zu einer schattierten zweidimensionalen Projektion auf einer Darstellungsfläche.

bringen jedoch häufig einen großen Anteil leerer Zellregionen mit sich, während in unstrukturierten Gittern lediglich die datentragenden Zellen enthalten sind.

- Hierarchische Unterteilung der Daten: Durch Strukturierung können effiziente Algorithmen zur Zellsuche zum Einsatz kommen. Die Verarbeitung von Volumendaten auf fein aufgelösten kartesischen Gittern, die nur zu einem geringen Teil mit Daten besetzt sind, ist durch hierarchische Verfeinerung möglich, durch die leere Regionen ausgespart werden.
- Vorverarbeitungsfilter: Die Anwendung von Filteroperationen kann zur Extraktion von Merkmalen in komplexen Datensätzen herangezogen werden, um die gesuchten Strukturen hervorzuheben.
- Simplifizierung: Die Anzahl darzustellender Dreiecke pro Bild wird durch geeignete Reduktionsalgorithmen auf ein Maß beschränkt, dass auch für ursprünglich große polygonale Modelle hohe Interaktionsraten und damit eine exploratorische Datenanalyse zulässt.
- Progressive Darstellung: Die Verwendung mehrerer Detailstufen (engl.: *levels of detail*) oder die Repräsentation von Oberflächen durch progressive Netze, die sich in Abhängigkeit des Abstands zwischen Betrachter und Objekt entsprechend verfeinern, beschleunigt ebenfalls den Darstellungsprozess.
- Szenengraphoptimierung: Die Beschreibung der Szene in Form eines gerichteten azyklischen Graphen entlastet die Graphik-Hardware durch Anwendung verschiedener Culling-Verfahren, bei denen ganze Szenenobjekte vorab als „nicht zum Bild beitragend“ klassifiziert werden. Durch ein optimiertes Szenengraph-Design können der Speicherbedarf sowie die Kosten für Änderungen an Einstellungen der Rendering-Pipeline, also die Anzahl der *state switches*, minimiert werden.
- Textur-basierte Darstellungsverfahren: Durch das Ausschöpfen verschiedener Funktionalität können Berechnungen direkt auf dem Graphiksubsystem durchgeführt werden, wodurch „in Hardware gegossene Berechnungsschritte“ ausgenutzt werden und der Bus als Flaschenhals zur CPU umgangen wird.

Im Rahmen der vorliegenden Arbeit sind diesen Ansätzen entsprechende Algorithmen neu entwickelt und angepasst worden, um die interaktive Visualisierung von Strukturmechanikdaten zu optimieren. Nachdem eine geeignete Datenrepräsentation in Form einer darstellbaren Szenenbeschreibung gefunden wurde, setzen sich Methoden der interaktiven Computergraphik damit auseinander, wie aus dem 3D-Modell in kürzester Zeit ein Bild generiert werden kann.

3.2 Interaktive Computergraphik

Interaktive Computergraphik hat das Ziel, durch die Verwendung effizienter Verfahren aus bereits definierten, meist dreidimensionalen Modellbeschreibungen möglichst schnell zweidimensionale Bilder zu erzeugen, um anschließend wieder auf die nächsten Einga-

ben des Benutzers reagieren zu können. Dieser Vorgang entspricht der letzten Stufe der Visualisierungs-Pipeline (Abbildung 3.1) und wird als Bildsynthese oder Rendering bezeichnet. Im Gegensatz zum fotorealistischen Rendering, bei dem das primäre Ziel höchster Bildqualität mit globalen Beleuchtungsmodellen wie Raytracing oder Radiosity trotz hohen Rechenaufwands verfolgt wird, kommen im Bereich interaktiver Graphikanwendungen nur lokale Beleuchtungsmodelle zum Einsatz, da diese von der Graphik-Hardware direkt unterstützt werden. Eine „interaktive“ Visualisierungsapplikation bietet dem Anwender jederzeit die Möglichkeit, über Eingabegeräte Einfluss auf die Darstellung zu nehmen: die Eingabe wird sofort verarbeitet und das Ergebnis wird dem Anwender unverzüglich in Form eines neuen Bildes angezeigt. Der Begriff der „Echtzeitvisualisierung“ stellt noch höhere Anforderungen: die Bildwiederholrate beträgt mindestens 25 Bilder pro Sekunde, so dass der Betrachter Veränderungen der Darstellung als fließend empfindet.

3.2.1 Rendering-Pipeline

Der Begriff der Rendering-Pipeline umfasst alle Verarbeitungsschritte, die notwendig sind, um von den Daten eines 3D-Modells zu einem 2D-Bild zu gelangen. In Abbildung 3.2 werden diese anhand der Koordinatensysteme skizziert, in die die Daten nacheinander überführt werden:

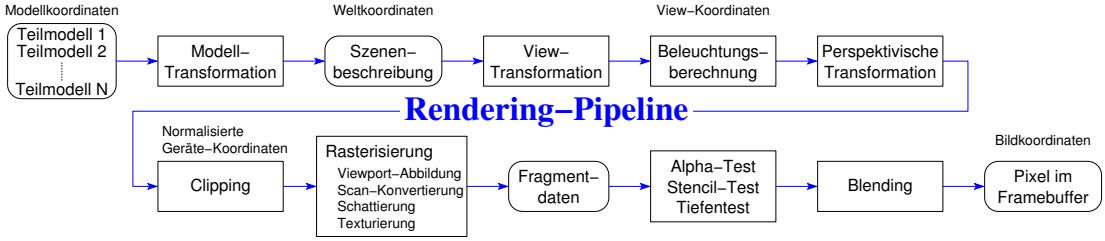


Abbildung 3.2: Ein grober Überblick der Rendering-Pipeline verdeutlicht, welchen Transformationen die 3D-Geometriedaten auf dem Weg zum 2D-Bild unterzogen werden.

- Modellkoordinatensystem: Jedes Objekt der Szene wird in lokalen Koordinaten beschrieben. Gegebenenfalls muss das Objekt zunächst tesseliert werden, um ein Dreiecksnetz zu erhalten.

- Weltkoordinatensystem: Durch die Modelltransformation werden die einzelnen Objekte in der Szene zu einem 3D-Modell zusammengeführt. Außerdem werden Position und Ausrichtung des Betrachters sowie der Lichtquellen festgelegt.

- View-Koordinatensystem: Nach Anwendung der Viewing-Transformation kann die Menge der weiterzuverarbeitenden Dreiecke durch View Frustum Culling gegen das Pyramidenstumpf-förmige Sichtvolumen reduziert werden.

- Normalisiertes Geräte-Koordinatensystem: Durch perspektivische Projektion, Clipping in homogenen Koordinaten und anschließender perspektivischer Division wird der Inhalt des View-Frustums auf ein normalisiertes, Quader-förmiges Volumen abgebildet. Im letzten Schritt, der Rasterisierung, werden die polygonalen Daten durch Scan-Konvertierung in Pixeldaten umgewandelt; dies schließt auch die Schattierungsberechnung und eine eventuelle Texturierung ein.

- Bildkoordinatensystem: Die Viewport-Abbildung befördert die Pixel nach bestandenem Alpha-, Stencil- und Tiefentest schließlich an die richtige Position im zweidimensionalen Bild.

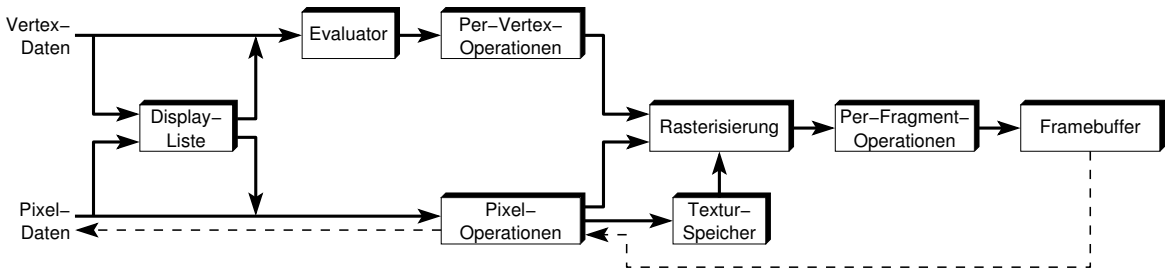


Abbildung 3.3: Die OpenGL-Pipeline skizziert die Pfade der Daten zwischen den einzelnen Verarbeitungsschritten während der Bildsynthese.

Die folgende Auflistung gibt am Beispiel der OpenGL-Pipeline einen Überblick zu den einzelnen Phasen während der Bildsynthese:

1. Auswertung des OpenGL-Befehls: OpenGL-Befehle können einzeln nacheinander oder zusammengefasst in so genannten Display-Listen der Pipeline zugeführt werden. Display-Listen werden schneller verarbeitet und können wiederverwendet werden.

2. *Per-Vertex*-Operationen: Für jeden Knoten müssen die oben angeführten Transformationen, die Beleuchtungsberechnung sowie das Clipping durchgeführt werden.

3. Rasterisierung: Hier werden aus den geometrischen Primitiven und Bild-basierten Daten durch Scan-Konvertierung, Schattierungsberechnung und Texturierung Fragmente gebildet; diese bestehen aus Bildspeicheradressen und Informationen zu den Pixel-Attributen, wie zum Beispiel Farbe und Transparenz.

4. *Per-Fragment*-Operationen: Bevor die Fragmentdaten im Bildspeicher abgelegt werden, durchlaufen die einzelnen Pixel noch diverse Tests, dazu gehören der Alpha-, Stencil-, Tiefen- und Scissor-Test. Darüber hinaus erfordern Attribute wie Transparenz gegebenenfalls eine Modifikation der Fragmentdaten durch bereits bestehende Bilddaten aus dem Bildspeicher.

3.2.2 Hierarchische Datenstrukturen zur räumlichen Unterteilung

Eine Basistechnik im Bereich der Graphischen Datenverarbeitung ist die hierarchische Unterteilung des darzustellenden Modells. Durch eine derartige Aufbereitung der Daten können nicht nur Teile der Bildsynthese (Culling-Verfahren), sondern auch Interaktionen (Picking, Force-Feedback-Motion) oder die Lösung anderer häufig gestellter Aufgaben („Finde den nächsten Nachbarn“, „Welche Objekte liegen in diesem Bereich?“) stark beschleunigt werden. Die meisten Unterteilungsverfahren lassen sich einer der beiden folgenden Kategorien zuordnen:

- **Binary Space Partitioning (BSP-Tree)**
  Die grundlegenden Arbeiten zu den in der Graphik angewendeten BSP-Trees sind der *Point-Quadtree* [18] (Abbildung 3.4 a) und der *kd-Tree* [3] von Bentley. Um eine 3D-Szene mit Hilfe eines *kd-Tree*s zu unterteilen, wird folgendermaßen vorgegangen: In jedem Schritt wird der 3D-Raum durch eine Ebene in zwei Teile unterteilt. Jedes Objekt wird daraufhin einer der beiden Untermengen zugeordnet. Dieses Vorgehen wird rekursiv fortgesetzt, bis eine vorher spezifizierte Anzahl maximal zulässiger Primitive nicht mehr überschritten wird oder eine vorgegebene Baumtiefe erreicht ist.
  In dem ursprünglichen Ansatz von Bentley et al. [3] verlaufen die Hauptachsenorthogonalen Ebenen jeweils durch einen Punkt der zu unterteilenden Punktmenge; außerdem wechselt die Ausrichtung der Unterteilungsebene in jedem Iterationsschritt. Die Unterteilung wird so vorgenommen, dass sich auf beiden Seiten der Ebene gleich viele Punkte befinden. Das führt zu einer optimalen Balancierung und resultiert somit für eine Suche in einer Laufzeitkomplexität von $O(\log n)$ (Abbildung 3.4 b).
  Der „optimierte *kd-Tree*“ [20] legt die Trennebene mittig zwischen die in beiden Untermengen nächstliegenden Punkte. Dadurch werden die eigentlichen Punktdaten nun nur noch durch die Blattknoten des Baumes repräsentiert und befinden sich in so genannten *Buckets*, also Zellen, die durch verschiedene Ebenen definiert werden (Abbildung 3.4 c). Später wird die Permutation der Ebenenrichtung aufgehoben.
  Fuchs et al. [25] richten in dem *Binary Space Partitioning Tree* die Trennebenen nach den zu unterteilenden Objekten aus. Sie passen sich dadurch der Szene besser an, verbrauchen allerdings mehr Speicher, da sie die volle Ebenenbeschreibung in jedem Hierarchie-Knoten halten müssen (Abbildung 3.4 d).

- **Bounding Volume Hierarchy (BV-Tree)**
  Die Hüllvolumenhierarchie speichert Informationen über Objekte ab, die ihre enthaltene Geometrie komplett umgeben. Häufig verwendete Hüllvolumen sind Kugeln und achsenparallele (AABB) oder auch objektorientierte (OBB) Quader. Sie eignen sich besonders zum Einsatz für Szenengraphen und zur Kollisionsdetektion. Um beispielsweise eine AABB-Hierarchie aufzubauen, werden die beiden folgenden Schritte für jede Hierarchiestufe nacheinander durchgeführt:

  1. Minimalen Hüllquader finden, der das Objekt oder die Teilszene umgibt

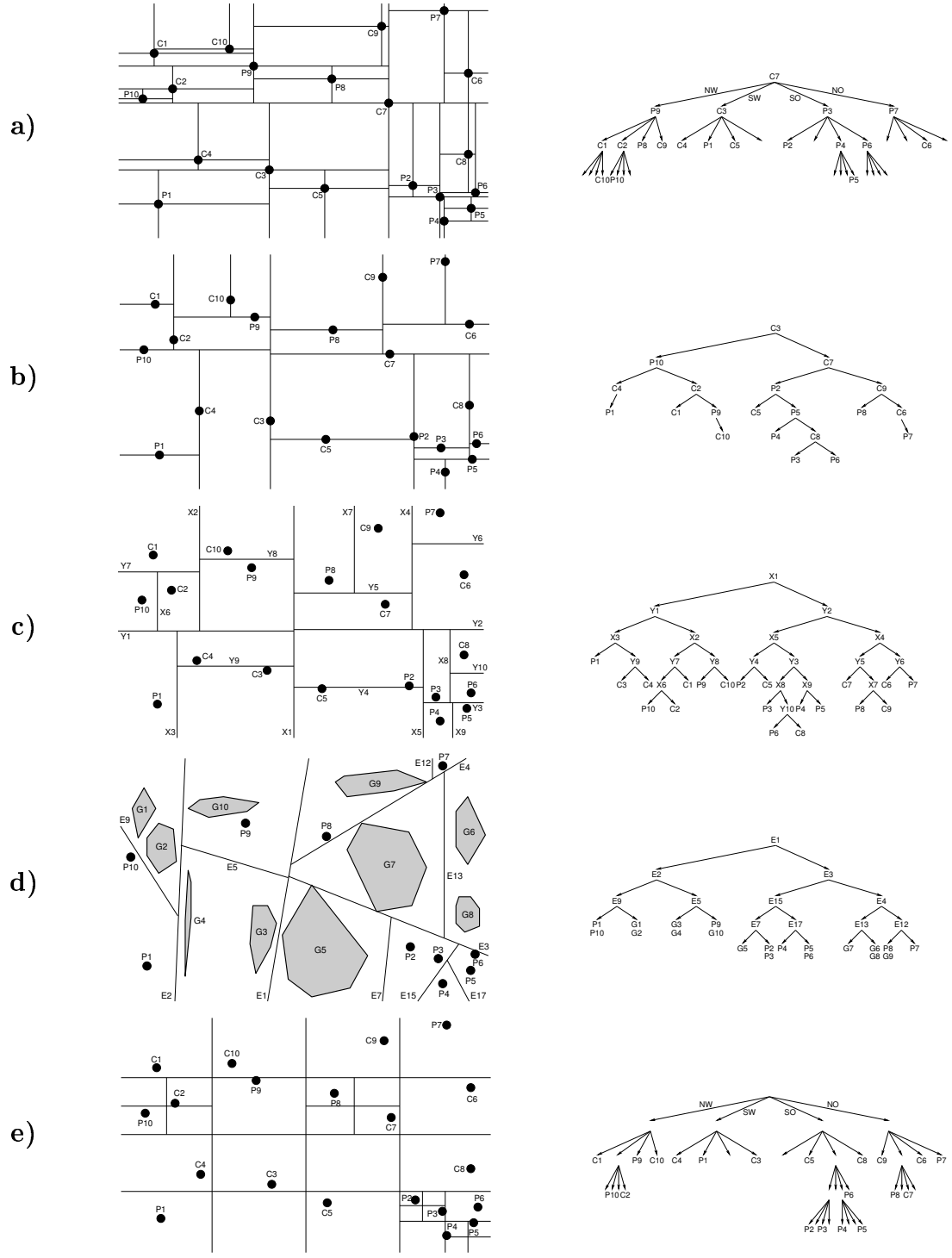


Abbildung 3.4: Beispiele hierarchischer Baumstrukturen: (a) *Point-Quadtree*, (b) *kd-Tree*, (c) Optimierter *kd-Tree*, (d) *BSP-Tree* und (e) *Region-Quadtree*.

2. Hüllquader entlang der längsten Achse unterteilen und die Objekte jeweils einem der beiden untergeordneten AABBs zuordnen.

Diese Vorgehensweise wird als *top-down* bezeichnet und verwendet als Abbruchkriterien die gleichen wie BSP-Trees. Bei vielen der im Rahmen dieser Arbeit entwickelten Berechnungs- und Visualisierungsmethoden ist die hierarchische Unterteilung der Finite-Element-Struktur in eine Hüllvolumenhierarchie unverzichtbar. Eine detaillierte Diskussion über die Arbeitsweise der eingesetzten Algorithmen zur Berechnung minimaler Abstände sowie der Detektion von Perforationen und den zu bevorzugenden Hüllvolumentyp findet sich in Abschnitt 7.1.1 ab Seite 108.

Der Raum im BSP-Tree wird nach dem Unterteilungsschritt in zwei disjunkte Volumenbereiche getrennt. Demgegenüber können sich die Volumenbereiche zweier Kindknoten in einem BV-Tree überlappen. Allerdings wird jedes Geometrieobjekt oder -primitiv nur einem der beiden Kindknoten zugeordnet.

Der *Octtree*, eine häufig im Zusammenhang mit Volumendaten verwendete Hierarchiestruktur, unterteilt den 3D-Raum auf die gleiche Weise wie der *Region-Quadtree* (Abbildung 3.4 e) im Zweidimensionalen: Ausgehend von einem achsenparallelen Hüllquader, der alles umschließt, wird dieser Raum in der Mitte durch drei ebenfalls achsenparallele Ebenen in acht kleinere Hüllquader zerlegt. Dieser Vorgang wird wiederum so lange wiederholt, bis die in einem Hüllvolumen enthaltene Information homogen ist, oder eine vorgegebene Unterteilungstiefe erreicht wird. Während die Unterteilung im *kd-Tree* in jedem Iterationsschritt jeweils nur durch eine Ebene vorgenommen wird und den Raum so unterteilt, dass beiden Seiten gleich viele Objekte zugeordnet werden, ist die Lage und Größe der Hüllquader beim *Octtree* durch regelmäßiges Teilen in der Mitte des zu unterteilenden Quaders festgelegt. Die optimale Balancierung von *kd-Tree*s führt bei einer Suche zu einer geringen Laufzeitkomplexität von $O(\log n)$, die im Allgemeinen weniger ausgewogene *Octtree*s nicht gewährleisten können. Auf der anderen Seite bieten *Quad-* und *Octtree*s durch ihre regelmäßige Struktur die Möglichkeit, effizient aus einem Hüllvolumen in ein angrenzendes zu wechseln – eine fundamental wichtige Eigenschaft für Aufgaben, auf die zum Beispiel Strahl- oder Partikelverfolgung zurückgreifen.

Darüber hinaus gibt es noch eine Reihe weiterer Ansätze, die sich bestimmten Problemstellungen widmen beziehungsweise die Nachteile der oben genannten Strukturen zu mindern versuchen. Einen guten Überblick geben Samet [59] sowie Gaede und Günther [26].

3.2.3 Einflussfaktoren bei der Bildsynthese

Das Nichterreichen der gewünschten Bildwiederholrate kann auf verschiedene Ursachen zurückzuführen sein. Wieviel Aufwand für die Berechnung eines Bildes von der Graphik-Hardware getrieben werden muss, hängt neben dem Haupteinflussfaktor, der Szenenkomplexität, von weiteren Faktoren ab, die im Folgenden aufgelistet werden.

Engpässe der Rendering-Pipeline

Bei Verwendung von Texturen stellt häufig eine hohe Bildauflösung bei beschränkter Rasterisierungsleistung einen Engpass für die Bildsynthese dar („rasterization bound“. Durch Herabsetzen der Bildauflösung während der Interaktion kann die Reaktionszeit der Anwendung verkürzt werden. Bei der Darstellung komplexer polygonaler Modelle handelt es sich allerdings in der Regel um eine zu große Menge von Geometriedaten, die von der Rendering-Pipeline verarbeitet werden müssen („geometry bound“). In diesem Fall ist es wichtig, die Weiterverarbeitung der Szene auf potenziell sichtbare 3D-Objekte zu beschränken und die Beschreibung der Geometriedaten für die Rendering-Pipeline zu optimieren. Schließlich beeinflusst die Anzahl der Lichtquellen, die Verarbeitungsreihenfolge von Objekten mit verschiedenen Attributen sowie die angewendeten Pixel-basierten Tests die Bilderstellungszeit.

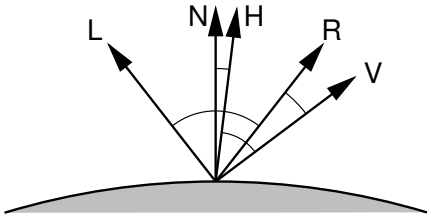
Beleuchtungsmodelle und Schattierungsverfahren

Neben Verdeckung und perspektivischer Projektion trägt auch die Beleuchtungsberechnung ganz entscheidend zur richtigen Wahrnehmung eines Bildes einer dreidimensionalen Szene bei. Das der Beleuchtungsberechnung am häufigsten zugrundeliegende Modell von Phong [51] setzt sich aus einem ambienten, einem diffusen und einem spekularen Anteil zusammen:

$$I_\lambda = I_{a\lambda}\, k_a O_{d\lambda} + \sum_{i=1}^{l} f_{AD_i}\, I_{L_i} \Big(k_d O_{d\lambda} \left(\vec{N} \cdot \vec{L_i}\right) + k_s O_{s\lambda} \left(\vec{R_i} \cdot \vec{V}\right)^n \Big) \tag{3.1}$$

Für die resultierende Farbe lassen sich die Intensitäten I_λ der RGB-Komponenten einzeln berechnen; λ steht hier stellvertretend für die jeweils betrachtete Farbkomponente. Der ambiente Term $I_{a\lambda}$ abstrahiert die Intensität des umgebenden Lichts, das auf indirektem Wege das Objekt erreicht, und das beeinflusst durch den ambienten Reflexionskoeffizienten k_a sowie die Objektfarbe $O_{d\lambda}$ von dort wieder abgestrahlt wird. Dieser ambiente Anteil sorgt unabhängig von der Ausrichtung des Lichts und des Objektes für eine Grundhelligkeit. Die Zusatzinformation für die Wahrnehmung des dreidimensionalen Objektes wird durch den diffusen und spekularen Anteil für jede der l Lichtquellen eingebracht. Die Intensität der i-ten Lichtquelle I_{L_i} wird zunächst durch einen Vorfaktor f_{AD_i} gemindert, der die atmosphärische Dämpfung approximiert. Des Weiteren gehen der diffuse Reflexionskoeffizient k_d und die Objektfarbe $O_{d\lambda}$ beeinflusst durch den Winkel zwischen der Oberflächen-Normale $\vec{N}$ und dem Richtungsvektor zur i-ten Lichtquelle $\vec{L_i}$ mit ein. Schließlich sorgt der spekulare Reflexionskoeffizient k_s zusammen mit der separat spezifizierbaren spekularen Objektfarbe $O_{s\lambda}$ in Abhängigkeit vom Winkel zwischen dem optimal reflektierten Lichtstrahl $\vec{R_i}$ und der Richtung $\vec{V}$, aus der das Objekt betrachtet wird, für zusätzliche Intensität durch gespiegeltes Licht (siehe Abbildung 3.5).

OpenGL bietet für die Beleuchtung einer polygonalen Szene drei Arten von Lichtquellen: *Directional*, *Point* und *Spot Lights*. Am häufigsten kommen *Directional Lights*, also gerichtete Lichtquellen zum Einsatz. Ausgehend vom Phong-Modell werden hier weitere



$\vec{L}$: Lichtvektor
$\vec{N}$: Normalenvektor am Objekt
$\vec{R}$: Optimaler Reflexionsvektor
$\vec{V}$: Vektor zum Betrachter
$\vec{H}$: *Halfway*-Vektor zwischen $\vec{L}$ und $\vec{V}$

Abbildung 3.5: Während das Beleuchtungsmodell von Phong die spekulare Reflexion in Abhängigkeit des Winkels zwischen dem optimal reflektierten Lichtstrahl $\vec{R}$ und der Betrachterrichtung $\vec{V}$ berücksichtigt, wird in OpenGL und Direct3D stattdessen die Abweichung des so genannten *Halfway*-Vektors $\vec{H} = \frac{\vec{L}+\vec{V}}{|\vec{L}+\vec{V}|}$ von der Normalen $\vec{N}$ herangezogen (Blinn-Phong-Modell [5]).

Vereinfachungen vorgenommen; anders als bei *Point* und *Spot Lights*, die in Abhängigkeit ihres Abstandes d_L zum beleuchteten Objekt die atmosphärische Dämpfung durch den Faktor

$$f_{AD} = \min\left(\frac{1}{A + Bd_L + Cd_L^2}, 1\right) \tag{3.2}$$

approximativ berücksichtigen, werden gerichtete Lichtquellen als im Unendlichen positionierte Lichter angenommen und der Term f_{AD} vernachlässigt. Des Weiteren wird der spekulare Reflexionsterm nicht mit dem potenzierten Kosinus des Winkels zwischen $\vec{R}$ und $\vec{V}$, sondern mit dem zwischen $\vec{H}$ und $\vec{N}$ multipliziert; sofern der Betrachter ebenfalls im Unendlichen angenommen wird, reduziert sich $\vec{H}$ zu einem konstanten Term.

Um die gesamte Szene basierend auf der oben erläuterten Beleuchtungsberechnung zu schattieren, gibt es drei Verfahren, die diese Berechnung pro Primitiv, pro Vertex oder pro Pixel durchführen und daher mit unterschiedlichem Rechenaufwand unterschiedliche Ergebnisse erzeugen:

- Das *Flat-* oder *Constant Shading* ordnet jedem Primitiv nur eine Flächennormale zu. Die Beleuchtungsberechnung wird nur einmal für das gesamte Primitiv durchgeführt; dadurch erscheint jeder Pixel des Primitivs in der gleichen Farbe. Vom Betrachter wird jedes Primitiv eines Polygonmodells als „flach“ wahrgenommen. Kanten zu benachbarten Primitiven mit abweichenden Normalen werden deutlich sichtbar.
- Beim *Gouraud-* oder *Smooth Shading* wird die Beleuchtungsberechnung an den Eckpunkten der Primitive vorgenommen. Die resultierenden Farbwerte an den Eckpunkten werden während der Rasterisierung für die anderen Pixel des Primitivs durch lineare Interpolation bestimmt. Die Übergänge zu benachbarten Primitiven sind weich und werden nicht als Kanten wahrgenommen, sofern gemeinsamen Knotenpunkten die gleichen Normalen zugeordnet werden.
- Beim *Phong-Shading* wird ausgehend von den Normalen in den Eckpunkten eines Primitivs durch Interpolation für jedes Pixel eine eigene Normale zur Beleuchtungsberechnung ermittelt. Lichtreflexionen werden dadurch sehr viel realistischer abgebildet

als beim Gouraud-Shading. Begrenzte spekulare Lichtreflexionen können somit auch innerhalb eines flachen Primitivs dargestellt werden.

Für die Schattierung von polygonalen Modellen werden von der Graphik-Hardware nur die beiden erstgenannten Verfahren direkt unterstützt (Abbildung 3.6). Phong-Shading-Reflexionseffekte können durch Verwendung von speziellen Texturen oder den Einsatz von Pixel-Shadern auf aktueller PC-Grafik-Hardware realisiert werden.

Darstellungsattribute

Neben Farbe und Reflexionseigenschaften werden zunehmend auch Texturen als Darstellungsattribut verwendet. Die im Vergleich zu den erstgenannten Attributen sehr aufwändige Texturierung hat sich durch die enorm gesteigerte Leistungsfähigkeit der Graphik-Hardware in den letzten Jahren zu einer Standard-Visualisierungstechnik entwickelt. Dabei werden den Primitiven eines geometrischen Objekts explizit Punkte im

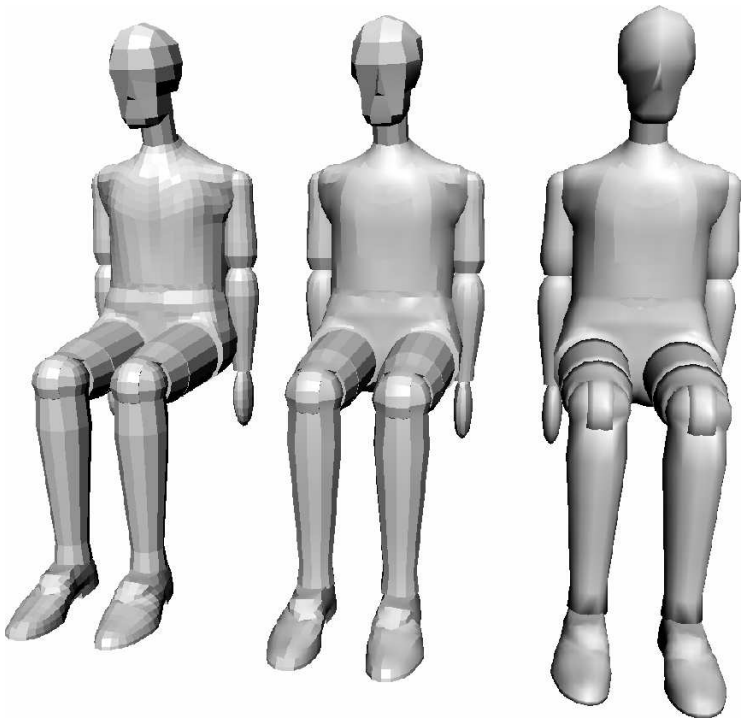


Abbildung 3.6: Während bei dem linken Dummy durch *Flat-Shading* sich die Finite-Elemente mit unterschiedlichen Normalen klar voneinander abgrenzen, verschwinden durch *Gouraud-Shading* (rechts) die inneren Kanten im gleichen Modell. Die Darstellung in der Mitte ist ebenfalls *Gouraud*-schattiert; allerdings wurden hier zuvor Elementkanten detektiert, an denen die Elementnormalen mehr als 15° voneinander abweichen, und den Knoten dieser Kanten mehr als eine Normale zugewiesen.

Texturbild zugeordnet. Innerhalb eines Primitivs werden diese Informationen während der Rasterisierung durch eine zuvor spezifizierte Interpolationsfunktion auf die Fragmente abgebildet. Der aus der Beleuchtungsberechnung hervorgegangene Farbwert des Objektes wird mit dem aus der zugehörigen Texturbildposition ermittelten Farbwert kombiniert oder durch ihn ersetzt. Dies geschieht in Abhängigkeit der gewählten Texturierungsfunktion (siehe auch Abschnitt 7.1.3). Schließlich wird der aus der Texturierung resultierende Farbwert in Abhängigkeit der Transparenz-Komponente nach dem Passieren der Pixelbasierten Tests mit der bereits bestehenden Pixelfarbe im *Compositing*-Schritt verrechnet.

Sofern die Texturierung nicht von der Graphik-Hardware unterstützt wird, können Szenen mit Texturen kaum mit Bildwiederholraten im interaktiven Bereich dargestellt werden. Inzwischen sind allerdings PC-Graphik-Karten im unteren Preisbereich schon in der Lage, mehr als 100 Millionen beleuchtete und texturierte Dreiecke pro Sekunde[2] darzustellen. Als reines Darstellungsattribut eignen sich Texturen zur Imitation der Oberflächenbeschaffenheit (zum Beispiel Holzmaserung). Darüber hinaus lassen sich durch Techniken wie *Bump Mapping* [6, 50] mit Hilfe von Texturen auch ohne zusätzliche geometrische Komplexität Unebenheiten simulieren. Abschnitt 6.3 zeigt, wie eindimensionale Texturen zur Visualisierung von skalaren Größen eingesetzt werden.

Culling-Techniken

Für Szenen mit hoher Komplexität, aus denen in einer interaktiven Applikation mehrere Bilder pro Sekunde erzeugt werden sollen, müssen Verfahren auf Modellebene dafür sorgen, dass nur potenziell sichtbare Objekte von der Rendering-Pipeline verarbeitet werden. Diese Vorverarbeitungsschritte auf Objekt-Ebene werden als *Culling* (englisch für „Aussortierung“) bezeichnet und sortieren alle Objekte aus, die nicht zum resultierenden Bild beitragen können.

- Das **View Frustum Culling** greift dem Clipping der Primitive an den Ebenen des Sichtvolumen auf Basis ganzer Objekte vor: Falls das Hüllvolumen eines Objektes nach der Transformation in das View-Koordinatensystem außerhalb des Sichtvolumens liegt, kann ausgeschlossen werden, dass Teile des Objektes sichtbar sind. Wie stark das View Frustum Culling die Bildsynthese beschleunigt, hängt neben der Granularität der Szene vor allem von der Position und Orientierung der Kamera ab. Der größte Effekt wird erzielt, sobald der Betrachter in die Szene eintaucht. Für Polygonnetze, die sich über die gesamte Szene erstrecken, ist eine im Zusammenhang mit OpenGL Optimizer als *Spatializing* bekannte Unterteilung in mehrere Teilnetze sinnvoll.

- Das **Portal Culling** findet vor allem in Spielen Anwendung, in denen sich der Spieler durch ein System von aneinandergrenzenden Räumen bewegt. Zunächst wird der Raum gezeichnet, in dem sich der Betrachter befindet. Anschließend wird für jedes sichtbare Loch in der den Raum begrenzenden Wand (zum Beispiel Fenster

[2] Stand Ende 2002

oder Türen) das View Frustum Culling entsprechend eingeschränkt. Mit Hilfe eines Graphen, der die Nachbarschaftsinformation der Szene speichert, wird dann der nächstliegende Raum gezeichnet. Die Bilderstellung wird beendet, sobald keine weiteren Räume mehr sichtbar sind oder die zur Verfügung stehende Zeit für das aktuelle Bild abgelaufen ist.

- Beim **Occlusion Culling** werden Objekte frühzeitig aussortiert, die durch andere, näher am Betrachter gelegene Objekte verdeckt werden. Occlusion Culling setzt eine Betrachter-abhängige Verarbeitungsreihenfolge der Objekte voraus, in der die nächstgelegenen Objekte vor weiter entfernten dargestellt werden, und basiert auf einer im Vergleich zum View Frustum Culling sehr aufwändigen Verdeckungsberechnung. Folglich profitiert das Rendering, wenn wenig großflächige Primitive viele Primitive anderer Objekte verdecken. Geräte-Modelle, deren Gehäuse ein komplexes Innenleben verdeckt, oder Spiele, in denen das Sichtfeld durch Raumwände begrenzt ist, sind Beispiele für Szenen, in denen Occlusion Culling zu einer starken Beschleunigung führen kann.

- Das **Detail Culling** unterschlägt Objekte, die aufgrund ihrer Darstellungsgröße kaum Informationen zu dem resultierenden Bild beitragen. Dazu wird die Größe des Hüllvolumens eines Objektes in Relation zum Sichtvolumen beziehungsweise der Entfernung zum Betrachter gesetzt. Sofern dieser Quotient unter dem vorgegebenen Schwellwert liegt, wird das Objekt als „unwichtig“ aussortiert.

Optimierung der Geometriedaten

Bei der Visualisierung großer Modelle bestehend aus hunderttausenden von Dreiecken ergeben sich zwei limitierende Faktoren: zum einen die Bandbreite des Datenbus und zum anderen die Rechenkapazität des Prozessors, der die Koordinatentransformationen für die Netzknoten durchführt. Deshalb spielt die optimale Aufbereitung der polygonalen Geometriedaten in Ergänzung zu den bereits genannten Filtermethoden für die Weiterverarbeitung in der Rendering-Pipeline eine bedeutende Rolle. Durch das Zusammenfassen benachbarter Primitive wird die Topologie eines Polygonnetzes in komprimierter Form beschrieben, so dass ein auf dem Graphik-Chip integrierter Zwischenspeicher auf die Resultate bereits vorangegangener Transformationsberechnungen zurückgreifen kann.

Die am häufigsten eingesetzte Technik beim Optimieren der Netzbeschreibung ist das Bilden von Dreiecksstreifen. Eine Modellbeschreibung, die sich aus beliebigen Polygonen zusammensetzt, muss dazu zunächst trianguliert werden. Anschließend werden benachbarte Dreiecke in Streifen zusammengefasst. Ausgehend von zwei Punkten, die eine Kante des Start-Dreiecks bilden, wird durch Hinzufügen jedes weiteren Punktes P_i das i-te Dreieck $P_{i-2}P_{i-1}P_i$ des Dreiecksstreifens definiert (Abbildung 3.7). Folglich werden für einen Streifen bestehend aus n benachbarten Dreiecken statt $3n$ lediglich $n+2$ Vertizes benötigt. Für ein Netz aus n Dreiecken, dass durch s Dreiecksstreifen repräsentiert werden kann, müssen also lediglich $n + 2s$ Punkte in der Rendering-Pipeline verarbeitet werden.

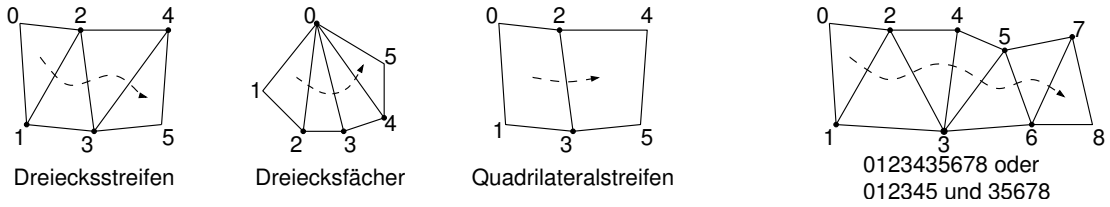


Abbildung 3.7: Bei der Verwendung von Dreiecksstreifen, -fächern oder Quadrilateralstreifen verringert sich die Anzahl der zu verarbeitenden Punktkoordinaten, da die hervorgehobenen Punkte für die Definition mehrerer Primitive herangezogen werden können. Das rechte Bild zeigt, dass es günstiger ist, einen Dreiecksstreifen durch einen Kantenrichtungswechsel fortzusetzen als zwei separate Dreiecksstreifen zu generieren.

Bei der Konvertierung eines Dreiecksnetzes in ein Netz von Dreiecksstreifen wird das Optimum erreicht, wenn die Anzahl der Referenzen von Netzknoten, die die Streifen definieren, minimiert werden kann. Ziel ist es also, möglichst wenige lange Streifen zu bilden. Es wurden bereits mehrere Ansätze zur Pfadsuche in Dreiecksnetzen publiziert: Eine frühe Implementierung [1] nutzte den *swap*-Befehl von IrisGL, einem Vorgänger von OpenGL aus, um einen Pfad auch dann fortsetzen zu können, wenn die Netzstruktur einen Kantenrichtungswechsel erforderte. Unter OpenGL wird die Unterbrechung eines Dreiecksstreifens durch Wiederholen des vorletzten Netzknotens und damit durch Erzeugen von „leeren" Dreiecken umgangen (vergleiche Abbildung 3.7 rechts).

In [17] stellen die Autoren verschiedene Strategien zur Optimierung von Dreiecksstreifen vor, die insbesondere für Polygonnetze mit vielen Quadrilateralen gute Ergebnisse liefern. Die Vorteile, die sich aus einer zusätzlichen globalen Suche nach Streifen maximaler Länge ergeben, werden in Kapitel 6.1 detaillierter diskutiert. Deering [13] schlägt vor, die gemeinsam verwendeten Punkte benachbarter Streifen durch Vergrößerung des Vertex-Cache wiederverwenden zu können. Chow [11] zeigt, wie ein Dreiecksnetz partitioniert werden muss, um den Vertex-Cache möglichst gut auszunutzen. Beide Ansätze komprimieren die zu verarbeitenden Daten, wodurch das Datenvolumen auf $\frac{1}{6}$ bis $\frac{1}{8}$ reduziert werden kann. Hoppe [35] schlägt ein Verfahren vor, das die Reihenfolge, in der die Primitive gezeichnet werden, umordnet und auch ohne Datenkompression durch Optimierung der Lokalität gegenüber einem herkömmlichen Dreiecksstreifen zu einer Entlastung des Bus um den Faktor 1.6 bis 1.9 kommt.

Simplifizierung polygonaler Modelle

Während das Zusammenfassen benachbarter Primitive zu Streifen die Rendering-Pipeline von Transformationslast befreit, indem die inneren Knoten eines Streifens nur einmal verarbeitet werden müssen, besteht in der Simplifizierung der Geometrie eine weitere Möglichkeit, Netzknoten einzusparen. Dabei wird die Geometrie am Ende der Simplifizierung durch weniger Polygone approximiert dargestellt, das heißt, dass die Simplifizierung

im Gegensatz zur Streifengenerierung in der Regel mit einem Qualitätsverlust behaftet ist.

In den vergangenen zehn Jahren wurde eine Vielzahl von Verfahren im Bereich der Polygonreduktion entwickelt. Beim *Re-tiling* von Turk [70] werden zunächst zusätzliche Punkte auf dem Polygonnetz bevorzugt in Regionen mit großer Krümmung platziert. Anschließend wird das Netz durch Einbeziehung der neuen Netzknoten trianguliert, bevor schließlich die ursprünglichen Netzknoten nacheinander entfernt werden. Beim *Vertex Clustering* [58] wird ein im Verhältnis zum Dreiecksnetz gröberes 3D-Gitter über das Netz gelegt und alle Netzknoten innerhalb einer Zelle in einer Position zusammengefasst; anschließend werden degenerierte Primitive sowie doppelte Knoten entfernt. Dieses Vorgehen ist zwar sehr schnell, erzeugt aber ebenfalls ein völlig neues Netz.

Ein anderer Ansatz wird von Schroeder et al. [60] gewählt: Nach der Klassifizierung eines Netzknotens durch seine topologischen und geometrischen Eigenschaften wird dieser bei Einhaltung eines lokalen Distanzkriteriums gelöscht. Durch Triangulierung der Nachbarknoten wird das entstandene Loch wieder verschlossen. Somit reduziert sich die Anzahl der Dreiecke für jeden Randknoten um eins und für innere Netzknoten um zwei. Basierend auf diesem Ansatz hat es zahlreiche Weiterentwicklungen gegeben, in denen durch Zusammenfallenlassen von Kanten oder Dreiecken sukzessiv das Dreiecksnetz ausgedünnt wird. Dabei werden mögliche Kandidaten für den nächsten Reduktionsschritt mit Hilfe einer Kostenfunktion sortiert, so dass dadurch der Fehler für jede Iteration minimiert wird. Klein et al. [39] verwendeten als globales Distanzkriterium den zweiseitigen Hausdorff-Abstand. Campagna [10] beschränkt sich auf die Berechnung des leichter zu berechnenden einseitigen Hausdorff-Abstandes (Abbildung 3.8). In [27] werden in der Kostenfunktion für die nächste auszuführende Kantenkontraktion zusätzliche Attribute der Netzknoten, wie zum Beispiel Farbwerte, Texturkoordinaten oder andere Randbedingungen berücksichtigt.

3.2.4 High-Level-3D-Graphikbibliotheken

Eine 3D-Graphikbibliothek wird von der anwendungsspezifischen Software als Schnittstelle zur Graphik-Hardware genutzt, um aus dreidimensionalen Szenenbeschreibungen ein zweidimensionales Bild zu erzeugen. Sie abstrahiert und ergänzt gegebenenfalls die Fähigkeiten der Hardware-seitig zur Verfügung gestellten Funktionalität. Die am meisten verwendeten 3D-Graphikbibliotheken sind unter Windows Microsofts Direct3D und ansonsten OpenGL, das 1992 aus dem SGI-proprietären IrisGL hervorging und inzwischen einen Plattformübergreifenden Industriestandard für Graphik-Software darstellt. Bei beiden handelt es sich um „Low-Level-Bibliotheken“, die dem Programmierer zwar alle Möglichkeiten bieten, in die Verarbeitungsschritte der Rendering-Pipeline einzugreifen, auf der anderen Seite jedoch keine Datenstrukturen zur Verfügung stellen, die eine einfache Beschreibung der Szene ermöglichen.

Im Gegensatz dazu wird die Erstellung einer Graphik-Anwendung von „High-Level-3D-Graphikbibliotheken“ durch komplexe Funktionalitäten mit einer einfachen Programmierschnittstelle unterstützt. Der Vergleich von High- zu Low-Level-Graphikprogrammierung ähnelt dem Vergleich von Hochsprachen- zu Assembler-Programmierung. Der Wandel von

Sei $\mathcal{P}$ die Menge aller Netzknoten eines Polygonnetzes $\mathcal{M}$ und bezeichne $\mathcal{S}$ die Menge aller Punkte auf der Oberfläche von $\mathcal{M}$, also $\mathcal{P} \subset \mathcal{S}$. Ferner ergebe $d(p, \mathcal{S}')$ den Abstand eines Punktes $p \in \mathcal{S}$ zur Fläche $\mathcal{S}'$ mit

$$d(p, \mathcal{S}') = \min_{p' \in \mathcal{S}'} \|p - p'\|_2,$$

wobei $\|\cdot\|_2$ für die Euklidische Norm steht. Dann ergibt sich der (einseitige) Hausdorff-Abstand $d(\mathcal{S}, \mathcal{S}')$ von der Fläche $\mathcal{S}$ zur Fläche $\mathcal{S}'$ als

$$d(\mathcal{S}, \mathcal{S}') = \max_{p \in \mathcal{S}} d(p, \mathcal{S}').$$

Zu beachten ist, dass $d(\mathcal{S}, \mathcal{S}') \neq d(\mathcal{S}', \mathcal{S})$ gilt und somit $d(\mathcal{S}, \mathcal{S}')$ nicht symmetrisch ist; vielmehr identifiziert $d(\mathcal{S}, \mathcal{S}')$ einen Punkt p auf der Fläche $\mathcal{S}$, der von allen Punkten p' der Fläche $\mathcal{S}'$ am weitesten entfernt ist. Der symmetrische oder auch zweiseitige Hausdorff-Abstand $d_s(\mathcal{S}, \mathcal{S}')$ gibt durch seine Definition als

$$d_s(\mathcal{S}, \mathcal{S}') = \max \left[d(\mathcal{S}, \mathcal{S}'), d(\mathcal{S}', \mathcal{S}) \right]$$

ein Maß für die maximale Abweichung zweier Polygonnetze an.

Abbildung 3.8: Definition des ein- und zweiseitigen Hausdorff-Abstandes

statischen zu dynamischen, eventuell animierten Szenen, in denen interaktiv navigiert werden soll, machte die Entwicklung von High-Level-Graphikbibliotheken notwendig. Wichtige Charakteristiken sind

- Szenengraph: Datenstruktur, die einen gerichteten azyklischen Graphen repräsentiert und es erlaubt, die 3D-Szene hierarchisch zu unterteilen.
- Traversierung: Objekte oder Mechanismen stehen zur Verfügung, die beim Durchlaufen des Szenengraphen Operationen auf den Datenstrukturen ausführen (zum Beispiel: „zeichne dich“).
- Sharing: Teile der Szenenbeschreibung können an anderen Stellen des Szenengraphen wiederverwendet werden; das minimiert den Speicherbedarf und sichert die Datenkonsistenz bei gleichartigen Szenenobjekten.
- Bounding-Volume-Hierarchie: Durch an den Szenengraphknoten integrierte Hüllvolumendaten kann die durch den Szenengraphaufbau vorgegebene Hierarchie dazu genutzt werden, Culling und Picking effizient durchzuführen.
- Dateiformate: Ein meist Bibliotheks-internes Dateiformat erlaubt es, die Szenenbeschreibung persistent abzuspeichern. Des Weiteren stellen viele Bibliotheken Hilfsfunktionen zur Verfügung, die das Einlesen weitverbreiteter Bilddatenformate zur Definition

von Bild-Texturen unterstützen.

Jede der im Folgenden vorgestellten „High-Level-Bibliotheken“ bietet zusätzlich zu den oben genannten Eigenschaften Unterstützung für die Lösung spezieller Probleme:

Open Inventor ist die wohl bekannteste Szenengraph-Bibliothek und bietet ein Dateiformat, das inzwischen für inkompatible Szenenbeschreibungen häufig als Austauschformat Verwendung findet. Der integrierte Satz an 3D-Manipulator-Objekten stellt im Vergleich zu anderen Graphik-Programmierschnittstellen eine große Besonderheit dar; mit den Manipulatoren lassen sich auf intuitive Weise Teile der Szene mit Hilfe der 2D-Maus im Raum manipulieren. Open Inventor steht seit 2000 als Open Source zur Verfügung und bietet eine sauber strukturierte und erweiterbare Programmierschnittstelle. Allerdings eignet sich Open Inventor aufgrund der Szenengraph-Semantik nicht für Parallelisierung und ist somit für komplexe Szenen nur beschränkt einsetzbar. [73, 74, 75]

OpenGL Performer wird von SGI entwickelt und ist außer für Irix inzwischen auch für Linux und Windows erhältlich. Verglichen mit den anderen Bibliotheken liegt hier der Schwerpunkt — wie der Name schon sagt — auf Performanz bei der Bildsynthese. Volle Multiprocessing-Unterstützung, Lastausgleich, asynchrones Ein-/Auslagern von Szenendaten sowie das Einhalten einer vorgegebenen Bildwiederholrate sind nur einige Eigenschaften von OpenGL Performer. Damit eignet es sich vor allem für den Einsatz in VR-Umgebungen (zum Beispiel CAVE, Powerwall, Workbench, HMD), in denen diese Vorzüge voll ausgeschöpft werden können. [37]

Java3D ist eine Szenengraphbibliothek, deren Entwicklung 1995 durch Sun, SGI, Apple und Intel begonnen wurde. Die erste Spezifikation wurde allerdings erst 1997 publiziert, nachdem sich SGI bereits wieder aus dem Entwicklungskonsortium zurückgezogen hatte. Java3D wurde ursprünglich zur Unterstützung Plattform-übergreifender Anwendungen entwickelt und ist derzeit außer unter Solaris auch für Irix, AIX, HP-UX, Linux und Windows verfügbar. Aus dieser Zielsetzung resultiert, dass Hardware-spezifische Schnittstellen zur Performanzsteigerung von Java3D nicht genutzt werden. Daher wird Java3D hauptsächlich für die Darstellung von Szenen geringerer bis mittlerer Komplexität eingesetzt. [66]

Cosmo3D / OpenGL Optimizer sollte ursprünglich nur eine Übergangslösung für eine nachfolgende Plattform-übergreifende Graphikbibliothek[3] sein (siehe 4.1) und adressiert die Probleme von CAD/CAM-Anwendungen. Der integrierte Tesselierer ermöglicht die Umwandlung von NURBS-Flächenbeschreibungen in Dreiecksnetze. Darüber hinaus werden durch Occlusion-Culling-, Simplifizierungs- und Streifengenerierungsfunktionalität die Probleme bei der Visualisierung sehr großer polygonaler Modelle adressiert. [36]

[3]Die Entwicklung von OpenGL++ als zukunftsweisende Szenengraphbibliothek ist über die Spezifikationsphase nicht hinaus gekommen.

Fahrenheit wurde infolge des gescheiterten OpenGL++ Ende 1997 von den Firmen Microsoft und SGI zusammen als zukünftiger High-Level-Graphikbibliotheken-Standard angepriesen, der die guten Eigenschaften vom intuitiven Open Inventor und dem schnellen OpenGL Performer vereinen und Cosmo3D / OpenGL Optimizer ablösen sollte. Die Zusammenarbeit an dem in drei Schichten geplanten Fahrenheit-Projekt kündigte SGI 1999 auf.

OpenSG wurde 1999 am Fraunhofer Institut für graphische Datenverarbeitung auch mit Unterstützung deutscher Automobilhersteller aus der Taufe gehoben. Die langjährigen Ankündigungen verschiedener Firmen zu neuen Szenengraphbibliotheken, deren Entwicklung teilweise schon vor einer ersten Implementierung wieder eingestellt wurden, motivierten zehn deutsche Forschungsinstitute beziehungsweise Universitäten, sich seit 2001 im Rahmen des BMBF[4]-Projektes „OpenSG PLUS“ an der Weiterentwicklung der bis dahin bestehenden Basisschnittstelle zu beteiligen. Vorrangige Ziele sind die Multi-Thread/Multi-Pipe-Unterstützung für VR-Umgebungen, aber auch Portabilität auf verschiedene Plattformen, um eine große Verbreitung und damit die Weiterentwicklung im Open-Source-Umfeld zu sichern. Ähnlich, wie für Fahrenheit geplant, setzt sich diese Szenengraphbibliothek aus mehreren Modulen zusammen, die hohe Flexibilität und Erweiterbarkeit gewährleisten sollen.

Open Scenegraph wird, aus der gleichen Motivation wie OpenSG, als Open-Source-Projekt allerdings mit einer nicht ganz so VR-lastigen Zielsetzung wie OpenSG ebenfalls im Hinblick auf hohe Performanz und Portabilität entwickelt.

VRML stellt an sich keine Graphikbibliothek dar, sondern dient als ***V**irtual **R**eality **M**odeling **L**anguage* lediglich der Beschreibung einer 3D-Szene. VRML 1 stellt im wesentlichen das Open Inventor Dateiformat 2.0 dar. VRML 2 bietet zusätzlich Animation, Sound, Keyframes und dergleichen mehr. Der 1997 erreichte ISO/IEC-Standard wird als VRML97 bezeichnet und diente als Basis der XML-basierten Beschreibungssprache „X3D“[5], die vom *web3D*-Konsortium entwickelt wird. Die Stärken von VRML liegen in der Portabilität, die Schwächen in der durch die Inventor-ähnliche Szenengraph-Semantik erreichbaren Performanz. Es kommt vor allem im Zusammenhang mit interaktiven 3D-Welten zum Einsatz, in denen mit Hilfe eines Web-Browsers mit einem entsprechenden Plugin navigiert werden kann.

Die oben aufgeführten Szenengraphbibliotheken basieren alle auf OpenGL; lediglich Java3D für Windows basiert auf DirectX. Zusätzlich zu den genannten Szenengraphbibliotheken gibt es noch viele hier nicht näher beschriebene, die meist als kleineres Open-Source-Projekt oder als Weiterentwicklung industriell begonnener Arbeiten existieren; dazu gehören auch Gizmo3D, Jupiter, Oops3D, OpenRM und Quesa.

[4] BMBF — Bundesministerium für Bildung und Forschung
[5] X3D — Extensible 3D

3.2.5 Darstellung polygonaler Daten

Bei der Darstellung des Fahrzeugmodells geht es darum, eine für das Einsatzfeld hinreichend genaue Repräsentation zu finden. In der Designphase kommt es zum Beispiel darauf an, eine möglichst genaue und realistische Abbildung vom zukünftigen Fahrzeugmodell zu erzeugen; dies beinhaltet eine glänzende, glatte Außenhaut, in der sich möglichst noch Objekte aus der virtuellen Umgebung widerspiegeln, sowie deutlich sichtbare Details (Kühlergrill, Blenden, Firmenemblem). Hier kommen hochaufgelöste Texturen zum Einsatz, um ein hohes Maß an Fotorealismus zu erreichen. Sofern die Fahrzeuggeometrie bereits in einem CAD-Modell beschrieben ist, werden aus den CAD-Daten in einem Vernetzungsschritt meist sehr feine Dreiecksnetze generiert, die zwar noch mit den oben beschriebenen Mitteln für die Weiterverarbeitung in der Rendering-Pipeline optimiert werden kann, jedoch stets als einschränkende Grenze die Nähe des Fahrzeugmodells zur Realität hat.

Demgegenüber wird im Simulationsumfeld der Fahrzeugentwicklung Wert auf Genauigkeit und Performanz gelegt; das heißt, das dargestellte Modell muss stets den zugrundeliegenden Daten entsprechen und sollte schnell vom Graphiksubsystem verarbeitet werden können, um eine interaktive Visualisierung der Daten zu ermöglichen. Bei der Wahl der Beleuchtungsparameter steht im Vordergrund, dass die visualisierten Werte unverfälscht widergegeben werden. Daher sollte zum Beispiel während einer Skalarvisualisierung durch Farben auf Lichtreflexion auf der Oberfläche zugunsten der Aussagekraft verzichtet werden.

Während an virtuellen Designmodellen für das fotorealistische Rendering in der Regel wochenlang herumgefeilt wird und einzelne Koordinaten teilweise „von Hand“ modifiziert werden, um das Ergebnis zu optimieren und Entscheidungsträger zu überzeugen, ist der entscheidende Faktor in der Datenvisualisierung die notwendige Aufbereitungszeit der zugrundeliegenden Daten. Daher können hier lediglich automatische Optimierungsverfahren zum Einsatz kommen, die keiner weiteren Interaktion bedürfen.

3.3 Visualisierungsmethoden

Die Visualisierung hat zum Ziel, die zugrundeliegenden Daten in visuelle Information so umzusetzen, dass der Betrachter hinsichtlich seiner Fragestellung möglichst schnell zu einer Aussage kommt. Um die richtige Visualisierungsmethode zu wählen, müssen zuvor verschiedene Aspekte betrachtet werden. Eine große Rolle spielt die Auswahl der visualisierten Daten. Bei einer statischen Visualisierung in Form eines Bildes mag es darauf ankommen, möglichst viele Aspekte der zugrundeliegenden Daten in einem Bild unterzubringen. Werden jedoch zu viele Parameter eines multivariaten Datensatzes in einem Bild untergebracht, ist es dem Betrachter nicht mehr möglich, die visuelle Information zu dekodieren, um die gestellte Frage zu beantworten. Um die Interpretation der Daten zu erleichtern, sollten lediglich die Größen visualisiert werden, die im Zusammenhang betrachtet werden müssen, um zu einer Aussage zu kommen. Der Datentyp (Skalar, Vektor, Tensor) schränkt den Kreis der möglichen Repräsentanten (Farbe, Piktogramme, Glyphen) ein. Anders als bei der statistischen Visualisierung multivariater Daten, in der die Position eines Glyphen

durch die Lage und Einteilung der Diagrammachsen impliziert wird, geht es im Rahmen dieser Arbeit ausschließlich um die Visualisierung ortsgebundener Daten. Datenträger ist stets ein Element des Finite-Element-Modells. Daher werden einzelne Skalare meist durch Farbkodierung direkt auf der Fahrzeuggeometrie visualisiert. Für Vektorgrößen bietet sich eine Darstellung in Form von Vektorpfeilen an, die jedoch bei hoher Datendichte mit dem Nachteil der Verdeckung behaftet ist (siehe Kapitel 8.2). Gegebenenfalls lassen sich die vektoriellen Größen auf Skalare reduzieren, wenn zum Beispiel die Beschleunigung oder Kraft in eine vorgegebene Richtung von Interesse ist. Komplexe Glyphen als zusätzliche Geometrie dienen beispielsweise der Visualisierung von Trägheitstensoren oder anderen höherdimensionaler Daten. Bei der Visualisierung von Simulationsergebnissen bekommen die Daten durch ihre Zeitabhängigkeit eine weitere Dimension. Zeitabhängige Daten lassen sich am besten durch Animation von Einzelbildern wiedergeben.

Ein weiterer wichtiger Aspekt ist das Ausgabemedium. Eine interaktive Visualisierungsapplikation bietet den Vorteil, dass der Betrachter mit den Daten interagieren kann und durch die dynamische Veränderung der Bilder mehr Informationen erhält als durch eine statische Darstellung. Insbesondere bei der Darstellung semitransparenter Volumina, aber auch schon bei komplexen Polygonmodellen erhält der Betrachter durch bewegte Bilder mehr Tiefeninformation. Darüber hinaus kann eine interaktive Applikation den Anwender bei der Suche nach bestimmten Merkmalen in den Daten unterstützen; zum Beispiel bei der Fragestellung: „In welchem Bereich überschreitet der visualisierte Parameter den vorgegebenen Schwellwert?“ Durch Visualisierung allein bleiben dem Betrachter diese kritischen Regionen in komplexen Modellen gegebenenfalls verborgen.

Visualisierung im Bereich der Crash-Simulation

Derzeit kommen in den Entwicklungsabteilungen der Automobilhersteller sowie ihrer Zulieferer eine ganze Reihe verschiedener Anwendungen zum Einsatz. Im Bereich der Crash-Simulation sind das neben dem Solver nicht nur interaktive Visualisierungsapplikationen für das Pre- und Postprocessing, sondern auch verschiedene Batch-Programme, die der Datenaufbereitung oder -konvertierung dienen. Ende der neunziger Jahre kamen besonders im Postprocessing Visualisierungsapplikationen zum Einsatz, die der Hersteller des Simulationsprogramms anbot. Das lag unter anderem daran, dass das binäre Dateiformat, in dem der Solver die Ergebnisdaten abspeichert, nicht dokumentiert war. Allerdings hatte das zur Folge, dass die implementierten Algorithmen nicht mehr dem sich schnell entwickelndem Stand der Technik in der Computergraphik und Visualisierung entsprachen. Zudem setzten die historisch gewachsenen Programme hohe Hürden für die Integration neuer Techniken. Die stetig wachsenden Modellgrößen ließen sich schon bald nicht mehr mit diesen Solver-nahen Visualisierungsapplikationen interaktiv darstellen, da dort weder spezialisierte Szenengraph-Bibliotheken noch andere Optimierungstechniken zur Steigerung der Rendering-Performanz zum Einsatz kamen. Bei den verfügbaren Preprozessoren wurden die Entwicklungsschwerpunkte auf die Abdeckung der für die Vorverarbeitung notwendigen Funktionalität gesetzt, weniger auf Darstellungsgeschwindigkeit und moderne

Interaktionskonzepte.

Die Forschung im Bereich der Computergraphik bot zu diesem Zeitpunkt bereits eine Vielzahl von Methoden zur Aufbereitung und Optimierung polygonaler Modelle an. Neben den in Abschnitt 3.2.3 genannten Möglichkeiten, die Darstellungsgeschwindigkeit zu erhöhen, wurden im akademischen Bereich bereits mit Szenengraph-Bibliotheken, die vom Einsatz verschiedener Detailstufen („Level of Detail"), dem View Frustum Culling und beschleunigten Picking-Mechanismen profitierten, positive Erfahrungen gesammelt. Darüber hinaus brachte der Einsatz Hardware-unterstützter Texturierung im Hinblick auf gesteigerte Darstellungsqualität verbunden mit einer schnellen Bildsynthese völlig neue Möglichkeiten mit sich. So konnten beispielsweise bei der Visualisierung Knotenbasierter Skalare durch Farben mit Hilfe von Texturen die Interpolationsberechnung der Farbübergänge auf die Graphik-Hardware ausgelagert werden.

Doch all diese Techniken kamen bis dahin in kommerziell verfügbaren Applikationen nicht zum Einsatz. Auf der anderen Seite gab es in der wissenschaftlichen Forschung im Bereich der Visualisierung nur wenige Vorarbeiten, die sich speziell mit den Bedürfnissen im Umfeld der Computer-gestützten Entwicklung durch Strukturmechaniksimulation auseinandersetzte.

3.4 Datenformate und -strukturen

Im Umfeld der Strukturmechaniksimulation in der Fahrzeugentwicklung werden Daten in vielen verschiedenen Formaten verarbeitet. Zum einen bringen das die unterschiedlich zugrunde liegenden Modelle mit sich, in denen das Fahrzeug über die verschiedenen Entwicklungsphasen repräsentiert wird. Zum anderen speichern die auf dem Markt verfügbaren Produkte die Daten meist in einem proprietären Format ab. Als Beispiele seien hier nur einige genannt: Das Produkt *CATIA* der Firma *Dassault Systemes* ist das derzeit am weitesten verbreitete Konstruktionsprogramm im Automobilsektor und speichert die Konstruktionsdaten, ähnlich wie *Pro/ENGINEER* [53], *I-DEAS*, *SolidEdge* oder andere in einem eigenen Format ab. Es hat viele Bestrebungen gegeben, ein produktunabhängiges Format zu schaffen: IGES (*Initial Graphics Exchange Specification* [48]), STEP (*Standard for the Exchange of Product Model Data* [38]), VDA-FS (*Verband der Automobilindustrie Flächen Schnittstelle* [71]) und OpenDWG [49]. Dennoch nutzen Applikationen, die CAD-Daten weiterverarbeiten, in der Regel diese Formate nur als Austauschformate und bieten zusätzlich Schnittstellen zu proprietären Formaten an. Da sich nicht alle Details in ein unabhängiges Format übertragen lassen, ist die Datenkonvertierung zumeist auch mit geringem Informationsverlust behaftet.

Das CAD-Modell wird durch Vernetzung in ein Finite-Element-Modell umgewandelt. Anwendungen wie *ANSA*, *HyperMesh* oder *MEDINA* erzeugen aus den mathematischen Flächenbeschreibungen, zum Beispiel durch Splines, ein diskretisiertes Elemente-Netz. Diese Daten werden mit einer Reihe von Preprozessoren zu einem Simulationsmodell aufbereitet und der Simulationssoftware, zum Beispiel *ABAQUS*, *LS-DYNA*, *NASTRAN* oder

PAM-CRASH zur Berechnung übergeben. Die Simulationsergebnisse werden schließlich mit einem Analyse-Werkzeug wie *Animator* oder *PAM-View* ausgewertet.

Dieses Kapitel beschreibt, welche Daten im Pre- und Postprocessing der Crash-Simulation vorkommen. Während Abschnitt 3.4.1 allgemein erläutert, aus welchen Strukturen sich ein Fahrzeugmodell zusammensetzt, beleuchtet der zweite Teil dieses Kapitels die Datenformate, die dem Pre- und Postprocessing der Crash-Simulation zugrunde liegen. Die Datenstrukturen, die im Rahmen dieser Arbeit entwickelt wurden, werden im Kapitel 5 ab Seite 73 vorgestellt.

3.4.1 Geometriestrukturen

Das gesamte Fahrzeugmodell setzt sich aus vielen einzelnen Bauteilen zusammen. Diese wurden vor fünf Jahren noch mit einem homogenen Netz überzogen, das heißt, aneinandergrenzende Bauteile wurden durch gemeinsame Netzknoten miteinander verbunden. Inzwischen ist es möglich, das Verhalten von Verbindungselementen wie Schweißpunkte, Schweißnähte oder Klebeschichten zu simulieren; dadurch können Bauteile nun unabhängig voneinander vernetzt und die inkompatiblen Gitter im Preprocessing miteinander verbunden werden. Jedes dieser Bauteilnetze beziehungsweise der Bauteilverbindungen setzt sich in der Regel aus mehreren Finite-Elementen des gleichen Typs zusammen. Die in Abbildung 3.9 dargestellten Finite-Elemente stellen eine Auswahl der im Rahmen dieser Arbeit wichtigsten Typen dar. Der größte Teil eines Crash-Modells besteht aus vierseitigen Schalen. Balkenelemente dienen in erster Linie der Bauteilanbindung, sofern es noch keine entsprechend spezialisierten Elementdefinitionen gibt, und Volumenelemente bilden zum Beispiel schaumartige Bauteile ab.

Ein Gesamtfahrzeugmodell setzt sich aus hunderten von Bauteilen zusammen, die ihrerseits aus tausenden von Finite-Elementen eines Typs bestehen können (Abbildung 3.10). Während sich im Verlauf der Crash-Simulation lediglich die Knotenkoordinaten ändern, die Netztopologie aber unverändert bleibt, wird in anderen Bereichen der virtuellen Fahrzeugentwicklung, wie zum Beispiel der Tiefziehsimulation, mit adaptiven Modellen gearbeitet, die sich zum Beispiel in Bereichen großer Krümmung lokal verfeinern (Abbildung 3.11). Die konstante Netztopologie in Crash-Modellen wird beim Szenengraph-Design für die Minimierung des Speicherbedarfs ausgenutzt (siehe Kapitel 4).

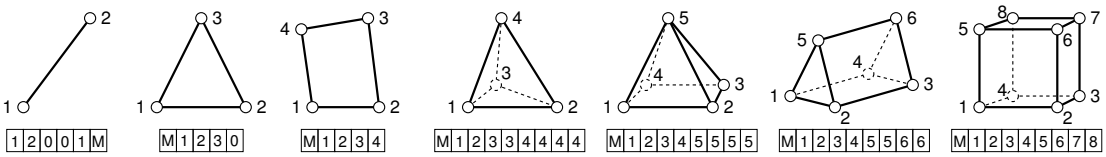


Abbildung 3.9: Jedes der Finite-Elemente ist durch zwei bis acht Referenzen auf Knotenpunkte definiert.

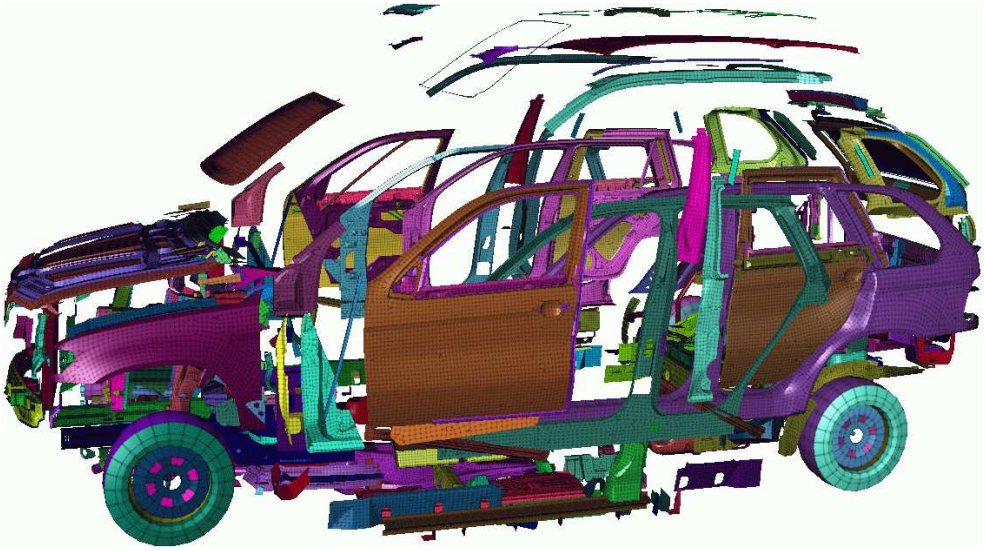


Abbildung 3.10: Die Explosionsdarstellung zeigt das Modell eines BMW X5 in der frühen Phase. Das Modell umfasst etwa 250 000 Elemente in 500 Bauteilen.

3.4.2 Ein-/Ausgabedaten

Es gibt viele verschiedene Datenformate, in denen Finite-Element-Modelle mit ihren Eigenschaften oder Ergebnissen gespeichert werden. In der Regel liegen Eingabedatensätze in einem ASCII-Format vor, um sie lesbar und für Skripte leichter modifizierbar zu machen. Ergebnisdaten werden aufgrund der großen Datenmengen bis auf wenige Ausnahmen in einem binären Datenformat gespeichert. Anhand der nachfolgend skizzierten Datenstrukturen wird veranschaulicht, wie Finite-Element-Modelle im Allgemeinen gespeichert werden.

Die Einträge, aus denen sich ein PAM-CRASH-Eingabedatensatz zusammensetzt, werden „Karten“ genannt (Abbildung 3.12). Sie bestehen aus einem Schlüsselwort gefolgt von einem „Label“ in Form einer positiven Ganzzahl. Jeder Knoten und jedes Element

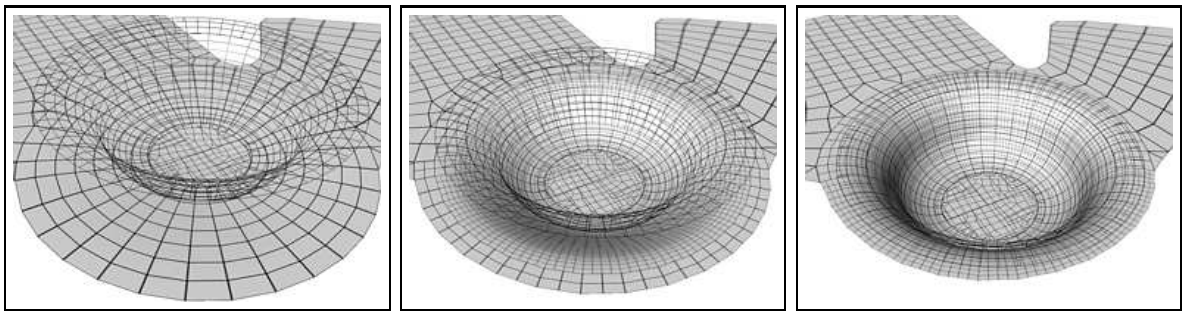


Abbildung 3.11: Topologieänderung während der Tiefziehsimulation

Bauteilattribute:	MATER	M_i	Typ, Dicke, Dichte, Titel, ...				
Knotenpunkt:	NODE	N_i	x_i	y_i	z_i		
Balkenelement:	BEAM	B_i	M_j	N_a	N_b		
Schalenelement:	SHELL	S_i	M_j	N_a	N_b	N_c	N_d
Volumenelement:	SOLID	V_i	M_j	N_a	... ...		N_h
Schweißpunkt:	SPOTW	P_i	G_j	x_i	y_i	z_i	
Schweißpunktgruppe:	SLINT2	G_i	M_j	M_k	Attribute		

Abbildung 3.12: Jede „Karte“ beginnt mit einem Schlüsselwort gefolgt von einem eindeutigen Label, einer positiven Ganzzahl. In der MATER-Karte werden die Attribute eines Bauteils definiert. Während in der NODE-Karte die Punktkoordinaten angegeben werden, verweist in den Elementkarten ein MATER-Label auf die Attributkarte des Bauteils, zu dem das Element gehört, und NODE-Labels spezifizieren die Element-Topologie. Schweißpunkte sind über ein SLINT2-Label einer Schweißpunktgruppe zugeordnet, die die zu verbindenden Bauteile mit MATER-Labels identifiziert.

ist über sein Label eindeutig identifizierbar; allerdings können sich die Label-Bereiche bei Knoten und unterschiedlichen Elementtypen überdecken. Die Knoten-Labels in den Element-Karten legen fest, welche Knotenpunkte die Geometrie eines Elementes definieren. Die Materialeigenschaften eines Bauteils werden in einer MATER-Karte spezifiziert. Bis zur PAM-CRASH-Version 2001 wird jede MATER-Karte ausschließlich von Elementen des gleichen Typs und des gleichen Bauteils referenziert.

Für andere Simulationscodes wie zum Beispiel *DYNA3D* oder *Nastran* gibt es ähnliche Schlüsselwort-basierte Dateiformate für die Eingabedaten. Allerdings kommt gerade für *LS-DYNA* auch noch das ältere sequenzielle Dateiformat zum Einsatz. Die Schwierigkeit bei sequenziellen Dateiformaten besteht darin, dass das komplette Format bekannt sein muss, um einen Datensatz korrekt zu interpretieren. Selbst für Daten, die von der Anwendung ignoriert werden soll, muss deren Länge bekannt sein, damit nachfolgende Daten weiter verarbeitet werden können. Die Reihenfolge, in der das Finite-Element-Modell beschrieben wird, ist für sequenzielle Datenformate fest vorgegeben. Die Komplexität, die sich aus dem sequenziellen Datenformat ergibt, hat zur Umstellung auf Schlüsselwort-basierte Formate geführt. Hier spielt die Reihenfolge der Karten in der Eingabedatei durch die Angabe von Schlüsselwörtern keine Rolle. So können die notwendigen Daten bauteilbezogen zusammengefasst werden.

Die Daten einer PAM-CRASH-Ergebnisdatei sind blockbasiert nach Elementtypen getrennt angeordnet. Die Binärdatei gliedert sich in einen Kontrollblock, der zum Beispiel über Formate der gespeicherten Daten und die Modellgröße Auskunft gibt. Darauf folgen die initiale Knotendefinition sowie die Topologie-Beschreibung des Fahrzeugmodells. Nach diesen Zeitschritt-unabhängigen Daten werden in den anschließenden Blöcken pro Zeitschritt die Knotenkoordinaten und Ergebniswerte gespeichert.

Die Modellbeschreibung setzt sich aus Label- und Werte-Arrays zusammen. So werden in je einem Array die Labels aller Knoten beziehungsweise Elemente für das gesamte Fahr-

zeugmodell zusammengefasst. Alle Knotenkoordinaten sind in einem separaten Feld gespeichert. Das Knoten-Label N_i korrespondiert dabei mit den Koordinaten (x_i, y_i, z_i). Dementsprechend verhält es sich auch mit den Element-Label-Arrays und den zugehörigen Konnektivitätslisten: für jedes BEAM-, SHELL- beziehungsweise SOLID-Label existieren sechs, fünf, beziehungsweise neun Einträge in den Konnektivitätslisten. Ein Konnektivitätstupel referenziert die Netzknoten und das Bauteil, zu dem das Element gehört. Während für Balkenelemente der Elementtyp explizit angegeben wird, bestimmt das mehrfache Vorkommen von Knoten-Labels implizit, um welchen Schalen- oder Volumenelementtyp es sich handelt (Abbildung 3.13). Zum Beispiel ist für dreieckige Schalenelemente N_4^* im SHELL-Konnektivitätstupel entweder Null oder gleich N_3.

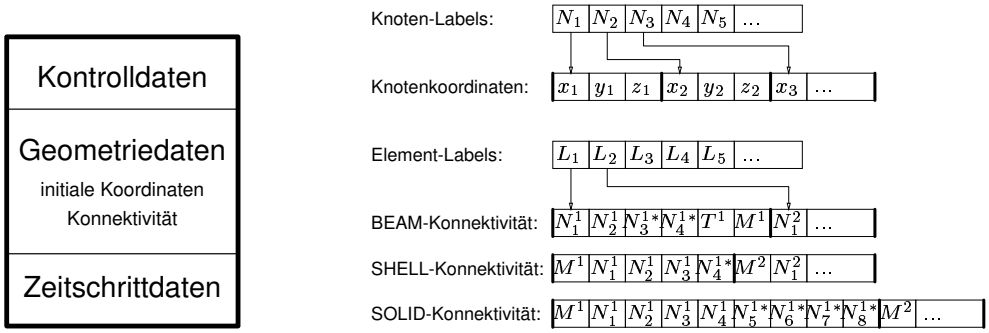


Abbildung 3.13: Eine Ergebnisdatei lässt sich grob in die links dargestellten Blöcke gliedern. Die Geometriedaten beinhalten die rechts abgebildeten Listen: Zu jedem Eintrag in den verschiedenen Label-Listen gibt es in einer korrespondierenden Liste in gleicher Reihenfolge ein Tupel mit Daten, die die Punktkoordinaten des Knotens angeben oder die Element-Topologie und Bauteilzugehörigkeit spezifizieren. Die hervorgehobenen Striche grenzen die Tupel gegeneinander ab. Nullen oder Label-Wiederholungen für die durch „*“ gekennzeichneten Knoten-Labels implizieren den Elementtyp.

Skalare Ergebnisse einer Variablen sind in einem Werte-Array gespeichert; auch hier findet die Zuordnung zu den Knoten beziehungsweise Elementen durch die gleiche Reihenfolge zur entsprechenden Label-Liste statt.

Kapitel 4

Effizientes Szenengraph-Design für zeitabhängige FE-Modelle

In den meisten Bereichen, in denen Computer-Simulation für die Fahrzeugentwicklung eingesetzt wird, stellt die Leistungsfähigkeit der Rechner eine obere Grenze für den Umfang der zu verarbeitenden Daten dar. Die Berechnungsdauer einer Crash-Simulation wird neben der Modellzusammensetzung, also der den verwendeten Finiten Elementen zugrunde liegenden Algorithmen, vor allem von der Modellgröße und der damit zusammenhängenden minimalen Elementkantenlänge vorgegeben, aus der sich wiederum der maximale Simulationszeitschritt ergibt. Für die interaktive Visualisierung von Strukturmodellen spielt neben einer geeigneten Repräsentation der Modelldaten die Eliminierung redundanter Informationen eine wesentliche Rolle. Die in ihrer Komplexität ständig steigenden Strukturmodelle erfordern hierarchische Datenstrukturen, mit denen die Interaktionen durch effizientes Picking und die Bildsynthese durch schnelle Culling-Methoden beschleunigt werden. Die Unterteilung der Geometriedaten in zusammengehörende Einheiten bietet sich wegen der zunehmenden Modularisierung der Fahrzeugmodelldaten an.

Szenengraphen beschreiben das darzustellende Modell in Form eines gerichteten azyklischen Graphen. Sie unterstützen sowohl High-Performance-Rendering als auch den Aufbau einer hierarchischen Ordnung der Szene. Um die umfangreichen und zudem zeitabhängigen Daten auch auf Arbeitsplatzrechnern mit geringerer Hauptspeicherausstattung handhaben zu können, ist es erforderlich, ein optimales Szenengraph-Design zu erarbeiten, das die Speicheranforderung unter Berücksichtigung der Kohärenzen minimiert.

Es wurden verschiedene Szenengraph-Bibliotheken hinsichtlich ihrer Leistungsfähigkeit für die Visualisierung umfangreicher Strukturmodelle betrachtet. Schließlich fiel die Entscheidung auf die Graphikbibliotheken *Cosmo3D* und *OpenGL Optimizer*, da sie zu dem Zeitpunkt im Bezug auf Ausrichtung, Weiterentwicklung und Portabilität sehr vielversprechend und zukunftsweisend erschienen.

Dieses Kapitel gibt einen Überblick über die im Rahmen dieser Arbeit verwendeten Graphikbibliotheken. Nachdem im Abschnitt 4.1 eine Einführung in die genutzten Objekte und Funktionalitäten gegeben wurde, beschreibt Abschnitt 4.2 ein effizientes Szenengraph-

Design für zeitabhängige, topologisch invariante Finite-Element-Modelle. Abschließend werden anhand zweier Beispiele die Erweiterungsmöglichkeiten von Cosmo3D erläutert.

4.1 Szenengraph-Bibliothek Cosmo3D

Bereits 1996 gab es Vorschläge im OpenGL Architecture Review Board (ARB), eine Szenengraph-Bibliothek zu standardisieren, die auf OpenGL basiert und weiter spezialisierten Graphikbibliotheken Basistechnologien zur Verfügung stellt. Die Firma SGI stellte in dem Zusammenhang Cosmo3D, ihre derzeitige Bibliothek zur Szenengraph-Manipulation, als möglichen Ausgangspunkt vor. Die Stärken der bis dahin bereits auf SGI-Plattformen verbreiteten Graphikbibliotheken Inventor und Performer sollten in die zukünftige Szenengraph-Bibliothek übernommen werden.

Anfang 1997 kündigte SGI die Entwicklung von OpenGL Optimizer an, das ein offenes API für Anwendungen aus dem CAD- und Analyse-Bereich werden sollte, welches auf vielen Plattformen, unter anderem IRIX, WindowsNT und Windows95, einsetzbar sein werde. Als Basis für OpenGL Optimizer sollte Cosmo3D dienen, bis es eine standardisierte Szenengraph-Bibliothek geben würde. Da die Mitglieder im OpenGL ARB sich jedoch nicht auf einen Standard für eine Szenengraph-Bibliothek einigen konnten, kündigten die Firmen Microsoft und SGI Ende 1997 das Fahrenheit-Projekt an, das aus drei Schichten bestand: zunächst sollte eine Szenengraph-Bibliothek entstehen, die auf SGI-Plattformen Cosmo3D ersetzen würde. Daran anschließend sollte OpenGL Optimizer auf diese neue Szenengraph-Bibliothek portiert werden.

Da der Portierungsaufwand für Anwendungen, die auf Cosmo3D / OpenGL Optimizer basieren würden, als sehr gering angekündigt wurden, stellten sich diese beiden Graphikbibliotheken als zukunftsweisend dar und wurden für den entstandenen Prototypen *crashViewer* als Zwischenschicht zu der Low-Level-Graphikbibliothek OpenGL gewählt.

4.1.1 Grundlegende Szenengraphobjekte

Dieser Abschnitt bietet einen Überblick über die im Rahmen der vorliegenden Arbeit wichtigsten Cosmo3D-Klassen. Ein Großteil lässt sich wie folgt unterteilen:

- **Basisklassen** — Dazu gehören nicht nur Datenklassen, wie beispielsweise csVec3f, csMatrix4f oder csSeg, um grundlegende Datenstrukturen mit Methoden zur Verfügung zu stellen, sondern auch folgende abstrakte Szenengraph-Basisklassen:
  - csObject ist die Basisklasse der meisten Objekte in der Graphikbibliothek. Sie unterstützt die damals in C++ noch nicht standardisierte *Runtime Type Identification* sowie einen Referenzzähler, der die Lebensdauer eines Objektes steuert und damit die Voraussetzung für die Mehrfachreferenzierung von Szenengraphobjekten ist (Abschnitt 4.1.2.1).

  - csContainer dient als Basisklasse für alle Szenenobjekte, die im Szenengraphen repräsentiert werden. Sie kann einen Namen und eine Referenz auf applikationsinterne Daten in einem csData-Objekt speichern. Darüber hinaus werden hier die das Szenengraphobjekt beschreibenden Daten durch Verwendung von csField-Objekten verwaltet und Methoden zur Handhabung dieser csField-Objekte implementiert.
  - csField ist die abstrakte Basisklasse von Datencontainer-Objekten, die ein oder mehrere gleichartige Daten mit Zugriffsmethoden kapseln. Sie lassen sich miteinander verbinden (siehe Abschnitt 4.1.2.1), um die Datenpropagierung zwischen voneinander abhängigen Objekten zu automatisieren, und bieten zusätzlich einen generellen Zugriffsmechanismus, der es erlaubt, auch die Daten neu kreierter Objekte zu verarbeiten.
  - csNode ist die Basisklasse aller Objekte, die die Szenengraph-Hierarchie definieren. Objekte, die von einer csAction traversiert werden, müssen vom Typ csNode sein. csNode speichert eine Bounding-Sphere[1], die die repräsentierten Szenenobjekte umgibt.

- **Szenengraphklassen** — Sie beschreiben den Aufbau des Szenengraphen:
  - csGroup ermöglicht die Gruppierung mehrerer csNode-Objekte.
  - csSwitch ist von csGroup abgeleitet und erlaubt zusätzlich, die Traversierung des Subgraphen zu steuern, indem die traversierende csAction an keinen, einen bestimmten oder alle Kindknoten weitergeleitet wird.
  - csTransform ist ebenfalls eine Spezialisierung von csGroup, in der zusätzlich eine 4×4-Transformationsmatrix auf alle Kindknoten angewendet wird.
  - csShape assoziiert Attribute des Aussehens (csAppearance) mit der darzustellenden Geometrie (csGeometry) und stellt damit auch die Blattknoten (in Form von Szenengraphklassen) des Szenengraphen dar.

- **Attributklassen** — Sie beeinflussen das Aussehen der Szenenobjekte:
  - csContext spezifiziert die Grundeinstellungen des Graphik-States (siehe auch Abschnitt 3.2.1, Seite 38), die für den gesamten Szenengraphen gelten.
  - csAppearance überschreibt einige der durch csContext definierten Grundeinstellungen, die das Aussehen der durch den im Szenengraphen darüberliegenden csShape-Knoten assoziierten Geometrie beeinflusst.
  - csMaterial speichert neben den Farbreflexionskoeffizienten weitere Eigenschaften, die für die Schattierung der Szenenobjekte von Bedeutung ist, wie zum Beispiel Spekularität, Emission und Transparenz.

[1] Bounding-Sphere (engl.): Hüllkugel

  - csTexture verwaltet Texturparameter und referenziert ein csImage, in dem die eigentliche Textur als zweidimensionales Bild abgelegt ist.

- **Geometrieklassen** — Sie repräsentieren die unter den aktuellen Einstellungen des OpenGL-States darzustellende Geometrie:

  - csGeometry speichert als die abstrakte Basisklasse aller Geometrieobjekte eine achsenparallele Bounding-Box[2], die das geometrische Objekt umgibt.
  - csBox, csSphere, csCone und csCylinder sind Beispiele für einfache geometrische Objekte, die entsprechend parametrisiert werden können.
  - csGeoSet verwaltet als Basisklasse von Objekten, wie zum Beispiel csPointSet, csLineSet, csTriSet, csQuadSet, csPolySet oder csTriStripSet, die mehrere gleiche Primitive repräsentieren, die benötigten Daten in je einem csCoordSet, csNormalSet, csTexCoordSet und gegebenenfalls in je einem dazugehörenden csIndexSet.

Zusätzlich zu den angesprochenen Objekten gibt es noch Klassen, die sich nicht unter einer der oben aufgeführten Kategorien einordnen lassen, aber hier dennoch eingeführt werden, um die Ausführungen in den weiteren Abschnitten zu verdeutlichen.

- csAction ist die abstrakte Basisklasse aller Objekte, die zur Traversierung des Szenengraphen eingesetzt werden (siehe Abschnitt 4.1.3).

- csData kann als Container für applikationsinterne Daten verwendet werden, damit Daten, die von der Applikation angelegt wurden, auch vom Szenengraphen aus genutzt werden können, ohne eine Kopie der Daten anlegen zu müssen.

4.1.2 Datenmanagement

4.1.2.1 Datenredundanz in der Szene

Wie bereits in Kapitel 3.2.4 erwähnt, ist ein wesentlicher Aspekt bei der Definition einer Szene unter Zuhilfenahme eines Szenengraphen die Wiederverwendbarkeit von Teilobjekten in der Szene. Im Zusammenhang mit Szenengraph-Bibliotheken wird dabei vom *Node-Sharing* gesprochen, das heißt, mehrere Objekte verwenden dieselbe Instanz eines Szenengraphobjektes, um das darzustellende Szenenobjekt zu repräsentieren (siehe auch Abbildung 4.2 auf Seite 66). *Node-Sharing* hat den Vorteil, dass die mehrfach genutzten Daten nur einmal im Hauptspeicher vorhanden sind und lediglich mehrfach referenziert werden. Zur Steuerung der Lebensdauer mehrfach referenzierter Instanzen gibt es an allen von csObject abgeleiteten Objekten einen Referenzzähler, der über Methoden in-/dekrementiert werden kann. Wenn dieser Zähler auf Null dekrementiert wird, endet die Lebensdauer der Instanz.

[2] Bounding-Box (engl.): Hüllquader

Die Handhabung einer anderen Art von Datenredundanz wird durch den so genannten *Field-Connection*-Mechanismus unterstützt. Hierbei wird kein Hauptspeicherplatz eingespart, sondern die Propagierung voneinander abhängender Daten automatisiert. Das heißt, wenn der Wert aus einer csField-Instanz über eine Zugriffsmethode ausgelesen wird, prüft diese zunächst, ob sie an eine andere csField-Instanz gebunden ist; sofern das der Fall ist, wird der Wert von dort geholt, bevor er zurückgeliefert wird.

Besonders wichtig ist der *Field-Connection*-Mechanismus im Zusammenhang mit csEngine-Objekten, die Daten über diesen Mechanismus einlesen, weiterverarbeiten und schließlich über eine weitere *Field-Connection* wieder abspeichern. csEngine-Objekte sind also in den Szenengraphen integrierte Rechenautomaten.

4.1.2.2 Datenredundanz zwischen Szenengraph und Applikation

Daten können in Abhängigkeit ihres Umfangs aufgrund des begrenzten Hauptspeichers nicht mehrfach gespeichert werden. Zum Beispiel können die Koordinaten und Normalen aller Dreiecke bei detaillierten Szenen bestehend aus mehreren Millionen Polygonen in der Regel nur einmal im Hauptspeicher gehalten werden. Sofern die Applikation Berechnungen auf diesen Daten durchführen muss, behindern Datenstrukturen, wie sie durch die Szenengraph-Bibliothek und den Aufbau des Szenengraphen vorgegeben sind, gegebenenfalls eine effiziente Datenverarbeitung.

Cosmo3D bietet zu diesem Zweck das csData-Objekt an. Mit ihm ist es möglich, Datenfelder, die von der Applikation intern verwendet werden, im Szenengraphen wiederzuverwenden, ohne eine zusätzliche Kopie der Daten zu erstellen.

4.1.3 Traversierung des Szenengraphen

Der Szenengraph wird für verschiedene Operationen von einer *Action* traversiert, das heißt, beginnend mit dem Wurzelknoten des Szenengraphen ruft ein von csAction abgeleitetes Objekt eine der Operation entsprechende Methode an dem Szenengraphknoten auf, bevor es zum nächsten Szenengraphknoten weitergeleitet wird.

Zur Erstellung eines Bildes wird der Szenengraph von einer csDrawAction traversiert. An jedem Szenengraphobjekt, an dem die *Action* angewendet wird, ruft sie eine Methode (in diesem Fall drawVisit(...) auf, so dass das Objekt in dem Moment entsprechende Aktionen ausführen kann. Zunächst wird das Viewfrustum-Culling durchgeführt, um sicherzustellen, dass sich das repräsentierte Szenenobjekt nicht komplett außerhalb der Sichtpyramide befindet. Anschließend hängt es vom Typ des Szenengraphobjektes ab, was die *Action* auslöst: während csGroup-Objekte die csDrawAction der Reihe nach an ihre Kindknoten weiterreichen, werden für csShape-Objekte zunächst die Einstellungen des OpenGL-States anhand des angehängten csAppearance-Knotens beziehungsweise anhand der Grundeinstellungen im csContext-Objekt vorgenommen, bevor an jedem der angehängten csGeometry-Knoten deren draw(...)-Methode aufgerufen wird, die schließlich das Zeichnen der Geometrie über OpenGL-Befehle durchführt.

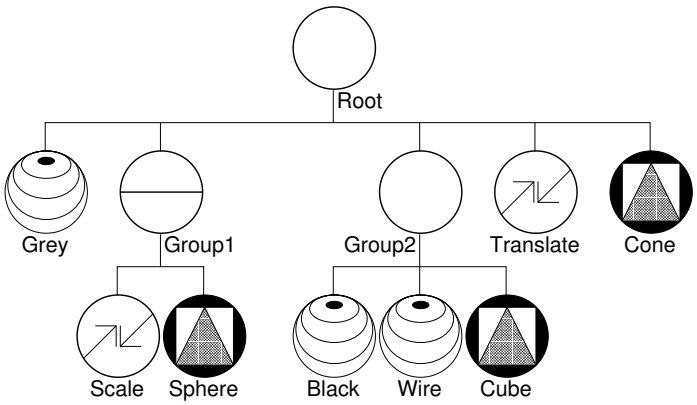


Abbildung 4.1: Unter **Open Inventor** wirken sich Transformationen und Darstellungsattribute durch Separator-Knoten beschränkt auf alle Szenengraphknoten rechts von ihnen aus. Während Group1 die Skalierung von Scale auf Sphere beschränkt, beeinflussen die Attribute Black und Wire nicht nur Cube, sondern auch Cone (Abbildung 4.2, rechts).

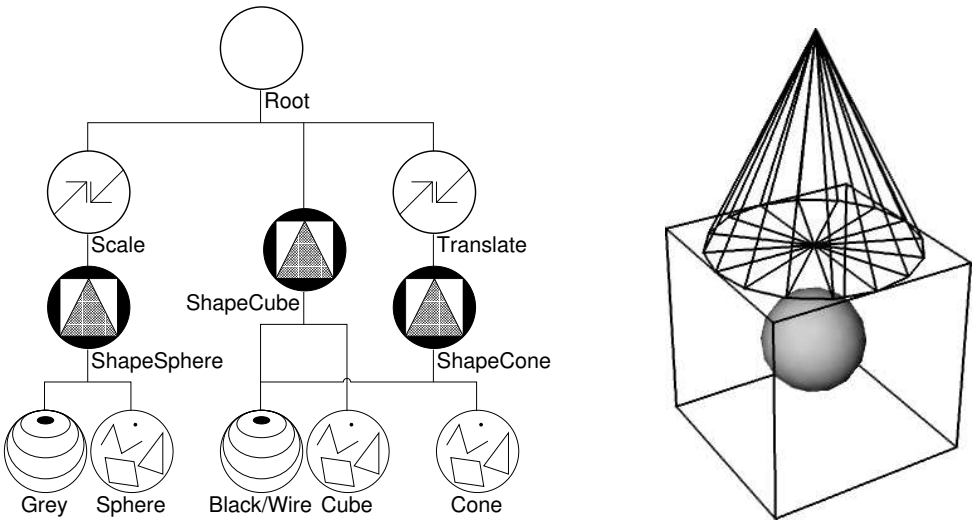


Abbildung 4.2: Transformationen und Darstellungsattribute vererben sich bei **Cosmo3D** nur nach unten und nach rechts im Szenengraphen. Der csAppearance-Knoten Black/Wire wird von ShapeCube und von ShapeCone referenziert (*Node-Sharing*). Das rechts dargestellte Bild könnte sowohl durch diesen als auch durch den in Abbildung 4.1 skizzierten Szenengraphen entstanden sein.

Die Traversierungsreihenfolge und der Einflussbereich von Szenengraphobjekten, die den OpenGL-State modifizieren, spielen eine wichtige Rolle: Im Vergleich mit anderen Szenengraph-Bibliotheken wie zum Beispiel Open Inventor (Abbildung 4.1) fällt auf, dass im Cosmo3D-Szenengraphen die Einstellungen am OpenGL-State lediglich nach unten und

unterhalb der csShape-Knoten nach rechts vererbt werden. Diese Einschränkung erlaubt es, Teilszenengraphen losgelöst von dem Rest des Szenengraphen bearbeiten zu können. Der in Abbildung 4.2 dargestellte Szenengraph ließe sich direkt unter dem Root-Knoten in drei Teile zerlegen, die in beliebiger Reihenfolge von einer csDrawAction abgearbeitet werden könnten, ohne dass sich das Bild ändern würde. Diese Eigenschaft wird von der Graphikbibliothek OpenGL Optimizer zum Beispiel beim Multiprocessing oder für eine Umsortierung des Szenengraphen zur Minimierung der Änderungen am OpenGL-State genutzt, um die Darstellungsgeschwindigkeit zu optimieren.

4.2 Szenengraphaufbau topologisch invarianter Netze

In Abschnitt 3.4 wurden bereits die zu visualisierenden Daten vorgestellt. Bei dem rapide wachsenden Datenumfang der Fahrzeugmodelle muss beim Design einer Szenengraphstruktur besonders für zeitabhängige Daten auf den Speicherbedarf geachtet werden. Eine nicht-indizierte Dreiecksdarstellung benötigt für die Daten, die lediglich die Geometrie eines Fahrzeugmodells, bestehend aus einer halben Million Finiten Schalenelementen und etwa gleich vielen Knoten, über 60 Zeitschritte beschreiben, mehr als 2 Gigabyte: 60 Zeitschritte · 500 000 Elemente · 2 Dreiecke · 3 Knoten · 3 Koordinaten · 4 Byte (pro Float) $>$ 2 GB ergeben sich allein aus der Geometrie ohne Normalen für eine korrekte Schattierung und ohne den Speicherbedarf für die Szenengraphobjekte. Der Speicherbedarf für die Geometrie lässt sich für die Darstellung von Crash-Modellen im Postprocessing jedoch durch folgende Aspekte minimieren:

- Die Bauteile, die aus Schalenelementen modelliert werden, bestehen zu etwa 90 Prozent aus 4-seitigen Elementen. Die Verwendung von Quadrilateralen bringt in diesem Fall eine Speicherplatzersparnis von knapp 27 Prozent.

- Im allgemeinen gilt, über das gesamte Fahrzeugmodell betrachtet, innerhalb von Bauteilnetzen: $\frac{\text{Knotenanzahl}}{\text{Elementanzahl}} \ll \frac{8}{3}$. Eine indizierte Speicherung der Geometrie benötigt statt 12 Koordinaten pro 4-seitigen Schalenelement lediglich 4 Indizes pro Element und zusätzlich 3 Koordinaten pro Knoten.

- Während sich in der Tiefziehsimulation das Finite-Element-Netz in Bereichen großer Umformung zwischen zwei Zeitschritten adaptiv verfeinert, bleibt die Topologie des Fahrzeugmodells in der Crash-Simulation über alle Zeitschritte erhalten. Diese Tatsache kann beim Aufbau des Szenengraphen für das Postprocessing von Crash-Modellen ausgenutzt werden, indem die Topologie lediglich einmal gespeichert und über alle Zeitschritte mit Hilfe des *Node-Sharing* genutzt wird.

Aus dem oben angeführten Beispiel ergibt sich demzufolge ein Speicherbedarf von (60 Zeitschritte · 500 000 Knoten · 3 Koordinaten + 500 000 Elemente · 4 Indizes) · 4 Byte $\approx$ 351 MB für die Geometrie. Abbildung 4.3 stellt das Szenengraph-Design unter Verwendung von Cosmo3D für zeitabhängige Crash-Netze dar. Die Wurzel des Szenengraphen bildet

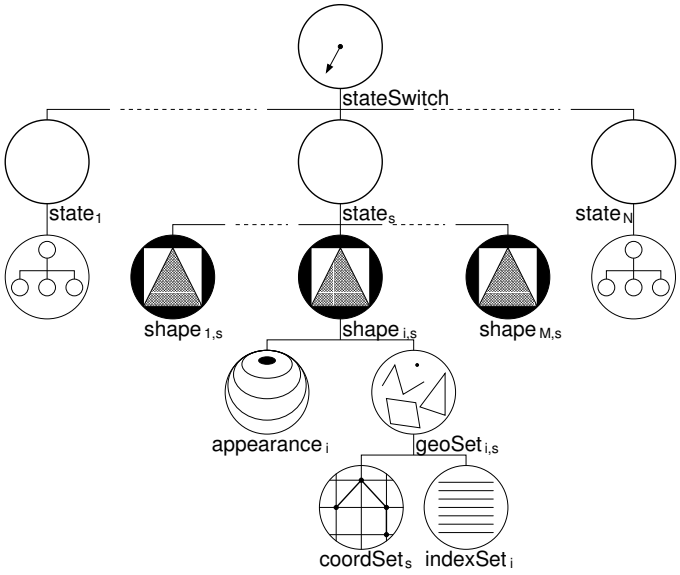
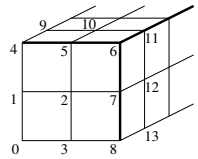


Abbildung 4.3: Dieser Ausschnitt eines Cosmo3D-Szenengraphen repräsentiert zeitabhängige, topologisch invariante Geometrie, in dem der csAppearance- und der csIndexSet-Knoten über alle Zeitschritte genutzt werden, während der csCoordSet-Knoten von allen csShape-Knoten eines Zeitschrittes referenziert wird.

der csSwitch-Knoten stateSwitch, der die Aufgabe hat, zwischen den N verschiedenen Zeitschritten umzuschalten. Für einen Zeitschritt s wird das gesamte Fahrzeugmodell von dem Subgraphen unterhalb des csGroup-Knotens state_s repräsentiert. Die M Bauteile, aus denen sich das Gesamtmodell zusammensetzt, werden jeweils durch einen csShape-Knoten $\mathsf{shape}_{i,s}$ abgebildet, indem der die Geometrie verwaltende csGeoSet-Knoten mit dem das Aussehen beeinflussende csAppearance-Knoten verknüpft wird. Dabei gibt es pro Bauteil nur einen csAppearance-Knoten $\mathsf{appearance}_i$, der jedoch über alle N Zeitschritte von den jeweiligen $\mathsf{shape}_{i,s}$ referenziert wird. Die Topologie eines Bauteilnetzes ist in den csIndexSet-Knoten definiert und wird ebenfalls für jeweils ein Bauteil über alle Zeitschritte gemeinsam verwendet. Die Koordinaten aller Netzknoten des Fahrzeugmodells können für einen Zeitschritt s in einem csCoordSet-Knoten $\mathsf{coordSet}_s$ gespeichert und von allen $\mathsf{geoSet}_{i,s}$ gemeinsam referenziert werden.

Für eine korrekte Beleuchtungsberechnung müssen außer den Geometrieinformationen auch Normalenvektoren gespeichert werden. Wie bereits in Kapitel 3.2 beschrieben, unterstützt der OpenGL-Standard 1.2 zwei Schattierungsmodelle: *Flat-Shading* – eine Normale pro Element – und *Gouraud-Shading* – eine oder mehrere Normalen pro Knoten. Beim *Flat-Shading* werden für das oben angegebene Beispielmodell nur weitere 6 MB für die Normalenvektoren benötigt. Das *Gouraud-Shading* bedarf mehrerer Normalenvektoren für die Netzknoten, die auf einer *Gouraud-Kante* liegen und einer Indizierung der Normalenvektoren. Ein eigenes csIndexSet für die Normalen sollte nur angelegt werden, sofern folgende Bedingung erfüllt ist: $3 \cdot \text{Zeitschrittanzahl} \cdot (\text{Normalenanzahl} - \text{Knotenanzahl}) \cdot 4 \geq$



csIndexSet	[0,3,2,1]	[1,2,6,4]	[2,11,8,6]	[3,13,11,2]	[5,7,16,15]	...
index:	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 ...
csCoordSet	0, 1, 2, 3,	4, 4, 5, 5,	6, 6, 6, 7,	7, 8, 8, 9,	10, 11, 11, 12,	13, ...
csNormalSet	0, 1, 2, 3,	$4^a, 4^b, 5^a, 5^b$,	$6^a, 6^b, 6^c, 7^a$,	$7^b, 8^a, 8^b, 9$,	$10, 11^a, 11^b, 12$,	13, ...

Abbildung 4.4: Dieses Beispiel aus [63] zeigt, wie das csIndexSet sowohl von dem csCoordSet als auch vom csNormalSet gemeinsam zur Indizierung genutzt werden kann. Die breiten Elementkanten entsprechen *Gouraud-Kanten*, das heißt, dass an den Netzknoten 4, 5, 7, 8, und 11 jeweils zwei und am Netzknoten 6 sogar drei verschiedene Normalenvektoren vorliegen. Daher müssen bei einem gemeinsam genutzten csIndexSet für diese Punkte die Koordinaten für jede zusätzliche Normale vervielfacht werden.

$4 \cdot$ Elementanzahl $\cdot\ 4 +$ sizeof(csIndexSet), wobei die Normalenanzahl hier alle Normalen an allen Knoten umfasst. Andernfall ist der Speicherbedarf geringer, wenn für Knotenkoordinaten und Normalenvektoren ein gemeinsamer Index für jeden Knoten verwendet wird, wobei die Koordinaten von Knoten auf *Gouraud-Kanten* mehrfach gespeichert werden müssen (Abbildung 4.4). Die Handhabung von OpenGL Vertex-Arrays wird seit Cosmo3D / OpenGL Optimizer Version 1.3.1 unterstützt, allerdings nur bei Verwendung eines gemeinsamen csIndexSet für Koordinaten und Normalen.

4.3 Erweiterungen unter Cosmo3D

Ein wichtiges Kriterium bei der Auswahl einer Szenengraphbibliothek ist die Erweiterbarkeit ihrer Schnittstelle um zusätzliche Objekte. Dies ist erforderlich, sobald die Funktionalität der zur Verfügung stehenden Klassen modifiziert werden soll oder die Bibliothek durch neue komplexe Objekte ergänzt wird. Da kaum Informationen zur Erweiterung dieser Szenengraphbibliothek verfügbar sind, beschreibt dieser Abschnitt an zwei unterschiedlichen Beispielen, wie bei der Integration neuer Objekte in die Szenengraphbibliothek vorzugehen ist.

Ähnlich wie Open Inventor lässt sich auch Cosmo3D durch weitere Szenengraphknoten und Actions erweitern. Das csField/csFieldInfo-Konzept sieht vor, Attribute neu hinzugefügter Objekte nach außen über eine generische Schnittstelle zugänglich zu machen. Der Datentyp eines Attributs kann spezifiziert und ihm eine ID zugewiesen werden, über die unter Verwendung vorgegebener Methoden auf die Daten zugegriffen wird.

4.3.1 Clip-Objekt

Ein vielfältig einsetzbares Werkzeug zur Analyse der Fahrzeugmodellstruktur stellen frei bewegliche Clip-Objekte dar. Mit ihrer Hilfe ist es möglich, verdeckende Teilstrukturen interaktiv auszublenden, um freien Einblick auf dahinterliegende Bereiche zu erhalten.

Ein einfaches und von OpenGL direkt unterstütztes Clip-Objekt ist die über glClipPlane steuerbare Clip-Ebene. Zusätzlich zu der Clip-Funktionalität muss der neue Cosmo3D-Szenengraphknoten eine Geometrie darstellen, die die Position und Lage der Clip-Ebene widerspiegelt und über die der Anwender mit dem Werkzeug interagieren kann. Der neue Cosmo3D-Szenengraphknoten ist ein csGroup-Knoten. Alle Objekte, die sich im Szenengraphen unter einem csClipGroup-Knoten befinden, können von seiner Clip-Ebene geschnitten werden. Durch entsprechende Positionierung im Szenengraphen ist es somit möglich, auch bauteil- oder bauteilgruppenbezogen Teilstrukturen wegzuschneiden. Die Deklaration von csClipGroup in Abbildung 4.5 zeigt exemplarisch, welche Methoden implementiert werden müssen, um einen neuen Szenengraphknoten in das Cosmo3D-Konzept einzubetten. Die am Ende des Kapitels in Abbildung 4.6 beschriebene Implementierung verdeutlicht am gleichen Beispiel, wie die Initialisierung von Cosmo3D-Szenengraphknoten vollzogen wird und wie im speziellen Fall von csClipGroup durch Hinzufügen einiger OpenGL-Anweisungen in der Rendering-Methode dieses neue Szenengraphobjekt entstanden ist, das so auch in anderen Cosmo3D-basierten Applikationen zur interaktiven Exploration polygonaler Modelle eingesetzt werden könnte (vergleiche SoClipKit in [61], Abschnitt 4.1.2).

```
class csClipGroup : public csGroup {
 private:
   static csType *_ClassType;   // für RTTI (Run Time Type Identification)
   static void initClass();     // initialisiert _ClassType und Fields

   csBool       activePlane;    // Flag, ob Clip-Ebene ein- oder ausgeschaltet
   csBool       negatePlane;    // ermöglicht Invertierung der Clip-Richtung
   csBool       showPlane;      // erlaubt das Ausblenden der Ebenengeometrie
   csTransform *planeXf;        // Root-Knoten der Ebenengeometrie
   GLdouble     clipPlaneEq[4]; // Clip-Ebenengleichung für glClipPlane
   static int   maxNumPlanes;   // maximale Anzahl aktiver OpenGL Clip-Ebenen
   static int   numCurActive;   // Anzahl derzeit aktiver Clip-Ebenen

 protected:
   void init();                 // von Konstruktoren verwendete Initialisierung

 public:
   enum { FIRST = csGroup::LAST-1,
      ACTIVE,     // über diesen Enumerator wird auf 'activePlane' zugegriffen
      NEGATE,     // entsprechend für 'negatePlane'
      SHOW,       // und für 'showPlane' (siehe auch initClass())
      LAST };     // Verwendung von FIRST/LAST sichert lückenloses Inkrementieren
   static csType *getClassType() {
                  if (!_ClassType) initClass(); return _ClassType; }
   csClipGroup() : csGroup(getClassType()) {
                  init(); }
   csClipGroup(csType *type) : csGroup(type) {
                  init(); }             // Konstruktor für abgeleitete Objekte
   virtual ~csClipGroup();              // zerstöre Clip-Ebenengeometrie
   virtual csContainer *instantiate() {  // z.B. bei Instanziierung durch Loader
                  return new csClipGroup(); }

   virtual csTravDirective drawVisit(csDrawAction *);   // zur Darstellung
   virtual void isect(csIsectAction *);                 // für Picking benötigt
};
```

Abbildung 4.5: Deklaration des neuen Cosmo3D-Szenengraphknoten csClipGroup.

Zusätzlich zu den hier aufgelisteten Methoden gibt es noch Funktionalität, um eine csClipGroup-Instanz sowohl im Szenengraphen als auch in der Szene zu positionieren und die Attribute zu modifizieren. Des Weiteren werden über den Field-Connection-Mechanismus csClipGroup-Instanzen synchronisiert, die in verschiedenen Zeitschritt-Subgraphen die gleiche Clip-Ebene repräsentieren.

Weitere Beispiele für die Erweiterung der Szenengraphbibliothek durch neue Knoten, die in dieser Arbeit entwickelt wurden, sind

- csCallbackNode — universell einsetzbarer Knoten, um zum Beispiel die Traversierung des Szenengraphen zu modifizieren oder bestimmte Aktionen auszulösen
- csFontNode — erlaubt die Einbettung von Annotationen in der Szene
- csQuadStripSet — ein Szenengraphknoten für Quadrilateralstreifen (siehe Abschnitt 6.1)
- csUnpickableLine — Linien, die nicht gepickt werden können, vermeiden eine unbeabsichtigte Selektion hervorgerufen durch das Hüllvolumen-basierte Picking von Linien

4.3.2 Traversierungsfunktionen

Die von der Graphikbibliothek Cosmo3D / OpenGL Optimizer zur Verfügung gestellten csAction-Objekte traversieren den Szenengraphen und führen eine Operation aus, indem entsprechende Methoden an dem Szenengraphknoten aufgerufen werden. Neue Operationen, die auf nur im Szenengraphen vorhandene Daten zurückgreifen, können als eine erweiterte Traversierungsfunktion implementiert werden. Eine derartige Erweiterung ist zum Beispiel die IntersectionAction, die mit Hilfe einer Selektionsgeometrie Szenengraphobjekte selektiert beziehungsweise schneidet. Im Folgenden wird lediglich die Arbeitsweise von Traversierungsfunktionen erläutert.

OpenGL Optimizer stellt mit der Basisklasse opDFTravAction den grundlegenden Traversierungsmechanismus zur Verfügung. Zur Implementierung einer eigenen Traversierungsfunktion können folgende Methoden überladen werden:

- begin wird vor Beginn der Traversierung aufgerufen und kann zur Initialisierung genutzt werden.
- end schließt die Szenengraphtraversierung ab.
- preNode wird jeweils mit dem als nächstes zu traversierenden Szenengraphknoten aufgerufen und bietet die Möglichkeit, Einfluss auf den Fortgang der Traversierung zu nehmen.
- postNode beendet die Traversierung eines Szenengraphknoten.

Spezialisierte Traversierungsfunktionen können durch das Wissen über die applikationsspezifische Szenengraphstruktur profitieren. Darüber hinaus können Unzulänglichkeiten der zur Verfügung gestellten Traversierungsfunktionen, wie zum Beispiel die fehlende Unterscheidung der csDrawAction zwischen semitransparenten und opaken Objekten, durch eigene Erweiterungen umgangen werden.

```
csType *csClipGroup::_ClassType = 0L;

void
csClipGroup::initClass()                    // === Klassen-Initialisierung ====
{
   csContext::getClassType();    // siehe Makro BEGIN_INIT_CSTYPE in csMacros.h
   _ClassType = new csType(csGroup::getClassType(), "ClipGroup");

   csSFBool::addTypeField(
     _ClassType,          // füge dieser Klasse ein neues Attribut hinzu
     "active",            // Bezeichner mit dem dieses Attribut spezifiziert wird
     csClipGroup::ACTIVE, // Enumerator über den auf das Attribut zugegriffen wird
     csBool(true),        // initialer Wert
     (csBool csContainer::*) &csClipGroup::activePlane);  // Speicher für Attribut

   // entsprechend werden auch 'negatePlane' und 'showPlane' als Attribute angemeldet
   numCurActive = 0;
   // initialisiere 'maxNumPlanes' über glGetIntegerv(GL_MAX_CLIP_PLANES)
}

void
csClipGroup::init()                         // === Instanz-Initialisierung ====
{
   getClassType();    // stellt sicher, dass _ClassType initialisiert ist
   initFieldDefaults(_ClassType);    // Attribute intialisieren
   // Subgraphen für planeXf aufbauen, der Clip-Ebenengeometrie repräsentiert
}

csTravDirective
csClipGroup::drawVisit(csDrawAction *da)    // === Rendering-Methode ==========
{
   // aktualisiere Clip-Ebenengleichung
   // zeichne Ebenengeometrie sofern 'showPlane' gesetzt
   bool    doClip = (activePlane && (csClipGroup::numCurActive < csClipGroup::maxNumPlanes));
   GLenum  planeIdx = GL_CLIP_PLANE0 + csClipGroup::numCurActive;
   if (doClip) {         // aktiviere OpenGL Clip-Ebene temporär
      glClipPlane(planeIdx, clipPlaneEq);
      glEnable(planeIdx);
      ++csClipGroup::numCurActive;
   }
   csTravDirective travDisp = csGroup::drawVisit(da);   // zeichne Subgraphen
   if (doClip) {         // schalte zugehörige OpenGL Clip-Ebene wieder ab
      --csClipGroup::numCurActive;
      glDisable(planeIdx);
   }
   return travDisp;      // Enum, ob SG-Traversierung fortgesetzt oder abgebrochen werden soll
}
```

Abbildung 4.6: Die Methode initClass zeigt, wie ein Attribut über das csFieldInfo-Konzept anderen Objekten zur Verfügung gestellt wird. drawVisit wird beim Rendering aufgerufen und stellt sicher, dass sich die frei bewegliche Clip-Ebene lediglich auf den eigenen Subgraphen auswirkt.

Kapitel 5

Architektur des Prototypen

Im Rahmen dieser Arbeit ist in enger Kooperation mit der Berechnungsabteilung EK-21 der *BMW Group* ein produktiv eingesetztes Visualisierungswerkzeug entstanden. Die Zielsetzung, die anfangs darin bestand, einen Postprocessor-Prototypen zu entwickeln, der durch Ausnutzung neuer Techniken die Analyse von Simulationsergebnissen beschleunigt, änderte sich durch die Teilnahme an dem BMBF-Projekt *Autobench* [68]. Ziel des Teilprojektes war die Entwicklung eines integrierten Prototypen für das Pre- und Postprocessing unabhängig voneinander vernetzter Bauteile. Hierzu wurden im Rahmen dieser Arbeit zahlreiche Beiträge geleistet, die in den nachfolgenden Kapiteln detailliert beschrieben werden.

In diesem Kapitel wird zunächst ein Überblick über die Software-Struktur gegeben und anschließend auf ausgewählte Module näher eingegangen. Dabei werden jeweils die entwickelten Funktionalitäten und die dazu notwendigen Datenstrukturen vorgestellt. Darüber hinaus wird das entwickelte Interaktionskonzept des entstandenen Prototypen *crashViewer* beleuchtet.

5.1 Objektorientiertes Design der Software

Die Software, die inzwischen circa 150 000 Zeilen C++-Quelltext umfasst, lässt sich grob in folgende Teilbereiche untergliedern (vergleiche Abbildung 5.1):

Ein-/Ausgabe-Module: Hierzu gehören außer den in Abschnitt 5.1.2 vorgestellten Reader-Modulen, die dem Einlesen verschiedener Dateiformate dienen, auch die in Abschnitt 5.1.4 entwickelten Parser-Module für selbstdefinierte Dateiformate.

Interne Datenstrukturen: Die Objekte, die aus einer objektorientierten Analyse der zu verarbeitenden Daten hervor gegangen sind, bilden den Kern der internen Datenstrukturen und werden im nachfolgenden Abschnitt eingeführt.

Szenengraph-Module: Für die beiden Szenengraph-Bibliotheken Cosmo3D und Open Inventor wurde jeweils eine auf GeoBase basierende Schnittstelle geschaffen, um die

internen Datenstrukturen auf Szenengraphobjekte abzubilden, die für eine effiziente Visualisierung genutzt werden. Während ExtCsOp stellvertretend für die bereits in Abschnitt 4.3 erläuterten Erweiterungen der Szenengraphbibliotheken steht, wird GeoCosmo, die Schnittstelle zu Cosmo3D, im Abschnitt 5.1.3 vorgestellt.

Funktionsmodule: Die Module, in denen Funktionalitäten zu den verschiedenen Anwendungsbereichen implementiert ist, werden aufgrund ihres Umfangs in den Kapiteln 6–8 separat behandelt.

Graphische Benutzerschnittstelle: Das zentrale Modul des Prototypen *crashViewer* stellt aus Sicht des Anwenders der in Abschnitt 5.2 beschriebene OpenGL Optimizer basierte Viewer dar, von dem die im darauffolgenden Abschnitt beschriebenen Mechanismen zur Datenanalyse gesteuert werden.

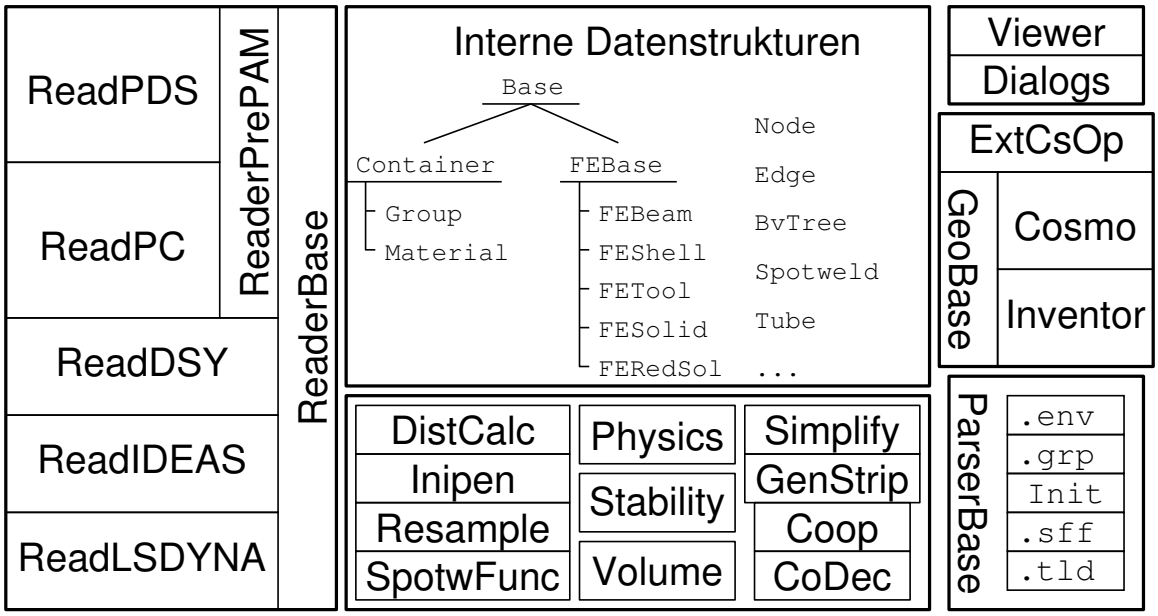


Abbildung 5.1: Überblick über die modulare Software-Struktur von *crashViewer*.

5.1.1 Interne Datenstrukturen

In einer objektorientierten Analyse wurden zunächst die grundlegenden Objekte identifiziert, aus denen sich die zu verarbeitenden Datensätze zusammensetzen, und schließlich folgende Klassen abgeleitet:

- Node: Diese Klasse repräsentiert die Knoten der Finite-Elemente. Auf die Koordinaten wird über einen Pointer auf ein zusammenhängendes Array von Koordinaten

verwiesen, welches beim Einlesen der Daten initialisiert wird. Dadurch werden beim Einlesen mehrerer Zeitschritte zusätzliche Kopiervorgänge vermieden. Das Objekt hält Verweise auf die angrenzenden FEBase-Elemente und speichert eine oder mehrere Normalen.

- Base: Fast alle in den Daten vorkommenden Objekte besitzen als Identifikator ein Label, das aus einer positiven Ganzzahl besteht und für den Objekttyp eindeutig ist. Außerdem sind die meisten Objekte einem anderen Objekt zugeordnet, zum Beispiel ist ein Finite-Element stets einem Bauteil (Material) zugeordnet.

- FEBase: Die Basisklasse der Finite-Elemente ist von Base abgeleitet und speichert eine Node-Liste der Netzknoten, die das Element definieren. Sie hat folgende Spezialisierungen:

  - FEBeam: Hier wird die Klasse der eindimensionalen Finite-Elemente repräsentiert.
  - FEShell: Die Klasse der zweidimensionalen Schalenelemente, die durch drei oder vier Knoten definiert sind, speichert außer einer Normalen auch skalare Parameter, wie zum Beispiel die Dicke oder die maximale plastische Dehnung.
  - FESolid: Die dreidimensionalen Volumenelemente werden in dieser Klasse zusammengefasst. Durch Einsatz von FERedSol ist es möglich, nur die äußere Hülle eines aus Volumenelementen bestehenden Bauteils darzustellen.

- Container: Zu dieser von Base abgeleiteten Klasse gehören Objekte, die mehrere Base-Objekte zusammenfassen. Ihnen kann ein Name, bestehend aus ASCII-Zeichen, zugewiesen werden und sie können mit einem Szenengraphobjekt verknüpft werden, das sie repräsentiert.

- Group: Eine Spezialisierung der Klasse Container, die der Modellstrukturierung dient. Hier können mehrere Container-Objekte zu einer Bauteilgruppe zusammengefasst werden.

- Material: Diese Unterklasse von Container bündelt gleichartige FEBase-Objekte und repräsentiert ein Bauteil. Jede Instanz referenziert eine Bounding-Volume-Hierarchie (BvTree), die die Bauteilelemente hierarchisch in Gruppen zusammenfasst.

- Spotweld: Die Schweißpunkte der Eingabedaten werden durch diese Klasse vertreten und entsprechend der Angaben zugehöriger Gruppierungsdaten in SpotweldGroup-Instanzen zusammengefasst.

- Rigbo: Die Starrkörperverbindungen (engl. *rigid body*), die im Wesentlichen aus einer Liste verschiedener untereinander nicht deformierbarer Netzknoten bestehen, werden durch die Klasse abgebildet.

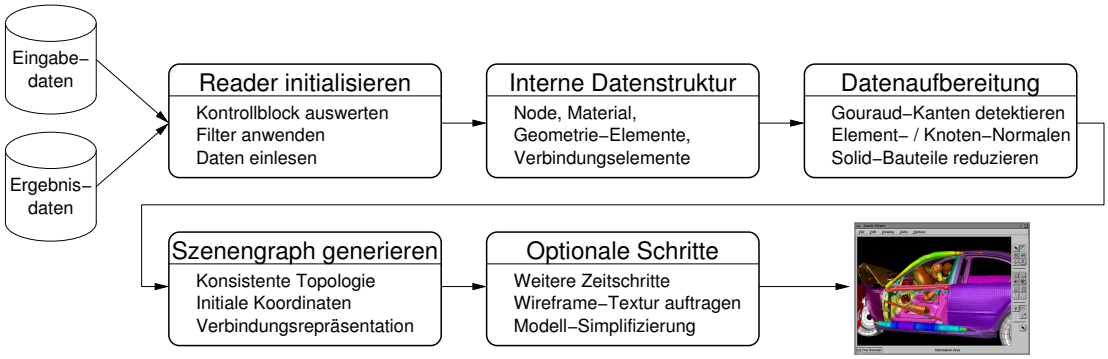


Abbildung 5.2: Dieses Diagramm skizziert den Datenfluss vom Einlesen der Eingabe- oder Ergebnisdaten über die Initialisierung interner Datenstrukturen bis hin zur Darstellung des generierten Szenengraphen.

Darüber hinaus wurden zahlreiche Hilfsobjekte implementiert, wie zum Beispiel Edge, das der Speicherung von Nachbarschaftsinformation zwischen Elementen dient und die *Gouraud-Kanten*-Erkennung beschleunigt. Den Datenfluss vom Einlesen der Daten bis zur Darstellung im Viewer wird von Abbildung 5.2 schematisch dargelegt.

5.1.2 Einlese-/Abspeicher-Module

Zum Einlesen der verschiedenen Datenformate sollten ursprünglich ausschließlich externe Bibliotheken verwendet werden, die für kommerzielle Produkte (zum Beispiel *PAM-View*) gepflegt werden, um den durch Änderungen der Datenformate verursachten Umstellungsaufwand zu minimieren und zugleich stets aktuelle Simulationsergebnisse durch Auswechseln der externen Bibliotheken einlesen zu können. Zu diesem Zweck stellte die Firma ESI im Rahmen des Autobench-Projekts stets die aktuellen Versionen ihrer *PDS*- und *DAISY*-Bibliotheken zum Einlesen von Pre- und Postprocessing-Daten zur Verfügung. Da die Daten, wie in Kapitel 3.4.2 erläutert, unterschiedlich strukturiert sind, wurden für diese beiden externen Bibliotheken die in den Abschnitten 5.1.2.1 und 5.1.2.2 beschriebenen Schnittstellen-Module implementiert. Darüber hinaus wurden eigene *Reader*-Module zum Einlesen von *LSDYNA*- und *IDEAS*-Datenformaten entwickelt, die im Anschluss vorgestellt werden.

Die Grundfunktionalität aller Einlese-Module wird in der Klasse ReaderBase durch virtuelle Methoden zur Verfügung gestellt. Der Ablauf beim Einlesen einer Datei geht in folgenden Schritten vor:

1. **Kontrollblock auswerten:** Bei der Instanziierung eines *Reader*-Moduls wird die Datei geöffnet und Informationen über die Größe und Zusammensetzung das Datensatzes eingeholt.

2. **Filter anwenden:** Durch Angabe einer Bitmaske für einzulesende Finite-Element-Typen, Verwendung einer Gruppierungsdatei (siehe auch Abschnitt 5.1.4) oder Spezifikation verschiedener *Label*(-Bereiche) kann das Einlesen auf eine Untermenge der Daten eingeschränkt werden, um Lade- und Darstellungszeiten zu verkürzen.

3. **Interne Datenstruktur initialisieren:** In der Initialisierungsphase werden die internen Datenobjekte klassenweise instanziiert. So werden die gesamten Koordinaten eines Zeitschrittes in einen zusammenhängenden Speicherbereich eingelesen, auf den von den Node-Instanzen verwiesen wird. Für jede Klasse werden nach dem Label sortierte Referenzlisten aufgebaut, die später unter Anwendung binärer Suchalgorithmen eine effiziente Generierung der Geometrie zulassen.

4. **Modell-Hierarchie aufbauen:** Der Aufbau einer Gruppenstruktur erfolgt nur bei Verwendung einer Gruppierungsdatei und dient dem Zusammenfassen einzelner Bauteile zu Bauteilgruppen.

Nach Abschluss der Initialisierungsphase sind nur die benötigten Grunddaten (Topologie, Geometrie, Hierarchie) initialisiert. Das *Reader*-Modul bleibt bis zu seiner Zerstörung in einem Zustand, in dem es zum Einlesen zusätzlich benötigter Daten (zum Beispiel von Knotenkoordinaten aus weiteren Zeitschritten oder von Ergebniswerten) genutzt werden kann.

5.1.2.1 Schnittstelle für *PAM-Crash* Eingabedateien

Zum Einlesen der *PAM-Crash* Eingabedateien, die im ASCII-Format vorliegen, startete die Firma ESI 1997 die Entwicklung einer neuen objektorientierten Produktplattform, dem *PAM Open Environment* (*POE*). In diesem Zusammenhang entstand die C++ basierte *Permanent Data Storage*- oder kurz *PDS*-Bibliothek, die zukünftig in der Lage sein sollte, auch binäre Ein- und Ausgabedaten verschiedener Simulationscodes zu verarbeiten.

Die im Rahmen dieser Arbeit unabhängig entwickelte interne Datenstruktur entsprach zwar überwiegend einer Untermenge der Objekte aus der *PDS*-Bibliothek, jedoch mussten die Daten in dieser Schnittstelle trotzdem kopiert werden, um eine enge Kopplung zu vermeiden. Bei Verwendung der *PDS*-Bibliothek werden also zwei komplette Repräsentationen des Fahrzeugmodells im Hauptspeicher gehalten. Der Speicherbedarf ist allerdings unkritisch, da für Eingabedaten jeweils nur die initiale Geometrie, also nur ein Zeitschritt definiert ist.

Bei der Klasse ReadPDS handelt es sich im Gegensatz zu den anderen *Reader*-Modulen um eine bidirektionale Schnittstelle, da hier auch interne Daten zurück in die *PDS*-Datenstrukturen transformiert werden, um sie anschließend mit den in der Bibliothek angebotenen Funktionen abspeichern zu können.

Sowohl die mangelnde Performanz der *PDS*-Bibliothek als auch der Umstand, dass neu entwickelte Objekte erst mit mehreren Monaten Verzögerung von nachfolgenden Bibliotheksversionen verarbeitet werden konnten, führte zur Entwicklung eines eigenen *Reader*-

Moduls (ReadPC), das keine externe Bibliothek verwendet und die internen Daten direkt aus den ASCII-Daten initialisiert. Um die Verwendung beider *Reader*-Module zu ermöglichen, wurde den beiden Klassen ReadPDS und ReadPC eine gemeinsame Oberklasse ReadPrePAM übergeordnet, die von ReaderBase abgeleitet ist.

Der Einsatz von *crashViewer* in der Berechnungsabteilung für Crash-Simulation der *BMW Group* erforderte, dass auch Datenobjekte verarbeitet werden konnten, die aus der Methodenentwicklung hervorgingen und noch nicht im Produktstandard enthalten waren. Daher wurde ein eigenes erweitertes *Reader*-Modul entwickelt, das die interne Datenstruktur direkt aus den Datenkarten in der Datei initialisiert. Durch die Einschränkung auf die wirklich benötigten Daten und die direkte Verarbeitung konnte die Zeit für das Einlesen von *PAM-Crash* Eingabedateien auf 15% der ursprünglichen benötigten Zeit beschränkt werden.

5.1.2.2 Schnittstelle für *PAM-Crash* Ergebnisdateien

Während der Crash-Simulation *PAM-CRASH* werden die Zwischenergebnisse für jeden zweitausendsten Berechnungsschritt in einem binären Dateiformat abgespeichert. Zum Einlesen der Ergebnisdaten wurde die in Fortran 77 implementierte *DAISY*-Bibliothek der Firma ESI eingesetzt. Die Ergebnisdaten werden im Gegensatz zu den Eingabedaten nach Objekttypen getrennt für das gesamte Fahrzeugmodell in einem Array abgelegt. Die Reihenfolge der Elemente entspricht der aus der Eingabedatei. Finite-Elemente eines Bauteils können über das gesamte Array des Elementtyps verstreut sein. Folglich kann das Einlesen von Teilbereichen dieser langen Arrays nur dann sinnvoll eingesetzt werden, wenn sichergestellt ist, dass die Daten bereits im Eingabemodell Bauteil-spezifisch gebündelt wurden.

Es folgen einige Beispiele von Bibliotheksfunktionen, die das Einlesen der Daten in den Hauptspeicher übernehmen:

- _dsyvar(...) holt die Anzahl der Netzknoten, Volumen-, Schalen- und Balkenelemente sowie die Koordinatendimension.
- _dsysta(...) ermittelt die Anzahl der abgespeicherten Zeitschritte.
- _dsylno(...) überträgt das Array der Knotenlabels in den Hauptspeicher.
- _dsycoo(...) übermittelt die Koordinaten der Netzknoten im initialen Zeitschritt.
- _dsydef(...) liest die Koordinaten des deformierten Finite-Element-Netzes zu einem spezifizierten Zeitschrittindex ein.
- _dsyvsh(...) kopiert die Werte für den angegebenen Schalenparameter (zum Beispiel die maximale plastische Dehnung) für alle Schalen des angegebenen Zeitschrittindexes.

Die Hälfte der rund 60 Zugriffsfunktionen auf Ergebnisdateien, die in einem undokumentierten binären Dateiformat vorliegen, werden im Modul ReadDSY genutzt, um die Datenarrays einzulesen. Die internen Datenobjekte werden aus diesen großen Arrays in Klassen-basierten Initialisierungsfunktionen generiert. Um das Einlesen mehrerer Zeit-

schritte möglichst effizient zu gestalten, werden die Knotenkoordinaten direkt in den Speicherbereich gelesen, auf den die Koordinaten-Pointer der internen Node-Instanzen verweisen. Dadurch wird zusätzliches Kopieren und Zerstückeln der Daten umgangen.

Verzögerungen durch geringe Bandbreiten beim Einlesen großer Datensätze von entfernten Fileservern werden durch Parallelisierung minimiert. Dabei werden bereits die Daten für den Zeitschrittindex $n+1$ über das Netzwerk in einen zweiten Hauptspeicherbereich geladen, während die internen Datenstrukturen mit dem Zeitschrittindex n initialisiert werden und ein entsprechender Szenengraph erzeugt wird (Abbildung 5.3).

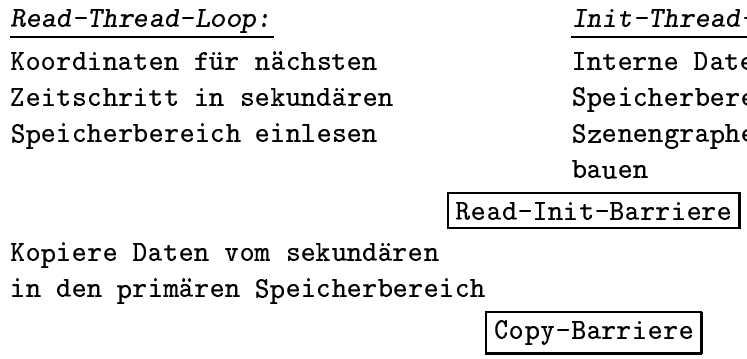


Abbildung 5.3: Dieser Pseudocode zeigt, wie die beiden Threads beim parallelen Einlesen ablaufen.

5.1.2.3 Einlese-Module für weitere Dateiformate

Da im Rahmen dieser Arbeit für Problemstellungen aus verschiedenen Bereichen nach Lösungen gesucht wurde, war die Unterstützung weiterer Datenformate notwendig. Für folgende Dateiformate sind eigene Einlesemodule entwickelt und implementiert worden:

- LS-DYNA Ergebnisdateien
  Mit dem Modul ReadLSDYNA ist es möglich, die interne Datenstruktur des Prototypen aus Ergebnisdaten, die mit dem Simulationsprogramm *LS-DYNA* der *Livermore Software Technology Coorporation* berechnet und in einem binären Dateiformat abgespeichert wurden, zu initialisieren. Das Format ähnelt dem der PAM-CRASH Ergebnisdateien.

- IDEAS Universal Dateien
  Um den Einsatz der Hardware-basierten Volumenvisualisierung im Umfeld der Strukturanalyse testen zu können, wurde für das Einlesen von Ergebnisdaten aus einer *PERMAS*-basierten Bauteiloptimierung ReadIDEAS implementiert. Dieses Einlesemodul kann aus den ASCII-basierten IDEAS Universal Dateien die Modellstruktur sowie zeitabhängige Ergebnisdaten herausfiltern und über die interne Datenstruktur der Visualisierung zuführen. Darüber hinaus dient ReadIDEAS dem Einlesen

von Schwingungs- und Akustikdaten aus der linearen Simulation. Das Universal-Dateiformat ist sehr umfangreich, da es den Anspruch hat, Daten aus völlig unterschiedlichen Einsatzgebieten der numerischen Simulation speichern zu können.

5.1.3 Schnittstelle zur Szenengraphbibliothek

Durch eine klare Trennung zwischen der internen Datenstruktur und der Szenengraphbibliothek wird die Möglichkeit gewährleistet, durch den Austausch der Schnittstelle weitere Szenengraphbibliotheken zum Einsatz bringen zu können. Zu diesem Zweck wurde die abstrakte Klasse GeoBase als Schnittstelle entwickelt. Von ihr leiten sich die Klassen GeoInventor und GeoCosmo ab, in denen die internen Objekte in entsprechende Szenengraphknoten umgewandelt werden. Für Cosmo3D-Szenengraphknoten bietet die Klasse csContainer die Möglichkeit, Benutzerdaten direkt am Szenengraphobjekt abzuspeichern. Dieser Mechanismus wird in GeoCosmo zur direkten Rückreferenzierung auf die Instanz eines internen Datenobjektes genutzt.

5.1.4 Parser für selbstdefinierte Dateiformate

Für verschiedene Zwecke wurden Dateiformate auf Basis von LALR(1)-Grammatiken definiert und *Parser*-Module implementiert, die derartige Dateien wieder einlesen können. Alle *Parser*-Module leiten sich von der abstrakten Klasse ParserBase ab. Die spezifizierten Grammatiken werden mit Hilfe des bison++-Compilers in einen C++-basierten Parser umgewandelt. Folgende Module wurden entwickelt:

- EnvFileParser liest Einstellungen wie zum Beispiel Kamerapositionen oder Hintergrundfarbe ein, die im Laufe einer Sitzung vom *crashViewer* abgespeichert wurden. Die verschiedenen Kamerapositionen stehen anschließend im Viewer wieder mit den Tastenkombinationen, für die sie abgespeichert wurden, zur Verfügung.

- GrpFileParser wertet Gruppendateien aus. Diese Dateien werden in der Regel auch vom Benutzer modifiziert und legen eine Hierarchie unter den Fahrzeugbauteilnetzen an, das heißt, mehrere Bauteile können so zu Bauteilgruppen zusammengefasst werden. Die spezifizierte Hierarchie wird in die interne Datenstruktur und damit anschließend auch in den Szenengraphen übernommen, wodurch die Handhabung (zum Beispiel Transformationen) von Bauteilgruppen als Einheit möglich wird.
  Darüber hinaus wurde das Dateiformat dahingehend erweitert, dass in einer Gruppendatei auf Basis von Bauteil-, Element- oder Knoten-IDs angegeben werden kann, welche Teile des Datensatzes visualisiert werden sollen und welche nicht.

- InitFileParser verarbeitet Konfigurationsdateien, die dazu dienen, die Parametereinstellungen im *crashViewer* nach den Wünschen des Anwenders zu modifizieren. Zu den konfigurierbaren Parametern gehören unter anderem die Standardverzeichnispfade für die diversen Dateitypen sowie Parameter zum Setzen von Schweißpunkten oder zum Erstellen von Kraftflussröhren.

- PathListParser ermittelt relative und absolute Datei- beziehungsweise Verzeichnispfade zur Verarbeitung mehrerer Ergebnisdateien bei der Detektion von Instabilitäten (siehe Kapitel 8.4 ab Seite 131).
- SffFileParser wertet die Schnittkraftdaten einer abgespeicherten Kraftflussröhre in einem *Section-Force-File* aus, wodurch die aufwändige Berechnung der Schnittkräfte direkt im Anschluss an die Simulation ohne Benutzerinteraktion geschehen kann und die Ergebnisse zur Visualisierung sofort zur Verfügung stehen.
- TldFileParser liest die Definition eines statischen oder dynamischen Polygonzugs aus einem *Trace-Line-Definition-File* ein, die den Verlauf einer Kraftflussröhre angibt. Diese Dateien werden entweder beim Abspeichern von Kraftflussröhren erzeugt oder können auch direkt vom Anwender mit einem Editor angelegt werden. Die Kraftflussröhren werden im Kapitel 8.3 ab Seite 127 besprochen.

5.1.5 Funktionsmodule

Dieser Abschnitt gibt einen Überblick über die Funktionalitäten, die der Prototyp *crashViewer* zur Verfügung stellt und die aufgrund ihrer Umfänge in späteren Kapiteln näher beschrieben wird. Für jede der nachfolgend aufgelisteten Funktionalitäten wurde ein Modul teilweise bestehend aus mehreren Klassen entwickelt.

- Für das kooperative Arbeiten auf mehreren Rechnern wurde in dem coop-Modul eine CORBA-basierte Übertragung von Events zwischen verschiedenen Viewer-Instanzen entwickelt und eine Bildübertragung via Sockets an Java-Clients integriert (Abschnitt 8.5, Seite 135ff).
- Das genstrip-Modul implementiert die Zusammenfassung benachbarter Primitive (Dreiecke oder Quadrilaterale) zu Streifen und wird ab Seite 89 erläutert.
- Die Reduzierung des Finite-Element-Netzes zu einer geringen Anzahl von Dreiecken dient ebenfalls der Darstellungsbeschleunigung und wird im simplify-Modul (Abschnitt 6.2) vollzogen.
- Das boundVol-Modul (ab Seite 108 beschrieben) stellt die Implementierung einer Bounding-Volume-Hierarchie dar, die das Fahrzeugmodell ab der Bauteilnetzebene weiter unterteilt und der effizienten Ermittlung von minimalen Abständen beziehungsweise der Detektion von Kollisionen dient.
- Im inipen-Modul wird über eine Schnittstelle eine Bibliothek der Firma ESI angesteuert, die anhand der berechneten initialen Kräfte iterativ die Knotenkoordinaten zu nahe aneinander liegender Bauteilnetze modifiziert (siehe Abschnitt 7.1.2).
- Die interaktive Definition und Modifikation von Schweißpunktdaten ist in dem spotwFunc-Modul realisiert worden und wird im Kapitel 7.2 ab Seite 118 vorgestellt.

- Das Modul resample implementiert die Übertragung von skalaren Parametern aus der Tiefziehsimulation auf das Crash-Netz und ermöglicht so zum Beispiel die Berücksichtigung unterschiedlicher Elementdicken innerhalb eines Bauteils in der Crash-Simulation (Kapitel 7.3).

- Die Berechnung von Schnittkräften und die visuelle Aufbereitung in Form von Kraftflussröhren übernimmt das forceTube-Modul, das in Kapitel 8.3 ab Seite 127 näher beleuchtet wird.

- Die Abweichungen in Simulationsergebnissen gleicher Eingabedaten, die zur Visualisierung von Instabilitäten genutzt wird, ermittelt das stability-Modul, das ab Seite 131 erläutert ist.

- Im Rahmen einer Diplomarbeit [72] wurde ein Volumizer-basierter Volumenknoten für Cosmo3D entwickelt. Die dort beschriebene Funktionalität wurde im volume-Modul gekapselt vom Prototypen *crashViewer* zur Akustikvisualisierung und Geometrieoptimierung genutzt.

5.2 Bedienelemente

Für die entwickelte Applikation *crashViewer* stand von Beginn an ein Arbeitsplatzrechner mit moderner Graphik-Hardware als Zielplattform fest. Der Benutzer sollte in der Lage sein, möglichst viele Funktionalitäten direkt im Darstellungsbereich durch Bedienung von Maus und Tastatur anzusteuern. Zusätzlich sollte der Viewer durch die Entwicklung von Interaktionsmechanismen mit Hilfsmitteln der Virtuellen Realität bereichert werden. Die folgenden Abschnitte geben einen Überblick über die implementierte Benutzerschnittstelle.

5.2.1 Eingabemedien

Als Eingabemedien stehen an jedem Arbeitsplatzrechner zunächst eine Tastatur und eine 2D-Maus zur Verfügung. Die Navigation wurde so realisiert, dass die zweidimensionalen Bewegungen bei gedrückter Maustaste in dreidimensionale Transformationen umgesetzt werden:

- linke Maustaste: Die Mausposition wird auf ein virtuelles Ellipsoid projiziert, dessen Mittelpunkt auf der Sichtlinie liegt. Dadurch initialisieren Mausbewegungen eine Rotation um den Mittelpunkt des Ellipsoids und ermöglichen Rotationen um beliebige Achsen im Raum.
- mittlere Maustaste: Die vertikale Mausbewegung wird in eine Translation entlang der Sichtlinie umgewandelt und erlaubt somit bei Verwendung einer perspektivischen Kamera das Zoomen. Bei Orthogonalprojektion wird der Bildausschnitt entsprechend vergrößert beziehungsweise verkleinert.

- rechte Maustaste: Die zweidimensionale Mausbewegung wird in eine Translation parallel zur Projektionsebene umgesetzt.

Es hat sich gezeigt, dass dem Anwender die Navigation erheblich erleichtert wird, indem weitere Hilfsmittel zur Verfügung stehen, mit denen zum Beispiel der Mittelpunkt des virtuellen Rotationsellipsoids festgelegt werden kann. Da die Untersuchung der Strukturmodelle immer wieder eine genaue Inspektion erfordert, wurden Mechanismen integriert, die es dem Ingenieur ermöglichen, über einen Tastenklick die Kamera auf einen konfigurierbaren Abstand an die Struktur heranzufahren oder ganze Bauteilnetze größtmöglich im Darstellungsbereich anzuzeigen. Es wurden zahlreiche tastatur- und dialoggesteuerte Interaktionsmechanismen entwickelt, um die Navigation mit der 2D-Maus zu optimieren.

Die Aktionen, die dem Anwender zur Verfügung gestellt werden, lassen sich in objektbezogene Aktionen, die sich nur auf ein selektiertes Objekt in der Szene auswirken sollen, und globale Aktionen unterteilen. Um einen expliziten Selektionsschritt zu umgehen, wurden die meisten Aktionen über Tastaturkombinationen zugänglich gemacht, so dass mit dem Mauszeiger das Objekt und mit der Tastatur die Aktion ausgewählt werden kann.

Um eine Vielzahl an Aktionen über die Tastatur mit möglichst wenigen Tastenbetätigungen ansteuern zu können und gleichzeitig den Funktionen noch intuitive Tastenkombinationen zuzuordnen, wurde ein Multi-Mode-System implementiert. Die Applikation hat somit in jedem Modus wieder die volle Tastatur für die Belegung mit Aktionen zur Verfügung. Derzeit können über 250 verschiedene Aktionen mit weniger als einem Drittel der Tasten ausgelöst werden. Die Modi setzen sich aus Präfix- und permanenten Modi zusammen. Die Präfix-Modi sind nur für den nächsten Tastendruck aktiv, die permanenten müssen explizit wieder verlassen werden. Ein Überblick über die wichtigsten Modi in *crashViewer*:

- *Main* – Von hier aus können alle anderen permanenten Modi erreicht und Aktionen ausgelöst werden, die sowohl im Pre- als auch im Postprocessing Verwendung finden.
- *Animation* – Steuerung der Zeitschrittanimation bei zeitabhängigen Daten.
- *Camera* – Abspeichern und Wiederherstellen von Blickpunkt/-richtung-Einstellungen
- *Dump* – Informationsausgabe zu ausgewählten Objekten
- *Edit* – Modifizieren von Darstellungseigenschaften
- *Select* – Beschränken der darzustellenden Szene durch Clip-Objekte
- *Spotweld* – Modifikation von Schweißpunktdaten
- *Tube* – Kraftflussvisualisierung mit Röhren
- *View* – Hilfsfunktionen zur Navigation in der Szene

Inzwischen sind durch weitere Forschungsaktivitäten anderer Doktoranden noch zusätzliche Modi hinzugekommen. Durch die Verwendung mehrerer Modi können gleichartige Arbeitsschritte, wie zum Beispiel das Setzen oder Löschen von Verbindungselementen mit gleichen Tasten vollzogen werden. Das hilft dem Anwender, der zusätzlich noch viele andere Anwendungen bedienen muss, sich die wesentlichen Tastenbelegungen einzuprägen.

5.2.2 Hilfsmittel der Virtuellen Realität am Arbeitsplatz

Obwohl inzwischen im Rahmen einer Studienarbeit [52] auch eine Portierung von *crashViewer* für die immersive Visualisierung in einer Cave oder an einer Powerwall entstanden ist, lag der Schwerpunkt für die Entwicklung von Interaktionsmechanismen auf der Bereicherung einer Bildschirm-basierten Version durch Hilfsmittel der Virtuellen Realität.

Abbildung 5.4: Mit diesem Eingabegerät, einer *Space Mouse*, ist es möglich, Translationen entlang beziehungsweise Rotationen um die drei Hauptachsen durchzuführen.

Außer den bereits beschriebenen Eingabemedien Tastatur und 2D-Maus werden ebenfalls die Daten einer *Space Mouse* (Abbildung 5.4) verarbeitet und in eine Transformation mit sechs Freiheitsgraden umgesetzt. Hier wurden zwei verschiedene Modi implementiert:

- Im *Model-Mode* wird die errechnete Transformation auf das Fahrzeugmodell oder auf ein aktuell selektiertes Bauteil angewendet. Das dargestellte Objekt verhält sich so wie der Griff der *Space Mouse* und lässt dadurch die Manipulation der Szene auf intuitive Art und Weise zu.
- Im *Fly-Mode* hingegen steuert der Anwender die Kamera mit Hilfe der *Space Mouse*. In diesem Modus ist es möglich, wie im virtuellen Flug in das Fahrzeugmodell einzutauchen und sich durch die Struktur des Finite-Element-Netzes zu bewegen.

Um die räumliche Wahrnehmung zu erleichtern, wurde ebenfalls ein Stereo-Modus implementiert, in dem jedes Bild zunächst für das linke und anschließend für das rechte Auge dargestellt wird. In diesem Modus wird die Kamera den konfigurierbaren Parametern Augenabstand und Konvergenztiefe entsprechend, wie in Abbildung 5.5 dargestellt, von dem eigentlichen Betrachterpunkt orthogonal abgerückt und rotiert. Damit das menschliche Auge beide Bilder wieder zu einem räumlichen Bild zusammensetzen kann, darf jedes Auge nur das Bild sehen, das für seine Position erzeugt wurde. Für die Stereo-Darstellung am Bildschirm gibt es so genannte *Shutter Glasses* – das sind Brillen, deren Gläser abwechselnd undurchsichtig gemacht werden. Die Synchronisation mit dem am Bildschirm dargestell-

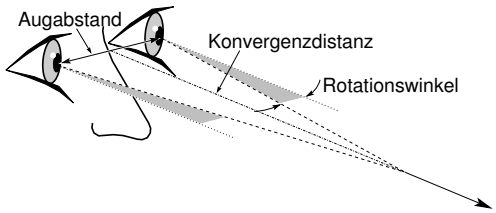


Abbildung 5.5: Im Stereo-Modus wird jedes Bild für beide Augen einzeln dargestellt und mit Hilfe einer aktiven oder passiven Stereo-Brille dem jeweiligen Auge zugeführt. Der Augenabstand und die Konvergenzdistanz lassen sich einstellen, um einen optimalen 3D-Eindruck zu gewährleisten.

ten Bild findet über einen zusätzlichen Infrarotsignalgeber statt. Dieses Verfahren wird auch als aktive Stereoskopie bezeichnet. Demgegenüber werden bei der passiven Stereoskopie die Bilder für das linke und rechte Auge zum Beispiel über zwei Projektoren durch senkrecht zueinander ausgerichtete Polarisationsfilter auf eine Projektionsfläche projiziert. Durch eine Brille mit Polarisationsgläsern werden die Strahlen der beiden Bilder wieder voneinander getrennt und erreichen somit nur eines der beiden Augen. Diese Technik wird eingesetzt, wenn der Einsatz von *Shutter Glasses* aus technischen Gründen oder aufgrund der Zuschaueranzahl nicht möglich ist.

5.2.3 Graphische Benutzerschnittstelle

Die graphische Benutzerschnittstelle des Prototypen *crashViewer* ist X11/Motif-basiert. Das Menü im Hauptfenster wird über einen Mechanismus generiert, der es erlaubt, neue Menüeinträge unterschiedlicher Art durch Hinzufügen einer Zeile zu erzeugen. Dies unterstützt die leichte Erweiterbarkeit der Oberfläche und trägt zusammen mit der modularen Softwarestruktur dazu bei, dass der Prototyp als Rahmen für weitere Forschungsaktivitäten genutzt werden kann.

Viele der Interaktionen bestehen aus Operationen auf dargestellten Objekten. Um die Vorteile einer Menüsteuerung gegenüber der Tastaturbenutzung ausnutzen zu können, gleichzeitig aber die Auswahl des Objektes zu erlauben, auf dem die Aktion ausgeführt werden soll, wurde zusätzlich zu der Hauptmenüleiste ein Kontext-sensitives Popup-Menü implementiert. Es kann über dem Darstellungsbereich geöffnet werden und bietet den Vorteil, dass die im Popup-Menü angewählte Funktion auf dem Objekt ausgeführt werden kann, über dem das Menü geöffnet wurde. Gleichzeitig werden dem Anwender über das Popup-Menü nur die Funktionen angeboten, die auf dem selektierten Objekt und in dem aktuellen Modus ausgeführt werden können.

Parametereingaben und umfangreichere Einstellungen, wie zum Beispiel die Modifikation eines Farbverlaufs zu einem zu spezifizierenden Wertebereich oder die Anpassung von Transferfunktionen für Parametertexturen wurden in separaten Dialogen realisiert.

5.3 Mechanismen zur Datenanalyse

Um die interaktive Datenanalyse zu unterstützen, wurden verschiedene Funktionalitäten entwickelt und im *crashViewer* implementiert.

- **Ausgabe von Informationen**
  Der Anwender kann mit dem Mauszeiger ein Bauteilnetz selektieren und sich Informationen dazu ausgeben lassen. Zusätzlich zum Bauteil wird auch das selektierte Finite-Element und der nächstliegende Netzknoten ermittelt. Die intern abgespeicherten Informationen beschränken sich im Wesentlichen auf die Bauteilbezeichnung und auf Labels in Form einer eindeutigen Ganzzahl für Bauteil, Element und Knoten. Weitergehende Informationen können allerdings über eine Anbindung von *crashViewer* an *CAE-Bench* direkt aus einem PDM-System eingeholt werden [21]. Zusätzlich kann sich der Anwender zu einem spezifizierten Label auch das Bauteil, das Element oder den Knoten anzeigen lassen. Die Kameraparameter werden so initialisiert, dass das entsprechende Objekt bildfüllend angezeigt wird.

- **Aus-/Einblenden von Bauteilen**
  Teile des Datensatzes können interaktiv aus- und wieder eingeblendet werden. Dabei werden die korrespondierenden Szenengraphknoten aus dem Szenengraphen entfernt und auf einem Stapel abgelegt. Die Datenverwaltung und Modifizierung des Szenengraphen ist in der Klasse ReplaceCluster implementiert.
  Für zeitabhängige Daten muss die Modifikation konsistent auch für zugehörige Szenengraphknoten aus den Subszenengraphen der anderen Zeitschritte geschehen (vergleiche Abbildung 4.3, Seite 68). Dazu wird der Pfad des selektierten Objektes in eine Indexliste umgewandelt, die auf jeden Subszenengraphen zum Auffinden des entsprechenden Bauteils angewendet werden kann, da jeder Zeitschritt-Subszenengraph mit gleicher Struktur aufgebaut ist.
  Bauteilnetze können entweder mit der Maus oder anhand ihrer Labels selektiert werden. Die dargestellten Bauteile können durch die nicht dargestellten ausgetauscht werden. Darüber hinaus können zu einem ausgewählten Bauteil alle über eine Bauteilverbindung (zum Beispiel Schweißpunkte) angebundenen Bauteile, die derzeit ausgeblendet sind, eingeblendet werden. Dieser Mechanismus dient der Validierung des Eingabedatensatzes im Preprocessing und hilft dabei, nicht angebundene Bauteile zu identifizieren.

- **Kameraeinstellungen**
  Position und Ausrichtung der Kamera können während einer Sitzung auf verschiedenen Tasten abgespeichert und später wieder abgerufen werden. Diese Tasturbelegung mit den Kameraeinstellungen kann auch abgespeichert werden, um sie für eine spätere Sitzung oder in einem zweiten *crashViewer* mit einem anderen Datensatz zu verwenden.

- **Bilder abspeichern**
  Das aktuell dargestellte Bild kann per Tastendruck abgespeichert werden, um zum Beispiel einen Ausdruck zu erstellen. Für hochauflösende Bilder, deren Format der Anwender frei wählen kann, wird die Szene mehrmals mit Hilfe einer csFrustumCamera gezeichnet, die es erlaubt, das eigentliche View-Frustum in beliebig viele asymmetrische View-Frusta zu unterteilen. Die erzeugten Zwischenbilder werden schließlich zu dem Ergebnisbild mit der spezifizierten Auflösung zusammengesetzt.

- **Animation mehrerer Zeitschritte**
  Wird für das Postprocessing ein Szenengraph mit mehreren Zeitschritten aufgebaut, so haben alle Subszenengraphen als gemeinsamen Vaterknoten einen csSwitch-Knoten. Mit Hilfe dieses csSwitch kann zwischen der Darstellung der einzelnen Zeitschritt-Subszenengraphen umgeschaltet werden. Eine Animation, bei der nacheinander alle Zeitschritte dargestellt werden, kann vorwärts oder rückwärts in einer Schleife laufen oder im *Swing*-Modus hin- und herpendeln.
  Für die detaillierte Beobachtung von Teilstrukturen muss verhindert werden, dass sich die beobachtete Teilstruktur durch die Zeitschrittanimation aus dem Darstellungsbereich bewegt; dazu kann der Anwender die Kamera mit dem *Locking*-Mechanismus an den Schwerpunkt eines Bauteils oder direkt an einen Netzknoten binden. Der Differenzvektor des sich verändernden Bezugspunktes wird auf die Kameraposition übertragen, wodurch der Bezugspunkt stets auf den gleichen Bildpunkt abgebildet wird.

- **Filmgenerierung**
  Zur Erstellung eines digitalen Videos können im *Record*-Modus alle erzeugten Einzelbilder abgespeichert und in einem Nachverarbeitungsschritt in eine Movie-Datei umgewandelt werden. Die Filmgenerierung kann auch offline durch ein Batch-Programm geschehen – siehe dazu Abschnitt 8.6.

- **Clipping-Mechanismen**
  Zusätzlich zu der Möglichkeit, die Darstellung des Fahrzeugmodells gleich beim Laden durch die Koordinaten-basierte Angabe eines Begrenzungsquaders einzuschränken, wurden eine interaktive, frei positionierbare Schnittebene, zwei semi-interaktive Selektionsobjekte und eine Textur-basierte Clipping-Technik entwickelt. Letztere wird im Zusammenhang mit der Distanzvisualisierung in Kapitel 7.1.3 detailliert erläutert. Die Schnittebene wurde in Form eines neuen universal verwendbaren Cosmo3D-Szenengraphknoten csClipGroup implementiert (Abschnitt 4.3.1). Bei den Selektionsobjekten handelt es sich um einen hauptachsenparallelen Quader und eine Kugel, deren Position und Größe interaktiv verändert werden kann. Hat der Anwender die Positionierung abgeschlossen, wird für alle Netzknoten bestimmt, ob sie sich inner- oder außerhalb des Selektionsobjektes befinden. Geometrien, die keinen Netzknoten innerhalb des Selektionsobjektes beinhalten, werden ausgeblendet. Für die Geometrien, die vom Selektionsobjekt geschnitten werden, wird ein neuer Szenengraphknoten

angelegt, der zwar das gleiche csCoordSet referenziert, für den jedoch nur die Elemente ins csIndexSet aufgenommen werden, die mindestens einen Netzknoten innerhalb des Selektionsobjektes haben. Schließlich ist es dem Anwender möglich, eine 2D-Box aufzuziehen, die durch Einbeziehung der Viewing-Transformation ein 3D-Volumen in Form eines Pyramidenstumpfes beschreibt. Mit Hilfes dieses leicht zu positionierenden Clip-Volumens können komplexe Modellstrukturen schnell auf den Bereich des momentanen Interesses eingeschränkt werden.

In diesem Kapitel wurde die grundlegende Softwarestruktur dargelegt und auf allgemeine Funktionalitäten hingewiesen, die für den entwicklungsbegleitenden Einsatz des Prototypen in der Kooperation mit den BMW-Ingenieuren unverzichtbar waren. Die nachfolgenden Kapitel gehen auf spezielle Ansätze zur Beschleunigung der Bildsynthese sowie auf Interaktions- und Visualisierungsmethoden für die Vor- und Nachbearbeitung von Strukturmechanikdaten ein.

Kapitel 6

Verfahren zur Darstellungsbeschleunigung

Die fortschreitende Entwicklung immer schnellerer Rechner und die wachsenden Ansprüche an Simulationsergebnisse ziehen bei der Erstellung des Fahrzeugmodells immer feinere Auflösungen des Finite-Element-Netzes nach sich. In den letzten zehn Jahren ist die Modellgröße von 50 000 Elementen im Jahre 1992 auf inzwischen über 800 000 Elemente stetig gewachsen. Diese Modellgrößen erfordern die Anwendung verschiedener Optimierungsverfahren zur Beschleunigung der Bildsynthese bereits in einem Vorverarbeitungsschritt, damit die darzustellende Geometrie durch möglichst wenig Daten beschrieben und damit das Graphiksubsystem entlastet werden kann. Dazu werden drei Ansätze verfolgt, die durch Reorganisation der Primitive, durch Ausdünnung entbehrlicher Geometrieinformation und durch Einbeziehung anderer Graphikkomponenten die Transformationslast des Geometrie-Subsystems verringern.

6.1 Streifengenerierung benachbarter Primitive

Die Vorteile der Geometriedaten-Optimierung wurden bereits in Kapitel 3.2.3, Seite 47 f erläutert. Evans et al. stellen in [17] verschiedene Strategien zur Optimierung von Dreiecksstreifen vor. Da sie insbesondere für Polygonnetze mit vielen Quadrilateralen gute Ergebnisse liefern, wurden diese Verfahren hinsichtlich ihrer Eignung für Strukturmechanikdaten näher untersucht. Die Autoren diskutieren verschiedene Ansätze, um die Streifengenerierung zu optimieren:

- Triangulierung von Polygonen bestehend aus mehr als drei Eckpunkten:
  Es wird zwischen der *statischen* und der *dynamischen* Triangulierung unterschieden. Bei statischer Triangulierung wird das gesamte Polygonnetz vor Beginn der Pfadsuche in Dreiecke zerlegt. Die dynamische Triangulierung bietet während der Pfadsuche mehr Flexibilität bei der Richtungswahl, um einen Streifen fortsetzen zu können.

- Strategien beim Fortschreiten an Verzweigungsstellen:
  Hier wird zwischen fünf verschiedenen Vorgehensweisen differenziert, die für das Polygon, in das der Pfad aktuell mündet, in Abhängigkeit des Nachbarschaftsgrades entscheiden, über welche Kante der Pfad das Polygon wieder verlässt.
- Sonderbehandlung rechteckiger Regionen von Quadrilateralen — *Patchification*:
  Bevor die oben aufgeführten lokalen Strategien angewendet werden, wird das Netz zunächst unter Vorgabe einer Mindestlänge in zwei Richtungen global nach aufeinanderfolgenden Quadrilateralen abgesucht, welche anschließend in einem *Full-patch*-Streifen oder in mehreren parallelen Zeilen- beziehungsweise Spalten-Streifen zusammengefasst werden.

Basierend auf diesen Ansätzen wurde im Rahmen einer Diplomarbeit [32] ein Streifengenerierungsmodul entwickelt, das aus den Bauteilnetzen der Fahrzeugmodelle unter Berücksichtigung des Schattierungsmodells optimale Streifen generiert. Sofern die Geometrie Gouraud-schattiert dargestellt werden soll, darf für jeden Knoten innerhalb eines Streifens nur eine Normale definiert werden; deshalb ist es nicht möglich, Streifen über *Gouraud-Kanten* hinweg fortzusetzen, da die Knoten an solchen Kanten mehrere Normalen besitzen.

Die Umwandlung der Finite-Element-Netze von Fahrzeugmodellen für die Crash-Simulation in Dreiecksstreifen hat zur Folge, dass es anschließend unmöglich ist, in einer Gitterliniendarstellung die Netzstruktur des Modells zu erkennen, die sich überwiegend aus vierseitigen Elementen zusammensetzt, da diese in jeweils zwei Dreiecke unterteilt wurden. Nur unter Verwendung einer *Wireframe-Textur* (Abschnitt 6.3.2) kann auch noch auf den Dreiecksstreifen die ursprüngliche Netzstruktur visualisiert werden.

Damit die Strukturinformation auch in einer Drahtgitterdarstellung nicht verloren geht, wurde das Streifengenerierungsmodul dahingehend erweitert, dass es auch Quadrilateralstreifen erzeugen kann. Da eine Richtungsänderung jedoch im Quadrilateralstreifen im Gegensatz zum Dreiecksstreifen aufwändiger ist, als einen neuen Streifen zu beginnen (siehe Abbildung 6.1), schied die Verwendung von Full-Patch-Streifen aus. Die Tatsache, dass

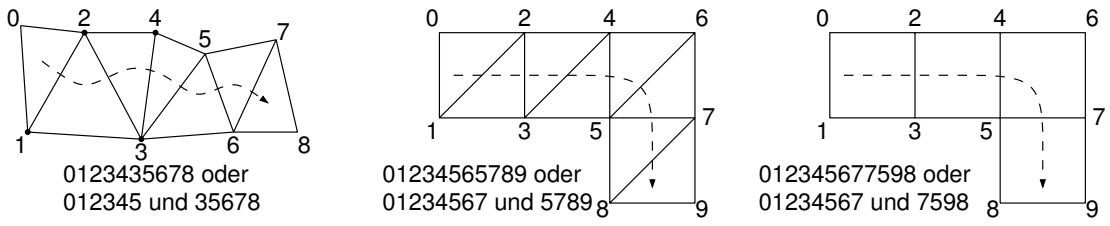
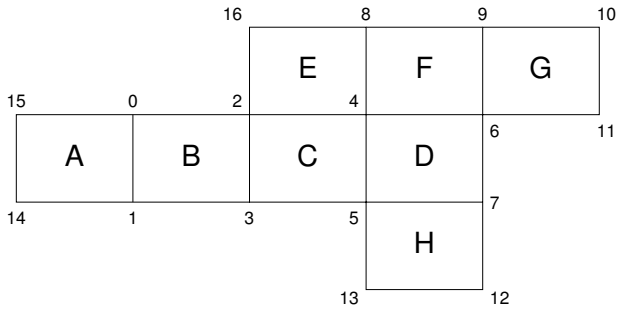


Abbildung 6.1: Das linke Bild zeigt, dass es günstiger ist, ein Dreiecksstreifen durch einen Kantenrichtungswechsel fortzusetzen als zwei separate Dreiecksstreifen zu generieren. Die beiden rechten Bilder vergleichen die „Wendigkeit“ von Dreiecks- und Quadrilateralstreifen: Um die Streifenrichtung durch Referenzierung möglichst weniger Knoten zu ändern, können auch zwei separate Quadrilateralstreifen erzeugt werden. Das „Abbiegen“ benötigt im Dreiecksstreifen nur einen, im Quadrilateralstreifen jedoch zwei zusätzliche Knoten.



Adjazenzliste
A: { B(0,1) }
B: { A(0,1) ; C(2,3) }
C: { B(2,3) ; D(4,5) ; E(2,4) }
D: { C(4,5) ; F(4,6) ; H(5,7) }
E: { C(2,4) ; F(4,8) }
F: { D(4,6) ; E(4,8) ; G(6,9) }
G: { F(6,9) }
H: { D(5,7) }

Abbildung 6.2: Die *Adjazenzliste* hält für jedes Polygon die Nachbarschaftsinformation, bestehend aus Verweisen auf Nachbarpolygone und die gemeinsam genutzte Kante.

die Bauteilnetze der Fahrzeugmodelle überwiegend eine regelmäßige Netzstruktur aufweisen, führte zur Entwicklung des im folgenden Abschnitt vorgestellten *Bandification*-Algorithmus.

6.1.1 Datenstrukturen und Algorithmus

Eine effiziente Pfadsuche wird durch temporär aufgebaute Nachbarschaftsinformationen auf Polygonen ermöglicht. Eine tragende Rolle spielt dabei die *Adjazenzliste* eines jeden Polygons; in ihr werden Verweise auf Nachbarpolygone und die gemeinsam genutzte Kante gespeichert (Abbildung 6.2).

Der *Bandification*-Algorithmus (Abbildung 6.3) sucht ausgehend von einem Startquadrilateral in beiden Dimensionen nach Streifen maximaler Länge, die eine vorgegebene

```
Markiere alle Quadrilaterale als 'aktiv'
while (Quadrilateralstreifen mit Länge >= minLänge gefunden) {
  Markiere alle 'aktiven' Quadrilaterale als 'verfügbar'
  Leere maxStreifenListe
  for (alle Quadrilaterale i, die als 'aktiv' UND 'verfügbar' markiert sind) {
    aktMaxStreifen := längster Streifen ausgehend vom Quadrilateral i
    if (aktMaxStreifen.länge() >= minLänge) {
      Markiere alle Quadrilaterale in aktMaxStreifen als 'nicht verfügbar'
      Füge aktMaxStreifen in die sortierte maxStreifenListe ein
    }
    else
      Markiere Quadrilateral i als 'nicht aktiv'
  }
  while (maxStreifenListe enthält noch Streifen) {
    aktMaxStreifen := längster Streifen der maxStreifenListe
    if (aktMaxStreifen.länge() ist noch unverändert)
      Markiere alle Quadrilaterale von aktMaxStreifen als 'nicht aktiv'
    else
      break
  }
}
```

Abbildung 6.3: Pseudocode für die Vorgehensweise im *Bandification*-Algorithmus.

Mindestlänge nicht unterschreiten dürfen, und fügt sie in die Liste potenzieller Quadrilateralstreifen ein. Bei der Pfadsuche werden Streifenrichtungswechsel ausgeschlossen, das aktuelle Quadrilateral wird also stets über die gegenüberliegende Kante zum benachbarten Quadrilateral verlassen. Da es für die traversierten Quadrilaterale keinen längeren Strei-

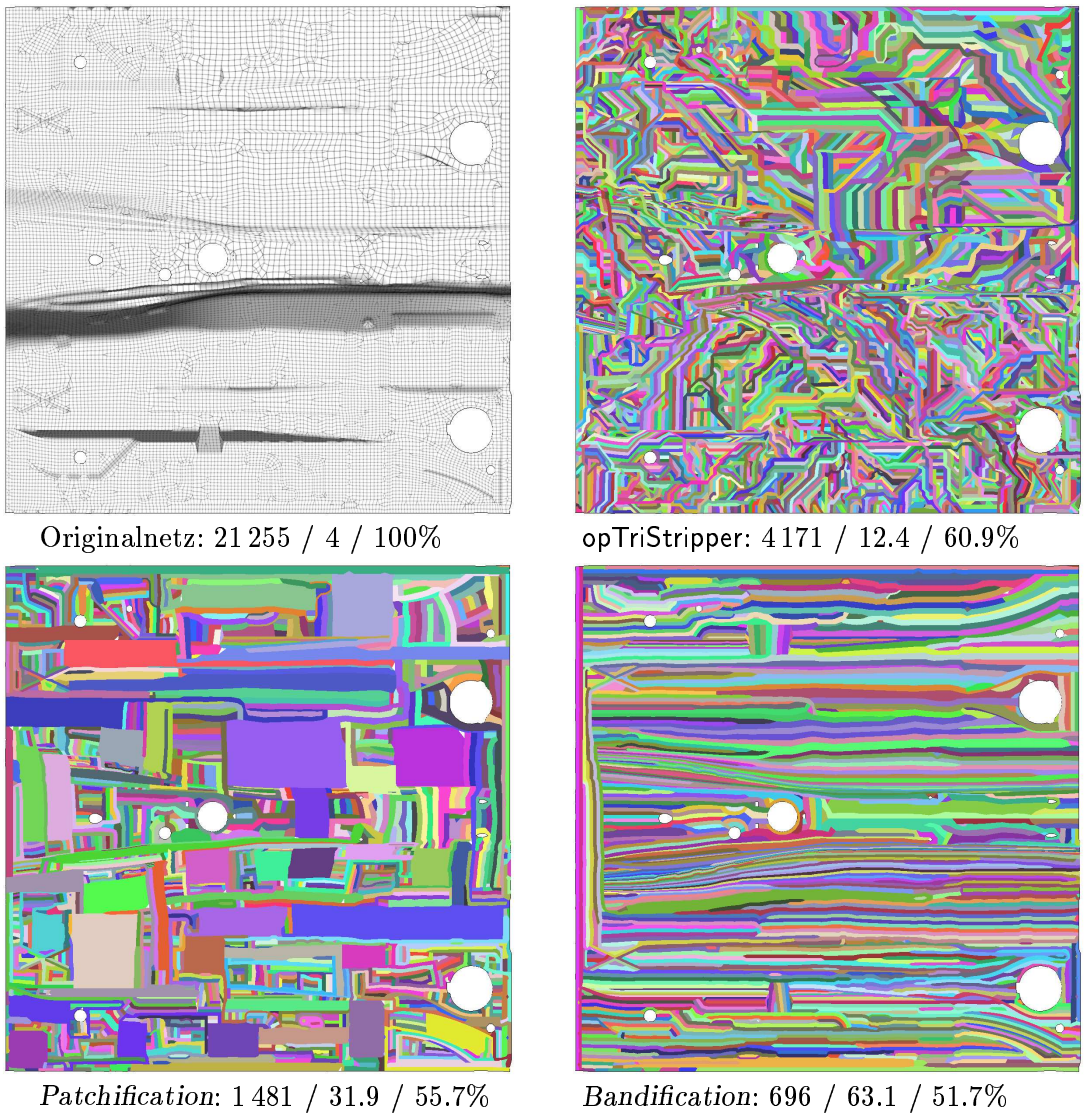


Abbildung 6.4: Die oben links dargestellte Bodenplatte wurde jeweils mit dem opTriStripper, dem *Patchification*- beziehungsweise dem *Bandification*-Algorithmus bearbeitet. Die Zahlen geben die Anzahl der Primitive, die durchschnittliche Strip-Länge sowie den Anteil der Knoten gegenüber einer Darstellung mit separaten Quadrilateralen an. Um die Unterschiede der Partitionierung zu verdeutlichen, wurde für die Gouraud-Kanten-Detektion ein Grenzwinkel von 60° gewählt und die Beleuchtung für die Streifendarstellung abgeschaltet.

fen in dieser Ausdehnungsrichtung gibt, werden sie aus der Liste der Startquadrilaterale für die entsprechende Dimension entfernt. Schließlich werden absteigender Länge nach die potenziellen Quadrilateralstreifen erzeugt, solange sie noch ihre ursprüngliche Länge haben, also noch nicht durch vorher erzeugte Streifen gekreuzt wurden. Wird der Algorithmus zusätzlich noch mit absteigender Mindestlänge durchlaufen, so enthält die Liste potenzieller Bänder, hervorgerufen durch die regelmäßige Netzstruktur, überwiegend parallele Bänder. Abbildung 6.4 veranschaulicht die Arbeitsweise des *Bandification*-Algorithmus anhand einer Gegenüberstellung zur *Patchification* und zu den Ergebnissen von opTriStripper aus OpenGL Optimizer.

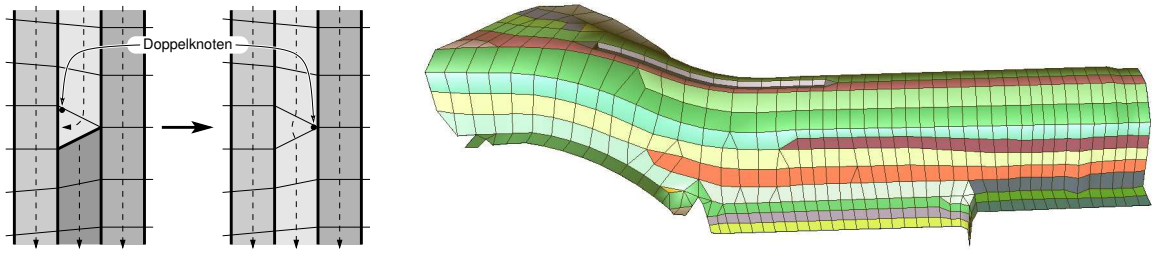


Abbildung 6.5: Eine dynamische Zuweisung, welcher der drei Eckpunkte in dreiseitigen Elementen dupliziert werden soll, ermöglicht einen topologischen Richtungswechsel, durch den die Streifenlänge maximiert werden kann. Im Beispiel rechts werden die „Doppelknoten“ stets so gewählt, dass maximal lange Streifen entstehen.

Da die Finite-Element-Struktur eines Bauteils neben Quadrilateralen auch Dreiecke enthalten können, das hybride Netz jedoch durch nur einen csGeometry-Knoten repräsentiert werden soll, werden dreiseitige Schalenelemente durch degenerierte Quadrilaterale dargestellt; das heißt, einer der drei Netzknoten wird zweimal referenziert. Der „Doppelknoten“ ist in den Finite-Element-Daten willkürlich festgelegt. Bei der Quadrilateralstreifenverfolgung führt ein Doppelknoten häufig zum „Abbiegen“ des Streifens (Abbildung 6.5, links), wodurch der Streifen gegebenenfalls nicht fortgesetzt werden kann, da benachbarte Elemente bereits einem anderen Streifen zugeordnet wurden. Wird der andere Knoten der Eintrittskante, über die das dreiseitige Element erreicht wurde, als Doppelknoten verwendet, kann der Quadrilateralstreifen, wie skizziert, fortgesetzt werden. Dieses Vorgehen entspricht einem *Swap* bei der Dreiecksstreifengenerierung. Die dynamische Topologieänderung der dreiseitigen Elemente ermöglicht also, den Streifen auch dann weiterzuverfolgen, wenn die der Eintrittskante gegenüberliegende Kante kollabiert ist oder das darauffolgende Quadrilateral bereits von einem anderen Streifen verbraucht wurde.

Während Quadrilaterale sowohl als einzelnes Primitiv als auch als Quadrilateralstreifen unter OpenGL unterstützt werden, enthält die Cosmo3D-Szenengraphbibliothek nur ein csQuadSet-Objekt, das mehrere einzelne Quadrilaterale repräsentiert. Für die Repräsentation von Quadrilateralstreifen wurde csQuadStripSet als neuer Szenengraphknoten implementiert. Er bietet die gleiche Schnittstelle wie sein Pendant csTriStripSet.

6.1.2 Resultate

Im Hinblick auf die besonderen Anforderungen bei der Visualisierung von Finite-Element-Netzen wurde ein Quadrilateralstreifengenerierer entwickelt, durch den die Bildwiederholrate deutlich gesteigert werden kann, ohne dass dabei Modelldetails verloren gehen. Eine Elementzerlegung in Dreiecke konnte dadurch umgangen werden.

Die Resultate der Streifengenerierung zeigen, dass sich die Anzahl der zu transformierenden Knoten für N Quadrilaterale durch Reorganisation der Primitive bereits erheblich reduzieren lässt. Das Optimum von $\frac{N+1}{2N} > \frac{1}{2}$ wird zwar kaum erreicht, da die Elemente jeder Teilstruktur im Allgemeinen von mehr als einem Streifen zusammengefasst werden.

Um einen Anhaltspunkt für die Bewertung der Streifenpartitionierung und die dadurch verminderte Referenzierung von Netzkonten zu haben, werden die Ergebnisse des neu entwickelten Quadrilateralstreifengenerierers QuadStripper denen des in OpenGL Optimizer enthaltenen Dreiecksstreifengenerierers opTriStripper gegenübergestellt (Abbildung 6.6).

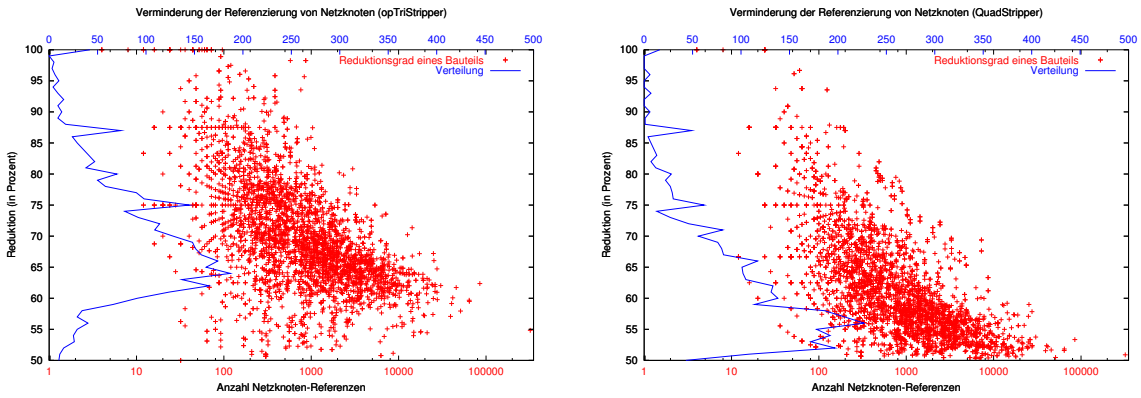


Abbildung 6.6: Die Graphik zeigt anhand von 3 247 Bauteilen unterschiedlicher Größe und Komplexität, auf wieviel Prozent (y-Achse) der ursprünglich notwendigen Knotenreferenzen (x-Achse, unten) das Bauteilnetz durch Quadrilateralstreifengenerierung reduziert werden konnte. Die Kurve veranschaulicht, wie häufig welcher Reduktionsgrad[1] erreicht wurde. Im Durchschnitt wurde eine Reduktion auf 63.5% (opTriStripper) beziehungsweise auf 54 % (QuadStripper) erreicht.

Im Vergleich der Ergebnisse fällt sofort auf, dass die erzeugten Dreiecksstreifen im Allgemeinen kürzer sind, obwohl durch Hinzufügen von *Swap*-Knoten Dreiecksstreifen eine größere Wendigkeit vorbehalten ist. Ausgehend von den gleichen Testdaten erreicht opTriStripper eine Reduktionsrate[1] von knapp 37%, während der QuadStripper trotz Hinzunahme leerer Dreiecke für die dreiseitigen Elemente eine Reduktion um 46% erreicht.

[1] Begriffsdefinition: Der *Reduktionsgrad* bezeichnet hier den Anteil, der nach der Reduktion bestehen bleibt. Die *Reduktionsrate* benennt demgegenüber den Anteil, um den reduziert wurde.
Reduktionsgrad = 100% − *Reduktionsrate*

Da von Cosmo3D / OpenGL Optimizer Version 1.2 die *Vertex Arrays* von OpenGL 1.1 nicht genutzt wurden, machte sich die Streifengenerierung mit einem Beschleunigungsfaktor von 4-5 für die Darstellung von Gesamtfahrzeugmodellen sehr stark bemerkbar. Seit der Version 1.3 werden für indizierte Geometrien *Vertex Arrays* verwendet, sofern für Normalen, Farben und Texturkoordinaten keine oder die gleichen Indexfelder benutzt werden wie für die Punktkoordinaten. Dadurch werden die Geometriedaten nur einmal transformiert und kommen anschließend durch die Indizierung mehrfach zur Anwendung. Dennoch ließ sich die Bildwiederholrate für ein Modell mit mehr als einer halben Million Dreiecken um den Faktor 1.7 erhöhen.

Um für eine weitergehende Darstellungsbeschleunigung noch höhere Reduktionsraten zu erreichen, muss ein anderes Verfahren angewendet werden: die Geometriesimplifizierung.

6.2 Simplifizierung

Für die Simplifizierung von Polygonnetzen gibt es, wie in Abschnitt 3.2.3, Seite 48 beschrieben, eine Vielzahl von Ansätzen. Eine Klasse der dort aufgeführten Reduktionsalgorithmen geht bei der Simplifizierung iterativ vor, das heißt, Schritt für Schritt werden einzelne Knoten aus dem Originalnetz entfernt. Bei der Netzmodifikation kommen *topologische* und *geometrische* Operatoren zum Einsatz; Topologische Operatoren verändern die Netztopologie ohne die Knotenkoordinaten zu modifizieren. Geometrische Operatoren verschieben Netzknoten und modifizieren dadurch die umliegenden Flächenprimitive, ohne jedoch die Topologie zu verändern. Letztere werden bei der Simplifizierung nur im Zusammenspiel mit topologischen Operatoren eingesetzt. Ein detaillierter Überblick wird in [10] gegeben. Beispiele für Operatoren, die bei der iterativen Dreiecksreduktion eingesetzt werden, sind:

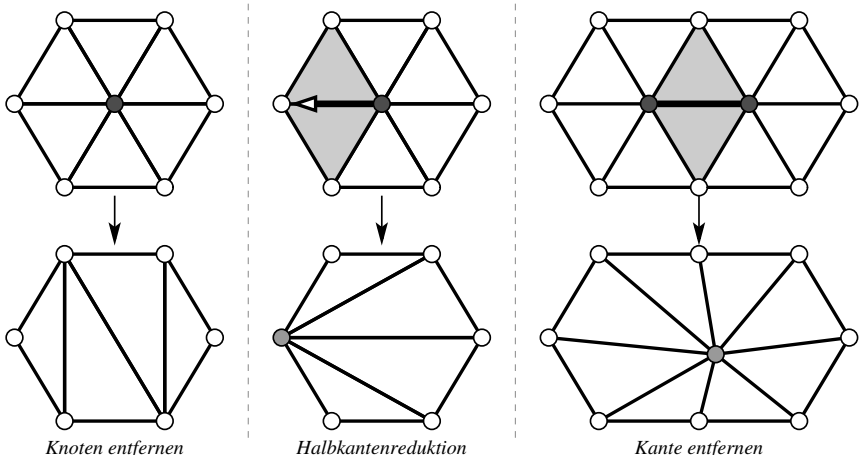


Abbildung 6.7: Drei Beispiele für Operatoren, die bei der iterativen Netzsimplifizierung zum Einsatz kommen. Der Operator „Kante entfernen“ (rechts) beinhaltet neben einem topologischen auch einen geometrischen Operator.

- *Knoten entfernen*: Durch Entfernen eines Netzknotens entsteht ein Polygon, das durch die Nachbarknoten definiert wird, die zuvor mit dem entfernten Knoten über eine Kante verbunden waren. Das entstandene Polygon wird unter Verwendung der Nachbarknoten neu trianguliert.

- *Halbkantenreduktion*: Hierbei wird der zu eliminierende Netzknoten entlang einer seiner Kanten in einen seiner Nachbarknoten verschoben. Durch die Verschmelzung werden die an diese Kante angrenzenden Dreiecke entfernt.

- *Kante entfernen*: Beim Entfernen einer Kante werden zunächst die Kantenendpunkte miteinander verschmolzen; anschließend wird der zusammengefallene Punkt jedoch durch Veränderung seiner Koordinaten so verschoben, dass die Teilfläche, die vor dem Reduktionsschritt bestand, möglichst gut approximiert wird.

Während die beiden erstgenannten Operatoren nur die Topologie modifizieren, setzt sich der Operator *Kante entfernen* aus einem topologischen und einem geometrischen Operator zusammen. Die schrittweise Dreiecksdezimierung wird solange durchgeführt, bis ein vorgegebenes Abbruchkriterium erfüllt ist.

Der Einsatz in der Visualisierung großer, zeitabhängiger Finite-Element-Netze beim Pre- und Postprocessing von Strukturmechaniksimulation bringt zwei Einschränkungen mit sich:

- Das Originalnetz muss weiterhin verfügbar bleiben.
  Da die Berechnungsingenieure insbesondere an der Netzstruktur der visualisierten Fahrzeugmodelle interessiert sind, darf die Originalgeometrie nur temporär durch eine grobe, angenäherte Geometrie ersetzt werden. Das hat zur Folge, dass sowohl die Daten für die Originalgeometrie als auch die für das reduzierte Modell im Speicher gehalten werden müssen.

- Minimaler Speicheraufwand kann nur unter Anwendung topologischer Operatoren erreicht werden.

Abbildung 6.8: Durch die Halbkantenreduktion $0 \rightarrow 1$ fallen die Dreiecke $\triangle 012$ und $\triangle 061$ weg. Der Reduktionsschritt ist allerdings nur erlaubt, sofern der einseitige Hausdorff-Abstand (vergleiche Abbildung 3.8, Seite 50) $d = \left\| 0, 0' \right\|$ des entfernten Knotens 0 zu dessen Projektionspunkt $0'$ im Dreieck $\triangle 145$ den vorgegebenen Toleranzwert nicht überschreitet.

Geometrische Operatoren erfordern das Speichern neuer Koordinaten. Da das Originalnetz aus oben genanntem Grund sowieso im Speicher gehalten werden muss, erlaubt die ausschließliche Anwendung topologischer Operatoren das Wiederverwerten der Koordinatendaten, indem lediglich die neue Topologie gespeichert und über zusätzliche Indizes auf die ursprünglichen Koordinatendaten zugegriffen wird.

Zur Optimierung der Bildwiederholrate während der Benutzerinteraktion wurde im Prototypen *crashViewer* ein Reduktionsalgorithmus implementiert. Er wendet den Operator Halbkantenreduktion solange iterativ an, bis kein Netzknoten mehr entfernt werden kann, ohne dass dadurch die Fehlertoleranz überschritten würde. Als Fehlermaß wird der einseitige Hausdorff-Abstand d, gemessen von einem entfernten Netzknoten 0 zur entstandenen reduzierten Fläche, herangezogen (Abbildung 6.8). Der Toleranzwert kann in *crashViewer* entweder absolut oder relativ zur Bounding-Box-Diagonale des zu simplifizierenden Bauteilnetzes als maximaler Hausdorff-Abstand angegeben werden. Randknoten können nur entlang des Randes verschoben werden, sofern das Abstandskriterium auch zur neu entstehenden Randkante eingehalten wird.

Der Algorithmus (Pseudocode siehe Abbildung 6.9) wurde in Anlehnung der in [10] vorgestellten Ergebnisse auf die Bedürfnisse der zugrunde liegenden Strukturmechanikdaten angepasst und in dem Modul HECSimplify implementiert. Eine detaillierte Beschreibung findet sich in Kapitel 5 oben genannter Arbeit. Im folgenden wird er dem von OpenGL Optimizer bereitgestellten *Successive Relaxation*-Algorithmus gegenübergestellt.

```
D := maximal zulässiger einseitiger Hausdorff-Abstand (Toleranzwert)
Q := Prioritätswarteschlange der als nächstes zu reduzierenden Knoten (leer)

wandle Quadrilateral- in Dreiecksnetz um
for (alle Knoten v des Dreiecksnetzes)
   d := bewerte beste potenzielle Halbkantenkontraktion [v,w] für v
   if (Kontraktion möglich  UND  d <= D)
      füge (d,[v,w]) an entsprechender Stelle in Q ein

while (Warteschlange Q nicht leer)
   hole nächsten und gleichzeitig besten Vorschlag [v,w] aus Q
   führe Halbkantenkontraktion [v,w] durch
   for (alle Knoten v' im 1-Ring von v)
      d := bewerte beste potenzielle Halbkantenkontraktion [v',w'] für v' neu
      aktualisiere Q durch (d,[v',w'])
```

Abbildung 6.9: Pseudocode der auf Halbkantenreduktion basierenden Simplifizierung.

6.2.1 *Successive Relaxation*-Algorithmus

Die Graphikbibliothek OpenGL Optimizer bietet zwei Verfahren zur Generierung gröberer Level-of-Detail-Geometrien an. Der *Spatial Lattice Simplifier* basiert auf dem *Vertex Clustering*-Ansatz von Rossignac [58], ist zwar schnell, aber aus den in Abschnitt 3.2.3 genannten Gründen für die Generierung eines Interaktionsmodells ungeeignet. Der in opSRASimplify implementierte *Successive Relaxation*-Algorithmus liefert demgegenüber wesentlich bessere Ergebnisse und ermöglicht die Weiterverwendung der Koordinaten des

Originalnetzes. Anhand der Kostenfunktion $f(v) = w_0 \cdot d(v) + w_1 \cdot \delta N_1(v) + w_2 \cdot \delta N_2(v)$ wird entschieden, welcher Knoten v als nächstes aus dem Netz entfernt wird. Dabei entsprechen w_0, w_1 und w_2 Gewichtungsfaktoren für die Fehlermaße, $d(v)$ dem einseitigen Hausdorff-Abstand, $\delta N_1(v)$ der gemittelten Abweichung der Dreiecksnormalen im 1-Ring von v und $\delta N_2(v)$ der maximalen Abweichung zu den Knotennormalen auf dem 1-Ring von v. Als Reduktionsoperator wird *Knoten entfernen* verwendet.

Da die Simplifizierungsmodule von OpenGL Optimizer für Erzeugung von Level-of-Detail-Geometrien ausgelegt sind, erwarten sie als Eingabeparameter die Anzahl oder den Anteil der zu reduzierenden Knoten. Es ist nicht ohne Weiteres möglich, eine Fehlertoleranz vorzugeben, die bei der Generierung eines maximal reduzierten Dreiecksnetzes als Abbruchkriterium dient.

6.2.2 *Successive Relaxation* versus Halbkantenreduktion

Ein Vergleich mit dem von OpenGL Optimizer zur Verfügung gestellten `opSRASimplify` lässt sich nur unter Einbezug der resultierenden Netzqualität vornehmen. In [32] werden die Kontrollparameter für die Simplifizierung mit opSRASimplify durch eine so genannte *Orakel*-Funktion vorgegeben, die verhindern soll, dass ein vorgegebenes Fehlermaß überschritten wird. Allerdings kann kein Einfluss auf die Reduktionsreihenfolge genommen werden. Das Seitenverhältnis der aus *Successive Relaxation* resultierenden Dreiecke ist größtenteils ungünstig: es entstehen lange, schmale Dreiecke, die zu auffälligen Schattierungseffekten führen (Abbildung 6.10, links). Der im Prototypen implementierte Halbkantenreduzierer HECSimplify erreicht bei noch höheren Reduktionsraten eine bessere Netzqualität.

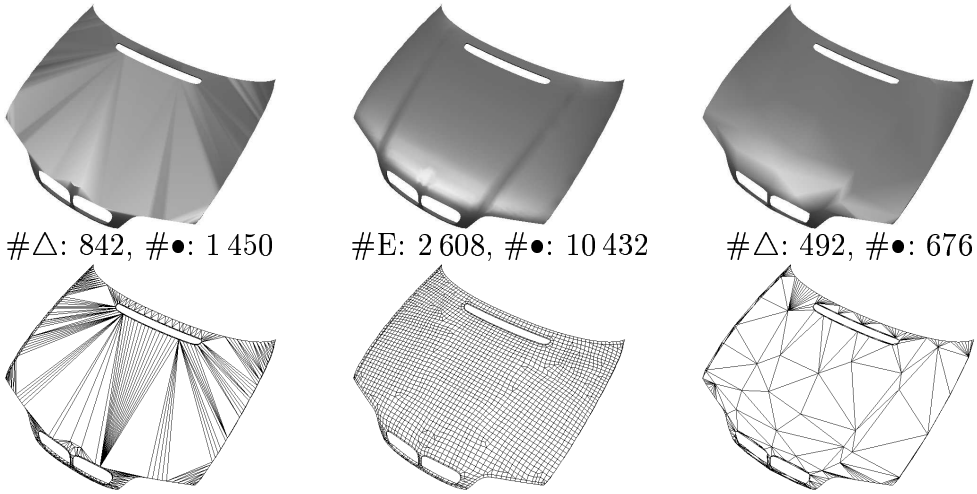


Abbildung 6.10: Die Gegenüberstellung der erzielten Ergebnisse mit dem opSRASimplify (links) und der Halbkantenreduktion (rechts) zu der Originalgeometrie (Mitte) zeigt, dass eine bessere Qualität der Dreiecke durch das implementierte Modul HECSimplify erzielt werden konnte; zudem wurde hier noch zusätzlich eine um ∼9,5% höhere Reduktionsrate erreicht.

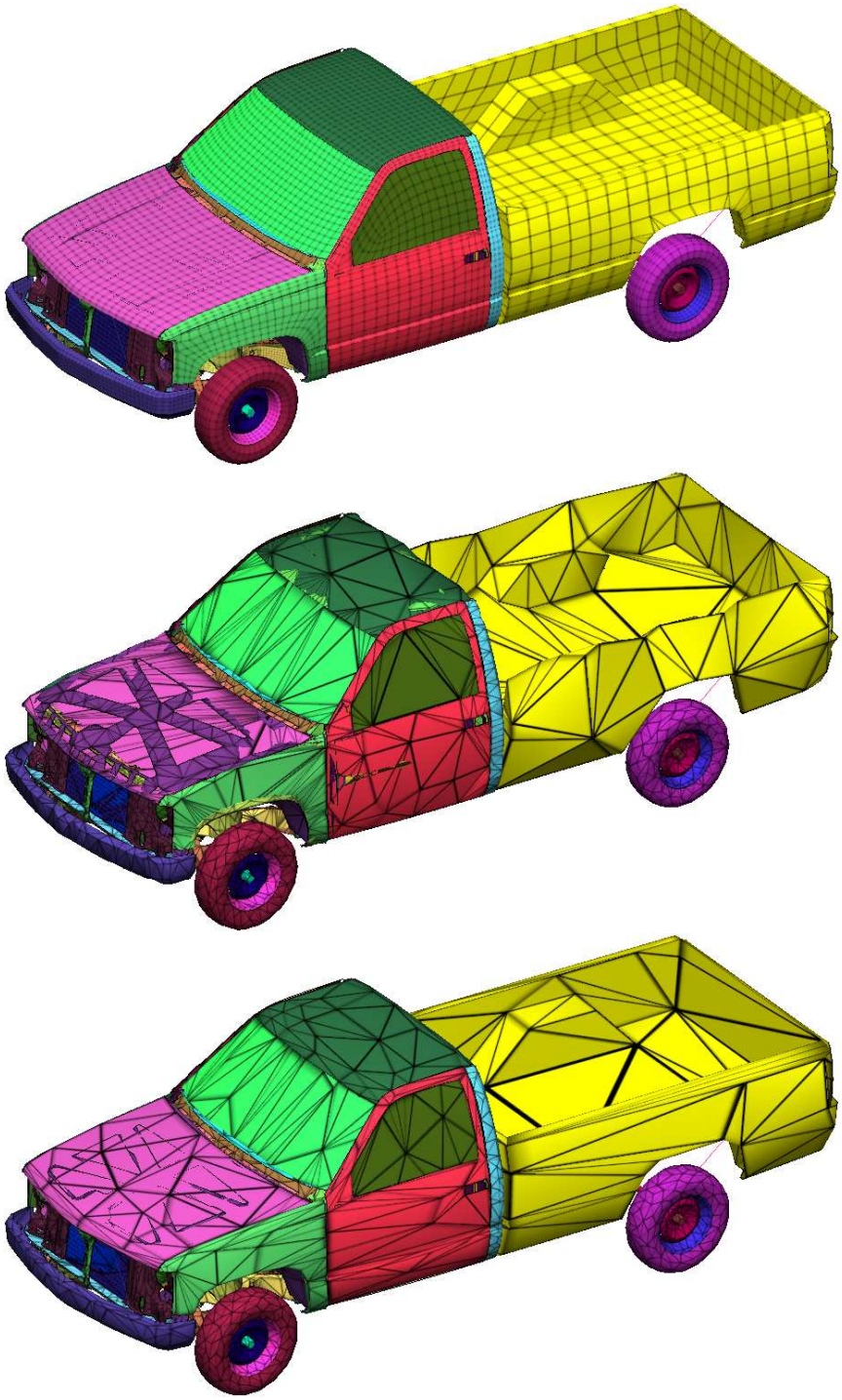


Abbildung 6.11: Dieses aus nur etwa 55 000 Finite-Elementen bestehende Pickup-Modell (oben) wurde sowohl mit opSRASimplify (Mitte) als auch mit HECSimplify (unten) auf 24% der ursprünglichen Dreiecke reduziert. Besonders in Bereichen, in denen benachbarte Bauteile nahe aneinander liegen (siehe Motorhaube), erzeugt HECSimplify weniger optisch störende Durchdringungen als opSRASimplify. Im oberen Bereich der Ladebordwand sowie am Radkasten werden die Qualitätsunterschiede besonders deutlich.

In Abbildung 6.11 werden die Ergebnisse für ein Gesamtfahrzeugmodell gegenübergestellt. Dazu wurde das Originalmodell zunächst mit dem auf Halbkantenreduktion basierenden Modul HECSimplify vergröbert, bis keine weiteren Iterationen möglich waren, ohne dass das Fehlerkriterium (0.5% der Bounding-Box-Diagonale) überschritten worden wäre. Somit lag beim Vergleich mit dem opSRASimplify für jede Teilgeometrie eine prozentuale Zielgröße vor, auf die das Originalmodell erneut durch *Successive Relaxation* reduziert wurde.

6.2.3 Resultate

Der Halbkantenkontraktionsalgorithmus wurde an Bauteilnetzen verschiedener Komplexität getestet. Ähnlich wie bei der Streifengenerierung führen auch hier Netze mit vielen Primitiven zu höheren Reduktionsraten; das liegt darin begründet, dass in Fahrzeugmodellen Bauteile mit vielen Finite-Elementen meistens eine Flächenstruktur mit wenigen, geraden Kanten besitzen, und sich daher für die Simplifizierung gut eignen.

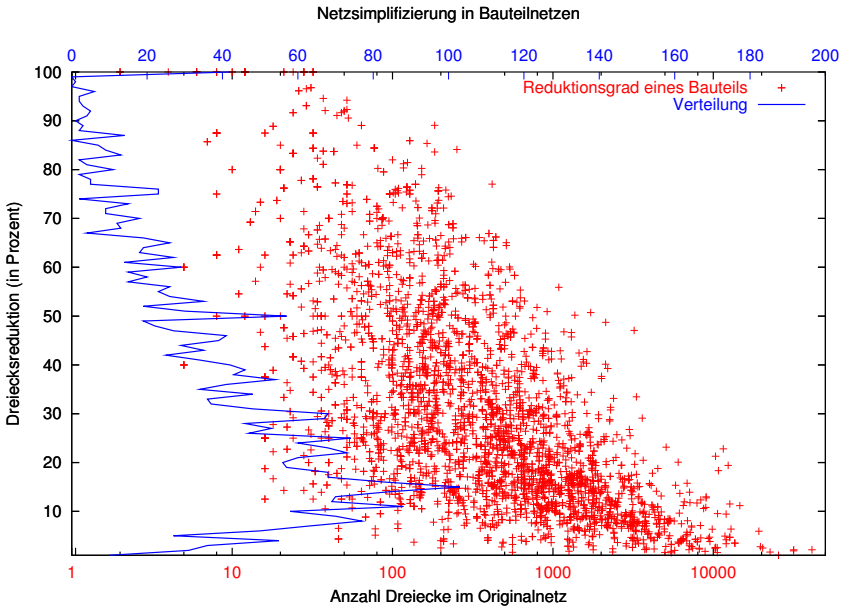


Abbildung 6.12: Aus der Anwendung des Halbkantenreduzierers HECSimplify auf über 3 200 verschiedene Bauteilnetze ergibt sich eine durchschnittliche Reduktion auf 15.3%. Während der Reduktionsgrad jedes Bauteils durch ein Kreuz repräsentiert wird, gibt die Kurve darüber Auskunft, wie häufig welcher Reduktionsgrad erreicht werden konnte.

Die mit HECSimplify erzielte durchschnittliche Reduktion auf 15.3% der ursprünglichen Dreiecke führt je nach Modellgröße zu den in Abbildung 6.13 dargestellten Beschleunigungsfaktoren. Die Zeiten wurden auf einem Linux-PC mit einer GeForce2 MX Graphik, einer AMD XP2400 CPU und 1 GB Hauptspeicher ermittelt.

Originalnetz			Simplifiziertes Netz		
Anzahl △	Bildrate	Reduktionsgrad	Anzahl △	Bildrate	Faktor
530 440	6.93	16.59 %	87 988	26.69	3.85
479 627	7.75	11.92 %	57 183	42.02	5.42
349 961	10.81	14.10 %	49 360	46.44	4.30
250 373	13.32	17.19 %	43 032	44.71	3.36
196 307	19.08	8.96 %	17 596	128.54	6.74
128 056	28.41	13.80 %	17 677	130.44	4.59
98 765	37.04	14.25 %	14 072	163.04	4.40

Abbildung 6.13: Anhand verschiedener Testdatensätze wird hier der Einfluss der Reduktion durch Simplifizierung auf die Darstellungsbeschleunigung gezeigt. Aufgelistet werden jeweils die Dreiecksanzahl und die Bildwiederholrate vor und nach der Simplifizierung inklusive des Reduktionsgrades sowie der sich ergebende Beschleunigungsfaktor.

Der Einsatz der Simplifizierung von Gesamtfahrzeugmodellen im Prototypen *crashViewer* hat gezeigt, dass ein während der Kamerainteraktion verwendetes reduziertes Dreiecksmodell vom Anwender sehr positiv angenommen wird. Es bietet wesentlich höhere Interaktionsraten und vermittelt eine bessere Orientierung im Modell als alternative Lösungen in kommerziellen Produkten, bei denen temporär nur Feature- und Randlinien dargestellt werden. Allerdings benötigt die Simplifizierung des Gesamtmodells in der Regel zu viel Zeit, um im alltäglichen Betrieb nach dem Einlesen des Originalmodells bei der Datenaufbereitung vom Anwender akzeptiert zu werden. Dieser Aspekt kommt besonders dann zum Tragen, wenn die Zeit, in der die Anwendung genutzt wird, relativ kurz ist. Eine mögliche Lösung ist, reduzierte Modelle vorab im Batch-Betrieb zu generieren und den Szenengraphen in einer Binärdatei zusätzlich zu den Originaldaten zur Verfügung zu stellen. Die Erzeugung und Abspeicherung solcher simplifizierten Interaktionsmodelle wurde durch das Batch-Programm *buildGraph* (Abschnitt 8.6, Seite 141) realisiert.

6.3 Texturen statt Geometrie

Bei der Visualisierung im Umfeld der Strukturmechaniksimulation wird die Bildwiederholrate durch die große Polygonanzahl beziehungsweise die Anzahl der von der Rendering-Pipeline zu verarbeitenden Netzknoten limitiert. Während sich die vorhergehenden Abschnitte mit der Minimierung zu transformierender Knoten für die Geometriedarstellung auseinandersetzten, werden im Folgenden Verfahren vorgestellt, die weitere Informationen visualisieren, ohne den Geometriepfad der Rendering-Pipeline zusätzlich zu belasten. Aus der Lastverteilung auf andere Graphiksubsysteme resultieren bei vergleichbaren visuellen Effekten höhere Bildwiederholraten.

Die beiden folgenden Abschnitte beschreiben im Prototypen implementierte Verfahren, die Texturen statt zusätzlicher Geometrie einsetzen und dadurch Last von der Geometrie- auf die Rasterisierungseinheit verlagern.

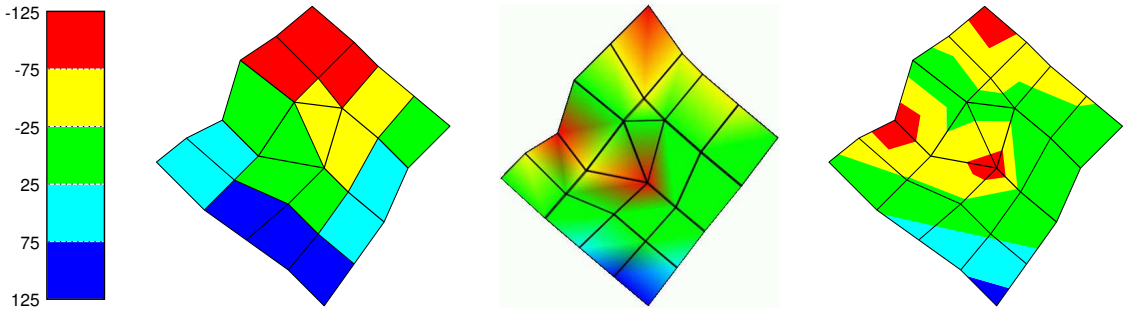
6.3.1 Visualisierung Knoten-basierter Skalare

Im Pre-, insbesondere aber im Postprocessing müssen skalare Daten (zum Beispiel Abstandsinformation, Beulgeschwindigkeit, plastische Dehnung) visuell aufbereitet werden, so dass aus der Visualisierung hervorgeht, welche Werte in welchen Regionen der Finite-Element-Struktur vorkommen. Diese skalaren Werte werden über eine Farbtabelle in einen Farbton umgewandelt und die Geometrie in dem Bereich entsprechend eingefärbt. Der Ursprungsort der skalaren Parameter – also ob die Werte an Netzknoten oder in Finite-Elementen auftreten – entscheidet, wie beim Einfärben der Geometrie vorgegangen werden kann:

- Parametervisualisierung an Elementen:
  Im *Flat-Shading*-Modus wird den Primitiven, die das Element repräsentieren, eine Farbe zugewiesen, mit der die entsprechenden Polygone uniform koloriert werden.
- Parametervisualisierung an Netzknoten:
  Da an den Eckpunkten eines Elementes im Allgemeinen unterschiedliche Werte vorliegen, muss innerhalb des Elementes eine Farbänderung dargestellt werden, der den als linear angenommenen Werteverlauf im Element approximiert.

Die Abbildung 6.14 zeigt, dass *Smooth-Shading* (c) für die Visualisierung Knoten-basierter Werte kein korrektes Bild liefert, da die Farbabbildung bereits vor der Interpolation durchgeführt wird. Um die diskrete Farbskala den Wertebereichen entsprechend, wie ganz rechts dargestellt, auf die Geometrie übertragen zu können, gibt es die Möglichkeit, die Elemente an den Stellen, an denen der Grenzwert zwischen zwei Wertebereichen, die jeweils durch ein Farbband repräsentiert werden, das Element geometrisch zu unterteilen und durch mehrere Polygone mit *Flat-Shading* zu repräsentieren. Allerdings würde durch die zusätzlichen Netzknoten die Geometrie-Einheit noch mehr belastet werden.

Eine wesentlich effizientere Möglichkeit ist die Verwendung einer eindimensionalen Farbtextur. Zunächst wird eine Textur entsprechend der Farbtabelle (a) definiert. Anschließend wird für jeden Netzknoten der skalare Parameter in eine Texturkoordinate $\in [0, 1]$ umgewandelt, die auf die zugehörige Farbe in der Textur verweist. Die Texturierung übernimmt die Rasterisierungseinheit in der Rendering-Pipeline. Dabei sind zwei Verfahren, das Pre- und das Post-Shading voneinander zu unterscheiden. Beim Pre-Shading wird die Texturkoordinate im Eckpunkt eines Primitivs ausgewertet, bevor die resultierende Farbe mit denen der anderen Vertizes über das Primitiv interpoliert wird; das Ergebnis entspricht dem der *Smooth-Shading*-Darstellung. Bei dem in (d) abgebildeten Post-Shading findet die Auswertung der Texturkoordinate (Texture Lookup) erst statt, nachdem die Texturkoordinaten als Repräsentanten der skalaren Parameter zwischen den Vertizes interpoliert wurden.



(a) (b) (c) (d)

Abbildung 6.14: Unterschiedliche Ansätze, um skalare Parameter, die entweder an Elementen oder Knoten vorliegen, durch Einfärbung der Geometrie zu visualisieren. Eine vom Anwender definierte diskrete Farbtabelle (a) ordnet jedem Skalar eine entsprechende Farbe zu. Für die Element-basierte Parametervisualisierung wird durch *Flat-Shading* (b) die Farbe für das gesamte Primitiv gesetzt. Bei der Visualisierung Knoten-basierter Werte kann mit Hilfe von *Smooth-Shading* (c) kein diskreter Farbverlauf innerhalb eines Primitivs erreicht werden. Wie beim Pre-Shading wird der skalare Wert am Knoten zunächst auf eine Farbe abgebildet und anschließend werden die Primitive durch lineare Interpolation der Farben koloriert. Die rechte Darstellung (d) visualisiert einen linearen Werteverlauf zwischen den Eckpunkten (Post-Shading).

In der Cosmo3D-Szenengraphbibliothek werden eindimensionale Texturen nicht unterstützt. Stattdessen kann eine zweidimensionale Textur verwendet werden, die in der einen Dimension nur ein Texel breit ist. Die Texturkoordinaten müssen also über ein csTexCoordSet2f in zwei Dimensionen (t_x, t_y) angegeben werden, wobei die zweite Komponente ignoriert wird. In *crashViewer* wird dieser Speicher dazu genutzt, den visualisierten Skalarwert abzuspeichern. Diese Werte können zum einen dazu verwendet werden, dem Anwender den dargestellten Wert auszugeben, und zum anderen kann mit einer modifizierten Wertezuordnung zur Farbtabelle ein effizientes Remapping des Skalars erfolgen, ohne dass dazu auf interne Daten zurückgegriffen werden muss:

$$t_x = \mathrm{Max}\left(\frac{1}{2 \cdot \mathrm{Texelbreite}} \quad , \quad \mathrm{Min}\left(1 - \frac{1}{2 \cdot \mathrm{Texelbreite}} \quad , \quad \frac{t_y - \mathrm{minVal}}{\mathrm{maxVal} - \mathrm{minVal}}\right)\right)$$

Der genaue Werteverlauf innerhalb der in der Crash-Simulation verwendeten linearen Finite-Elemente kann durch die Textur-basierte Visualisierung nur für dreiseitige Elemente exakt wiedergegeben werden; bei vierseitigen Schalenelementen hängt es von der durch das Graphiksubsystem vorgenommenen Triangulierung des Quadrilaterals ab, wie die Isolinie innerhalb des Primitivs verläuft (Abbildung 6.15). In der Praxis hat diese Abweichung innerhalb der Elemente allerdings nur geringe Relevanz und der Fehler wird durch die zunehmend feinere Auflösung der Finite-Element-Modelle entsprechend geringerer.

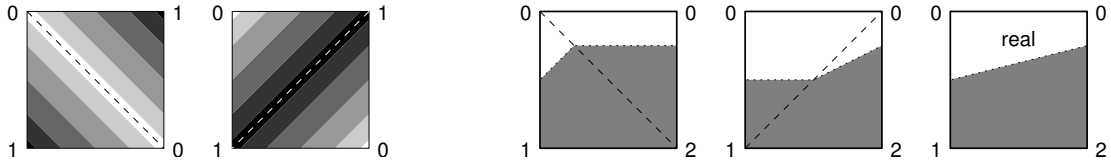


Abbildung 6.15: Die beiden skizzierten Beispiele zeigen, dass die Unterteilung (gestrichelte Linie) eines Quadrilaterals entscheidet, wie der Übergang von einem in ein anderes Farbband, also eine Isolinie, innerhalb des Quadrilaterals verläuft, wenn die Werte an den Eckpunkten gegeben sind. Links kommt es bei stark voneinander abweichenden Farben in den Eckpunkten zu großen Unterschieden. Rechts wird der Verlauf der Isolinie für einen Isowert von 0.5 dem realen Werteverlauf im Element gegenübergestellt.

Wie die Textur-basierte Visualisierungsmethode zur selektiven Geometriedarstellung in Abhängigkeit der zugeordneten Werte genutzt werden kann, wird anhand der Darstellung potenzieller Flansche in Kapitel 7.1.3.1, Seite 115 vorgestellt.

6.3.2 Visualisierung der Netzstruktur

Die Struktur des Finite-Element-Netzes ist neben den Materialeigenschaften ein wesentlicher Einflussfaktor für das Verformungsverhalten des Fahrzeugbauteils in der Crash-Simulation. Daher ist die Visualisierung der Netzstruktur von großer Bedeutung. Da die reine Darstellung eines Gitterdrahtmodells keine Tiefeninformationen enthält, ist eine kombinierte Darstellung der schattierten Oberfläche zusammen mit den Elementgrenzen wünschenswert.

Das konventionelle Verfahren, um diese Darstellung zu erzielen, ist ein Zwei-Schritt-Verfahren: zuerst wird die Geometrie durch schattierte Dreiecke gezeichnet und in einem zweiten Durchlauf werden die gleichen Daten dazu verwendet, um im Linien-Modus die Elementgrenzen anzuzeigen. Der Aufwand verdoppelt beziehungsweise die Bildwiederholrate halbiert sich dadurch. Zudem ist die Darstellung mit „z-Buffer-Fighting“ behaftet, das heißt, ohne Ausnutzung spezieller Erweiterungen von OpenGL (glPolygonOffset) werden die Linien-Fragmente während des Tiefentests teilweise als hinter der Fläche liegend eingestuft und erscheinen daher unregelmäßig gestrichelt oder gar nicht (siehe auch Abbildung 6.17).

Dieses Zwei-Schritt-Verfahren kann umgangen werden, indem jedes Element mit einer Luminanz-Textur belegt wird, die pro Texel nur ein Bit benötigt. Dieses Bit gibt an, ob das Pixel nach den aktivierten Berechnungsvorschriften schattiert oder schwarz dargestellt werden soll. Bei der Texturierung ergibt sich die Farbe bei GL_MODULATE als Textur-Funktion aus: $C_v = L_t \cdot C_f$. Dabei entspricht C_f der schattierten Fragmentfarbe und L_t dem Texelwert ($L_t = 0$ am Elementrand und $L_t = 1$ im Elementinneren). Das Texturbild ist in Abbildung 6.16, links zu sehen. Um nach Anwendung der Textur trotzdem noch Dreiecks- oder Quadrilateralstreifen generieren zu können, müssen die Informationen, also auch die Texturkoordinaten, an den inneren Knoten der Streifen übereinstimmen; daher

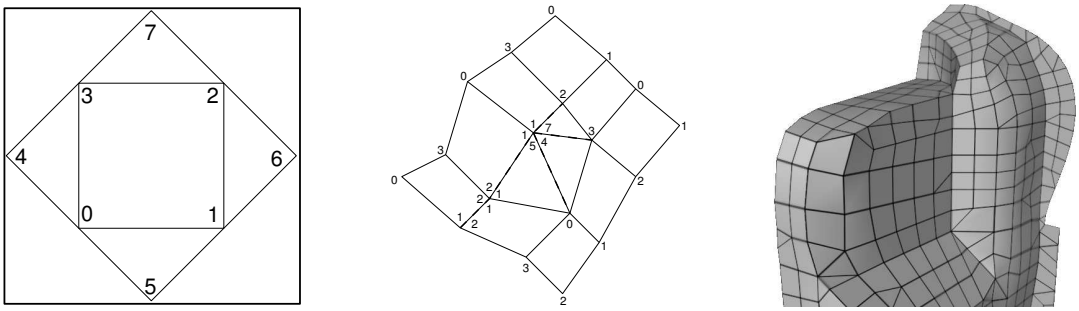


Abbildung 6.16: Das *Wireframe*-Texturbild (links) wurde im Vergleich zu [43] um die umliegenden Dreiecke erweitert. Degenerierte Quadrilaterale, die zur Darstellung dreieckiger Schalenelemente verwendet werden, bekommen die Texturkoordinaten der Dreiecke zugewiesen. Die Spiegelung der Textur (Mitte) an benachbarten Elementen erlaubt weiterhin die Generierung von Dreiecks- oder Quadrilateralstreifen. Das rechte Bild zeigt ein Gouraudschattiertes Bauteilnetz mit der *Wireframe*-Textur auf jedem Element.

wird die Textur an inneren Kanten gespiegelt.

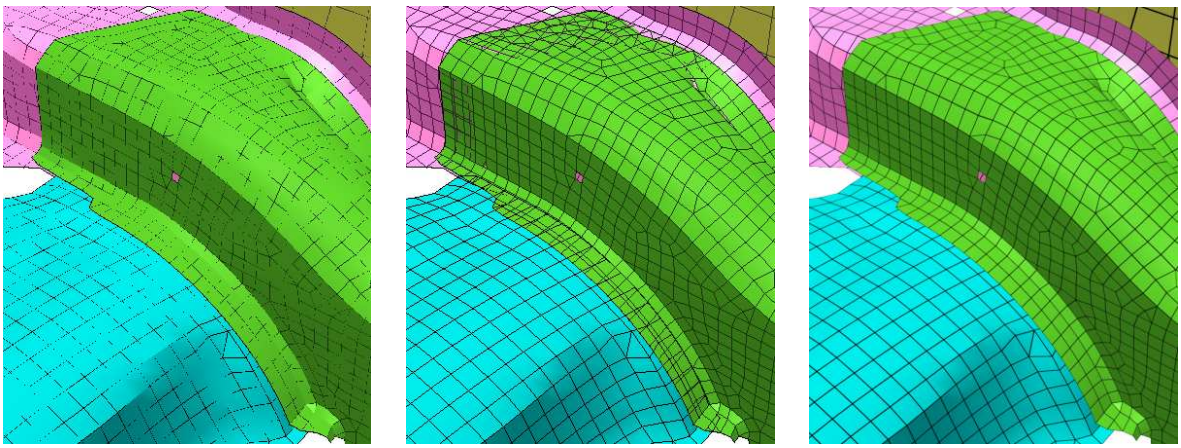


Abbildung 6.17: Die konventionelle Darstellung der Elementgrenzen auf einer schattierten Oberfläche benötigt zweimaliges Zeichnen der Szene. Das „z-Buffer-Fighting“ (links) kann zwar durch Verwendung von glPolygonOffset unterbunden werden (Mitte); jedoch können dann in Flanschbereichen eng benachbarter Bauteile die Elementgrenzen der hintenliegenden Bauteile unerwünschterweise hervortreten. Die Anwendung der *Wireframe*-Textur (rechts) resultiert in einer höheren Bildwiederholrate und einer besseren Darstellungsqualität.

Da bisher weder Cosmo3D / OpenGL Optimizer noch die SGI-Graphik-Hardware Multi-Texturing, also die Anwendung mehrerer Texturen auf eine Geometrie unterstützt, schließt die Verwendung der *Wireframe*-Textur das gleichzeitige Visualisieren von Parame-

tern mit Texturen, wie im vorhergehenden Abschnitt beschrieben, aus. Im Zusammenhang mit der Textur-basierten Parametervisualisierung muss auf das konventionelle Zwei-Schritt-Verfahren zur Netzstruktur-Darstellung zurückgegriffen werden. Diese wirkt sich allerdings für große Fahrzeugmodelle hinsichtlich der Bildwiederholrate sehr negativ aus. Für ein Modell bestehend aus etwa 530 000 Dreiecken, die schattiert mit 6.9 Bildern pro Sekunde dargestellt werden, fällt die Bildwiederholrate bei Hinzunahme des Wireframe-Schrittes auf 1.16 ab. Demgegenüber werden unter Einsatz der *Wireframe*-Textur immer noch 5.5 Bilder pro Sekunde erzeugt.

Kapitel 7

Spezielle Pre-Processing Funktionalitäten

Bis vor wenigen Jahren wurden die virtuellen Fahrzeugmodelle noch homogen vernetzt, das heißt, die Finite-Element-Netze aneinandergrenzender Bauteile mussten so miteinander abgeglichen werden, dass sie in den Randbereichen gemeinsame Knoten verwendeten. Dadurch brachte der Austausch eines Bauteils durch eine andere Variante stets großen Vernetzungsaufwand mit sich. Inzwischen können immer mehr Bauteilverbindungen, wie zum Beispiel Schweißpunkte, Schweißnähte oder Klebeverbindungen im Simulationsprozess abgebildet werden. Somit spielt die Assemblierung unabhängig voneinander vernetzter Bauteile eine wichtige Rolle im Vorverarbeitungsschritt der Crash-Simulation (Abbildung 7.1).

Dieses Kapitel widmet sich der Distanzvisualisierung und insbesondere der Lokalisierung und Beseitigung initialer Penetrationen zwischen unabhängig voneinander vernetzten Bauteilen. Darüber hinaus werden die Möglichkeiten zur interaktiven Modifikation von Schweißpunktdaten erläutert, bevor abschließend ein Graphik-Hardware-basiertes Verfahren beschrieben wird, welches erlaubt, skalare Parameter von einem Gitter auf ein anderes zu übertragen.

7.1 Distanzvisualisierung

Die Vernetzung von CAD-Daten durch die Diskretisierung der Bauteilflächen bringt Approximationsfehler mit sich. Zum einen kommt es durch die Datenkonvertierung über verschiedene Datenformate, in denen die CAD-Flächenbeschreibung teilweise als Mittel- teilweise aber auch als Randfläche interpretiert wird, zum anderen durch die unabhängige Vernetzung zu unterschiedlichen Abständen der diskretisierten Bauteilflächen. Da das Finite-Element-Netz für die Crash-Simulation die Mittelfläche des Bauteils darstellt, ist zu beiden Flächenseiten die halbe Bauteildicke (Blechstärke) zu berücksichtigen. Dadurch kommt es in einem assemblierten Fahrzeugmodell in Flanschbereichen oder zwischen benachbarten Bauteilen gegebenenfalls zu initialen Penetrationen oder gar zu Perforationen,

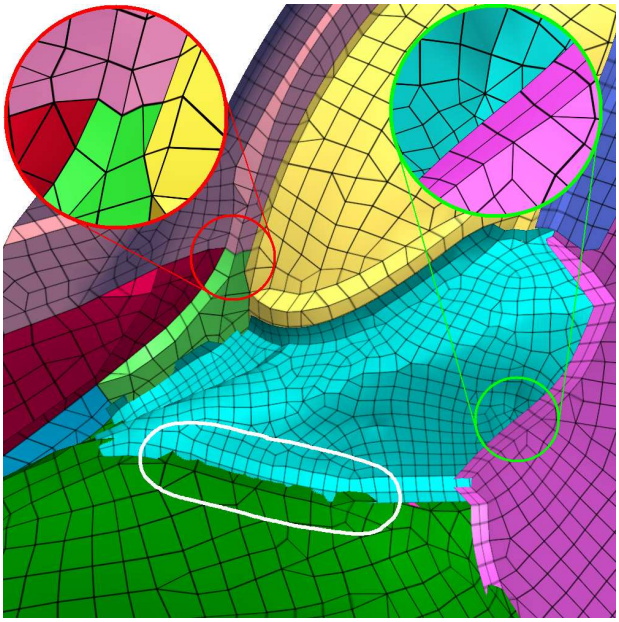


Abbildung 7.1: Während in homogenen Netzen aneinandergrenzende Bauteile durch gemeinsam genutzte Knoten implizit miteinander verbunden sind (links oben), werden unabhängig voneinander vernetzte Bauteile in inhomogen Flanschbereichen (rechts oben) explizit durch Verbindungselemente aneinandergebunden. Die unabhängige Vernetzung einzelner Bauteile kann allerdings zu Netzdurchdringungen, so genannten Perforationen, führen (unten).

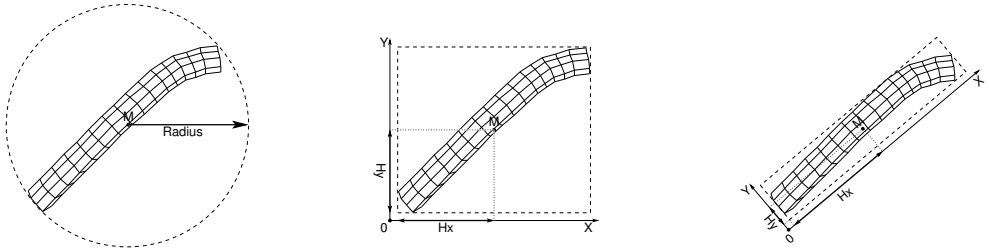
die beim Start der Simulation initiale Kräfte verursachen und damit die Simulationsergebnisse verfälschen (vergleiche Abbildung 2.4, Seite 32).

Dem Berechnungsingenieur fehlten bisher Werkzeuge, um derartige Bereiche zu detektieren, zu visualisieren und schließlich selektiv zu beseitigen. Abschnitt 7.1.1 beschreibt die Grundlagen der eingesetzten Bounding-Volume-Hierarchie, die eine effiziente Berechnung minimaler Abstände erlaubt. Abschnitt 7.1.2 setzt sich mit der Anwendung dieser Bounding-Volume-Hierarchie zur Detektion initialer Penetrationen auseinander und im Unterkapitel 7.1.3 werden schließlich zwei Varianten zur Visualisierung potenzieller Flansche vorgestellt.

7.1.1 Bounding-Volume-Hierarchie auf FE-Netzen

Es gibt eine Vielzahl von Arbeiten, die sich mit der Unterteilung komplexer polygonaler Modelle auseinandersetzen (siehe Abschnitt 3.2.2, Seite 40). Für eine hierarchische Unterteilung der Finite-Element-Netze wurde der Ansatz der objektorientierten Bounding-Boxen zur Kollisionsdetektion von Gottschalk et al. [31] im Rahmen einer Diplomarbeit [40] untersucht und an das Einsatzgebiet in der Strukturmechaniksimulation angepasst. In [31]

vergleichen die Autoren den Einsatz von Begrenzungskugeln beziehungsweise von hauptachsenparallelen Begrenzungsquadern (AABB) mit dem von Begrenzungsquadern, deren Orientierung an den eingehüllten Strukturen ausgerichtet ist (OBB). Dabei unterscheiden sich die drei Begrenzungsvolumenarten im Allgemeinen nicht nur durch ihre räumliche Ausdehnung, sondern auch durch Umfang der zu speichernden Daten und den Berechnungsaufwand bei der Erstellung der BV-Hierarchie sowie bei der Abstandsberechnung (Abbildung 7.2). Während sich Lage und Größe der Bounding-Boxen durch die Suche nach den Extrempunkten der eingehüllten Geometrie bezüglich der Box-Achsen festlegen lassen, kann für die Bestimmung einer annähernd optimalen Begrenzungskugel das von Ritter[54] vorgestellte Verfahren angewendet werden. Eine aufwendigere Optimierung der Begrenzungskugeln durch Methoden wie sie in [28] präsentiert werden, kommen im Prototypen nicht zur Anwendung, da der Aufbau der BV-Hierarchie nicht ausreichend effizient vollzogen werden könnte.



(a) Sphere (b) AABB (c) OBB

Abbildung 7.2: Die drei Begrenzungsvolumenarten unterscheiden sich in ihrer räumlichen Ausdehnung, den zu speichernden Daten pro Instanz und beim Berechnungsaufwand für den Abstandstest. M bezeichnet den Mittelpunkt des Hüllvolumens, H_x und H_y entsprechen den halben Seitenlängen der Bounding-Boxen.

Bei der Verwendung orientierter Bounding-Boxen (OBB) wird zum Aufbau der BV-Hierarchie rekursiv in folgenden Schritten vorgegangen:

1. Ermittlung der Kovarianzmatrix eingehüllter Geometrieprimitive
   Die normalisierten Eigenvektoren der symmetrischen Kovarianzmatrix bilden die Orthonormalbasis für die Achsen des ausgerichteten Begrenzungsquaders.

2. Festlegung der Ausdehnung der orientierten Bounding-Box
   Die extremen Punkte werden durch die Projektion aller eingehüllten Knotenpunkte auf die drei Achsen ermittelt und bestimmen damit die Kantenlänge des Begrenzungsquaders. Sofern die Anzahl maximal zulässiger Geometrieprimitive pro Blattknoten der Bounding-Volume-Hierarchie nicht überschritten ist, kann die Rekursion für diesen Zweig hier abgebrochen werden.

3. Aufteilung der umhüllten Geometrieprimitive in zwei Untermengen
   Die Schwerpunkte der Geometrieprimitive werden auf den ursprünglich längsten Eigenvektor projiziert, der parallel zur größten Ausdehnung des orientierten Begrenzungsquaders verläuft. Die Partitionierungsebene verläuft orthogonal zum längsten Eigenvektor (Projektionsachse) durch den gemittelten Schwerpunkt aller umhüllten Primitive. Die Lage des auf die Projektionsachse abgebildeten Primitivschwerpunktes entscheidet, welchem der beiden Untermengen das Primitiv zugeordnet wird.

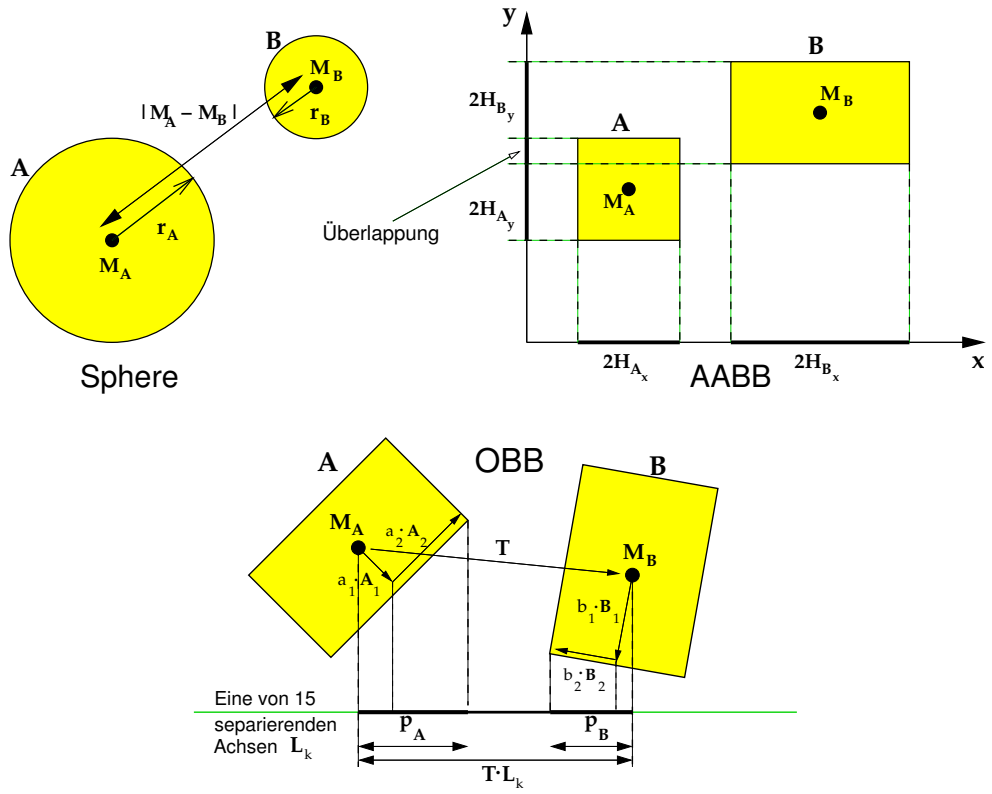


Abbildung 7.3: Überlappungstest für die Hüllkörpertypen Sphere, AABB und OBB

Für den Kollisionstest zwischen Geometrieprimitiven beziehungsweise die Ermittlung minimaler Abstände werden die Berechnungen zunächst jeweils zwischen zwei Hüllvolumen der Hierarchiestufe i durchgeführt. Sofern das Abbruchkriterium – fehlende Überlappung der Hüllvolumen beziehungsweise Überschreitung eines vorher festgelegten Höchstabstandes – nicht erfüllt ist, wird mit den Begrenzungsquadern der Hierarchiestufe $i+1$ fortgefahren. Wenn in beiden Hierarchien Blattknoten erreicht wurden, die keine weiteren Hüllvolumen enthalten, werden die enthaltenen Primitive des einen Hüllvolumens gegen die des anderen getestet.

Der Überlappungstest ergibt sich je nach Hüllvolumentyp aus folgenden Formeln, bei denen M_i jeweils die Hüllkörpermittelpunkte bezeichnen (vergleiche Abbildung 7.3):

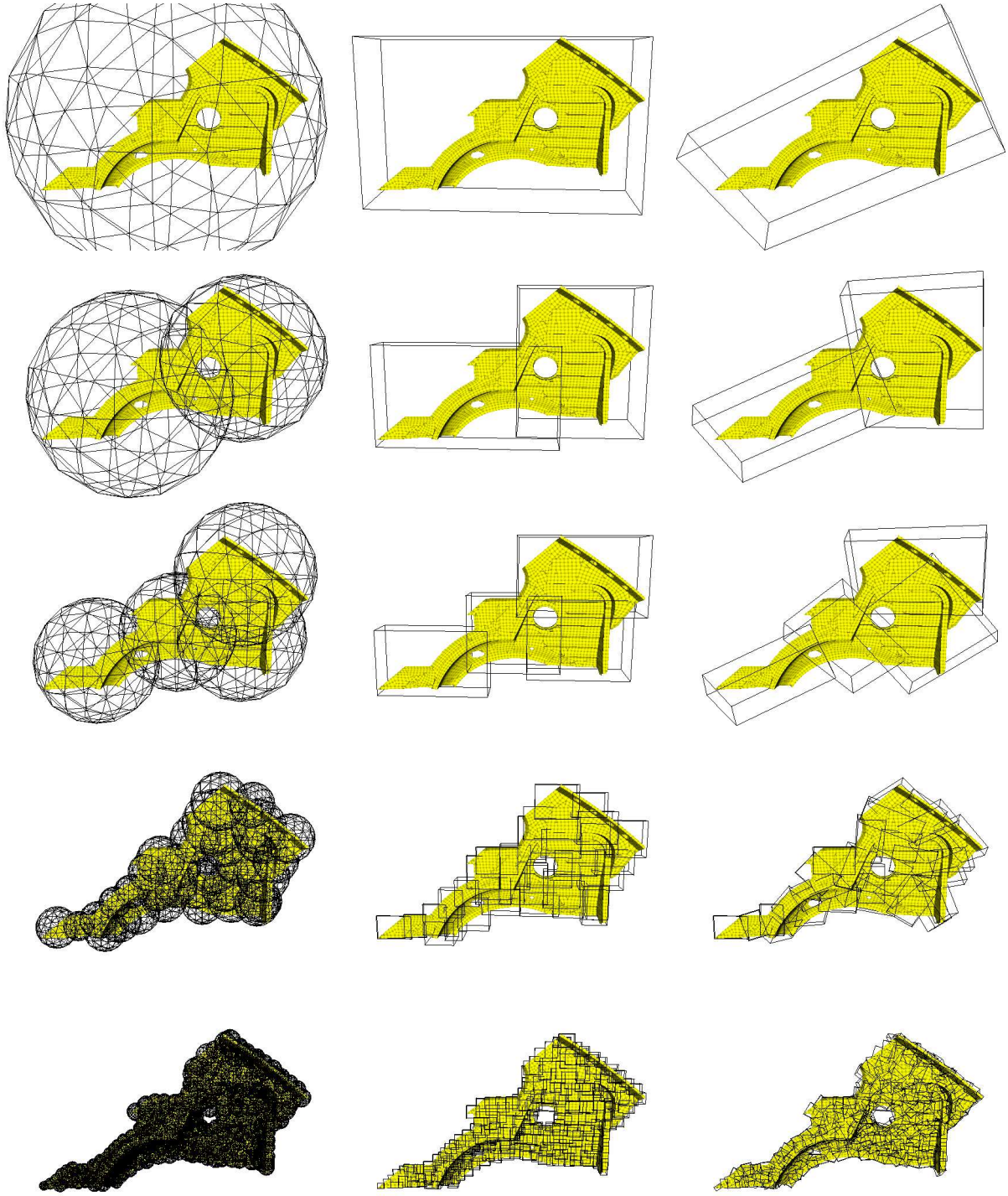


Abbildung 7.4: Gegenüberstellung der Hüllkörpertypen Sphere, AABB und OBB am Beispiel der Bounding-Volume-Hierarchie-Ebenen 0, 1, 2, 5 und 8 eines Bauteils

Sphere: $\|M_A - M_B\| - (r_A + r_B) \; < \; 0$ — r_i: Kugelradien

AABB: $j \in \{x, y, z\} : \; \big(\, |M_A - M_B| - (H_A + H_B) \,\big)_j \; < \; 0$ — H_i: halbe Seitenlängen; j: Komponente x, y, z

OBB: $\forall k : \; \Big(|L_k \cdot (M_A - M_B)| - \big(|L_k \cdot D_A| + |L_k \cdot D_B| \big) \Big) \; < \; 0$ — D_i: halbe Raumdiagonale; L_k: separierende Achse

In [31] wird vorgestellt, wie der Überlappungstest für objektorientierte Hüllquader (OBB) durch die Projektion des Verbindungsvektors $\mathbf{T}$ zwischen beiden Mittelpunkten $\mathbf{M_A}$ und $\mathbf{M_B}$ und den Halbdiagonalen jeder Box auf eine von 15 separierenden Achsen $\mathbf{L_k}$ durchgeführt wird. Zwei konvexe Polyeder überlappen sich nämlich genau dann nicht, wenn es eine Ebene gibt, die parallel zu einer der Polyederflächen ist und beide Objekte voneinander trennt, ohne diese zu schneiden. Darauf aufbauend sind die Volumina zweier Polyeder disjunkt, wenn es orthogonal zu einer Polyederfläche oder orthogonal zu einem Kantenpaar beider Polyeder eine Achse $\mathbf{L_k}$ gibt, die beide Polyeder voneinander trennt. Da es für jede Box drei unterschiedliche Ausdehnungsrichtungen für Flächen und Kanten gibt, resultiert daraus: $2 \cdot 3$ Flächen $+ \, 3 \cdot 3$ Kantenpaare $= 15$ separierende Achsen.

Sofern eine Überlappung der umhüllten Geometrie festgestellt wird, kann die weitere Traversierung für eine reine Kollisionserkennung abgebrochen werden. Wenn es jedoch darum geht, sich durchdringende Finite-Elemente zu detektieren, werden gefundene Elementpaarungen in einer Liste gesammelt und die gesamte Hüllvolumenhierarchie bis zum Ende durchlaufen.

Für die Ermittlung des minimalen Abstands zweier sich nicht durchdringender polygonaler Netze wurde der Algorithmus von Cameron [9] verwendet. Dabei handelt es sich um eine beschleunigte Variante des bereits neun Jahre vorher publizierten Verfahrens von Gilbert et al. [29], bei dem der minimale Abstand zum Ursprung im durch die Minkowski-Differenz zweier konvexer Polyeder definierten TC-Raum berechnet wird. Die Minkowski-Differenz im $I\!R^3$ ist definiert als

$$A - B = \big\{ a - b \quad : \quad a \in A, \; b \in B \big\}, \text{ mit } A, B \subset I\!R^3$$

Auf dessen konvexen Hülle, als *Translational C-Space Obstacle* bezeichnet, wird in einer iterativen Vorgehensweise das Minimierungsproblem gelöst (Abbildung 7.5).

Die Bounding-Volume-Hierarchie erlaubt die effiziente Kollisionsdetektion zwischen polygonalen Netzen, zum Beispiel zur Lokalisierung von Perforationen, wird jedoch hauptsächlich zur Abstandsberechnung zwischen einem gegebenen Punkt und dem hierarchisch unterteilten Netz eingesetzt. Außer für die Detektion initialer Penetrationen und die Visualisierung potenzieller Flansche wird im Prototypen die Abstandsberechnung auch bei der interaktiven Modifikation von Schweißpunktdaten und der automatischen Pfadsuche bei flächigen Bauteilverbindungen verwendet [23]. Die effiziente Berechnung minimaler

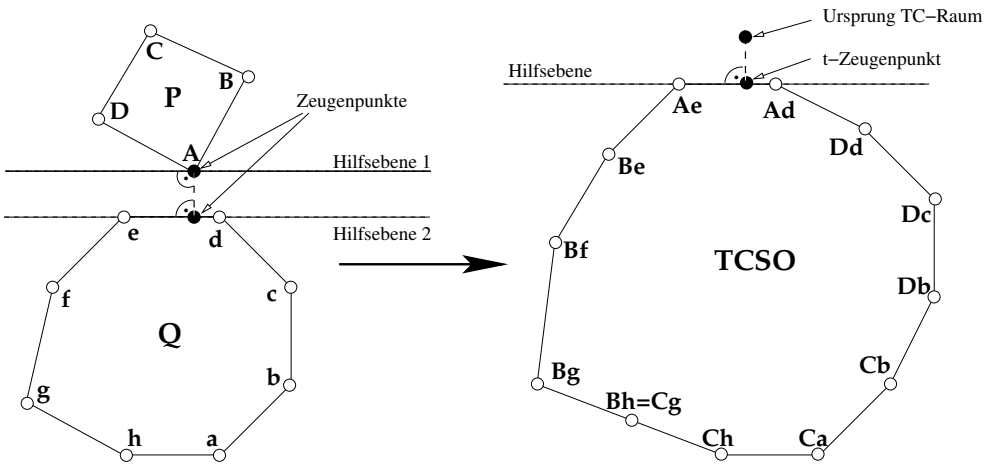


Abbildung 7.5: Der TC-Raum entspricht der Minkowski-Differenz der beiden links dargestellten Polyeder **P** und **Q**. Der minimale Abstand zwischen **P** und **Q** entspricht im TC-Raum dem kürzesten Abstand zwischen Ursprung und konvexer Hülle der Minkowski-Differenz, also dem rechts skizzierten *Translational C-Space Obstacle* (TCSO). Die Punkte des TC-Raumes, die nicht auf der konvexen Hülle liegen, wurden der Übersichtlichkeit halber nicht eingezeichnet.

Abstände bildet die Grundlage für viele weitergehende Funktionen, zum Beispiel für die abschließende Validierung von Bauteilverbindungen, bei der potenzielle Flanschbereiche gesucht werden, in denen die Bauteile nicht durch Verbindungselemente angebunden sind. Entscheidend für hohe Performanz der Abstandsberechnung ist die Vorgabe eines möglichst geringen Wertes für die maximal zu betrachtende Distanz um einen Knoten. Sofern dieser initiale Wert bei der Abstandsberechnung zwischen Punkt und Hüllvolumen nicht unterschritten wird, brauchen die Kinder des Hüllvolumens nicht betrachtet zu werden. Nachfolgend werden zwei Einsatzgebiete dieser Algorithmen näher erläutert.

7.1.2 Detektion und Beseitigung initialer Penetrationen

Das Auftreten initialer Kräfte, hervorgerufen durch initiale Penetrationen benachbarter Bauteile, muss vermieden werden, um Verfälschungen der Simulationsergebnisse auszuschließen. Um Regionen, in denen initiale Penetrationen vorkommen, lokalisieren zu können, werden zunächst die minimalen Abstände aller Netzknoten zu einem anderen Bauteil berechnet. Dies geschieht unter Zuhilfenahme der im letzten Abschnitt angesprochenen Algorithmen. Die Zeit, die für die Ermittlung der minimalen Knoten-Element-Abstände benötigt wird, hängt entscheidend von der maximalen Distanz ab, bis zu der die genauen Abstände berechnet werden sollen. Liegt diese Distanz nur geringfügig über der Kontaktdicke zweier Bauteile, so kann der initiale Abstandswert für jeden Netzknoten, der bei der Traversierung der BV-Hierarchie als Abbruchkriterium dient, entsprechend klein gewählt

werden. Das hat zur Folge, dass nur wenige BV-BV- beziehungsweise Knoten-BV-Distanzen berechnet werden müssen, bevor der minimale Abstand zwischen dem Knoten und einer geringen Anzahl von Elementen berechnet werden kann.

In der Datenstruktur PenetrationCluster werden Verweise auf den penetrierenden Knoten und das betroffene Element gespeichert. Die Detektion liefert zusätzlich zu den minimalen Knoten-Element-Abständen eine Liste solcher PenetrationCluster, aus der hervorgeht, welche Bauteile ineinander eindringen. Darüber hinaus kann durch den Anwender die Auswahl der modifizierbaren Bauteilnetze eingeschränkt werden, was für die Anpassung von Bauteilvarianten an ein bereits bereinigtes Fahrzeugmodell eine entscheidende Rolle spielt, da alle anderen Bauteilnetze während der Beseitigung initialer Penetrationen konserviert werden sollen. Das Ergebnis der Detektion wird in Form von Listen, die Knotenkoordinaten sowie Knoten- und Element-Labels enthalten, an die *Inipen*-Bibliothek[1] übergeben. Die Schnittstelle entspricht der der DAISY-Bibliothek. Durch ein zusätzliches Bit-Array wird die Modifizierbarkeit der Netzknoten festgelegt. Die Bibliothek übernimmt das iterative Verfahren, in dem zwischen den Bauteilen zunächst Kräfte berechnet und die Netzknoten in Bereichen initialer Penetration auseinandergetrieben werden, bis keine Eindringung mehr vorliegt. Nach diesem Vorgang werden die verschobenen Knotenkoordinaten zunächst in die interne und schließlich auch in die Szenengraph-Datenstruktur übernommen, so dass das Ergebnis der Penetrationsbeseitigung vom Anwender begutachtet werden kann.

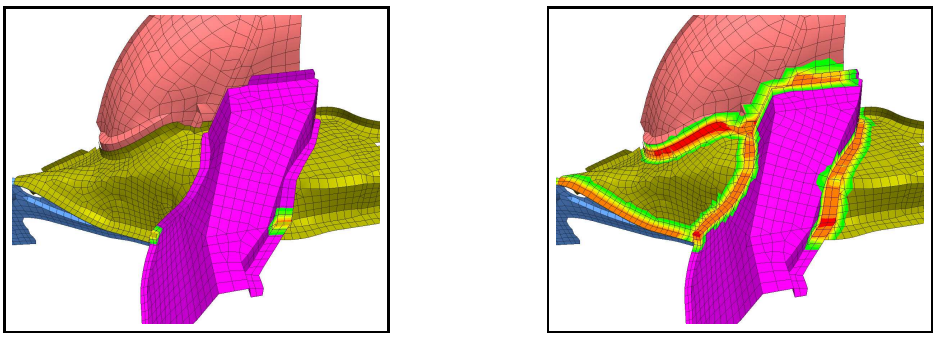


Abbildung 7.6: Durch Einfärbung sich perforierender (links) beziehungsweise penetrierender Elemente (rechts) werden fehlerhafte Netzregion hervorgehoben.

Durch die selektive Bereinigung penetrierender Netze wird dem Berechnungsingenieur erstmals die Möglichkeit gegeben, die Modifikation auf bestimmte Bauteile einzuschränken und anschließend das modifizierte Netz gegebenfalls noch zu überarbeiten, bevor der Simulationslauf gestartet wird. Derzeit können sowohl Perforationen als auch Penetrationen detektiert und visualisiert werden; letztere können auch beseitigt werden. Es wäre denkbar, auch Perforationen durch ein Zwei-Schritt-Verfahren zu beseitigen, indem im ersten Schritt perforierende Knoten hinter das zugehörige Element aus dem PenetrationCluster projiziert werden und anschließend die dann vorliegende Penetration, wie oben beschrieben, besei-

[1] Die *Inipen*-Bibliothek wurde freundlicherweise von der Firma ESI zur Verfügung gestellt.

tigt wird. Mit der Beseitigung von Perforationen und einer anschließenden Gitterrelaxation setzt sich [42] auseinander.

7.1.3 Visualisierung potenzieller Flansche

Bei der Assemblierung einzelner Bauteile beziehungsweise Bauteilgruppen zu einem kompletten Fahrzeugmodell für die Crash-Simulation spielt die korrekte Definition der Nebenbedingungen von Bauteilen untereinander eine große Rolle. Zu diesen Nebenbedingungen gehören unter anderem Kontaktspezifikationen und Bauteilverbindungen, die in Bereichen, in denen zwei oder mehr Bauteilflächen einander sehr nahe kommen, definiert werden können. Aufgrund der Komplexität der Simulationsmodelle ist es für den Berechnungsingenieur hilfreich, die Visualisierung des Finite-Element-Netzes auf die Bereiche zu beschränken, in denen Bauteile einen geringen Abstand zu benachbarten Bauteilen haben. In der Regel handelt es sich hierbei um die Flanschbereiche, in denen Bauteilverbindungen definiert werden müssen.

In dem entstandenen Prototypen *crashViewer* wurden für diese Anforderung zwei unterschiedliche Lösungen entwickelt und implementiert. Während das eine Verfahren es erlaubt, unter Ausnutzung der Graphik-Hardware die Fahrzeuggeometrie anhand der für die Distanzvisualisierung berechneten Minimalabstände interaktiv auszublenden, ohne die Geometriedaten modifizieren zu müssen, reduziert das andere Verfahren das Finite-Element-Netz auf einen kleinen Bruchteil und sorgt somit für eine wesentlich bessere Performanz bei der Handhabung des Modells. Beide Verfahren werden im Folgenden vorgestellt.

7.1.3.1 Graphik-basiertes Ausblenden der Fahrzeuggeometrie

Nachdem der Anwender zunächst einen Abstandsbereich spezifiziert hat und die Minimalabstände der Netzknoten zu benachbarten Bauteilen, wie oben beschrieben, ermittelt wurden, werden die Distanzwerte analog zur Parametervisualisierung (Abschnitt 6.3.1, Seite 102) auf Texturkoordinaten abgebildet. Als Textur wird eine eindimensionale RGBα-Textur verwendet, deren Farbverlauf über den in Abbildung 7.7 dargestellten Dialog interaktiv verändert werden kann. Der zusätzliche α-Kanal in der Textur ermöglicht in Abhängigkeit der verwendeten OpenGL-Texturierungsfunktion folgendes:

- Sofern GL_DECAL als Funktion gewählt wird, bleibt der Opazitätswert texturierter Fragmente unverändert. Der α-Wert beeinflusst lediglich, inwieweit der Farbwert C_f durch die Texelfarbe C_t modifiziert wird; so ergibt sich die resultierende Farbe aus $(1 - \alpha_t)C_f + \alpha_t C_t$. Das heißt, dass in Bereichen, in denen der α-Kanal für die Textur auf vollkommen transparent gesetzt ist, eine Wertevisualisierung unterbleibt und stattdessen die ursprünglich schattierte Fahrzeuggeometrie sichtbar ist.

- Falls GL_MODULATE während der Texturierung verwendet wird, werden sowohl die Farb- als auch die Opazitätswerte des Fragmentes und der Textur miteinander mul-

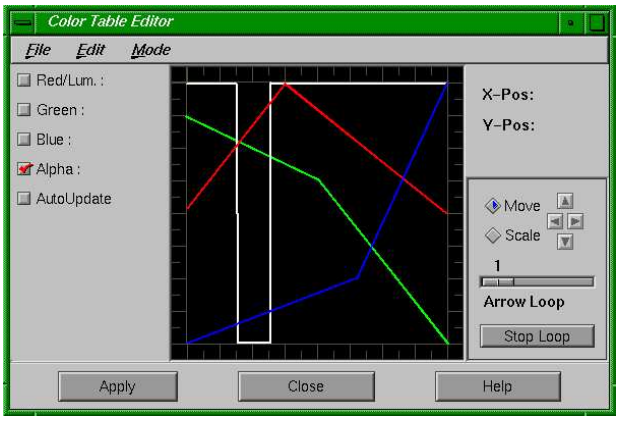


Abbildung 7.7: Das Dialogfenster unterstützt den Anwender bei der Definition von Farbverläufen über den spezifizierten Wertebereich.

tipliziert. Dies hat zur Folge, dass in Bereichen, auf die transparente Texturelemente abgebildet werden, die Fahrzeuggeometrie nur unsichtbar dargestellt wird. Wenn während der Bildsynthese zwischen opaken und transparenten Polygonen nicht unterschieden wird — dies ist unter OpenGL Optimizer / Cosmo3D der Fall —, hat das zur Folge, dass der z-Puffer durch transparente, nahe am Betrachter liegende Fragmente „verschmutzt“ wird. Dadurch scheitern weiter entfernte, opake Fragmente, die später verarbeitet werden, am z-Test und bleiben somit unsichtbar. Abhilfe schafft die zusätzliche Aktivierung des α-Tests: Fragmente, deren α-Komponente unterhalb des Referenzwertes für den α-Test liegen, werden somit schon vor einem eventuellen Passieren des z-Tests abgefangen (vergleiche Abbildung 3.3).

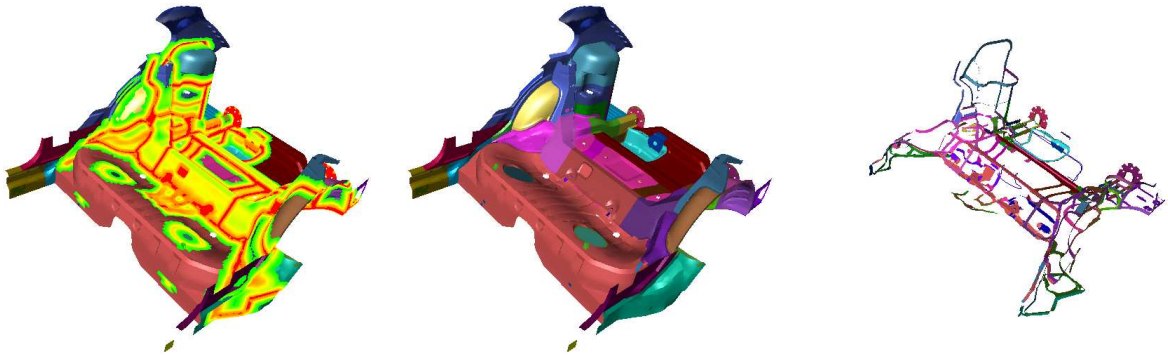


Abbildung 7.8: Während links mit der GL_DECAL-Funktion nur die Visualisierung der Werte durch Farben auf bestimmte Wertebereiche eingeschränkt wird, kann durch Modifizierung des α-Kanals während der Texturierung mit GL_MODULATE interaktiv die Fahrzeuggeometrie ausgeblendet werden.

Dieses Verfahren eignet sich besonders gut, um einen Datensatz interaktiv zu analysieren. So kann der Anwender sich während einer Postprocessing-Sitzung mit der Graphik-basierten Technik auch in komplexen Fahrzeugmodellen sofort einen Überblick verschaffen, wo Netzknoten beziehungsweise -elemente kritische Werte aufweisen, indem uninteressante Wertebereiche ausgeblendet werden. Doch auch, wenn nur noch ein Bruchteil des Gesamtmodells sichtbar ist, müssen für jedes Bild die gesamten Szenendaten verarbeitet werden; daher resultieren aus dieser Vorgehensweise keine höheren Bildwiederholraten.

7.1.3.2 Werte-basierte Reduzierung des Fahrzeugmodells

Nach der Assemblierung unabhängig voneinander vernetzter Bauteile spielen hohe Bildwiederholraten während der Definition von Bauteilverbindungen eine große Rolle. Darüber hinaus unterscheiden die verwendeten Picking-Algorithmen nicht zwischen opaken und transparenten Polygonen. Um also das Setzen von Schweißpunkten möglichst Benutzerfreundlich zu gestalten, wurde zusätzlich zu dem graphischen Verfahren ein Reduktionsalgorithmus implementiert, der das Modell auf die potenziellen Flanschbereiche einschränkt. Zunächst werden dazu die minimalen Knoten-Element-Abstände ermittelt. Anschließend traversiert die ParamCutAction, eine opDFTravAction (Abschnitt 4.1.3, Seite 65), den Szenengraphen und geht für jedes Bauteilnetz nach dem Algorithmus in Abbildung 7.9 vor. Nachdem der gesamte Szenengraph traversiert wurde, bleiben nur die Teile des Fahrzeugmodells sichtbar, deren Werte in vorgegebenen Bereich liegen. Da nun wesentlich weniger Dreiecke verarbeitet werden müssen, können auch für ursprünglich sehr große Modelle hohe Bildwiederholraten erreicht werden, so dass der Anwender in Echtzeit im reduzierten Fahrzeugmodell navigieren kann.

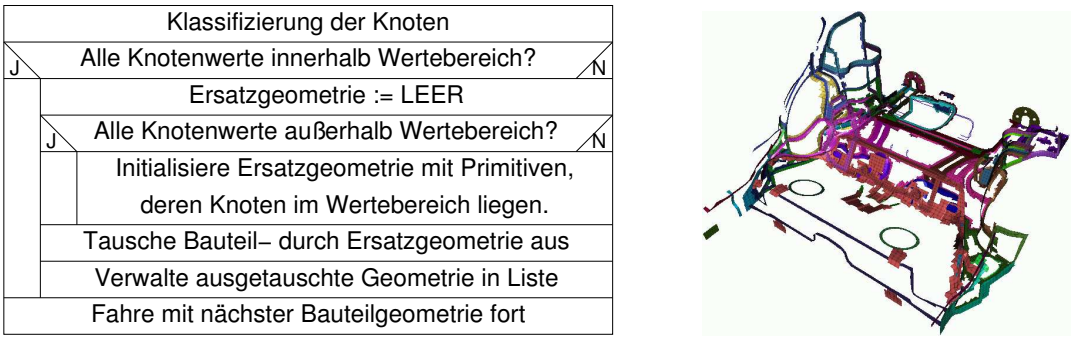


Abbildung 7.9: Die links dargestellten Schritte werden auf jedes Bauteil angewendet, das im Szenengraphen von der ParamCutAction traversiert wird. Als Ergebnis bleiben nur die Finite-Elemente sichtbar, deren Daten im spezifizierten Wertebereich liegen.

7.2 Interaktives Modifizieren von Schweißpunktdaten

Eine der wesentlichen Aufgaben, die mit dem im Rahmen dieser Arbeit entstandenen Programm *crashViewer* gehandhabt werden, ist das interaktive Setzen, Verändern und Löschen von Schweißpunkten. Die endgültigen Schweißpunkte werden von dem Konstruktionsingenieur in den CAD-Daten definiert. Da diese Daten in der frühen Phase eines neuen Fahrzeugprojektes jedoch nicht vorliegen, grundlegende Konstruktionsentscheidungen aber auf der Basis erster Simulationsergebnisse getroffen werden sollen, müssen unabhängig voneinander vernetzte Bauteilmodelle vom Berechnungsingenieur miteinander verbunden werden. Zunächst war die Positionierung von Schweißpunkten nur über die Eingabe ihrer Koordinaten möglich. Daher wurde eine Lösung gesucht, die es dem Berechnungsingenieur erlaubt, interaktiv per Mausklick Schweißpunkte zu definieren beziehungsweise bereits existierende Schweißpunkte zu verschieben oder zu löschen.

Die erste Repräsentation der Schweißpunkte besteht im Simulationseingabedatensatz aus zwei Teilen, der SPOTW-Karte, die außer einem Bezeichner und der Position auch die Zugehörigkeit zu einer Schweißpunktgruppe spezifiziert, und der SLINT2-Karte, in der die Attribute aller Schweißpunkte einer Schweißpunktgruppe angegeben sind. Dementsprechend wurden zwei Klassen Spotweld und SpotweldGroup implementiert, um Schweißpunkte im internen Datenmodell zu repräsentieren (Abbildung 7.10). Die PLINK-Karte als neuere Schweißpunktrepräsentation enthält in ihrem ursprünglichen Format alle der oben genannten Informationen und umgeht damit die Indirektion von Schweißpunkten zu verbundenen Bauteilen.

Abbildung 7.10: Links wird der Zusammenhang der Karten illustriert, die Schweißpunktverbindungen repräsentieren. Rechts sind zwei Schweißpunktgeometrien abgebildet. In Abhängigkeit vom Fehlerstatus eines Schweißpunktes wird die Farbe variiert.

Im Szenengraphen entspricht jede Schweißpunkt-Instanz einem Transformationsknoten, der die Position und Lage des Schweißpunktes widerspiegelt. Alle diese csTransform-Knoten verweisen auf jeweils einen von sechzehn csShape-Knoten, die die eigentliche Schweißpunktgeometrie mit den seinem Status entsprechenden Darstellungsattributen beinhalten.

Beim Setzen eines Schweißpunktes wird zunächst eines der beiden zu verbindenden Bauteile selektiert, das daraufhin temporär als Drahtgittermodell dargestellt wird, um

darunter liegende Bauteile sichtbar werden zu lassen; mit dem zweiten Mausklick legt der Anwender das andere zu verbindende Bauteil und gleichzeitig die Koordinaten eines Suchstrahls fest. Von diesem Punkt aus wird ein senkrecht zur Ebene des Finite-Elementes stehendes Strahlensegment verwendet, um den Schnittpunkt mit dem anderen Bauteilnetz zu ermitteln. Wird in einem maximal zulässigen Abstand ein Element des gewählten Bauteilnetzes gefunden und werden auch alle anderen Kriterien erfüllt, kann der neue Schweißpunkt im Mittelpunkt des Schnittstrahls definiert werden. Außer dem Elementabstand wird überprüft, ob der Mindestabstand zu benachbarten Schweißpunkten sowie eine maximale Winkelabweichung zwischen den Normalen der verbundenen Elemente eingehalten wurde, und ob Elemente dritter Bauteile zwischen den zu verbindenden Elementen liegen. Nach wie vor lassen sich einzelne Schweißpunkte auch über Koordinatenangabe setzen; falls die zu verbindenden Bauteile zuvor nicht selektiert wurden, werden die beiden nächstgelegenen Bauteile verschweißt. Auch hier kommt wieder die Distanzberechnung mit Hilfe der BV-Hierarchie zum Einsatz. Mehrere Schweißpunkte können entlang eines auf das Bauteil projizierten Liniensegments definiert werden, indem der Start- und Endpunkt des Segments durch Mausklick vorgegeben wird. Eine detaillierte Beschreibung der notwendigen Berechnungsschritte wird in [40], Kapitel 4.1 gegeben.

Schweißpunkte, die bereits in dem Eingabedatensatz definiert waren, werden in der Initialisierungsphase ebenfalls auf die oben genannten Kriterien überprüft und entsprechend klassifiziert. Der Anwender kann fehlerfreie von fehlerhaften Schweißpunkten durch unterschiedliche Farben und gegebenfalls eine andere geometrische Repräsentation unterscheiden. Um ein sukzessives Korrigieren fehlerhafter Schweißpunkte zu ermöglichen, wurde zudem eine Suchfunktionalität implementiert, die die Viewing-Matrix so einstellt, dass der nächste Schweißpunkt aus der Fehlerliste den Darstellungsbereich füllend visualisiert wird.

Schweißpunktverbindungen über mehr als zwei Bauteile werden derzeit durch Definition mehrerer Zweifachverbindungen mit gleichen Koordinaten und unterschiedlichen Schweißpunktgruppen abgebildet. Da jeder dieser Schweißpunkte mit unterschiedlichen Bauteilpaarungen durch einen Quader repräsentiert wird, der lediglich durch seinen Mittelpunkt und seine längste Achse spezifiziert ist, können sie sich gegenseitig verdecken. Um Mehrfachschweißpunkte von einfachen zu unterscheiden, werden die Quader um ihre längste Achse auseinander rotiert (Abbildung 7.11).

Schließlich werden in *crashViewer* noch Mechanismen zur Verfügung gestellt, die es dem Anwender erlauben, Schweißpunktverbindungen zu prüfen. Wird ein Schweißpunkt selektiert, so werden die durch diesen Schweißpunkt verbundenen Bauteilnetze temporär als Drahtgitter dargestellt. Umgekehrt kann auch ein Bauteil ausgewählt werden, woraufhin sich alle Schweißpunkte an diesem Bauteil verfärben. Durch eine zweite Bauteilauswahl werden von diesen Schweißpunkten diejenigen hervorgehoben, die beide Bauteile verbinden. Außerdem gibt es noch die Möglichkeit, ausgehend von einem allein dargestellten Bauteilnetz sich schrittweise alle bis dahin ausgeblendeten Bauteile anzeigen zu lassen, die mit einem dargestellten Bauteil verbunden sind. So kann am Ende einer Assemblierungssitzung noch einmal getestet werden, ob alle notwendigen Bauteilanbindungen definiert wurden oder ob Bauteile noch unangebunden im Modell „schweben“.

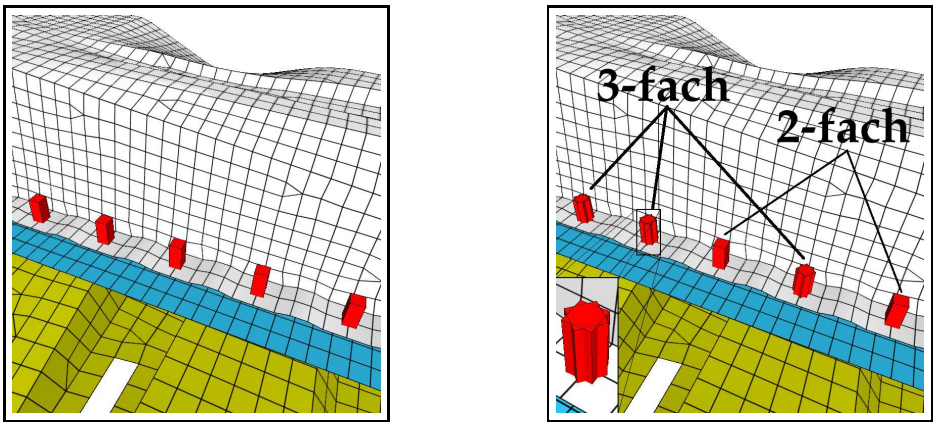


Abbildung 7.11: Schweißpunkte, die mehr als zwei Bauteile verbinden, werden mit der SPOTW-Karte durch mehrere Zweifachverbindungen im gleichen Punkt mit unterschiedlichen Bauteilpaarungen modelliert. Eine zusätzliche Rotation der geometrischen Repräsentation macht derartige Mehrfachverbindungen von den normalen Zweifachverbindungen für den Betrachter unterscheidbar.

7.3 Parameterübertragung zwischen inkompatiblen Gittern

Im virtuellen Fahrzeugentwicklungsprozess bekommt die Verzahnung der Berechnungsergebnisse aus den unterschiedlichen Bereichen, in denen die numerische Simulation eingesetzt wird, einen wachsenden Stellenwert. Die derzeit in der Crash-Simulation eingesetzten Berechnungsmodelle verwenden Werkstoffkenngrößen, die zuvor an Halbzeugen[2] ermittelt wurden. Dadurch bleiben bisher weitere Fertigungseinflüsse, die bei der Umformung von Halbzeugen zu fertigen Bauteilen durch Abstreck- und Stauchvorgänge auftreten, in der Crash-Simulation unberücksichtigt [34]. Zum Beispiel wird beim Tiefziehen in Bereichen großer Krümmung die Dicke des Bauteilbleches vermindert. Statt in der Crash-Simulation von einer gleichbleibenden Blechdicke über alle Elemente des Bauteils auszugehen, sollen die Ergebnisse einer Tiefziehsimulation in der Crash-Berechnung weiterverwendet werden.

In einer Tiefziehsimulation werden Faltenbildung und Materialversagen des Umformteils untersucht und Werkzeuge wie Matrize, Niederhalter oder Stempel optimiert. Um aussagekräftige Ergebnisse zu erhalten, werden in Tiefziehsimulationen wesentlich feinere Finite-Element-Netze eingesetzt als in Crash-Simulationen. Zudem verfeinert sich die Netzstruktur in Bereichen großer Krümmung. Um nun die aus der Fertigung resultierenden Elementdicken des feinen Tiefziehnetzes auf das gröbere Crash-Netz übertragen zu können, müssen für jedes Element des Crash-Netzes die Elemente des Tiefziehnetzes ermittelt werden, die dann zum Beispiel nach Ihrer Fläche gewichtet auf die zu berechnende

[2] *Halbzeug* — Stahlkörper, die bereits eine Formgebung im Stahlwerk erhalten haben und aus denen im Allgemeinen durch weitere Umformung Fertigerzeugnisse hergestellt werden

Elementdicke Einfluss nehmen. Nachfolgend wird hier das zusammen mit Kobbelt und Westermann entwickelte Verfahren beschrieben, das in den Prototypen *crashViewer* integriert und zum Einsatz gebracht wurde.

```
1    Kodiere Indizes der Elemente Ef des feinen Netzes als RGB-Farbe
2    Initialisiere Raumpartitionierung und Display-Listen mit enthaltenen Ef
3    Setze allgemeine Darstellungsparameter für OpenGL-State

4    for (alle Elemente Eg des groben Netzes)
5       Berechne für Eg Mittelpunkt M, Normale N und Look-up-Vektor LuV
6       Positioniere Kamera in M mit Orientierung (N, LuV)
7       Initialisiere orthographische Projektion anhand längster Kante von Eg
8       P := Partitionen die von Bounding-Box(Eg) geschnitten werden
9       Wähle Viewport-Auflösung groß genug und beschränke Scissor-Bereich
10      Initialisiere Stencil-Buffer mit Fläche von Eg
11      Zeichne alle Ef aus P innerhalb des Stencil-Bereichs
12      Lese Scissor-Bereich aus dem Framebuffer in den Hauptspeicher
13      Dekodiere aus den Pixelfarben die Indizes der beitragenden Ef
14      und wende gewählte Abbildungsfunktion an
15      Weise ermittelte Größe(n) Eg zu
```

Abbildung 7.12: Pseudocode der Transformation von Element-basierten Größen zwischen Netzen unterschiedlicher Auflösung

Als Eingangsdaten für den in Abbildung 7.12 dargestellten Pseudocode für die Hardware-basierte Transformation Element-basierter Größen liegen zunächst die beiden Netze unterschiedlicher Auflösung aber gleicher geometrischer Form, Position und Orientierung vor. Der Algorithmus nutzt die Graphik-Hardware, um zu ermitteln, welche Elemente $\mathtt{E_f}$ des feinen Netzes ein Element $\mathtt{E_g}$ des groben Netzes überdecken. Dazu ist es notwendig, die Elemente $\mathtt{E_f}$ durch einen Identifikator – in diesem Fall einen aufsteigenden Index – eindeutig zu markieren. Um die Information im Graphiksubsystem verarbeiten zu können, wird jeder Index nach folgendem Schema binär in eine Farbe kodiert (1):

$$C = R_{r-1} \ldots R_0 G_{g-1} \ldots G_0 B_{b-1} \ldots B_0 = I_{i-1} \ldots I_0 + 1$$

Dabei entsprechen r, g und b den Bittiefen des Bildspeichers für den Rot-, Grün- und Blaukanal. Sie werden jeweils in den Bereich $[0, 1]$ verschoben. Sollte die Elementanzahl des feinen Netzes die Bildspeichertiefe überschreiten, also $I \geq 2^i - 1 > 2^{r+g+b}$ gelten, dann kommt ein Mehrschrittverfahren zum Einsatz, in dem jedem Element $\mathtt{E_f}$ mehrere Farben für die verschiedenen Bitbereiche zugeordnet werden. Die Farbe Schwarz bleibt für uninitialisierte Bereiche reserviert, daher die '+1'.

Um die Graphik zu entlasten und für jedes $\mathtt{E_g}$ nur potenziell beitragende $\mathtt{E_f}$ zu berücksichtigen, wird das feine Netz durch eine Raumpartitionierung unterteilt und jeder Zelle eine Display-Liste zugeordnet, die die in ihr enthaltenen $\mathtt{E_f}$ in ihren jeweiligen Farben zeichnet (2).

Da die in den Farben kodierten Indizes nicht verfälscht werden dürfen, wird Antialiasing, Beleuchtungsberechnung und Blending aus- und *Flat-Shading* eingeschaltet (3). Die nachfolgend beschriebenen Schritte werden für jedes Element $\mathtt{E_g}$ des groben Netzes durchgeführt

(4). Die Kamera wird in der Mitte von E_g positioniert und blickt entlang der Elementnormalen (5+6). Die längste Kante von E_g gibt die Höhe und Breite des Sichtvolumens vor, dessen Tiefe durch einen einstellbaren Toleranzwert, dem maximal zulässigen Abstand zwischen beiden Netzgeometrien, spezifiziert wird (7). Nachdem die Zellen der Raumpartitionierung ermittelt wurden, die das Sichtvolumen schneiden (8), wird der Viewport-Bereich, in dem die graphischen Operationen durchgeführt werden, so groß gewählt, das selbst das kleinste aktuell ausgewählte Element E_f auf mindestens einem Pixel dargestellt wird (9). Mit glScissor werden alle Pixeloperationen auf diesen Bereich beschränkt. Mit Hilfe des Stencil-Buffers wird sichergestellt, dass nur Pixel beschrieben werden, die auch von E_g überdeckt werden (10). Nachdem alle das Sichtvolumen schneidenden Elemente E_f mit ihren jeweiligen „Index-Farben“ gezeichnet wurden (11), wird das resultierende Bild in den Hauptspeicher geholt (12) und die Farbe jedes Pixels in den Index des beitragenden Elementes E_f zurücktransformiert(13). Schwarz gebliebene Pixel weisen darauf hin, dass an der Stelle kein zugehöriges E_f gefunden wurde. Welche der zu transformierenden Größen von den Elementen E_f auf E_g zu übertragen sind, hängt von der gewählten Abbildungsfunktion ab (14). POINT bestimmt E_f über das Pixel in der Mitte des Viewport-Bereichs, während AREA die jeweilige Größe über alle E_f mittelt. Darüber hinaus können mit MIN und MAX auch minimale oder maximale Werte übertragen werden. Die transformierten Werte stehen anschließend für jedes E_g zur Verfügung (15) und können somit für Berechnungen mit dem groben Netz weiterverwendet werden. Abbildung 7.13 zeigt die Übertragung der Elementbasierten Blechdicke von einem lokal verfeinerten Tiefzieh- auf ein Crash-Netz mit gröberer Auflösung.

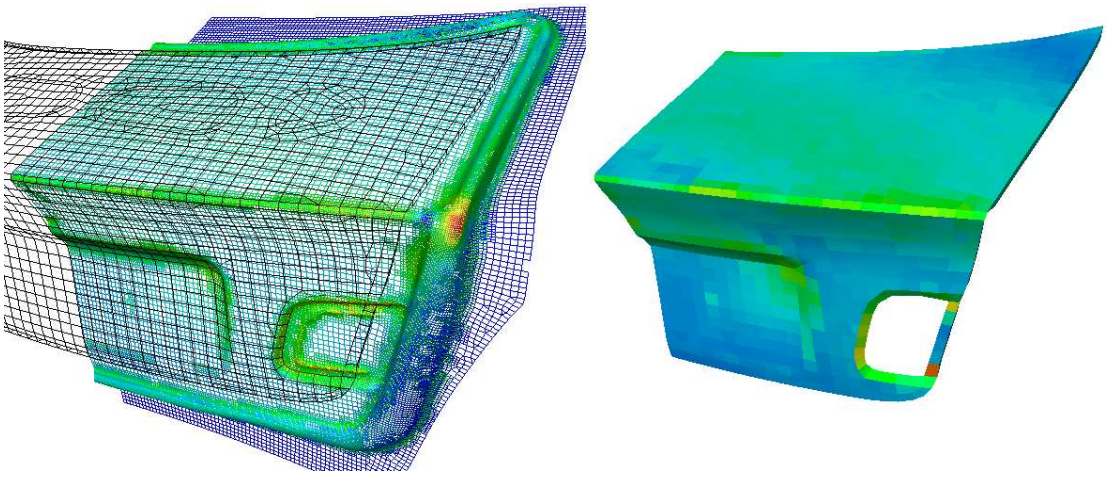


Abbildung 7.13: Links liegen die beiden Finite-Element-Netze unterschiedlicher Auflösung in gleicher Position und Lage übereinander – das Crash-Netz schwarz und das Tiefziehnetz mit der visualisierten Blechdicke. Rechts werden die übertragenen Werte als Ergebnis der Hardware-basierten Transformation auf dem Crash-Netz dargestellt.

Kapitel 8

Spezielle Post-Processing Funktionalitäten

Die großen Datenmengen, die als Ergebnisse aus einem Crash-Simulationsprozess hervorgehen, müssen für die Analyse aufbereitet und visualisiert werden. In diesem Kapitel wird der Einsatz von Mapping-Verfahren zur Visualisierung skalarer Größen auf der Finite-Element-Struktur an verschiedenen Beispielen diskutiert. Darüber hinaus zeigt die Verwendung animierter Vektorpfeile und farbiger Kraftflussröhren, wie auch mehrdimensionale Daten durch Einbringen zusätzlicher Geometrie veranschaulicht werden können. Abschließend wird das Kommandozeilen-gesteuerte Batch-Programm *buildGraph* und dessen Einsatzgebiete beschrieben.

8.1 Skalarwerte auf der Fahrzeuggeometrie

Um einen möglichst engen Bezug zwischen Daten und dem Ort, an dem die Daten vorliegen, herzustellen, bietet sich die Visualisierung skalarer Größen durch entsprechende Einfärbung der Geometrie an. In Kapitel 6.3.1, Seite 102 wurde bereits die Nutzung eindimensionaler RGBα-Texturen zur Visualisierung Knoten-basierter Skalare erläutert. Anhand einiger Beispiele werden hier Bilder gezeigt, in denen die Daten Textur-basiert visualisiert wurden. Neben den bereits in der Simulation berechneten Skalaren, wie zum Beispiel die minimale / maximale plastische Dehnung der Elemente, die direkt auf Farbwerte abgebildet werden können, dienen die abgespeicherten Größen in den nachfolgenden Beispielen als Eingangsgrößen für die Berechnung der zu visualisierenden Werte.

Beulgeschwindigkeit

Die Darstellung der Beulgeschwindigkeit soll Regionen auf Längsstrukturen hervorheben, in denen Elemente seitlich ausbeulen. Zur Berechnung eines skalaren Wertes, der anschließend farbkodiert dargestellt werden kann, wird die Knotengeschwindigkeit in Richtung

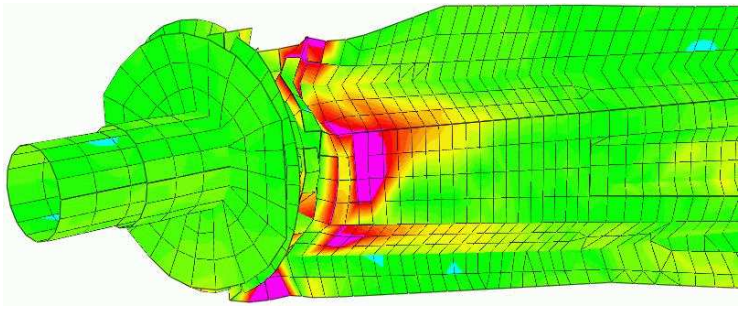


Abbildung 8.1: Die Visualisierung der Beulgeschwindigkeit hebt insbesondere auf Strukturen, deren Ebenen in Richtung der Starrkörperbewegung verlaufen, ausbeulende Bereiche hervor, wie hier am Längsträger sichtbar.

des Normalenvektors ermittelt. Durch diese Vorgehensweise kann allerdings lediglich über Strukturen, deren Normale annähernd senkrecht zur Starrkörperbewegung des Fahrzeugs orientiert ist, eine Aussage der Beulgeschwindigkeit gemacht werden, da andernfalls die Ausbeulung von der Starrkörperbewegung überlagert wird.

Hourglassing

Durch die auf einen Punkt reduzierte Integration können in vierseitigen Schalenelementen Null-Energie-Moden, so genannte *Hourglass*-Effekte auftreten. Dabei handelt es sich um eine oszillierende zick-zack-artige Verformung benachbarter Schalenelemente (siehe Abbildung 8.2) in der Elementebene, die die Simulationsergebnisse unbrauchbar machen. Derartige *Hourglass*-Deformationen lassen sich durch eine uniforme Vernetzung und Ausschluss zu großer punktförmiger Lasten einschränken. Darüber hinaus können signifikante *Hourglass*-Deformationen durch Entgegenwirken der *Hourglass*-Energien während der Simulation kontrolliert werden. Um sicherzustellen, dass ein zulässiges Maß an *Hourglass*-Energie nicht überschritten wurde, beziehungsweise die Elemente mit verstärkt auftretendem *Hourglassing* zu identifizieren, wird das Verhältnis zwischen der *Hourglass*- und der internen Energie eines Schalenelementes als Quotient Element-basiert visualisiert.

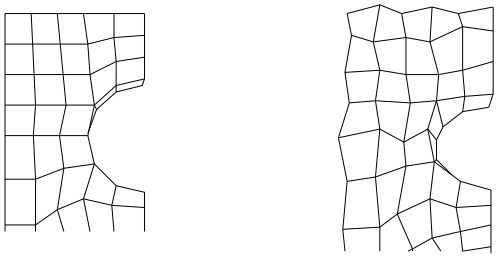


Abbildung 8.2: Signifikante *Hourglass*-Effekte verfälschen die Simulationsergebnisse. Daher ist eine Überprüfung aufgetretener *Hourglass*-Energien unverzichtbar.

Intrusion

Die Eindringtiefe in die Fahrgastzelle ist eine kritische Größe bei der Beurteilung der passiven Sicherheit während eines Front- beziehungsweise Seitenaufpralls. Um die Eindringtiefe zu visualisieren, wird eine Referenzebene definiert, zu der der jeweilige Netzknotenabstand zunächst gemessen und dann in Relation zu seinem ursprünglichen Abstand gesetzt wird. Da sich das Fahrzeug während eines Seitenaufpralls verschiebt, kann die Referenzebene dynamisch durch Selektion dreier Netzknoten auf der anderen Fahrzeugseite definiert werden. Der Algorithmus geht wie folgt vor:

```
Initialisiere Normale N_p^0 der Referenzebene und ihren Abstand d_p^0 zum Ursprung
Für jeden Knoten i
    Berechne den initialen Abstand für Punkt P_i zur Referenzebene: d_i^0 = P_i · N_p^0 − d_p^0
Für jeden Zeitschritt {
    Aktualisiere Referenzebene (N_p und d_p) anhand der Referenzknotenpositionen
    Für jeden Knoten i {
        Berechne den aktuellen Abstand für Punkt P_i zur Referenzebene: d_i = P_i · N_p − d_p
        Stelle Eindringtiefe t_i als Farbe dar, t_i = d_i − d_i^0
    }
}
```

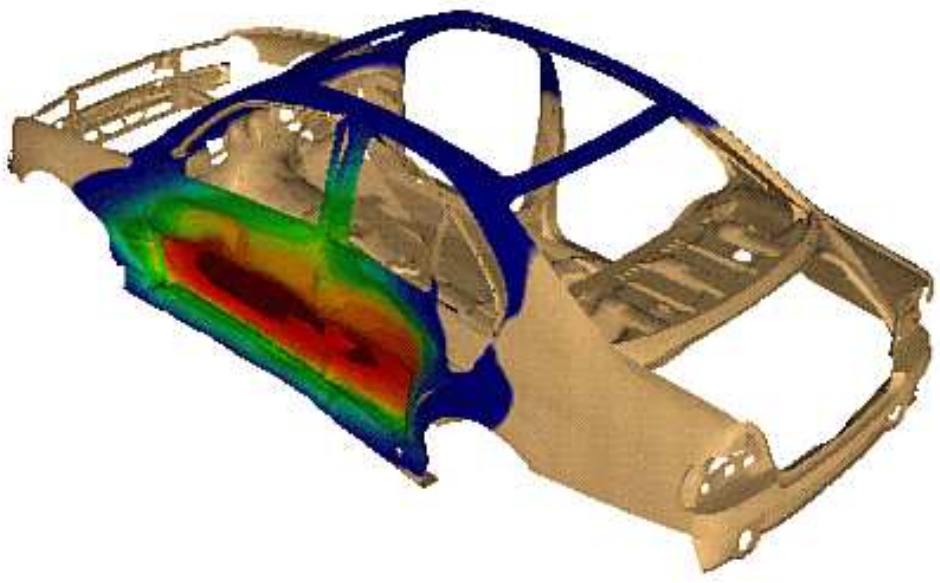


Abbildung 8.3: Darstellung der Eindringtiefe beim Seitenaufprall

8.2 Animierte Darstellung vektorieller Daten

Als Simulationsergebnisse werden nicht nur skalare Werte ausgegeben; auch vektorielle Größen, wie zum Beispiel die Knotengeschwindigkeit oder -beschleunigung, gilt es, zu visualisieren. Häufig werden Vektoren als Pfeile dargestellt: eine Linie repräsentiert die Lage des Vektors und mit zwei weiteren Linien (Pfeilspitze) wird die Richtung angegeben. Die Verwendung solcher Pfeile hat bei einer hohen Dichte den Nachteil, dass darunterliegende Strukturen verdeckt werden.

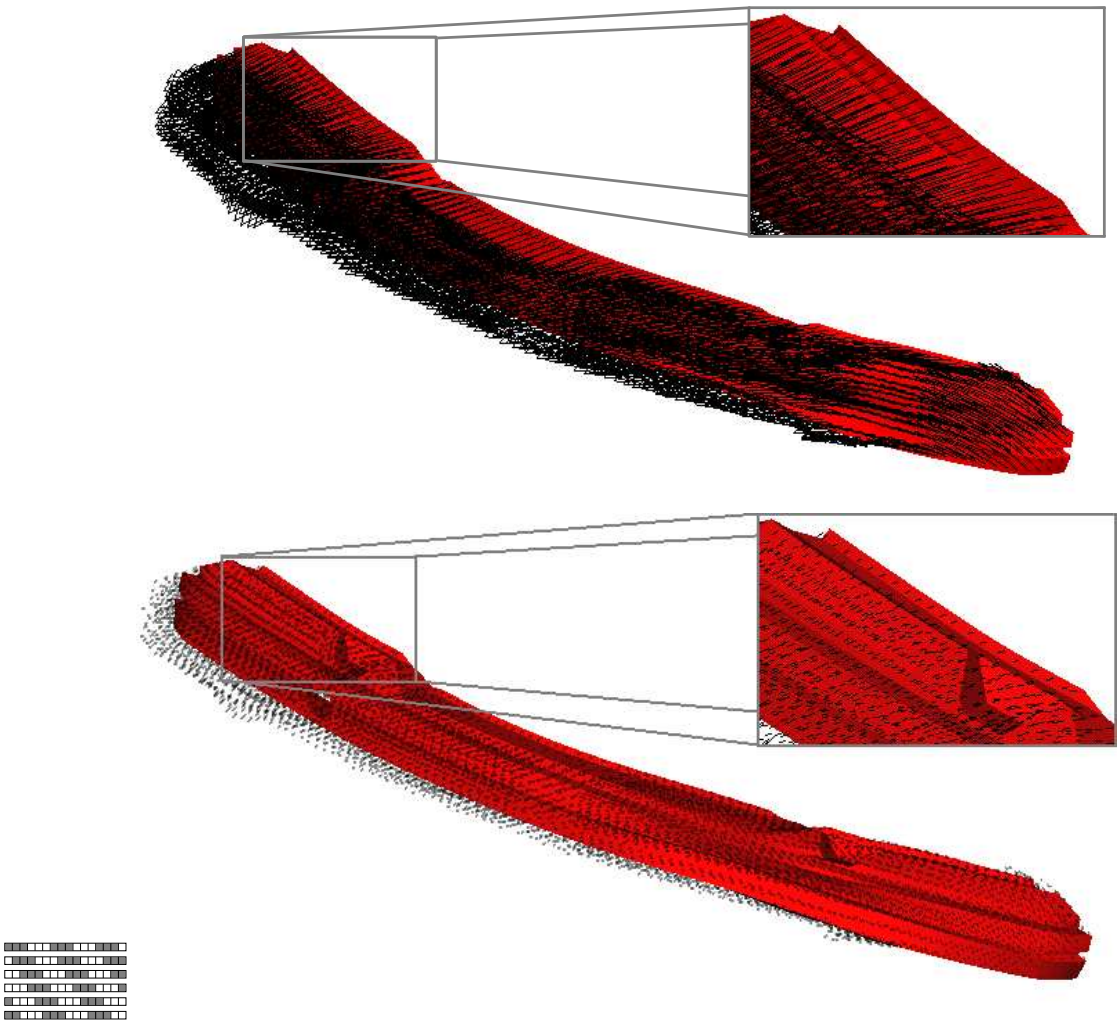


Abbildung 8.4: Das obere Bild zeigt die Knotenbeschleunigung an dem Finite-Element-Modell einer vorderen Stoßstange in der herkömmlichen Darstellungsweise durch Vektorpfeile. Die Struktur der Stoßstange bleibt weitestgehend unter den dichten Vektorpfeilen verborgen. Im Bild unten werden die Vektoren durch Linien mit animierter Opazität visualisiert, wodurch eine bessere Sicht auf die darunterliegende Stoßstangenstruktur gewährleistet ist. Unten links sind die verwendeten 1D-α-Texturen skizziert.

Die Abbildung 8.4, oben zeigt, wie durch die traditionelle Darstellung von Vektoren die Sicht auf das darunterliegende Finite-Element-Modell verdeckt wird. Um Vektordaten zu visualisieren, ohne die datentragenden Strukturen zu verdecken, wurde eine Idee von Yamrom [76] aufgegriffen, in der Strömungsdaten durch animierte Texturen auf Linien repräsentiert werden, und an die Darstellung vektorieller Größen im Bereich der nichtlinearen Strukturmechaniksimulation angepasst [46].

Im Gegensatz zur herkömmlichen Darstellungsweise werden keine Pfeilspitzen verwendet; stattdessen wird die Richtung durch die Bewegung der opaken Liniensegmente codiert. Die scheinbare Bewegung wird durch sechs 1D-α-Texturen hervorgerufen, die permutierend auf die Linien angewendet werden. Jede dieser Texturen besteht aus 16 Texeln, die in verschobener Reihenfolge opak oder transparent sind. Außer einer geringeren Verdeckung hat diese Visualisierungstechnik auch den Vorteil, dass nur ein Drittel der Linien gezeichnet werden muss. Für statische Abbildungen, zum Beispiel einen Ausdruck der Auswertung, muss gegebenenfalls auf eine Pfeildarstellung zurückgegriffen werden, sofern die Vektorrichtung nicht schon wie in Abbildung 8.4 aus der Anordnung der Linie hervorgeht.

8.3 Kraftflussvisualisierung

Wenn ein Fahrzeug im Crash-Test auf eine Barriere trifft, wird ein Großteil der Bewegungsenergie durch die Verformung der Fahrzeugstruktur absorbiert. Bei der Auswertung sind die Ingenieure daran interessiert, wieviel Kraft durch welche Fahrzeugbauteile weitergeleitet beziehungsweise durch Verformungen absorbiert wird. Um den Kraftfluss durch Teile der Fahrzeugstruktur darzustellen, wurde eine von Kuschfeldt et al. [44] vorgestellte und patentierte [45] Visualisierungstechnik weiterentwickelt und damit dem produktiven Einsatz in der Simulationsergebnisanalyse zur Verfügung gestellt.

Hierbei wird zunächst ein Linienzug, im Folgenden als *Traceline* bezeichnet, im Modell spezifiziert, entlang dessen der Kraftfluss durch umliegende Strukturen untersucht werden soll. Anschließend werden in geringen Abständen Schnittebenen entlang dieser Traceline definiert, die jeweils orthogonal zu dem zugehörigen Liniensegment liegen. Schließlich wird die Schnittkraft durch das Modell ermittelt und in Form eines Ringes in der Schnittebene dargestellt. Die zu einer Röhre verbundenen Ringe veranschaulichen den Kraftfluss durch Durchmesser, Form und Farbe (Abbildung 8.5).

Um den Ansatz der Kraftflussvisualisierung in der prototypischen Anwendung *crashViewer* für Berechnungsingenieure produktiv einsetzbar zu machen, mussten folgende Anforderungen erfüllt werden:

- Interaktionsmechanismus zur interaktiven Spezifikation der Traceline
- Effiziente Berechnung der Kraftflussröhren
- Entkoppelte Vorberechnung von Kraftflussröhren mit der Möglichkeit, diese Daten während einer Visualisierungssitzung einzulesen

Die folgenden Abschnitte erläutern, wie diesen Anforderungen Rechnung getragen wurde.

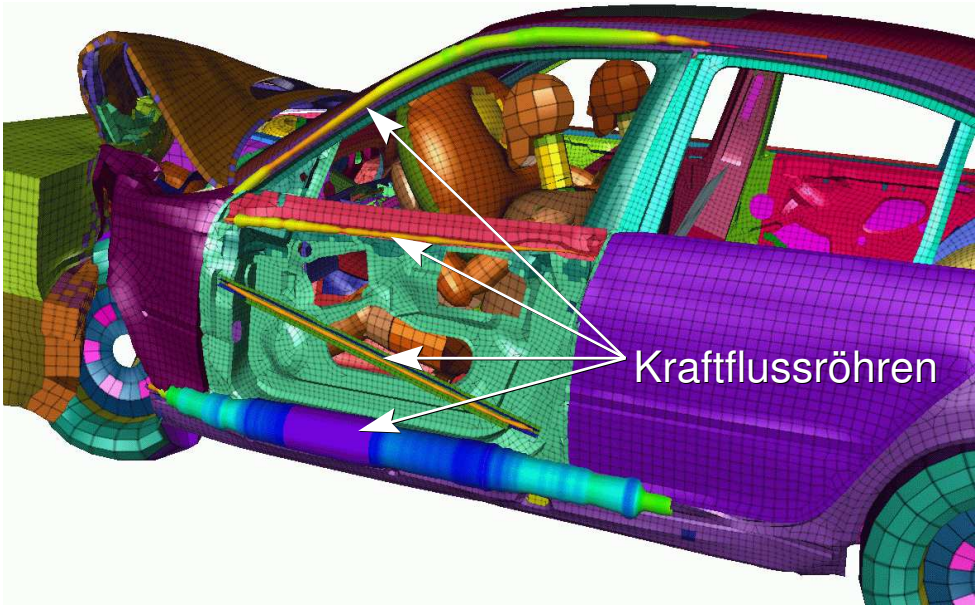


Abbildung 8.5: Die Kraftflussvisualisierung durch Kraftflussröhren veranschaulicht in jedem Zeitschritt, wie groß die Schnittkräfte sind, die in den jeweils ausgewählten Strukturen auftreten. Die Möglichkeit, den Kraftfluss in verschiedenen Fahrzeugstrukturen direkt miteinander vergleichen zu können, führt zu einer verbesserten Wahrnehmung gegenüber herkömmlicher 2D-Schnittkraftdiagramme.

8.3.1 Interaktive Definition von Tracelines

Eine Traceline wird entweder durch statische oder dynamische Stützpunkte definiert. Bei einer statischen Traceline sind die Stützpunkte durch feste Raumpunkte fixiert und bleiben für die Berechnung der Kraftflussröhre über alle Zeitschritte unverändert. In einer dynamischen Traceline werden die Stützpunkte mit Netzknoten der Fahrzeugstruktur identifiziert; dadurch passt sich die Traceline im zeitlichen Verlauf der sich deformierenden Fahrzeugstruktur an. Die Stützpunkte der Traceline können vom Anwender interaktiv per Mausklick an der Fahrzeugstruktur festgelegt werden.

Die Berechnung des Kraftflusses innerhalb von Teilstrukturen eines Gesamtfahrzeugmodells setzt voraus, dass die zu berücksichtigenden Fahrzeugteile vom Anwender zuvor selektiert werden können. Darüber hinaus ist es erforderlich, innerhalb eines Fahrzeugteils die Auswahl auf einen Teil der Finite-Elemente beschränken zu können: Abbildung 8.6 zeigt, wie mit Hilfe eines halbtransparenten Selektionsschlauches um den oberen Bereich des Seitenrahmens eine engere Auswahl der für die Kraftflussberechnung zu berücksichtigenden Finite-Elemente getroffen werden kann. Andernfalls wäre es nicht möglich, den Kraftfluss innerhalb des Seitenrahmens zwischen Dach- und Schwellerbereich (vergleiche Abbildung 8.5) zu unterscheiden. Nachdem der Radius des Selektionsschlauches, in dessen Mitte sich die Traceline befindet, und gegebenenfalls ein Verschiebungsvektor festgelegt

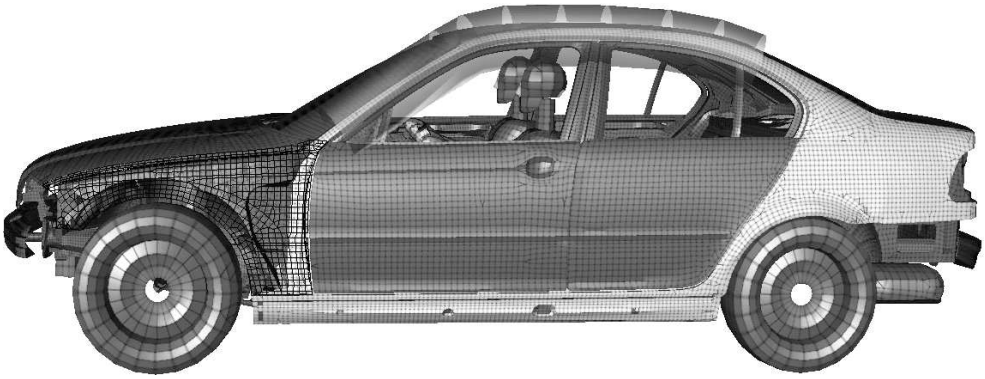


Abbildung 8.6: Der semitransparent dargestellte, interaktiv modifizierbare Selektionsschlauch schränkt für die Kraftflussberechnung die Auswahl der zu berücksichtigenden Finite-Elemente im Seitenrahmen auf den oberen Bereich ein.

wurde, werden die Elemente der ausgewählten Fahrzeugteile ermittelt, von denen mindestens ein Netzknoten innerhalb des Selektionsschlauches liegt. Dies ergibt die Menge der beitragenden Elemente für die im folgenden Abschnitt beschriebene Kraftflussberechnung.

8.3.2 Kraftflussberechnung

Die Ermittlung der von einer Ebene geschnittenen Elemente wurde durch den Einsatz der Bounding-Volume-Hierarchie (Kapitel 7.1.1, Seite 108) stark beschleunigt, so dass diese Visualisierungsmethode nun auch interaktiv einsetzbar ist. Dabei wird die Menge der durch die Bauteilauswahl und den Selektionsschlauch eingegrenzten Finite-Elemente in einer Bounding-Volume-Hierarchie unterteilt. Als Volumenprimitiv wird hierbei die Kugel gewählt, da sich somit der Test, ob die aktuelle Schnittebene die umhüllten Elemente schneiden könnte, auf

$$\text{Kugelradius} < \left\| \left(\text{Kugelmitte} - \text{Ebenenpunkt}\right) \cdot \text{Ebenennormale} \right\|$$

beschränkt. Für die wenigen Elemente, die nach diesem Test noch als geschnitten in Frage kommen, wird die Lage ihrer Knoten bezüglich der Ebene getestet. Sofern Elementknoten auf beiden Seiten der Ebene liegen, wird das Element in die Liste der beitragenden Elemente aufgenommen. Schließlich wird mit dem in Abbildung 8.7 beschriebenen Algorithmus die Schnittkraftsumme für die aktuelle Schnittebene berechnet. Das Resultat wird als Ausgangswert für den skalierten Radius eines Ringes genommen, der in der Schnittebene liegt und um die Traceline verläuft.

8.3.3 Entkoppelte Vorberechnung

Die Vorberechnung des Kraftflusses entlang spezifizierter Tracelines aus den Simulationsergebnissen beschleunigt die Analyse. Für ein Fahrzeugmodell können die interaktiv defi-

```
Für alle geschnittenen Elemente
    Ermittle lokales Koordinatensystem
    Für alle Knoten des aktuellen Elementes
        Wenn Knoten auf beitragender Seite der Ebene liegt,
            summiere aus den Simulationsdaten eingelesene Knotenkraft auf
    Transformiere lokal aufsummierte Knotenkräfte ins Schnittebenensystem
    Summiere den Schnittkraftbeitrag für die Schnittebene auf
```

Abbildung 8.7: Pseudocode für die Aufsummierung der Schnittkräfte beitragender Elemente für eine Schnittebene.

nierten Tracelines separat abgespeichert werden (Abbildung 8.8). Über eine weitere Datei, in der die Definitionsdateien der vorzuberechnenden Kraftflussröhren aufgelistet sind, wird das Batch-Programm *buildGraph* gesteuert. Die Vorgehensweise beschreibt der Pseudocode in Abbildung 8.9. Die errechneten Schnittkraftsummen werden ebenfalls im ASCII-Format abgespeichert, um das direkte Lesen mit einem Editor und die Weiterverarbeitung mit anderen Werkzeugen zur Diagrammerstellung zu ermöglichen. Außer je einer Schnittkraftsumme für jede Schnittebene und jeden Zeitschritt enthält die Datei einen Verweis auf die Traceline-Definitionsdatei. Der Prototyp *crashViewer* kann die Ergebnisdatei interaktiv einlesen und den vorberechneten Kraftfluss entlang der Traceline, wie in Abbildung 8.5 dargestellt, visualisieren, ohne dass der Anwender auf weitere Berechnungen warten muss.

```
SelectionTube: 100.0 0 0 0
StartSelectedMaterialLabels
    15011 15021 15031 15051 15151 15221
    15251 16021 16031 17021 17041 20000
EndSelectedMaterialLabels
Type:          DYNAMIC
TorusSegs:     12
PlaneDist:     8.0
ScaleFactor:   0.002
StartTraceLineDefinition
    Point:    799.940    -733.350    715.260
    Point:   1078.750    -658.640    885.520
    Point:   1347.420    -585.570   1008.320
    Point:   1845.730    -555.165   1054.910
    Point:   2351.240    -557.580   1037.340
    Point:   2628.320    -609.430    952.260
    Point:   2826.870    -668.180    833.970
EndTraceLineDefinition
```

Abbildung 8.8: Dieses Beispiel einer Traceline-Definitionsdatei erzeugt eine Kraftflussröhre, die aus sechs Segmenten besteht. Entlang des Polygonzugs werden die Schnitte durch die zwölf selektierten Modellstrukturen im angegebenen Abstand gelegt. Die Auswahl der zu berücksichtigenden Elemente wird zusätzlich durch einen Selektionsschlauch mit einem Radius von 100 mm eingegrenzt.

```
Ermittle alle benötigten Bauteilstrukturen
Einlesen eines minimalen Modells
Erstelle Liste zu berücksichtigender Elemente pro Kraftflussröhre
Für alle Zeitschritte der Simulationsergebnisse
    Für alle Traceline-Definitionen
        Berechne die Schnittkraftsumme pro Schnittebene
Für alle Traceline-Definitionen
    Speichere die Daten in Schnittkraftdateien
```

Abbildung 8.9: Pseudocode für das Vorgehen bei der Vorberechnung von Kraftflussröhren im Batch-Programm *buildGraph*.

8.4 Visualisierung von Instabilitäten

Die Ergebnisse der Crash-Simulation unterliegen einer großen Streuung, deren Ursachen sich wie folgt gliedern lassen:

- Äußere Randbedingungen, wie zum Beispiel Steifigkeit der Barriere oder Auftreffgeschwindigkeit und -winkel, beeinflussen die Simulationsergebnisse massiv.
- Wechselnde Materialeigenschaften der verwendeten Bauteile verändern das Crash-Verhalten eines Fahrzeugs.
- Form der Bauteile und die Art ihrer Verbindung untereinander haben ebenfalls maßgeblichen Einfluss.
- Natürliche Streuung in Form von Zufallsentscheidungen tritt an Verzweigungsstellen auf, an denen zum Beispiel eine schalenförmige Bauteilfläche senkrecht auf einen Widerstand trifft und daraufhin entweder zur einen oder anderen Seite abgleiten kann.
- Numerische Rundungsfehler führen besonders dann zu großen Abweichungen, wenn sie an Verzweigungsstellen zu einem anderen Verlauf der Verformung führen.

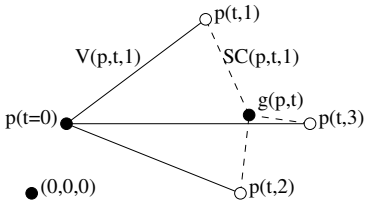
Durch Variation der Einflussfaktoren, die unter die ersten drei Punkte fallen, soll im Rahmen stochastischer Simulationsverfahren eine allgemeinere Aussage über das Crash-Verhalten eines Fahrzeugmodells gemacht werden können. Darüber hinaus sind die Fahrzeugentwickler daran interessiert, durch Veränderung der Materialeigenschaften und der Bauteilformen neben dem Crash-Verhalten, auch weitere Zielgrößen wie beispielsweise das Fahrzeuggewicht zu optimieren. Die beiden letztgenannten Kategorien von Ursachen, die für Streuung in den Simulationsergebnissen verantwortlich sein können, sorgen sogar bei gleichbleibenden Eingabedaten (Randbedingungen, Fahrzeugmodell etc.) für abweichende Ergebnisse und werden als *Instabilität* des Simulationsverfahrens bezeichnet.

Die Voraussetzung für eine gesteigerte Aussagekraft der Simulationsergebnisse und die Möglichkeit einer effektiven Optimierung des Fahrzeugmodells durch Variation des Modellaufbaus ist die Identifizierung und Beseitigung solcher Instabilitäten. In [64] werden erstmals die folgenden Methoden zur Lokalisierung vorgestellt und deren Implementierung beschrieben.

8.4.1 Maße für Instabilität

Um Abweichungen bei Simulationsergebnissen, die aus den gleichen Eingabedaten hervorgegangen sind, zu erkennen, können verschiedene Maße eingesetzt werden. Die Maße lassen sich danach unterscheiden, ob sie ein globales oder ein lokales Kriterium repräsentieren und ob die Daten ein- oder mehrdimensional betrachtet werden.

Abbildung 8.10: Die Skizze zeigt den Netzknoten p im initialen Zeitschritt und nach t Zeitschritten für drei verschiedene Simulationsläufe. Die durchgezogenen Linien markieren die Verschiebungsfunktion $\mathrm{V}(p,t,r)$, die gestrichelten Linien die Streufunktion $\mathrm{SC}(p,t,r)$.



Die beiden in Abbildung 8.10 skizzierten Funktionen stellen Beispiele für globale Maße der Instabilität dar und sind wie folgt definiert:

- Die Funktion $\mathrm{V}(p,t,r) = p(t,r) - p(0,r)$ misst die Verschiebung eines Netzknotens p im Zeitschritt t von seiner ursprünglichen Position im initialen Zeitschritt für den Simulationslauf r. Diese kann komponentenweise oder euklidisch gemessen werden. Die Länge des Verschiebungsvektors wird über alle Simulationsläufe R miteinander verglichen.
- Die Funktion $\mathrm{SC}(p,t,r)$ ermittelt die Distanz eines Netzknotens p im Zeitschritt t von dessen geometrischen Schwerpunkt $g_c(p,t) = \frac{1}{R}\sum_{r=1}^{R} p(t,r)$, wobei R die Anzahl der Simulationsläufe ist.

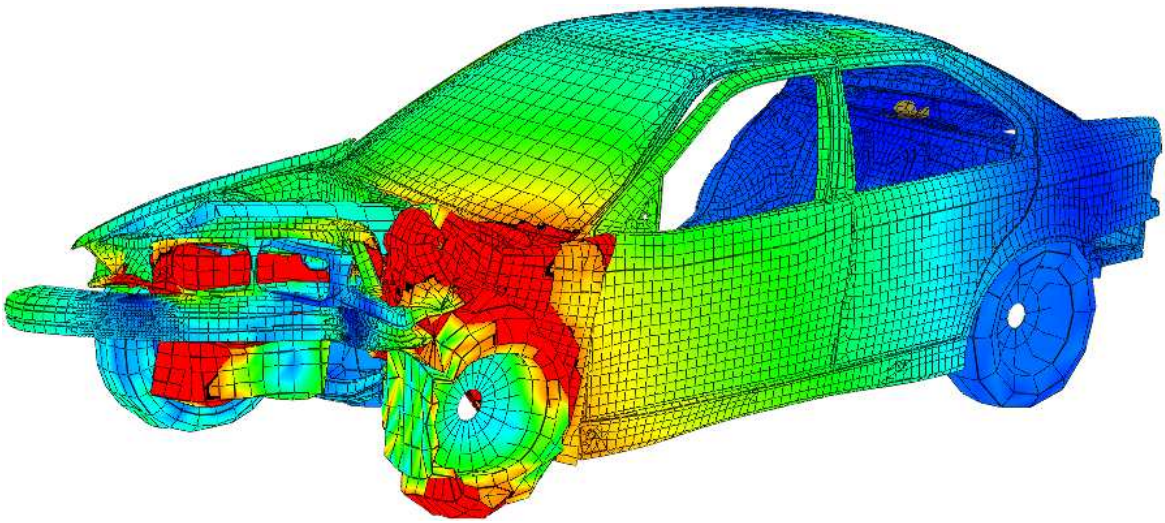


Abbildung 8.11: Die Farbe stellt die Länge der Differenzvektoren korrespondierender Netzknoten dar. Nach 80 Millisekunden ist die Abweichung im rot eingefärbten linken vorderen Bereich am größten und im hinteren Fahrzeugbereich geringer.

Der Nachteil bei Verwendung globaler Maße ist, dass sich die Verzweigungsstellen nicht hervorheben lassen. Wenn beispielsweise bei einem Front-Crash im vorderen Bereich eine Verzweigung auftritt, wird diese weitreichende Auswirkungen auf die Umgebung haben und im weiteren Verlauf auch Koordinatenabweichungen der Netzknoten im hinteren Fahrzeugbereich verursachen (Abbildung 8.11).

Um Verzweigungsstellen zu lokalisieren, muss ein Maß benutzt werden, das globale Einflüsse, die durch vorhergehende Verzweigungen verursacht wurden, weitestgehend ignoriert und eine Bewertung der Abweichung bezüglich der direkten Nachbarschaft vornimmt. Eine Funktion, die diesem Anspruch gerecht wird, wurde im Institut für Algorithmen und Wissenschaftliches Rechnen (SCAI) am Forschungszentrum Informationstechnik GmbH (GMD) im Rahmen des Autobench-Projekts entwickelt. Die Funktion DNM(p,t,r) berechnet die Verschiebung eines Netzknotens p in einem Zeitschritt t eines Simulationslaufs r bezüglich seiner Nachbarknoten hinsichtlich seiner Ausgangslage im initialen Zeitschritt. Das Funktionsergebnis stellt ein lokales Maß für die Verformung der benachbarten Elemente dar (Abbildung 8.12).

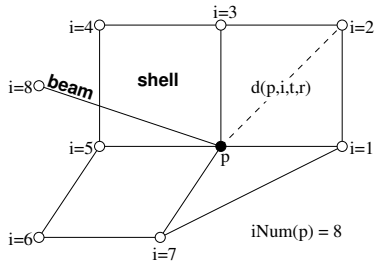


Abbildung 8.12: $d(p,i,t,r)$ berechnet sich aus dem Euklidischen Abstand (Gleichung 8.1) des Netzknotens p zu seinem Nachbarknoten i im Zeitschritt t des r-ten Simulationslaufs.

Die Deformation DNM(p,t,r) um einen Knoten p mit iNum(p) Nachbarknoten berechnet sich dabei aus der durchschnittlichen Summe der Abweichungen aller Knotendistanzen $d(p,i,t,r)$ gegenüber ihren ursprünglichen Abständen $d(p,i,0,r)$.

$$d(p,i,t,r) = \left\| \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} - \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} \right\|_{t,r} \quad , \quad \begin{array}{l} t : \text{Zeitschrittindex} \\ r : \text{Simulationslaufindex} \end{array} \tag{8.1}$$

$$\text{DNM}(p,t,r) = \frac{1}{N} \sum_{i=1}^{N} \left| d(p,i,t,r) - d(p,i,0,r) \right| \quad , \quad N := \text{iNum}(p) \tag{8.2}$$

Die Funktion DNM(p,t,r) wird für jeden Netzknoten p in jedem Zeitschritt t und für alle Simulationsläufe r berechnet. Der Ergebniswert wird, im Gegensatz zu den oben vorgestellten globalen Funktionen, lediglich durch seine lokale Umgebung beeinflusst. Schließlich wird

der Erwartungswert der Deformation

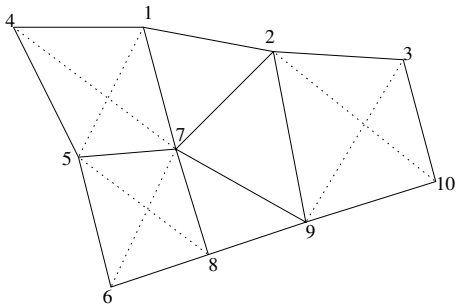
$$\mathrm{DNMAV}(p,t) = \frac{1}{R}\sum_{r=1}^{R}\mathrm{DNM}(p,t,r) \qquad , \qquad R : \#\ \text{Simulationsläufe} \tag{8.3}$$

aller R Simulationsläufe ermittelt und dessen Standardabweichung $\sigma(p,t)$ als lokales Maß für Instabilität um den Knoten p verwendet.

$$\sigma(p,t) = \sqrt{\frac{1}{R}\sum_{r=1}^{R}\left(\mathrm{DNM}(p,t,r) - \mathrm{DNMAV}(p,t)\right)^2} \tag{8.4}$$

8.4.2 Effiziente Berechnung der mittleren lokalen Deformation

Um eine effiziente Berechnung der mittleren lokalen Deformation zu ermöglichen, wird zunächst die in Abbildung 8.13 dargestellte Index-Paar-Liste für jeden Knoten aufgebaut. Jedes Paar besteht aus einem Verweis auf den Knoten, der mit dem zur Liste gehörenden Knoten benachbart ist, und einem Index für den schnellen Zugriff in zwei Arrays mit den Distanzwerten $d(p,i,t,r)$ des initialen und des aktuellen Zeitschrittes. Die Reihenfolge in den Index-Paar-Listen stellt sicher, dass der Abstand jeder Knotenpaarung nur einmal berechnet wird.



Knoten	(Nachbarknoten/Distanz)-Index-Paare
1	**(2,0) (4,1) (5,2) (7,3)**
2	(1,0) **(3,4) (7,5) (9,6) (10,7)**
3	(2,4) **(9,8) (10,9)**
4	(1,1) **(5,10)(7,11)**
5	(1,2) (4,10) **(6,12)(7,13) (8,14)**
6	(5,12)**(7,15)(8,16)**
7	(1,3) (2,5) (4,11) (5,13) (6,15) **(8,17)(9,18)**
8	(5,14) (6,16) (7,17) **(9,19)**
9	(2,6) (3,8) (7,18) (8,19) **(10,20)**
10	(2,7) (3,9) (9,20)

Abbildung 8.13: Dieses Beispiel zeigt die Struktur der Tabelle, die für jeden Knoten aus einer Liste (Reihe) besteht, welche einen Verweis auf den benachbarten Knoten und auf den Distanzwert in einem separaten Array enthält.

Aus der internen Datenstruktur werden zunächst die Index-Paare initialisiert und anschließend die Distanzwerte für den initialen Zeitschritt ermittelt und in einem der beiden Distanzwerte-Arrays abgespeichert, das im weiteren für alle Simulationsläufe unverändert bleibt. Beginnend mit dem ersten Simulationslauf werden zunächst die Differenzen zwischen den Knotenabständen des aktuellen und des initialen Zeitschrittes berechnet und in dem zweiten Array gespeichert. Dazu wird die dargestellte Tabelle reihenweise traversiert und Knotenabstände nur dann neu berechnet, wenn der Knotenindex des Nachbarknotens

größer ist als der des zur aktuellen Reihe gehörenden Knotens (fettgedruckte Paare in Abbildung 8.13); diese Vorgehensweise stellt sicher, dass Knotenabstände lediglich einmal berechnet werden. Die mittlere Deformation $\mathrm{DNM}(p,t,r)$ ergibt sich aus den aufsummierten, in einer Tabellenreihe referenzierten Werten im zweiten Array geteilt durch die Anzahl der Einträge in dieser Reihe. In einem weiteren Array werden die $\mathrm{DNM}(p,t,r)$ aufsummiert, um nach Auswertung aller Simulationsläufe den Erwartungswert $\mathrm{DNMAV}(p,t)$ berechnen zu können. Aufgrund der großen Datenmengen können die benötigten Daten nicht komplett im Hauptspeicher gehalten werden; deshalb werden die $\mathrm{DNM}(p,t,r)$ für jeden Simulationslauf nach Abarbeitung aller Zeitschritte in eine binäre Datei ausgelagert, die später zur Ermittlung der Standardabweichung $\sigma(p,t)$ nochmal eingelesen wird.

Der Algorithmus entkoppelt den Speicherbedarf von der Anzahl der Simulationsläufe, indem die Zwischenergebnisse für einen Simulationslauf auf die Festplatte ausgelagert werden. Zusätzlich zu der internen Datenstruktur fallen somit lediglich 24 Byte pro Knotenpaar, 32 Byte pro Knoten und weitere 4 Byte pro Knoten pro Zeitschritt an. Durch geringfügige Modifikation kann der Speicherbedarf weiter verringert werden, indem die Zwischenergebnisse zusätzlich für jeden Zeitschritt ausgelagert werden.

8.4.3 Resultate

In der Praxis hat sich gezeigt, dass sich das Deformationsfunktional zur Lokalisierung von Instabilitäten eignet, da sich in der direkten Umgebung von Knoten mit hohen Standardabweichungen der mittleren lokalen Deformation Verzweigungen finden lassen. Da das Deformationsfunktional allerdings keine Auskunft über die absolute Streuung in den Simulationsergebnissen geben kann und nur Stauchung beziehungsweise Streckung in der Elementebene, jedoch keine Biegungen berücksichtigt, hat sich die zusätzliche Visualisierung der maximalen Streuung mit dem Scatter-Funktional als hilfreich erwiesen.

8.5 Kooperatives Arbeiten

Die Anforderung der Automobilhersteller, möglichst schnell auf die Nachfrage des Marktes reagieren zu können, führt in den Entwicklungsabteilungen zu Arbeitslastspitzen, die von der fest angestellten Belegschaft nicht abgedeckt werden können. Daher entwickelt sich in den letzten Jahren der Trend, mehr und mehr Entwicklungsleistungen bei externen Ingenieurbüros in Auftrag zu geben. Darüber hinaus entsteht auch durch Firmenfusionen und den daraufhin angestrebten Wissensaustausch in korrespondierenden Fachabteilungen ein erhöhter Kommunikationsbedarf zwischen kooperierenden aber räumlich getrennten Entwicklern.

Im Folgenden werden zwei Verfahren erläutert, die das kooperative Arbeiten mit dem Prototypen *crashViewer* ermöglichen. Sie verfolgen einen unterschiedlichen Ansatz: Die Event-basierte Methode tauscht lediglich Ereignisse und Zustandsinformationen zwischen den synchronisierten Applikationsinstanzen aus, führt jedoch die eigentliche Visualisierung

der Daten jeweils lokal aus. Im Gegensatz dazu werden bei dem Bild-basierten Verfahren auf einem Rechner anhand der dort vorliegenden Daten Bilder generiert und an angeschlossene Clients verschickt. Das gemeinsame Ziel ist, den teilnehmenden Anwendern jederzeit die gleiche Sicht auf die Daten zu bieten und eine wechselseitige Interaktion mit dem dargestellten Modell zu ermöglichen, um eine parallel stattfindende Kommunikation, zum Beispiel via Telefon, zu unterstützen.

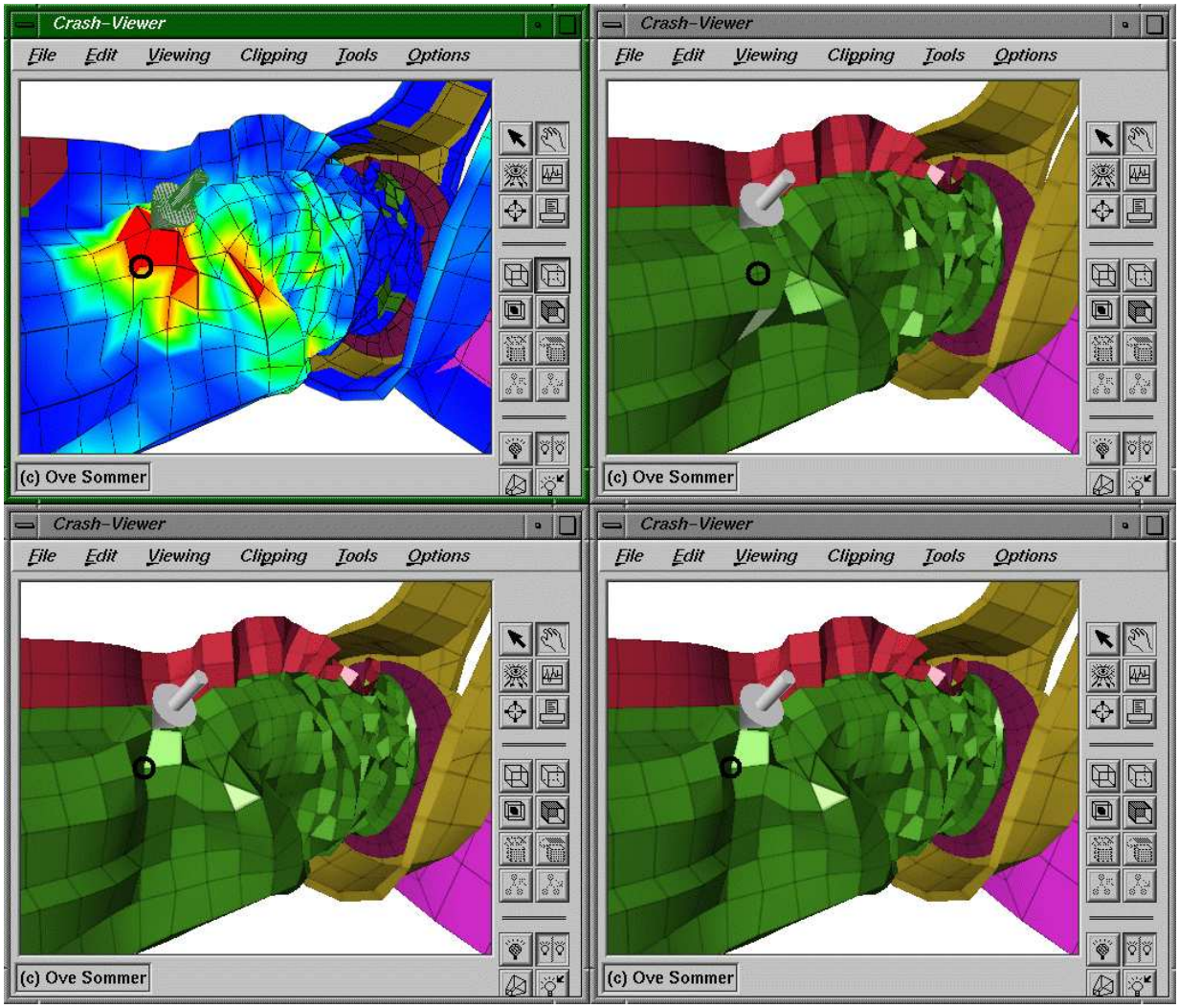


Abbildung 8.14: Der *crashViewer* wurde hier viermal gestartet und zeigt abweichende Simulationsergebnisse des gleichen Eingabedatensatzes. Die Kamera wird, wie in Kapitel 8.5.1 beschrieben, über eine CORBA-Verbindung synchronisiert. Der 3D-Pfeil markiert den gleichen Raumpunkt, während der schwarze Kreis den gleichen Netzknoten kennzeichnet. Die visualisierte Standardabweichung in dem *crashViewer* oben links hebt die Regionen mit stark abweichendem Deformationsverhalten hervor.

8.5.1 Event-basiert

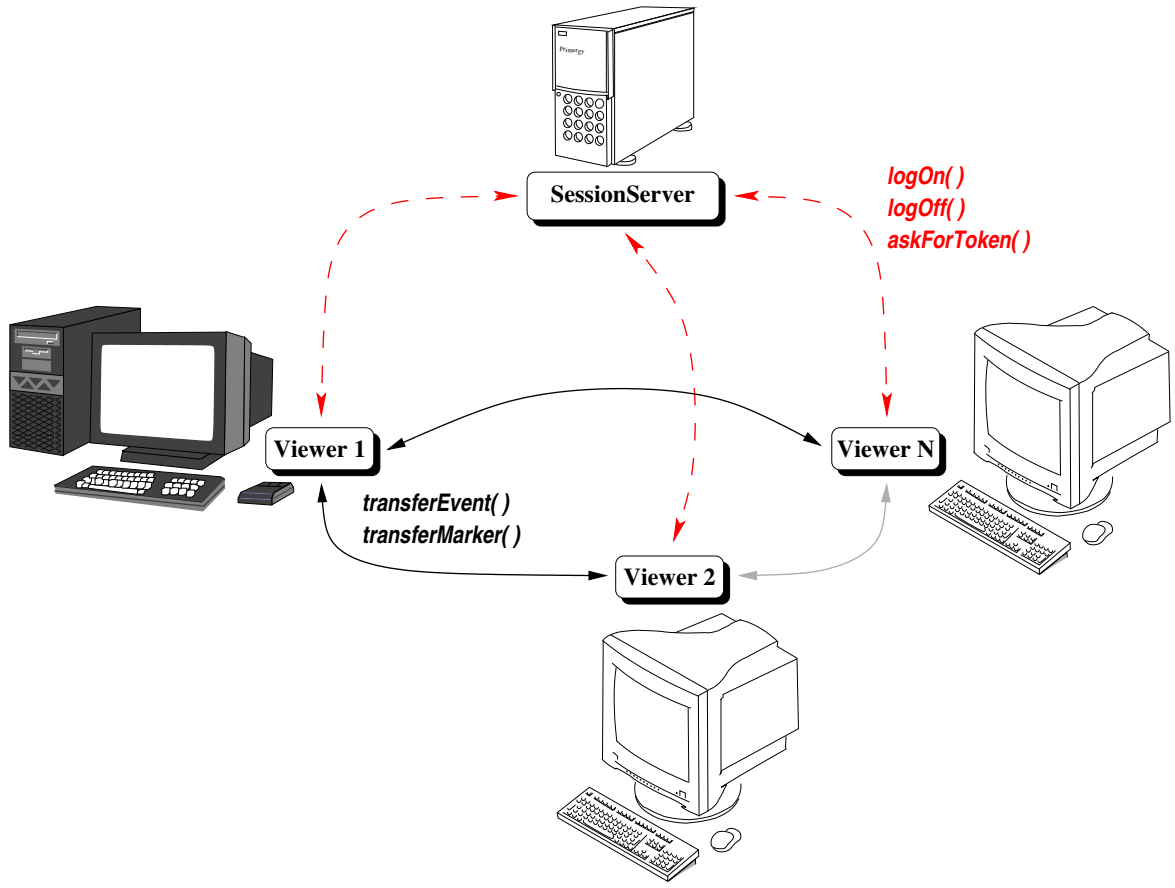


Abbildung 8.15: Der *Session-Server* verwaltet die kooperative Sitzung mehrerer *crashViewer*-Instanzen und vergibt das *Master-Token*. Die Instanz, der das *Master-Token* zugeteilt ist (hier *Viewer 1*), sendet ihre Events direkt an die anderen teilnehmenden *crashViewer*-Instanzen.

Die Event-basierte Variante des kooperativen Arbeitens im Prototypen *crashViewer* nutzt CORBA[1] für die Kommunikation zwischen den an der kooperativen Sitzung beteiligten Rechnern. Abbildung 8.15 stellt eine kooperative Sitzung mit mehreren Teilnehmern schematisch dar. Der die Sitzung initialisierende Anwender startet zunächst einen *Session-Server*; dieser verwaltet die kooperative Sitzung und trägt Sorge dafür, dass An- und Abmeldungen von *crashViewer*-Instanzen beim *Session-Server* an alle teilnehmenden *crashViewer*-Instanzen propagiert werden. Außerdem teilt der *Session-Server* das *Master-Token* zu, das stets nur eine *crashViewer*-Instanzen der Sitzung als diejenige auszeichnet, die ihre Events an die anderen Instanzen weitergibt und damit die Sitzung steuert.

[1] CORBA — **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

Nachdem der *Session-Server* gestartet wurde, wird eine CORBA-Referenz in Form einer langen Zeichenkette in einer Datei abgelegt. Diese Zeichenkette, in der Informationen über die IP-Adresse, die Portnummer etc. kodiert enthalten sind, muss den Anwendern, die an der Sitzung teilnehmen möchten, zugänglich gemacht werden. Über die CORBA-Referenz des *Session-Server* können sich nun *crashViewer*-Instanzen an den *Session-Server* wenden, um ihre eigene CORBA-Referenz unter den bereits teilnehmenden *crashViewer*-Instanzen zu propagieren und selbst auch eine Liste von CORBA-Referenzen zu erhalten. Die sich zuerst anmeldende *crashViewer*-Instanz erhält automatisch das *Master-Token*.

Eine neu hinzukommende *crashViewer*-Instanz wird vom *Session-Server* bei den bereits teilnehmenden *crashViewer*-Instanzen angemeldet und erhält abschließend die Liste der Teilnehmer als CORBA-Referenzen. Das Abmelden wird ebenfalls über den *Session-Server* abgewickelt.

Sofern der *crashViewer* im Besitz des *Master-Token* ist, propagiert er aufgetretene Events (zum Beispiel Kamerabewegungen oder Zeitschrittanimation) direkt an die ihm bekannten *crashViewer*-Instanzen; der *Session-Server* ist hier nicht involviert. Jeder Teilnehmer kann das *Master-Token* beim *Session-Server* beantragen, die Anfrage wird daraufhin an den *Master-crashViewer* weitergeleitet und die CORBA-Referenz der beantragenden *crashViewer*-Instanz in eine FIFO-Warteschlange aufgenommen. Das *Master-Token* wird, sobald keine weiteren Interaktionen durchgeführt werden, vom *crashViewer* an den *Session-Server* zurückgegeben. Jetzt reicht der *Session-Server* das *Master-Token* an die erste CORBA-Referenz in der Warteschlange weiter, wodurch die Sitzung nun von der dazugehörigen *crashViewer*-Instanz gesteuert wird.

Der implementierte Mechanismus lässt sich mit einer moderierten Diskussion vergleichen: dabei steuert der *Session-Server* als Moderator die Diskussion und teilt mit dem *Master-Token* einem Teilnehmer eine Rede-Berechtigung zu, während alle anderen in dieser Zeit nur zuhören dürfen.

Es wurde vorausgesetzt, dass dem Anwender ein Telefon oder ähnliche Konferenzwerkzeuge zur Verfügung stehen, um mit den anderen Teilnehmern verbal zu kommunizieren. Zur Unterstützung wurde eine Marker-Funktionalität implementiert, die es dem Anwender erlaubt, Strukturen durch 3D-Pfeile zu markieren; diese werden ebenfalls, wie die Kamerabewegungen, an alle teilnehmenden *crashViewer*-Instanzen übertragen und dienen der besseren Verständigung zwischen den Teilnehmern.

Die zu visualisierenden Daten können entweder von jeder *crashViewer*-Instanz selbst eingelesen werden, sofern sie jeweils lokal vorliegen, oder auch in Form des erstellten Szenengraphen via CORBA vom *Master-* zum *Slave-crashViewer* transferiert werden. Das hat den Nachteil, dass auf interne Daten, die nicht im Szenengraphen abgespeichert sind, nicht zurückgegriffen werden kann. Eine Zuordnung geometrischer Daten auf Modellbestandteile wäre dann nur über die Instanz möglich, die den Szenengraphen generiert und verteilt hat. Als Vorteil kann allerdings gesehen werden, dass nur die für die Visualisierung notwendigen Daten übertragen werden und sich damit oftmals der Datenumfang auf einen Bruchteil der Originaldaten beschränkt.

Dieses Verfahren des Event-basierten kooperativen Arbeitens lässt sich ebenfalls sehr gut zum direkten visuellen Vergleich von Varianten einsetzen, indem an einem Rechner mehrere *crashViewer* mit variierenden Fahrzeugmodellen gestartet werden. Marker, die an gleichen Raumpunkten gesetzt werden, geben zusätzlich zu den identischen Kamerapositionen Anhaltspunkte, um Abweichungen im Verlauf der Zeitschrittanimation sowie Netzmodifikationen bei der Beseitigung initialer Penetrationen (Kapitel 7.1.2) zu verdeutlichen. Insbesondere für die detaillierte Analyse von Instabilitätsuntersuchungen bietet dieser Ansatz effektive Unterstützung (Abbildung 8.14, Seite 136).

Während die Event-basierte Variante einerseits den Vorteil eines schnellen Abgleichs zwischen den kooperierenden *crashViewer*-Instanzen durch geringen Datenaustausch hat und die Möglichkeit der Interaktion für jeden Teilnehmer bietet, setzt sie andererseits voraus, dass jeder Teilnehmer die notwendige Hard- und Software hat, um *crashViewer* starten zu können, und dass die Daten bei jedem Teilnehmer vorliegen oder der Szenengraph dorthin übertragen wurde. Da diese Voraussetzungen nicht immer erfüllt werden können, wurde ein weiteres Verfahren zum kooperativen Arbeiten in *crashViewer* integriert, das im Anschluss an die Bildsynthese den Bildspeicherinhalt überträgt.

8.5.2 Bild-basiert

Im Alltag des Berechnungsingenieurs kommt es regelmäßig vor, dass Simulationsergebnisse oder Konstruktionsdetails mit Kollegen, die gegebenenfalls auch in anderen Abteilungen arbeiten, diskutiert werden müssen. Wenn zum Beispiel ein Berechnungsingenieur einen Konstrukteur fragen möchte, ob kleine Änderungen an einem Fahrzeugbauteil möglich sind, ist dafür in der Regel ein Treffen der beiden notwendig, damit der Berechnungsingenieur seine Vorstellungen am Bild des Finite-Element-Modells erläutern kann.

Um derartige Rückfragen zu beschleunigen und gleichzeitig die Visualisierung auf Low-End-Hardware bis hin zum PDA[2] zu ermöglichen, wurde zusätzlich zum Event-basierten Verfahren ein Bild-basierter Client-Server-Ansatz zum kooperativen Arbeiten in *crashViewer* integriert. Dabei wird lediglich eine *crashViewer*-Instanz als Visualisierungsserver gestartet. Ein erzeugtes Bild wird codiert, über eine Socket-Verbindung zum Client übertragen, dort decodiert und als Bild wieder dargestellt.

Sowohl für den Verbindungsaufbau als auch für das Codieren und anschließende Versenden der Bilder wird jeweils ein zusätzlicher Thread gestartet. Abbildung 8.16 stellt schematisch das Zusammenwirken zwischen dem *Render-* und dem *Send-Thread* dar. Die Verwendung von Doublebuffering reduziert die Zeit, die beide Threads in einem kritischen Abschnitt — in dem auf gemeinsam genutzte Daten zugegriffen wird — zubringen, auf ein Minimum. Die Threads synchronisieren sich durch die Verwendung von Semaphoren und schließen somit aus, dass der *Send-Thread* nach dem Transfer des letzten Bildes einen Pufferwechsel macht, während der *Render-Thread* gleichzeitig noch Bilddaten kopiert.

[2]Der **P**ersonal **D**igital **A**ssistant, aufgrund seiner geringen Größe auch als *Handheld* bekannt, ist ein elektronischer Organizer, der entweder über Touchpad – per Stift – oder Tastatur gesteuert wird.

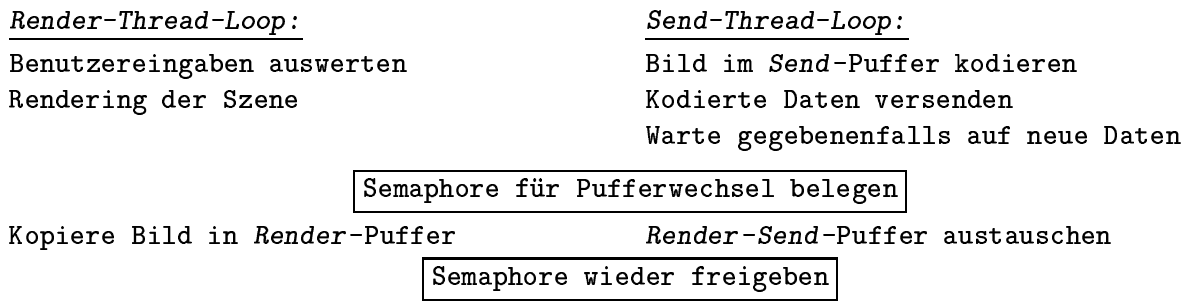


Abbildung 8.16: Der gleichzeitige modifizierende Zugriff auf den Puffer, der die Bilddaten speichert, wird durch die Verwendung von Semaphoren ausgeschlossen. Um die Zeit, die die Threads in kritischen Abschnitten verbringen, zu minimieren, wurden zwei Bildspeicher verwendet: einer, in den die aktuellen Bilddaten vom *Render-Thread* kopiert werden, und ein anderer, auf dem die Codierung stattfindet und der anschließend dem Transfer als Puffer dient.

Als Codierungsverfahren wurden die folgenden drei getestet:

- *RAW*: Die Bilddaten werden unkomprimiert transferiert. Der Transfer der Originaldaten kann lediglich in Netzwerken mit hoher Bandbreite ohne große Verzögerungen gewährleistet werden.
- *RLE*: Das verlustfreie *Run-length Encoding* ist zwar nicht sehr rechenintensiv, erreicht in der Regel aber auch nur geringe Kompressionsraten und ist daher für Netzwerke mit mittleren Bandbreiten geeignet.
- *ZLIB*: Höhere Kompressionsraten wurden mit dem verlustfreien Kompressionsverfahren der ZLIB-Bibliothek[55] erzielt. Damit eignet sich dieses Verfahren für Verbindungen mit geringer Bandbreite. Außerdem wird die Dekompression, im Gegensatz zum *RLE*-Verfahren, vom schnelleren Native-Code durchgeführt, da ZLIB-Kompression inzwischen ein Standard-Feature von Java ist.

Auf der Client-Seite werden die codierten Bilddaten von einem Java-Applet decodiert und schließlich als Bild dargestellt (Abbildung 8.17).

Dieses Szenario kann noch dahingehend erweitert werden, dass Benutzereingaben wieder von der Client-Anwendung zum *crashViewer* als Visualisierungsserver übermittelt und dort interpretiert werden. Dadurch wäre zusammen mit einem Offscreen-Rendering die Realisierung eines Visualisierungsservers möglich, der ausschließlich von Client-Applikationen gesteuert wird, die geringe Hardwareanforderungen haben und dennoch über den Bildbasierten Mechanismus auf Kapazitäten größerer Rechner zugreifen können. Ein solches Client-Server-System zur interaktiven Visualisierung von medizinischen Daten wird in [15] beschrieben.

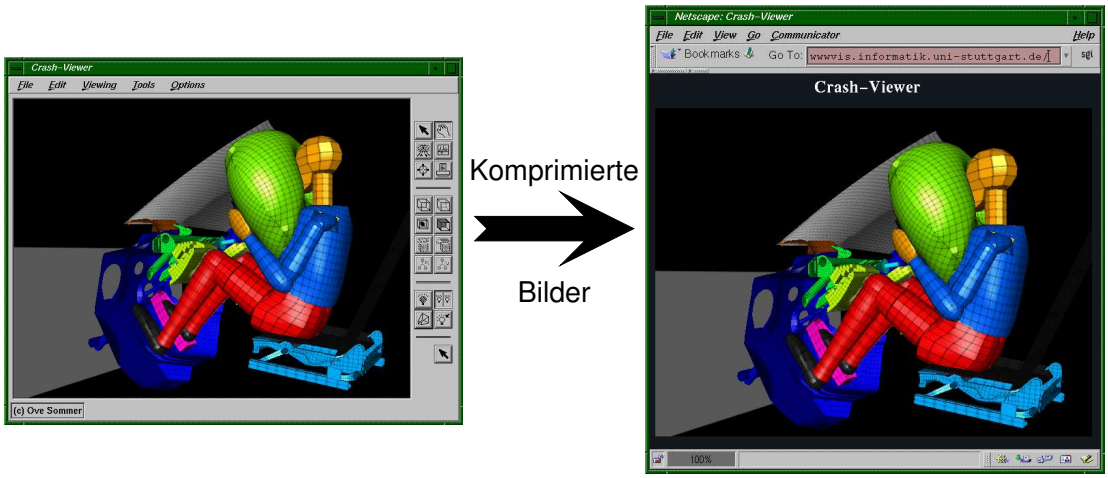


Abbildung 8.17: Die durch *crashViewer* visualisierten Daten werden als komprimierte 2D-Bilder an ein Java-Applet übertragen, das hier in einem Netscape-Browser gestartet wurde. Auf der Client-Seite müssen die Bilder lediglich entpackt und angezeigt werden.

8.6 Batch-Programm

Zusätzlich zu der Visualisierungsapplikation *crashViewer* ist das Batch-Programm *buildGraph* entwickelt worden. Es nutzt die gleichen Module wie *crashViewer*, läuft aber im Batch-Betrieb und kann somit einerseits für zeitintensive Vorberechnungen, auf die der Benutzer während einer *crashViewer*-Sitzung nicht warten möchte, genutzt werden. Andererseits können mit *buildGraph* Verarbeitungsschritte, die keiner Benutzerinteraktion bedürfen, automatisiert werden, indem sie in den Berechnungsablauf im Anschluss an die Simulation eingebunden werden. Zu den Einsatzgebieten von *buildGraph* gehören die

- Szenengraphgenerierung für eine anschließende Bild- oder Filmerstellung (siehe unten),
- Simplifizierung von Finite-Element-Modellen und Speicherung des reduzierten Modells als Szenengraphen im Cosmo3D oder Open Inventor Dateiformat,
- Kraftflussberechnung durch die Finite-Element-Struktur entlang vordefinierter Bahnen (Abschnitt 8.3, Seite 127),
- Ermittlung von Instabilitäten beim Vergleich mehrerer Simulationsläufe gleicher Eingabedaten (Abschnitt 8.4, Seite 131).

Szenengraphgenerierung für die Bild-/Filmerstellung

Das Batch-Programm *buildGraph* wird beim Aufruf über Kommandozeilenparameter gesteuert und kann die eingelesenen Daten sowohl als Cosmo3D- als auch als Open Inventor Szenengraphen abspeichern, welcher dann zur Weiterverarbeitung durch andere Anwendungen zur Verfügung steht. Dies wurde im Zusammenhang mit der Entwicklung einer HTML-

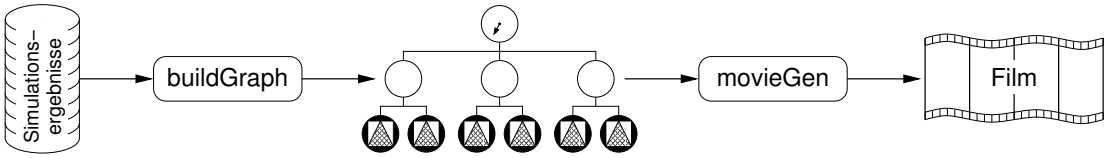


Abbildung 8.18: Datenfluss bei der Batch-basierten Generierung digitaler Filme aus animierten Simulationsergebnissen mit Hilfe von *buildGraph* und *movieGen*.

basierten Benutzerschnittstelle zu einer Datenbank, in der sämtliche Finite-Element-Netz-Varianten der Fahrzeugbauteile verwaltet werden, für die Erstellung von Bauteilabbildungen genutzt. Der abgespeicherte Open Inventor Szenengraph wird von einem weiteren Batch-Programm wieder eingelesen, das aus vorher spezifizierten Blickrichtungen Bilder der Bauteilnetze *offscreen* – ohne dass dafür ein Fenster geöffnet werden muss – erstellt und abspeichert.

In einem weiteren Projekt, das automatisiert Filme generiert, die dem Berechnungsingenieur beim Postprocessing einen ersten Überblick über das Crash-Verhalten des Fahrzeugmodells geben, wurde ebenfalls *buildGraph* zur Szenengrapherstellung eingesetzt. Abbildung 8.18 skizziert den Datenfluss von den Simulationsergebnissen bis zum digitalen Film. Der generierte Open Inventor Szenengraph enthält alle berechneten Zeitschritte der sich verformenden Fahrzeugstruktur unter einem *Switch*-Knoten. *movieGen*, ein weiteres Batch-Programm liest den Szenengraphen wieder ein, erstellt mit Hilfe des SoOffscreenRenderer Einzelbilder von der animierten Szene. Dabei kann sich die Kamera auf einer vorgegebenen Flugbahn um das Modell herum bewegen, oder sich relativ zu einem spezifizierten Knoten mit der sich verformenden Struktur mitbewegen. Die Einzelbilder werden schließlich zu einem digitalen Video zusammengesetzt. Diese Videos können daraufhin zusammen mit den berechneten Daten in einer Datenbank abgelegt werden und später als eine Art visuelle Kurzfassung zur Identifikation der Simulationsergebnisse vom Anwender herangezogen werden, ohne dass dafür ein Postprocessor gestartet und die Ergebnisdaten geladen werden müssen.

Kapitel 9

Ergebnisse

Im Rahmen der vorliegenden Arbeit wurden Methoden entwickelt, die zuvor analysierte Probleme im virtuellen Fahrzeugentwicklungsprozess lösen und darüber hinaus Wege aufzeigen, wie moderne Visualisierungs- und Interaktionstechniken genutzt werden können, um den Arbeitsablauf im Pre- und Postprocessing von Strukturmechanikdaten zu beschleunigen. Die Resultate im Bereich des kooperativen Arbeitens bieten Lösungsansätze für eine Zusammenarbeit über räumliche Grenzen hinweg. Die Integration von Pre- und Postprocessing in dem entstandenen Prototypen *crashViewer* hat den Weg für eine enge Kopplung an die Berechnung geebnet. Durch den modularen Aufbau der Prototypen-Software konnte schließlich eine Basis für weiterführende Arbeiten geschaffen werden. Der praktische Einsatz von *crashViewer* im Preprocessing zeigt, dass die Analyse herkömmlicher Arbeitsschritte und die Entwicklung und Umsetzung moderner Interaktions- und Visualisierungskonzepte in enger Zusammenarbeit mit den Anwendern zu einer signifikanten Verkürzung der Entwicklungszeiten führen kann.

9.1 Neue Methoden im virtuellen Fahrzeugentwicklungsprozess

Diese Arbeit wurde in enger Kooperation mit der Berechnungsabteilung für nicht-lineare Strukturdynamik der *BMW Group* erstellt. Die Analyse des Arbeitsablaufs bei der Vor- und Nachverarbeitung der Fahrzeugmodelle hat durch neue Methoden in der virtuellen Fahrzeugentwicklung sowie durch stetig anwachsende Modellgrößen bedingte Probleme aufgezeigt, die von den bis dahin zur Verfügung stehenden Software-Werkzeugen nicht oder nur unzureichend adressiert wurden. Daher beschäftigte sich ein Teil der Arbeit mit der Entwicklung neuer Techniken sowie der Adaption bereits bestehender Methoden speziell an die Bedürfnisse und Rahmenbedingungen der Strukturmechaniksimulation. Um die gewählten Methoden im Einsatz bewerten und optimieren zu können, wurden prototypische Anwendungen implementiert.

Konzepte zur Minimierung benötigter Ressourcen

Die stetig steigende Komplexität der Simulationsmodelle in der Strukturdynamik führt bei der Bearbeitung von Gesamtfahrzeugmodellen zu immer längeren Ladezeiten, die den Arbeitsablauf des Berechnungsingenieurs verzögern. Die benötigte Zeit für das Einlesen zeitabhängiger Daten zur Auswertung von Simulationsergebnissen konnte zum einen durch Parallelisierung, zum anderen durch speziell angepasste interne Datenstrukturen minimiert werden. Der Speicherbedarf für den Szenengraphen konnte ebenfalls durch ein auf die Eigenschaften der zeitabhängigen aber topologisch invarianten Finite-Element-Netze ausgerichtetes Szenengraph-Design auf ein Minimum reduziert werden. Die hierarchische Unterteilung der Bauteilnetze mit Hilfe von Hüllvolumen ermöglicht nicht nur die interaktive Distanzvisualisierung zur Detektion von Netzperforationen und -penetrationen; darüber hinaus bildet die über die Szenengraphstruktur hinaus gehende feinere Zerlegung des Finite-Element-Modells eine Grundlage für alle zeitintensiven Distanz-basierten Algorithmen, zum Beispiel der Flanschdetektion und der interaktiven Berechnung von Kraftflussröhren.

Visualisierungstechniken

Diese Arbeit hat neue Visualisierungstechniken im CAE-Umfeld zum Einsatz gebracht. Die Verwendung der Wireframe-Textur wurde weiterentwickelt und im Prototypen für Quadrilateralnetze anwendbar gemacht. Dadurch konnte die Bildwiederholrate für die Darstellung der Finite-Element-Diskretisierung auf der schattierten Geometrie im Vergleich zum konventionellen Zwei-Schritt-Verfahren mehr als verdoppelt werden. Des Weiteren wurde ein Verfahren zur Quadrilateralstreifengenerierung entwickelt und auf die Bauteilnetze angewendet, um die Geometrierepräsentation für die Weiterverarbeitung in der Rendering-Pipeline zu optimieren.

Nachdem die Methodenentwicklung der Strukturmechaniksimulation netzunabhängige Schweißpunkte hervorgebracht hatte, um unabhängig voneinander vernetzte Bauteile ohne Neuvernetzung für die Simulation vorbereiten zu können, stellten sich die in kommerziellen Werkzeugen zur Verfügung stehenden Visualisierungsverfahren für die Schweißpunktdarstellung als unzureichend heraus. Die Aussagekraft der Bilder im Pre- und Postprocessing konnte durch Verwendung zusätzlicher Geometrie deutlich gesteigert werden. Anhand der Darstellungseigenschaften eines Schweißpunktes in Form und Farbe ist der Berechnungsingenieur nun sofort in der Lage, eine fehlerhafte Schweißpunktverbindung zu klassifizieren. Zusammen mit den Interaktionsmöglichkeiten des entstandenen Prototypen trägt die Schweißpunktvisualisierung entscheidend zur Beschleunigung der Modellvalidierung bei. Inzwischen haben diese Ergebnisse auch Einfluss auf die Weiterentwicklung in kommerziellen Produkten genommen. So wurde beispielsweise die graphische Repräsentation der Schweißpunkte aus dem Prototypen *crashViewer* in den Postprocessor PAM-VIEW der Firma ESI übernommen.

Die Textur-basierte Visualisierung von knotengebundenen skalaren Größen konnte sowohl für das Pre- als auch das Postprocessing über die reine Farbdarstellung hinausgehend

zur Datenexploration genutzt werden. Es wurde gezeigt, wie die Graphik-Hardware dazu dienen kann, Geometrie Werte-bezogen auszublenden, um ausschließlich Strukturen darzustellen, deren visualisierter Parameter in einem interaktiv spezifizierbaren Wertebereich liegt. Da die Änderung des Wertebereichs allein durch Modifikation der Texturtabelle umgesetzt wird, eignet sich diese Visualisierungstechnik besonders, um sich bei der Datenauswertung schnell einen Überblick über die Simulationsergebnisse zu verschaffen. Ebenso lassen sich in der Vorverarbeitung sehr schnell potenzielle Flanschbereiche sowie Regionen initialer Penetration und Perforation entdecken.

Interaktionstechniken

Ein Bestandteil dieser Arbeit war die Entwicklung neuer Interaktionskonzepte, die den Arbeitsablauf im Pre- und Postprocessing von Strukturdynamikdaten beschleunigen sollten. Dazu wurde ein Ansatz gewählt, der es dem Benutzer mit möglichst wenigen Eingaben ermöglicht, die gewünschten Funktionen auszuführen. Die Kombination der Modusabhängigen Tastaturbelegung in Kombination mit der Maus hat sich bei Anwendern, die regelmäßig mit dem Prototypen arbeiteten, bewährt. Nachdem allerdings immer neue Funktionen mit neuen Tastenbelegungen hinzugekommen waren, hat sich gezeigt, dass das Tastatur-basierte Bedienkonzept für Neueinsteiger ein Hindernis darstellt und zusätzlich eine Graphische Benutzerschnittstelle benötigt wird, über die dann jede Funktionalität angesteuert werden kann. Erfahrenere Anwender sprachen sich für die Beibehaltung der schnellen Funktionsansteuerung über die Tastatur aus. Für die Navigation mit 2D- und 3D-Maus im virtuellen Fahrzeugmodell wurden zahlreiche Hilfsfunktionen implementiert, die den Arbeitsablauf vereinfachen und beschleunigen und somit die Akzeptanz beim Anwender steigerten. Dieser Aspekt spielte bei der Implementierung des Prototypen *crashViewer* stets eine Rolle, da die entwickelten Konzepte nur anhand der prototypischen Anwendung im alltäglichen Einsatz beim Berechnungsingenieur bewertet und optimiert werden konnten.

Der Prototyp wurde bereits in seiner frühen Entwicklungsphase trotz des im Vergleich zu kommerziellen Vorverarbeitungswerkzeugen recht beschränkten Funktionsumfangs von verschiedenen Anwendern produktiv eingesetzt, weil damit interaktiv und effizient Schweißpunktdaten generiert und validiert werden konnten. Gegenüber den bis dahin verfügbaren Werkzeugen konnte eine große Beschleunigung in der Modellaufbereitung für die frühe Produktentwicklungsphase erlangt werden. Die automatische Navigation zu fehlerhaften Verbindungselementen und Netzfehlern trug dazu bei, dass allein die Validierung für ein Gesamtfahrzeugmodell um 40 Arbeitsstunden auf ein Drittel reduziert werden konnte.

Für die Auswertung der Kraftflussverteilung in der Analyse von Crash-Simulationsergebnissen war es notwendig, den Verlauf der zu visualisierenden Kraftflussröhren interaktiv festlegen zu können. Durch Selektion der zu berücksichtigenden Bauteile, Festlegung der Kraftflussröhrenstützpunkte und Beschränkung auf eine Untermenge von Finite-Elementen mit Hilfe eines Selektionsschlauches wurde ein Interaktionsmechanismus entwickelt, der für die Definition einer auszuwertenden Kraftflussröhre nur

wenige Sekunden in Anspruch nimmt. Durch die Zuordnung der Stützpunkte zu Netzknoten wurde zudem eine dynamische Anpassung des Kraftflussröhrenverlaufs an die sich verformende Fahrzeugstruktur erreicht. Die persistente Speicherung erlaubt die Wiederverwendung einmal definierter Kraftflussröhrenverläufe in Varianten-Simulationen sowie die Batch-Auswertung zur Vorberechnung der Kraftflussröhren im direkten Anschluss an die Simulation.

Die Exploration der Strukturverformung, die zu Intrusionen in die Fahrgastzelle führen, spielt in der Analyse eine große Rolle. Ein Hilfsmittel stellt dabei die Visualisierung der Eindringtiefe mit Hilfe einer an die Fahrzeugstruktur gebundenen Referenzebene dar. Diese Ebene kann im Prototypen interaktiv definiert werden. Ein weiteres wichtiges Werkzeug ist die frei bewegliche Schnittebene. Sie hilft bei den immer komplexer werdenden Modellen auch in stark verformten unübersichtlichen Regionen einen Einblick zu bekommen (Abbildung 9.1). Mehrere solcher Schnittebenen lassen sich nicht nur auf das Gesamtfahrzeugmodell, sondern auch auf Teilstrukturen anwenden, wodurch dieses Analysewerkzeug noch flexibler zum Einsatz gebracht werden kann. In diesem Zusammenhang wurde auch

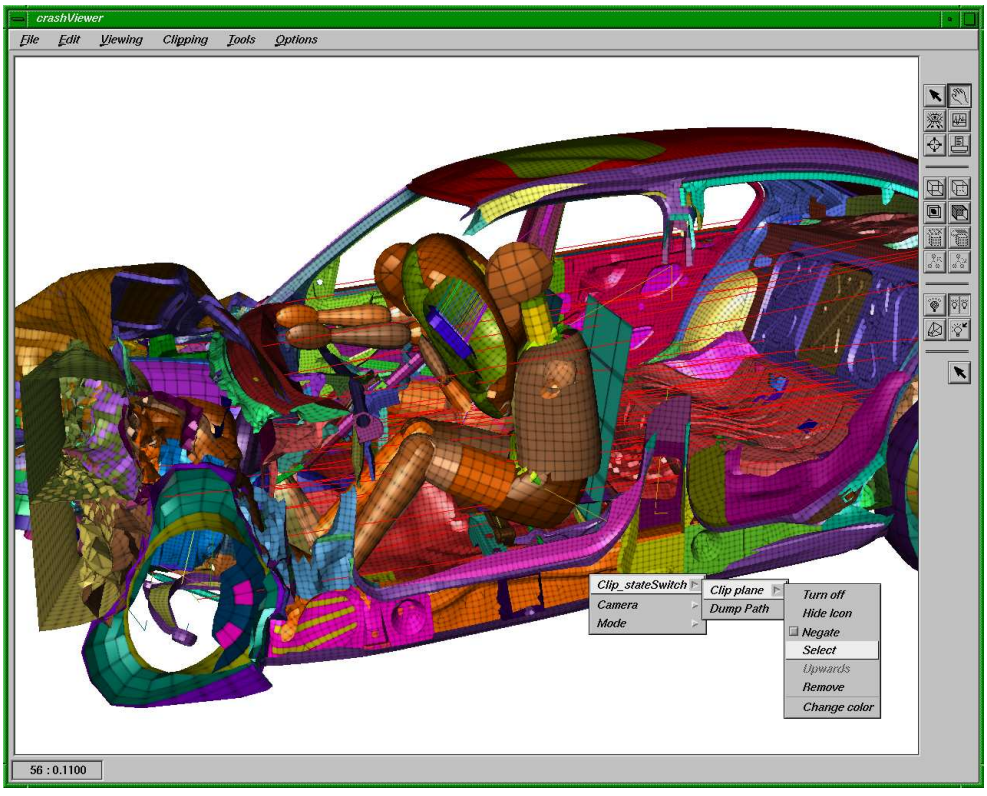


Abbildung 9.1: Die frei beweglichen Schnittebenen können unter anderem über das Kontext-sensitive Popup-Menü kontrolliert werden. Mit ihrer Hilfe kann der Berechnungsingenieur innere Finite-Element-Strukturen betrachten. Ohne dieses Werkzeug wäre eine Analyse gerade in Bereichen starker Verformung kaum möglich.

die Verwendung Kontext-sensitiver Popup-Menüs untersucht. Die Beschränkung des Funktionsumfangs anhand des selektierten Objekts macht das Menü übersichtlich und seine Einträge schnell erreichbar. Durch diesen Mechanismus werden Objekt- und Funktionsauswahl in einer Interaktion miteinander verschmolzen.

Die flüssige Navigation durch große Modelle wurde mit Hilfe einer simplifizierten Modelldarstellung erreicht. Für die Netzreduktion wurde ein auf die Anforderungen ausgerichteter Dezimierungsalgorithmus implementiert. Eine vergröberte Darstellung sorgt während der Navigation für deutlich höhere Bildwiederholraten, und nach Abschluss der Kamerabewegung werden dem Ingenieur wieder alle Details des Fahrzeugmodells visualisiert. Dies geschieht auf Basis gemeinsam genutzter Koordinaten für beide Detailstufen, da während der Simplifizierung ausschließlich topologische Operatoren zum Einsatz kommen.

Integration von Pre- und Postprocessing

Ein großes Ziel in der virtuellen Fahrzeugentwicklung ist die Verschmelzung von Konstruktion und Berechnung, um die derzeit notwendigen Synchronisationspunkte zwischen beiden Arbeitsbereichen zu beseitigen und damit den Entwicklungsprozess weiter zu beschleunigen. Anhand des entstandenen Prototypen wurde gezeigt, dass sich Pre- und Postprocessing-Funktionalität sehr wohl in einem Programm vereinigen lassen. Ein Vorteil für den Anwender, wenn eine Applikation mehrere Arbeitsschritte abdeckt, ist die Reduktion der Anwendungen, die bedient werden müssen. Darüber hinaus können Zusatzinformation, die während der Berechnung verloren gehen würden, aus der Vorverarbeitung in die Analyse der Simulationsergebnisse übertragen werden.

Um sowohl Eingabe- als auch Ergebnisdaten verarbeiten zu können, wurden flexible Datenstrukturen konzipiert, die den Zielgrößen beider Einsatzgebiete – interaktive Modifikation von Eingabedaten und minimaler Speicherbedarf für zeitabhängige Ergebnisdaten – gerecht werden. Damit konnte der Grundstein für die direkte Anbindung an den Simulationsprozess gelegt werden, die es dem Ingenieur erlaubt, einen Einblick in den aktuellen Stand der Berechnung zu erhalten [21].

Kooperatives Arbeiten

Der Kommunikationsbedarf räumlich getrennter kooperierender Entwicklungsingenieure ist in den vergangenen Jahren stark angestiegen. Das liegt sowohl am Wachstum fusionierender Firmen, deren Berechnungsabteilungen aus Synergiegründen möglichst eng zusammenarbeiten sollen, als auch an der zunehmenden Auslagerung von Entwicklungsaufgaben an externe Dienstleistungunternehmen. Eine Zusammenarbeit ohne eine gemeinsame Sicht auf das zu entwickelnde Fahrzeugmodell ist unmöglich. Projekttreffen mit allen Beteiligten sind vor allem bei größeren Entfernungen zeit- und kostenintensiv. Um einen Lösungsansatz für diese Probleme zu bieten, wurde im Rahmen dieser Arbeit eine Methode entwickelt, die Prototyp-Instanzen mehrerer Sitzungsteilnehmer an ihren Arbeitsplatzrechnern über eine zentrale Server-Applikation synchronisiert. Markierungsobjekte stellen einen

virtuellen Zeigestock dar und vermeiden bei einer parallel stattfindenden Telefonkonferenz Missverständnisse unter den Teilnehmern. Außer den Interaktions-Events können in einer bestehenden Sitzung über den CORBA-basierten Kommunikationskanal auch die für die 3D-Visualisierung benötigten Modelldaten zu Teilnehmern übertragen werden, die auf den aktuellen Datenstand keinen Zugriff haben. Ein Vorteil der erarbeiteten Event-basierten Lösung besteht darin, dass nur geringe Datenmengen an die anderen Rechner übertragen werden müssen und die lokale Visualisierung für ähnlich hohe Bildwiederholraten sorgt, wie im nicht-kooperativen Einsatz.

Der Einschränkung dieses Event-basierten Ansatzes, der auf jedem Client den Plattform-abhängigen Prototypen voraussetzt, wurde mit einem weiteren Bild-basierten Client-Server-Modell begegnet: dabei fungiert der Prototyp als Rendering-Server und verschickt die erzeugten Bilder als komprimierten Datenstrom an Clients, die lediglich in der Lage sein müssen, Bilddaten darzustellen. Als Ergebnis einer Diplomarbeit aus dem Umfeld der medizinischen Datenvisualisierung [16] und Forschungsergebnissen zur Remote-Visualisierung [14] wurde ein Java-Applet entwickelt, das es ermöglicht, der Visualisierungssitzung mit Hilfe eines Web-Browsers zu folgen. Dieser Ansatz bietet darüber hinaus die Möglichkeit, durch Rückübertragung der Interaktions-Events an den Prototypen die Server-Applikation zu steuern und stellt damit vor allem für Sitzungen über Breitbandverbindungen eine Alternative zum Synchronisationsansatz dar.

Die Synchronisierung mehrerer Visualisierungsinstanzen bietet allerdings auch noch einen Lösungsansatz für ein weiteres Problemfeld: die vergleichende Visualisierung. Neben der Möglichkeit, die Unterschiede zwischen zwei oder mehreren Datensätzen farblich auf einem der Modelle abzubilden, können mit Hilfe der Synchronisationsmethode Modellvarianten sowie unterschiedliche Crash-Verläufe in mehreren Instanzen des Prototypen, die lokal auf einem Rechner gestartet wurden, direkt visuell verglichen werden. In der Vorverarbeitung lassen sich durch Ausblenden identischer Fahrzeugstrukturen die Unterschiede der Modellvarianten schneller erfassen, während in der Analyse synchronisiert visualisierter Berechnungsergebnisse die Ursache verschiedener Strukturverformungen leichter nachvollzogen werden kann.

9.2 Prototypische Anwendungen

Die im Rahmen dieser Arbeit entwickelten Interaktions- und Visualisierungskonzepte konnten nur anhand der implementierten prototypischen Anwendungen evaluiert und optimiert werden. Während es für spezielle Problemlösungen ausreichte, sich eng an die Rahmenbedingungen zu halten, bestand bei der Entwicklung und Implementierung der Kerndatenstrukturen der Anspruch, auch für weiterführende wissenschaftliche Arbeiten im Umfeld der virtuellen Fahrzeugentwicklung eine Plattform zu schaffen, in die schnell neue Algorithmen integriert werden können. Dieser Abschnitt fasst nochmal die wichtigsten Prototypen zusammen.

Bild- und Filmgenerierung zur Standardauswertung

Die Berechnungsabteilung der *BMW Group* hat Ende der neunziger Jahre in Zusammenarbeit mit SGI die Entwicklung eines Web-basierten Systems zum gesteuerten transparenten Datenmanagement namens *CAE-Bench* [33] initiiert. Das System soll allen am Entwicklungsprozess beteiligten Ingenieuren die aktuellen Daten und Ergebnisse bereits durchgeführter Berechnungen zugänglich machen und eine einheitliche Dokumentation der Auswertungen fördern. Ursprünglich sollte der Berechnungsingenieur bei dem Modell-Assembly die unabhängig voneinander vernetzten und separat abgespeicherten Bauteile über die Web-Oberfläche zusammenführen. Kleine, das Bauteilnetz darstellende Piktogramme dienten zur Visualisierung des Datenbankbestandes. Darüber hinaus sollten im Anschluss an Simulationsrechnungen automatische Standardauswertungen vorgenommen werden.

Erste Ergebnisse dieser Arbeit konnten dazu verwendet werden, aus Modelldaten im Batch-Betrieb mit Hilfe eines Open Inventor basierten Offscreen-Renderers Bilder zu erzeugen. Während die Bilder der Einzelbauteile zu Piktogrammen verkleinert werden, können Ergebnisdaten unter Verwendung der Engine-Objekte in Open Inventor animiert werden. Zusätzlich werden Flugbahnen festgelegt, von denen aus die Kamera die sich verformende Struktur aufnimmt. Schließlich werden die hochaufgelösten Einzelbilder zu einem digitalen Video konvertiert, das zusammen mit den Ergebnisdaten in der Datenbank abgelegt wird und später zur Ergebnisanalyse dem Berechnungsingenieur zu einer ersten Ansicht zur Verfügung steht.

Der Postprocessing-Schritt wird durch derartige Standardauswertungen, die durch 2D-Diagramme und 3D-Visualisierungen der wichtigsten Parameter vervollständigt werden, stark beschleunigt. Das System wird inzwischen von der Firma MSC unter dem Namen *Virtual Insight* vertrieben und verwendet ein internes Visualisierungsmodul.

Kraftflussröhrenberechnung

Die von Kuschfeldt et al. [44] vorgestellte Kraftflussvisualisierung durch schlauchförmige Glyphen konnten weiterentwickelt und zur interaktiven Anwendung gebracht werden. Es wurden Dateiformate für Definitionen der Kraftflussröhrenverläufe sowie für die vorberechneten Kraftflussröhren entwickelt. Dadurch wurde es möglich, die vorab interaktiv festgelegten Hauptlastpfade im Anschluss an die Simulationsrechnung auszuwerten. Die im Batch-Betrieb berechneten Kraftflussröhren lassen sich während der Analyse in den Prototypen einlesen und werden sofort in eine entsprechende Schlauchrepräsentation umgewandelt. Die Kraftflussröhre kann dann zusammen mit der sich verformenden Fahrzeugstruktur animiert werden.

Ermittlung von Instabilitäten

Die Ermittlung von Instabilitäten in Simulationsergebnissen, die aus gleichen Eingabedaten hervorgegangen sind, spielt für die Bewertung des Modellaufbaus und der Simulationssoftware eine wichtige Rolle. Es wurde aufgezeigt, wie eine im Prinzip beliebige Anzahl solcher Simulationsläufe auf ihre Stabilität hin untersucht werden können. Dabei haben sich lokale Maße bewährt, um schnell den jeweiligen Ursprung einer Instabilität entdecken zu können. Das Textur-basierte Geometrie-Clipping konnte auch in diesem Zusammenhang Gewinn bringend eingesetzt werden, um Ort und Zeitpunkt einer Verzweigung in den verschiedenen Simulationsläufen auszumachen. Analog zur Kraftflussberechnung wurde eine Zweiteilung der Problemlösung gewählt: Die Anwendung eines der implementierten Instabilitätsfunktionale findet im Batch-Betrieb statt und liefert als Resultat einen zu visualisierenden Skalarwert pro Netzknoten, der die Ergebnisabweichung an diesem Ort repräsentiert.

Die skalaren Größen quantifizieren zwar die Abweichungen, lassen jedoch keinen direkten Rückschluss auf das Spektrum der unterschiedlichen Strukturverformungen zu. Mit Hilfe der Event-basierten vergleichenden Visualisierung der am meisten voneinander abweichenden Simulationsläufe können diese Differenzen veranschaulicht und direkt miteinander verglichen werden. Eine Klassifizierung der Instabilität wird erleichtert und der Anwender wird bei der Auswahl eventuell vorzunehmender konstruktiver Maßnahmen durch den Überblick über die Verformungsmöglichkeiten unterstützt.

Pre- und Postprocessing mit *crashViewer*

Das angestrebte Fernziel im Bereich der Strukturmechaniksimulation ist die enge Kopplung von Pre- und Postprocessing. Die benötigte Rechenzeit für einen Simulationslauf soll durch massive Parallelisierung der Simulationsverfahren und Ausnutzung leistungsfähiger Simulationscluster sowie durch Weiterentwicklung der Solver in Richtung Wiederverwendbarkeit von Teilergebnissen drastisch gesenkt werden. Mit der Entwicklung interner Datenstrukturen und Algorithmen, die in der Lage sind, sowohl Simulationseingabedaten als auch Simulationsresultate effizient zu verarbeiten, konnte bereits jetzt gezeigt werden, dass Pre- und Postprocessing-Funktionalität in eine Visualisierungsapplikation integriert werden können, ohne Performanzeinbußen in Kauf nehmen zu müssen. Im Rahmen des *Autobench*-Projektes konnte von einem anderen Doktoranden basierend auf den *crashViewer*-Datenstrukturen gezeigt werden, wie ein Fahrzeugmodell zunächst mit dem Prototypen vorverarbeitet und anschließend während der laufenden Simulation mit Hilfe einer weiteren CORBA-Schnittstelle das Crash-Verhalten vom Berechnungsingenieur überwacht werden kann (siehe auch Abschnitt 9.3). Darüber hinaus erleichtert die Fusion von Pre- und Postprocessing-Funktionalität in einem Software-Werkzeug die Steuerung und Handhabung durch den Anwender, da dieser sich nicht auf eine Vielzahl von Benutzerschnittstellen einarbeiten muss.

9.3 Weiterführende Arbeiten

Im Folgenden wird noch auf Projekte und Arbeiten anderer Doktoranden verwiesen, die auf den im Rahmen dieser Arbeit geschaffenen Grundlagen basieren und den entstandenen Prototypen um weitere Problemlösungsansätze bereichern. Zunächst sei hier auf das vom Bundesministerium für Bildung und Forschung finanzierte *Autobench*-Projekt verwiesen, in dessen Rahmen *crashViewer* unter Mitwirkung der beiden Doktoranden Norbert Frisch und Dirc Rose weiterentwickelt wurde. So wurde zum Beispiel ein Flanschverfolgungsalgorithmus entwickelt, der es ermöglicht, zwei interaktiv spezifizierte Bauteile über den kompletten Flanschbereich durch Bauteilverbindungen aneinander zu koppeln [24]. Darüber hinaus wurden Schweißnähte und Klebeschichten als weitere Bauteilverbindungsarten in den Prototypen integriert [57] und eine Anbindung an einen Löser-unabhängigen Datenserver über eine weitere CORBA-Schnittstelle realisiert [21]. In [56] wird ein in *crashViewer* implementierter, auf Normalen-Texturen basierender Ansatz zur Steigerung der Darstellungsqualität bei der Visualisierung von Strukturmodellen unter Nutzung simplifizierter Polygonnetze vorgestellt. Derzeit wird in einem Nachfolgeprojekt namens *Auto-Opt* daran gearbeitet, das Netz des Finite-Element-Modells mit Hilfe intelligenter Manipulator-Objekte interaktiv modifizieren zu können [22].

Der aus diesen Arbeiten entstandene Prototyp *crashViewer* wurde bereits relativ früh von den Berechnungsingenieuren der *BMW Group* und deren Zulieferern eingesetzt, da dadurch das Preprocessing in der Crash-Berechnung signifikant beschleunigt werden konnte. Auf Drängen der Anwender ist der Prototyp inzwischen kommerzialisiert worden. Das entstandene Produkt *scFEMod* [65] wird bei der Firma *science + computing* weiterentwickelt und ist derzeit bei zwei Automobilherstellern und mehreren Dienstleistungunternehmen produktiv im Einsatz.

Kapitel 10

Zusammenfassung und Ausblick

Die Automobilhersteller müssen in immer kürzer werdenden Abständen neue Produkte auf den Markt bringen, um im zunehmenden Konkurrenzkampf bestehen und auf sich ändernde Kundenwünsche reagieren zu können. Gleichzeitig nimmt bei vielen Herstellern die Anzahl der entwickelten Fahrzeuglinien (Limousine, Coupé, Cabriolet, Mini-Van, Geländeoder Kleinwagen) und deren Variationen zu, um einen Großteil der Marktsegmente abdecken zu können. Das setzt eine Zeit- und Kostenersparnis voraus, die nur erreicht werden kann, wenn der gesamte Entwicklungsprozess vom Design bis zur Produktion immer wieder überprüft und optimiert wird. Ein wichtiges Glied in der Produktentstehungkette ist die numerische Simulation, die es schon in der frühen Planungsphase ermöglicht, Konzeptentwürfe auf ihre Machbarkeit zu überprüfen. Berechnungsergebnisse sind wesentlich günstiger und schneller zu erhalten als Resultate aus Versuchen mit realen Fahrzeugprototypen und sie führen im weiteren Verlauf zu detaillierteren Aussagen, die der Steigerung der Produktqualität dienen.

Ziel der vorliegenden Dissertation war es, die Arbeitsabläufe in einem Teilbereich der virtuellen Fahrzeugentwicklung, im Pre- und Postprocessing der Strukturmechaniksimulation zu analysieren, und den Prozess durch die Entwicklung neuer Visualisierungstechniken und Interaktionsmechanismen zu beschleunigen. Durch die enge Kooperation mit der Crash-Berechnungsabteilung der *BMW Group* war es möglich, einen detaillierten Einblick in den simulationsbasierten Entwicklungsprozess zu bekommen. In der Diskussion mit den Berechnungsingenieuren konnten Probleme im damaligen Entwicklungsprozess identifiziert werden, die zum einen durch die Umstellung der Berechnungsmethode auf inhomogen vernetzte Bauteile begründet waren und sich zum anderen daraus ergaben, dass die Aufgaben mit den eingesetzten Visualisierungswerkzeugen nicht oder nur unbefriedigend gelöst werden konnten. Zudem sollte der durch die zunehmende Modellkomplexität abnehmenden Effizienz in der Strukturmechanikvisualisierung durch Methoden entgegengewirkt werden, die speziell auf das Einsatzgebiet zugeschnitten sind. Die Lösungsvorschläge wurden im Dialog mit den Anwendern erarbeitet, um aus einer Reihe möglicher Ansätze jeweils denjenigen auszuwählen, der einerseits technisch umsetzbar ist und andererseits den Entwicklungsprozess durch die erzeugten Ergebnisse signifikant verbessern und beschleunigen kann.

Durch die Entwicklung eines effizienten Szenengraph-Designs ist es gelungen, den Speicherbedarf für die Repräsentation zeitabhängiger, topologisch invarianter Finite-Element-Modelle, wie sie aus der Strukturmechaniksimulation resultieren, deutlich zu senken. Zusammen mit internen Datenstrukturen, die auf den Aufbau der Simulationsergebnisdateien ausgerichtet sind, und einem parallelisierten Initialisierungsprozess wurde das Einlesen und Verarbeiten der Ergebnisse einer Gesamtfahrzeugsimulation über 60 Zeitschritte auf einem Arbeitsplatzrechner ermöglicht. Insbesondere die erzielten kurzen Ladezeiten für Pre- und Postprocessing-Daten schufen die Grundlage dafür, dass sich die ständig unter Projektdruck stehenden Berechnungsingenieure hin und wieder die Zeit nahmen, die prototypisch implementierten Forschungsergebnisse anzuwenden und wertvolles Feedback zu geben. Gespräche mit den Anwendern über die entwickelten Interaktionsmechanismen haben ergeben, dass die komfortable Kamerasteuerung, die den Berechnungsingenieur bei der Suche nach Fehlern im Modell aktiv unterstützt, sehr positiv angenommen worden ist. Auf die Implementierung einer umfangreichen graphischen Benutzerschnittstelle, über die alle Funktionalitäten angesteuert werden können, wurde im Prototypen verzichtet; allerdings hätte dies Neueinsteigern sicherlich die Einarbeitung erleichtert.

Mit der speziell für dieses Anwendungsfeld entwickelten Quadrilateralstreifengenerierung konnte ein Weg aufgezeigt werden, bei dem ohne Detailverlust die Darstellungsgeschwindigkeit der Finite-Element-Struktur deutlich erhöht wird. Das implementierte Simplifizierungsverfahren nutzt das Szenengraph-Design und ermöglicht auch für große Modelle bei minimalem Speichermehrbedarf eine Beschleunigung der Interaktionsraten um den Faktor 4-6. Darüber hinaus konnten durch Anwendung Textur-basierter Methoden skalare und vektorielle Größen visualisiert werden. Es wurde ein Lastausgleich auf der Graphik-Hardware durch Entlastung der Geometrie-Einheit und Ausnutzung freier Kapazitäten des Rasterisierungssubsystems geschaffen. Diese Entwicklungen tragen im Vergleich zu den konventionellen Alternativen ebenfalls zu einer weiteren Steigerung der Bildwiederholrate bei und erlauben gleichzeitig das Ausblenden von Teilstrukturen in Abhängigkeit der zugehörigen Skalarwerte.

Die zusätzlich zum Szenengraphen verwendete hierarchische Datenstruktur bildet zusammen mit den adaptierten Algorithmen zur Kollisionsdetektion und Abstandsberechnung die Grundlage für viele Funktionalitäten im Bereich der Netzmodifikation und der Modellvalidierung. Im Rahmen der vorliegenden Arbeit fand die Bounding-Volume-Hierarchie in erster Linie für die Detektion initialer Penetrationen sowie für die effiziente Überprüfung von Schweißpunktdaten Verwendung. Bisher konnten Penetrationen im Modell lediglich durch Anrechnen „unkontrollierbar“ behoben werden. Ein Resultat dieser Arbeit ist die interaktive und vor allem kontrollierte Beseitigung der Penetrationen, die erstmals das Anpassen einer Bauteilvariante an ihre Umgebung effizient ermöglicht. Durch die Markierung fehlerhafter Schweißpunkte und die Möglichkeit, diese Fehler mit wenigen Interaktionen zu beheben, konnte der Berechnungsprozess, der zuvor aufgrund fehlerhafter Verbindungselemente hin und wieder abstürzte, stabilisiert und damit Zeit und Kosten eingespart werden.

Es wurde ein Interaktionskonzept erarbeitet, durch das die patentierte Kraftflussvisualisierung in der Analyse von Crash-Simulationsergebnissen erstmals interaktiv zum Einsatz

gebracht werden konnte. Darüber hinaus wurde eine Methode zur Visualisierung von Instabilitäten in der Crash-Simulation entwickelt. Insbesondere die Möglichkeit, mehrere Instanzen des entstanden Prototypen *crashViewer* über eine CORBA-Verbindung zu synchronisieren, erleichtert den Berechnungsingenieuren die Erforschung der Ursprünge und der Weiterentwicklung derartiger Verzweigungen im Crash-Verhalten. Mit der Batch-basierten Vorabberechnung der Instabilitäten und der Kraftflussröhren, sowie mit den off-screen generierten digitalen Filmen konnte mit dieser Arbeit ein Beitrag zur Automatisierung und Beschleunigung des Prozesses in der digitalen Karosserieentwicklung geleistet werden.

In Zukunft wird der Anteil der Volumenelemente durch den Einsatz neuer Materialien im Fahrzeugbau, zum Beispiel Hartschäume, aber auch durch eine genauere Abbildung der Verbindungstechnik für die Versagensanalyse stark zunehmen. Um das Verformungsverhalten innerhalb von Volumenbauteilen besser analysieren zu können, wird das Hardwarebasierte Volume-Rendering als neue Technik in der Strukturmechanikvisualisierung Einzug halten. Es wurden bereits erste Untersuchungen im Zusammenhang mit der Strukturoptimierung von Last aufnehmenden Bauteilen und der Akustikvisualisierung in der Fahrgastzelle unternommen. Dazu wurde basierend auf einem Szenengraph-basierten Konzept zur interaktiven Visualisierung medizinischer Daten [62] ein Volume-Rendering-Knoten für Cosmo3D / OpenGL Optimizer im Rahmen einer Diplomarbeit [72] entwickelt und in den Prototypen *crashViewer* integriert.

Aber auch die Forschung in dem als *Commodity*-Sektor bezeichneten PC-Bereich wird in Zukunft zu neuen Lösungen im Berechnungsumfeld beitragen. Schon seit geraumer Zeit weichen in den großen Berechnungsabteilungen teure Supercomputer den günstigeren Compute-Clustern, bestehend aus mehreren hundert Knoten, um die Rechenzeit durch die sich schnell weiter entwickelnden CPUs zu verkürzen. Darüber hinaus werden immer neue Wege gesucht, um die Komponenten moderner Graphik-Hardware „zweckentfremdet“ zur Beschleunigung von Berechnungsaufgaben einzusetzten. Die Hardware-basierte Parameterübertragung zwischen inkompatiblen Finite-Element-Netzen hat bereits im Rahmen dieser Arbeit gezeigt, wie durch Ausnutzung der Rechenkapazität des Graphiksubsystems bestimmte Berechnungen wesentlich schneller durchgeführt werden können als auf der CPU. Durch die rasante Weiterentwicklung der Graphik-Hardware im PC-Bereich und den Umstieg von einer vor wenigen Jahren noch fest verdrahteten (*fixed function pipeline*) zu einer in Teilen frei programmierbaren Graphik-Pipeline kann aus der GPU[1] großer Nutzen für Berechnungen gezogen werden, die bisher nur mit Hilfe der CPU gelöst wurden. Lindholm et al. [47] stellen die Programmierung der GeForce3-Vertex-Engine vor und zeigen verschiedene Effekte mit Hilfe von Vertex-Programmen, wie zum Beispiel Morphing, anisotrope Beleuchtung oder Bump- und Environment-Mapping. Thompson et al. [69] präsentieren ein Framework, das die GPU für Berechnungen allgemeiner Art nutzt. Eine Analyse ergibt, dass die GPU große Matrizen mit vielen hundert Spalten und Zeilen um ein Vielfaches schneller verarbeiten kann, als die CPU. Aktuelle Arbeiten [7, 30, 41] zeigen auf, wie die Kapazität der GPU zum Lösen dünnbesetzter Matrizen mit Hilfe der Konjugierten-

[1] GPU — **G**raphics **P**rocessing **U**nit

Gradienten-Methode oder dem Multigrid-Verfahren genutzt werden kann. Somit wird derzeit vielleicht der Grundstein dafür gelegt, dass in naher Zukunft das Graphiksubsystem nicht nur zur Visualisierung, sondern auch zur Berechnung von Simulationsergebnissen eingesetzt wird.

Damit die Visualisierung in den Berechnungsabteilungen der Automobilhersteller und der ihnen zuliefernden Dienstleistungsunternehmen von den Entwicklungen im Graphik-Bereich profitieren kann, muss dem Berechnungsingenieur zunächst einmal die entsprechende Hardware am Arbeitsplatz zur Verfügung gestellt werden. Dieser Umstellungsprozess wird langfristig geplant, da alle Software-Werkzeuge, die unverzichtbarer Bestandteil des Fahrzeugentwicklungsprozesses sind, auf die neue Hardware portiert werden müssen. Sobald dieser Schritt vollzogen sein wird, werden Hardware-basierte Algorithmen völlig neue Möglichkeiten der Visualisierung im digitalen Entwicklungsprozess anbieten, denn bis dahin wird sich die Leistung der PC-Graphik-Karten, getrieben von der Spiele-Industrie, wieder mehrfach verdoppelt haben.

Literaturverzeichnis

[1] Kurt Akeley, Peter Haeberli, and David Burns. `tomesh.c`. C Program on SGI Developer's Toolbox CD, 1990.

[2] Altair Engineering. HyperMesh. http://www.altair.com.

[3] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[4] BETA CAE Systems. ANSA — Automatic Net-generation for Structural Analysis. http://www.ansa-usa.com.

[5] J.F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, July 1977.

[6] J.F. Blinn. Simulation of Wrinkled Surfaces. In *SIGGRAPH '78 Conference Proceedings*, pages 286–292. ACM SIGGRAPH, 1978.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *Proc. of ACM SIGGRAPH 2003*. ACM SIGGRAPH, 2003.

[8] M. Braitmaier, M. Haiss, M. Knauß, S. Langauf, S. Opferkuch, G. Stein, and P. Stolz. OpenManip — Manipulatoren für Cosmo3D. Abschlussbericht zum Studienprojekt, Abt. f. Visualisierung und Interaktive Systeme, IFI, Universität Stuttgart, 2000/2001.

[9] Stephen Cameron. A Comparison of Two Fast Algorithms for Computing the Distance Between Convex Polyhedra. *IEEE Transactions on Robotics and Automation*, 13(6):915–920, December 1997.

[10] Swen Campagna. *Polygonreduktion zur effizienten Speicherung, Übertragung und Darstellung komplexer polygonaler Modelle*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1998. ISBN 3-89675-480-7.

[11] M. M. Chow. Optimized Geometry Compression for Real-Time Rendering. In *Proc. IEEE Visualization '97*, pages pages 346–354. IEEE Computer Society Press, November 1997.

[12] Dassault Systemes. CATIA. http://www.catia.com.

[13] M. Deering. Geometry Compression. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 13–20, 1995.

[14] Klaus Engel, Ove Sommer, Christian Ernst, and Thomas Ertl. Remote 3D Visualization using Image-Streaming Techniques. In *Advances in Intelligent Computing and Multimedia Systems (ISIMADE '99)*, pages 91–96, 1999.

[15] Klaus Engel, Ove Sommer, and Thomas Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Proc. of EG/IEEE TCVG Symposium on Visualization VisSym 2000*, pages 167–177,291. Springer Wien/New York, May 2000.

[16] Christian Ernst. Medizinische Visualisierung im WWW mittels 3D-Texturen. Diplomarbeit, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg, Mai 1999.

[17] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing Triangle Strips for Fast Rendering. In Yagel and Nielson, editors, *Proc. IEEE Visualization '96*, pages 319–326, 1996.

[18] Raphael A. Finkel and Bentley Jon L. Quad Trees: A data structure for retrieval on composite keys. In *Acta Informatica*, volume 4, pages 1–9. Springer, 1974.

[19] Thomas Frank. Crashsimulation — Stand der Technik – Möglichkeiten – Herausforderungen. Vortrag gehalten bei science + computing ag, Tübingen, Dezember 2002.

[20] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[21] Norbert Frisch and Thomas Ertl. Embedding Visualization Software into a Simulation Environment. In *Proceedings of the Spring Conference on Computer Graphics, Bratislava*, pages 105–113, April 2000.

[22] Norbert Frisch and Thomas Ertl. Deformation Of Finite Element Meshes Using Directly Manipulated Free-Form Deformation. In *Proceedings of Seventh ACM Symposium on Solid Modeling and Applications 2002*, pages 249–256, 2002.

[23] Norbert Frisch, Dirc Rose, Ove Sommer, and Thomas Ertl. Pre-processing of Car Geometry Data for Crash Simulation and Visualization. In Vaclav Skala, editor, *WSCG 2001 - The Ninth International Conference in Central Europe on Computer Graphics and Visualization*, pages 25–32, February 2001.

[24] Norbert Frisch, Dirc Rose, Ove Sommer, and Thomas Ertl. Visualization and Preprocessing of Independent Finite Element Meshes for Car Crash Simulations. *The Visual Computer*, 18(4):236–249, 2002.

[25] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *SIGGRAPH '80 Conference Proceedings*, volume 14, pages 124–133, July 1980.

[26] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[27] Michael Garland and Paul S. Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 263–270, 1998.

[28] Bernd Gärtner. Fast and Robust Smallest Enclosing Balls. In *Proc. 7th Annual European Symposium on Algorithms (ESA)*, pages 325–338. Springer Verlag, 1999.

[29] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three dimensional space. *IEEE Transactions on Robotics and Automation*, 4:193–203, April 1988.

[30] Nolan Goodnight, Gregory Lewin, David Luebke, and Kevin Skadron. A Multigrid Solver for Boundary-Value Problems Using Programmable Graphics Hardware. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware 2003*, 2003.

[31] 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.

[32] Horst Hadler. Evaluierung und Implementierung verschiedener Optimierungsverfahren für die effiziente Visualisierung komplexer Fahrzeugmodelle. Diplomarbeit, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg, Dezember 1998.

[33] J. Hägele, U. Hänle, A. Kropp, M. Streit, C. Kerner, and M. Schlenkrich. The CAE-Bench Project — A Web-based System for Data, Documentation and Information to Improve Simulation Processes. In *Proceedings of 2nd MSC Worldwide Automotive Conference*, 2000.

[34] Michael Holzner, Touraj Gholami, and Horst-Uwe Mader. Virtuelles Crashlabor: Zielsetzung, Anforderungen und Entwicklungsstand. In *VDI Berichte 1411: Berechnungen im Automobilbau, Tagung Würzburg*. VDI Gesellschaft Fahrzeug und Verkehrstechnik Düsseldorf, September 1998.

[35] Hugues Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. In *ACM Computer Graphics, Proc. SIGGRAPH '99*, pages 269–276. ACM SIGGRAPH, August 1999.

[36] Silicon Graphics Inc. OpenGL Optimizer™ Programmer's Guide: An Open API for Large-Model Visualization. Silicon Graphics Inc., IRIS Insight Library, 1998. http://techpubs.sgi.com/.

[37] Silicon Graphics Inc. OpenGL Performer Programer's Guide. Silicon Graphics Inc., IRIS Insight Library, 2002. http://techpubs.sgi.com/.

[38] ISO — International Organization for Standardization. STEP — Standard for the Exchange of Product Model Data. http://pdesinc.aticorp.org/.

[39] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. In *Proceedings IEEE Visualization '96*, pages 311–318, October 1996. ISBN 0-89791-864-9.

[40] Jan Kraheberger. Interaktive Aufbereitung von vorvernetzten Bauteilgeometrien für die Fahrzeugberechnung. Diplomarbeit, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg, Oktober 1998.

[41] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *Proc. of ACM SIGGRAPH 2003*. ACM SIGGRAPH, 2003.

[42] Jens Kuenzl. Visualisierung und Beseitigung initialer Durchdringungen bei Finite Elemente Gittern. Studienarbeit, Abt. f. Visualisierung und Interaktive Systeme, IFI, Universität Stuttgart, 2001.

[43] Sven Kuschfeldt. *Effiziente Visualisierungsverfahren zur besseren Erfassung von Crash-Simulationen im Fahrzeugbau.* Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1998.

[44] Sven Kuschfeldt, Thomas Ertl, and Michael Holzner. Efficient visualization of physical and structural properties in crash-worthiness simulations. In Yagel and Hagen, editors, *Proc. IEEE Visualization '97*, pages 487–490,583. IEEE Computer Society Press, October 1997. ISBN 1-58113-011-2.

[45] Sven Kuschfeldt and Michael Holzner. Visualisierung von Kraftfluss- und Momentenverläufen in einer Struktur oder in Elementen aus dieser Struktur. Deutsches Patent- und Markenamt, Oktober 1999. Offenlegungsschrift DE19818582A1.

[46] Sven Kuschfeldt, Ove Sommer, and Thomas Ertl. Efficient Visualization of Crash-Worthiness Simulations. *IEEE Computer Graphics and Applications*, 18(4):60–65, July/August 1998.

[47] E. Lindholm, M. J. Kligard, and H. Moreton. A User-Programmable Vertex Engine. In *Proc. of ACM SIGGRAPH 2001*, pages 149–158. ACM SIGGRAPH, July 2001.

[48] NTIS — National Technical Information Service. IGES. http://www.nist.gov/iges/.

[49] OpenDWG Alliance. OpenDWG. http://www.opendwg.org.

[50] M. Peercy, J. Airy, and B. Cabral. Efficient Bump Mapping Hardware. *Computer Graphics, Proc. SIGGRAPH '97*, pages 303–307, July 1997.

[51] B.-T. Phong. Illumination of Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[52] Thomas Piekarski. Anpassung des crashViewer für immersive Visualisierung. Studienarbeit, Abt. f. Visualisierung und Interaktive Systeme, IFI, Universität Stuttgart, Januar 2001.

[53] PTC — Parametric Technologies Corporation. Pro/ENGINEER. http://www.ptc.com.

[54] Jack Ritter. An efficient bounding sphere. In Andrew Glassner, editor, *Graphic Gems*, pages 301–303. Academic Press, 1990.

[55] Greg Roelofs. ZLIB. http://www.gzip.org/zlib/.

[56] Dirc Rose and Thomas Ertl. Rendering Details on Simplified Meshes by Texture Based Shading. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 239–245, 2000.

[57] Dirc Rose, Norbert Frisch, Thomas Ruehr, and Thomas Ertl. Interaktive Visualisierung neuer Elemente im virtuellen Automobil-Crashversuch. In *Tagungsband SimVis '02, Magdeburg*, 2002.

[58] J. Rossignac and Borrel P. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T.L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[59] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[60] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *ACM Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[61] Ove Sommer. Hardware-unterstützte Beleuchtungsberechnung in der Volumenvisualisierung. Diplomarbeit, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg, Juni 1997.

[62] Ove Sommer, Alexander Dietz, Rüdiger Westermann, and Thomas Ertl. An Interactive Visualization and Navigation Tool for Medical Volume Data. *Computers & Graphics*, 2:233–244, 1999.

[63] Ove Sommer and Thomas Ertl. Geometry and Rendering Optimizations for the Interactive Visualization of Crash-Worthiness Simultations. In *Proceedings of IT&T/SPIE Electronic Imaging, Visual Data Exploration and Analysis VII*, volume 3960, pages 124–134, January 2000.

[64] Ove Sommer and Thomas Ertl. Comparative Visualization of Instibilities in Crash-Worthiness Simulations. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 319–328,364. Springer, May 2001.

[65] Ove Sommer, Norbert Frisch, Dirc Rose, and Thomas Ertl. scFEMod — The New Preprocessor for Efficient Assembly and Model Validation. In *Procceedings of 4th European LS-DYNA User's Conference*, pages F–I–21–32. DYNAmore GmbH, May 2003. ISBN 3-00-011175-1.

[66] Henry A. Sowizral, David R. Nadeau, Michael J. Bailey, and Michael F. Deering. Introduction to programming with java3d. ACM SIGGRAPH '98 Course Notes, July 1998.

[67] T-Systems — Digital Engineering Solutions. MEDINA. http://www.c3pdm.com/des/products/medina/.

[68] Clemens-August Thole, editor. *AUTOBENCH: Integrierte Entwicklungsumgebung für virtuelle Automobil-Prototypen.* Number 145 in GMD-Report. GMD – Forschungszentrum Informationstechnik GmbH, Juli 2001. ISSN 1435-2702.

[69] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *Procceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[70] Greg Turk. Re-tiling polygonal surfaces. *ACM Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):55–64, July 1992.

[71] Verband der Automobilindustrie. VDA-FS — Verband der Automobilindustrie Flächen Schnittstelle. http://www.vda.de/.

[72] Manfred Weiler. Evaluierung und Einsatz von OpenGL Volumizer zur Volumenvisualisierung auf strukturierten und unstrukturierten Gittern. Diplomarbeit, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg, November 1999.

[73] J. Wernecke. *Open Inventor C++ Reference Manual.* Addision-Wesley, 1994.

[74] J. Wernecke. *The Inventor Mentor, Programming Object-Oriented 3D Graphics with OpenInventor.* Addision-Wesley, 1994.

[75] J. Wernecke. *The Inventor Toolmaker, Extending OpenInventor.* Addision-Wesley, 1994.

[76] B. Yamrom and K. Martin. Vector Field Animation with Texture Maps. *IEEE Computer Graphics and Applications*, 15(2):22–24, March 1995.

Lebenslauf

Ove Sommer

geboren am 30. Juli 1967 in Kiel
verheiratet, drei Söhne

Schulausbildung:

1974 – 1978	Grundschule, Schafflund
1978 – 1982	Auguste-Viktoria-Gymnasium, Flensburg
1982 – 1984	Realschule, Schafflund
05/1984	Abschluss: Mittlere Reife
1984 – 1988	Fachgymnasium — technischer Zweig, Flensburg
05/1988	Abschluss: Algemeine Hochschulreife

Wehrdienst:

1988 – 1990	Soldat auf Zeit für zwei Jahre

Hochschulausbildung:

1990 – 1997	Studium der Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg
07/1997 – 06/1999	Wissenschaftlicher Mitarbeiter an dem Lehrstuhl für Graphische Datenverarbeitung (IMMD IX) der Friedrich-Alexander-Universität Erlangen-Nürnberg
08/1997 – 07/2000	Promotionsstipendium der BMW Group, München
07/1999 – 06/2001	Wissenschaftlicher Mitarbeiter in der Abteilung für Visualisierung und Interaktive Systeme des Instituts für Informatik der Universität Stuttgart

Berufstätigkeit:

seit 07/2001	Produktmanager des Visualisierungssystems *scFEMod* bei der science + computing ag, Tübingen