EG'99 Full Day Tutorial

—

Advanced Graphics Programming using OpenGL and Extensions

Lecturers:

R. Westermann
Dept. of Computer Science, University of Utah
W. Heidrich
Max-Planck-Institute for Computer Science, Saarbrücken
O. Sommer
Dept. of Computer Science, University of Stuttgart

Motivation

With fast 3D graphics hardware becoming more and more available even on low end platforms, the focus in developing new algorithms is beginning to shift towards higher quality rendering and additional functionality instead of simply higher performance implementations of the traditional graphics pipeline.

Graphics libraries like OpenGL and its extensions provide access to advanced graphics operations in the geometry and the rasterization stage and therefore allow for the design and implementation of completely new classes of rendering algorithms. Prominent examples can be found in realistic image synthesis (shading, bump/environment mapping, reflections) and scientific visualization applications (volume rendering, vector field visualization, data analysis). OpenGL Optimizer and Cosmo3D, on the other hand, are platform-independent APIs which are supported on SGI workstations and NT systems. Designed as high-end graphics APIs built on top of OpenGL they offer a variety of useful built-in algorithms specifically designed to allow for efficient rendering of complex polygonal models. Cosmo3D, or in the future the Fahrenheit Scene Graph, will be used as a base for different kinds of high-level applications.

The goal of the proposed tutorial is twofold: To give both a state-of-the-art overview of advanced graphics programming using the extended OpenGL graphics library and the Optimizer toolkit and to present a number of selected graphics applications in which hardware supported operations are paramount. The first part of this course summarizes the most important fundamentals and features of the graphics libraries OpenGL and Optimizer where individual recipes for the practical and efficient design of algorithms are emphasized. The second part of the tutorial is dedicated to various subfields of computer graphics ranging from realistic image synthesis and scientific visualization to structural mechanics. In each of them hardware accelerated graphics operations are used thus allowing interactive, high quality rendering and analysis of complex polygonal models and volume data sets.

Preface and Overview

These tutorial notes provide insight to particular applications in which OpenGL and its extensions and the Optimizer toolkit are heavily employed for scientific visualization, realistic image synthesis and interactive handling and manipulation of complex scenes. The emphasis of these notes lies on demonstrating the functionality and power of the used libraries as well as their relevance in computer graphics applications. In each of the examples described below we outline scenarios in which different features that are available in the core or extended OpenGL function set and in the Optimizer toolkit are exploited to considerably improve rendering performance and quality.

In particular, we introduce new concepts for the design of graphics algorithms by identifying a powerful set of orthogonal features to be implemented in hardware, which can then be flexibly combined to form new algorithms. Our major goal is to provide participants with sustained knowledge concerning state-of-the-art graphics programming using dedicated graphics hardware, and to demonstrate novel

and future directions in the design and implementation of graphics algorithms. Participants should become familiar with sophisticated OpenGL extensions and a variety of algorithms integrated into the Optimizer/Cosmo3D toolkits to allow for efficient rendering of complex polygonal models.

Schedule

At the beginning of this tutorial all materials used for presentation will be hand out to the participants. In particular this includes hard-copies of all slides that will be shown. An electronic version that is also available for non participants will be available after the tutorial from the Eurographics web-pages. In contrast to the notes provided hereafter these handouts also include additional descriptions of the main ingredients used in the proposed examples.

Session 1 (9.00 - 9.45):
Introduction to OpenGL
This session gives an introduction to the advanced OpenGL functionality. We will start with simple examples to explain some of the features of OpenGL that are important in the proposed applications. In particular we will focus on a detailed description of the rendering pipeline demonstrating the potential use and benefits of operations performed in different processing stages. A brief description of the functionality available in the geometry unit will be followed by a detailed investigation and explanation of the rasterization stage, the frame buffer hardware and advanced pixel transfer operations. Throughout this session we will demonstrate the benefits of the presented features by examples from practical applications.

Session 2 (10.00 - 10.45):
OpenGL Optimizer
In the second session we will proceed with an introduction to the large model visualization API OpenGL Optimizer and the underlying scene graph layer Cosmo3D. We will give an overview of various tools which are made available by Optimizer. In particular we will focus on efficient strategies to modify the Cosmo3D scene graph in order to obtain optimal frame rates. Different examples will demonstrate the benefits which can be easily inherited from the built-in features. We will refer to special techniques like view frustum and occlusion culling, especially with respect to the term Anti-Graphics-API Optimizer stands for. Throughout this session we will give practical advises how to improve the performance of graphics applications using the presented APIs.

Session 3 (11.00 - 11.45):
OpenGL extensions and the imaging subset
OpenGL extensions are a powerful mechanism for providing innovative functionality in new graphics systems. In this session we will describe how to use OpenGL extensions, and we will give an overview of some of the most important extensions. In particular we will emphasize the importance of the so-called "imaging subset" in OpenGL version 1.2, which consists of advanced per-pixel operations like color matrices, lookup tables, convolutions and histograms. Additional extensions covered include the new multi-texture extension and pixel textures. Many of the presented extensions will also be used in the "Applications" session.

Lunch break (11.45 - 13.00)

Session 4 (13.00 - 13.45):
High-quality shading and lighting
In this session we will present techniques for the simulation of realistic shading and lighting effects with OpenGL. We cover different methods for generating shadows, and discuss several techniques for bump mapping on contemporary graphics hardware. Then we present the concept of view-independent environment maps, and use them to visualize global illumination solutions on glossy surfaces. We conclude with a discussion of how to achieve anisotropic effects using graphics hardware. Throughout this section we will use both standard OpenGL functionality and some of the extensions presented in the first session.

Session 5 (14.00 - 14.45):
Advanced volume rendering techniques
In this session we will demonstrate the use of 3D textures in order to make interactive navigation and parameter modification possible for the rendering of scalar data sets defined on regular grids. We will outline various ways to take advantage of graphics hardware for the rendering of volumetric data. First, we will briefly summarize the fundamental algorithm for 3D texture based volume rendering. Then we will introduce easy and intuitive use of arbitrary clipping geometries based on stencil buffer operations and polygon tesselation. Additionally we will show that lighted and shaded iso-surfaces can be generated with 3D texture mapping in real-time without extracting any polygonal representation. Finally, we will outline a novel and practical approach for the rendering of unstructured grids, thus establishing a general framework for hardware accelerated volume rendering.

Session 6 (15.00 - 15.45):
Cosmo3D / OpenGL Optimizer in CAE applications
In this final part we will demonstrate the application of some of the previously described features in a particular visualization environment; a virtual car crash test-bed. In this context we will outline strategies for the efficient design of scene graphs allowing for the handling of large polygonal models. In our special example we will focus on finite element meshes resulting from numerical simulations. We will show the benefits of 1D and 2D textures for the mapping of parameter values to arbitrary geometries and for the localization and analysis of scalar parameter quantities. Furthermore, methods will be proposed for combined rendering of FE models and polygonal scenes. Although we focus on a particular example most of the described techniques can be directly used in other applications where the efficient handling and visualization of large scale polygonal models is the major concern.

Session S4:

High-quality shading and lighting

In this section, we explore the use of OpenGL features for high-quality shading and lighting effects. We start by an overview of several shadow algorithms, continue with some environment mapping techniques for visualizing global illumination effects, and conclude with bump-mapping techniques for contemporary graphics hardware.

Projected Geometry

Besides precomputed shadow textures, projected geometry [3] is currently the most often used implementation of shadows in interactive applications. This method assumes that a set of small occluders casts shadows onto a few large, flat objects. Since this assumption does not hold for most scenes, applications using this approach typically do not render all the shadows in a scene, but only some of the visually most important. In particular, shadows of concave objects onto themselves are typically not handled.

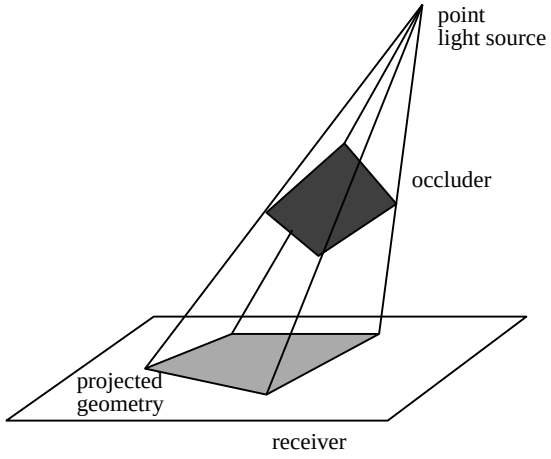


Figure 1: *One way of implementing shadows cast onto large, planar objects is to project the geometry onto the surface.*

The method works as follows. Given a point or directional light source, the shadowed region of a planar receiver can be computed by projecting the geometry of the occluder onto the receiver (see Figure 1). The projection is achieved by applying a specific model/view transformation to the original surface geometry. This transformation is given as follows:

$$S = V \cdot P \cdot M, \tag{1}$$

where V is the viewing transformation, M is the modeling transformation, and P is the projection matrix that would be used for rendering the scene from the light source position with the receiver as an image plane.

The occluder is then rendered twice, once with the regular model/view matrix $V \cdot M$ and the regular colors and textures, and once with the matrix S and the color of the shadow.

All kinds of variations on this basic algorithm are possible. For example, if parts of the projected geometry fall outside the receiver, a stencil buffer can be used to clamp it to the receiving polygon. Also, instead of simply drawing the shadow in black or dark gray, alpha blending can be used to modulate the brightness of the receiver in the shadow regions. Although this is not actually physically accurate, it looks more realistic, since the texture of the underlying surface remains visible.

As pointed out above, projected geometry is only appropriate for a small number of large, planar receiving objects. As the number of receivers grows, the method quickly becomes infeasible due to the large number of required rendering passes. Thus, this method can hardly be called a general solution to the shadowing problem; rather it simply adds shadows as a special effect in areas where they have the most visual impact.

Shadow Volumes

The second implementation of shadows are shadow volumes [9], which relies on a polyhedral representation of the spatial region that a given object shadows (see Figure 2).

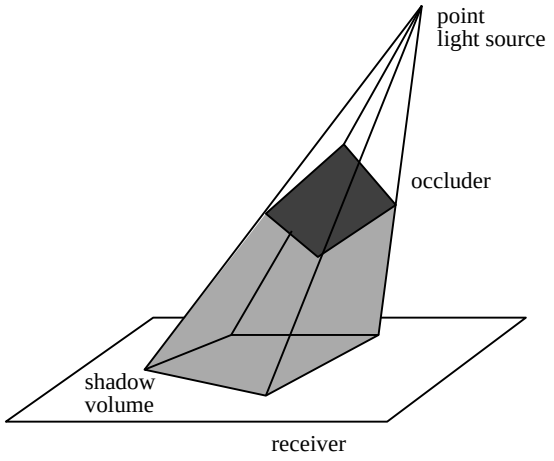


Figure 2: *The shadow volume approach uses a polyhedral representation of the spatial region shadowed by an occluder.*

This polyhedron is generated in a preprocessing step by projecting each silhouette edge of the object away from the light source. With graphics hardware and a z-buffer, the shadowing algorithm then works as follows [4, 14, 13]: First, the geometry is rendered without the contribution of the point

light casting the shadow, and the stencil buffer is cleared. Then, without clearing the *z*-buffer, the front-facing polygons of all shadow volumes are rendered without actually drawing to the color buffers. Each time a pixel of a shadow volume passes the depth test, the corresponding entry in the stencil buffer is incremented. Then, the back facing regions of the shadow volume are rendered in a similar fashion, but this time, the stencil buffer entry is decremented. Afterwards, the stencil buffer is zero for lit regions, and larger than zero for shadowed regions. A final conditional rendering pass adds the illumination for the lit regions.

Many details have to be solved to make this algorithm work in practice. For example, if the eye point lies inside a shadow volume, the meaning of the stencil bits is inverted. Even worse, if the near plane of the perspective projection penetrates one of the shadow volume boundaries, there are some areas on the screen where the meaning of the stencil bits is inverted, and some for which it is not.

In [14] and [13], this algorithm was used for rendering the complete set of shadows of a rather complex scene. However, the rendering times were far from interactive. Although today's graphics hardware is faster than the one used by Diefenbach, shadow volumes are typically still too costly to be applied to a complete scene. Another issue is the size of the data structures required for the shadow volumes, which can exceed several hundred megabytes [13]. The use of simplified geometry for generating the shadow volumes can help to reduce these problems. Nonetheless, most interactive applications and games only apply shadow volumes to a subset of the scene geometry, much in the same way projected geometry is used. Even then, the regeneration of the shadow volumes for moving light sources is costly.

Shadow Maps

In contrast to the analytic shadow volume approach, shadow maps [60] are a sampling-based method. First, the scene is rendered from the position of the light source, using a virtual image plane (see Figure 3). The depth image stored in the *z*-buffer is then used to test whether a point is in shadow or not.

To this end, each fragment as seen from the camera needs to be projected onto the depth image of the light source. If the distance of the fragment to the light source is equal to the depth stored for the respective pixel, then the fragment is lit. If the fragment is further away, is is in shadow.

A hardware multi-pass implementation of this principle has been proposed in [45]. The first step is the acquisition of the shadow map by rendering the scene from the light source position. For walkthroughs, this is a preprocessing step, for dynamic scenes it needs to be performed each frame. Then, for each frame, the scene is rendered without the illumination contribution from the light source. In a second rendering pass, the shadow map is specified as a projective texture, and

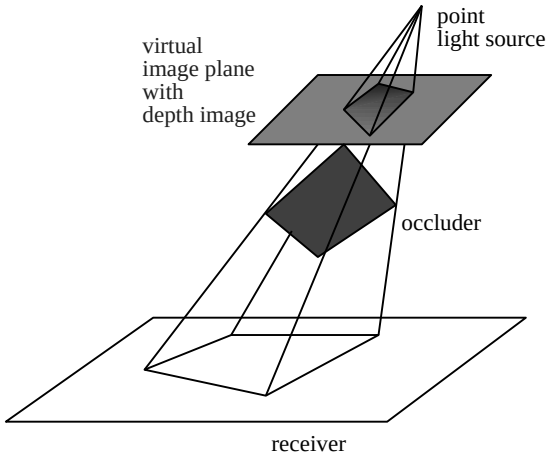


Figure 3: *Shadow maps use the z-buffer of an image of the scene rendered from the light source.*

a specific hardware extension is used to map each pixel into the local coordinate space of the light source and perform the depth comparison. Pixels passing this depth test are marked in the stencil buffer. Finally, as was the case for projected geometry, the illumination contribution of the light source is added to the lit regions by a third rendering pass.

The advantage of the shadow map algorithm is that it is a general method for computing all shadows in the scene, and that it is very fast, since the representation of the shadows is independent of the scene complexity. On the down side, there are artifacts due to the discrete sampling and the quantization of the depth. One benefit of the shadow map algorithm is that the rendering quality scales with the available hardware. The method could be implemented on fairly low end systems, but for high end systems a higher resolution or deeper *z*-buffer could be chosen, so that the quality increases with the available texture memory. Unfortunately, the necessary hardware extensions to perform the depth comparison on a per-fragment basis are currently only available on two high-end systems, the RealityEngine [2] and the InfiniteReality [41].

Shadow Maps Using the Alpha Test

Instead of relying on a dedicated shadow map extension, it is also possible to use projective textures and the alpha test. Basically, this method is similar to the method described in [45], but it efficiently takes advantage of automatic texture coordinate generation and the alpha test to generate shadow masks on a per-pixel basis. This method takes one rendering pass more than required with the appropriate hardware extension.

In contrast to traditional shadow maps, which use the contents of a *z*-buffer for the depth comparison, we use a depth

map with a *linear* mapping of the z values in light source coordinates. This allows us to compute the depth values via automatic texture coordinate generation instead of a per-pixel division. Moreover, this choice improves the quality of the depth comparison, because the depth range is sampled uniformly, while a z-buffer represents close points with higher accuracy than far points.

As before, the entire scene is rendered from the light source position in a first pass. Automatic texture coordinate generation is used to set the texture coordinate of each vertex to the depth as seen from the light source, and a 1-dimensional texture is used to define a linear mapping of this depth to alpha values. Since the alpha values are restricted to the range $[0 \dots 1]$, near and far planes have to be selected, whose depths are then mapped to alpha values 0 and 1, respectively. The result of this is an image in which the red, green, and blue channels have arbitrary values, but the alpha channel stores the depth information of the scene as seen from the light source. This image can later be used as a texture.

In a second rendering pass, the scene is now rendered from the camera point. Again, automatic texture coordinate generation and the 1-dimensional texture of the previous pass are applied. This results in an image as seen from the camera, but with light source depth values in the alpha channel of each pixel.

The third pass is used to render the scene from the camera position with the alpha-coded depth image from the first rendering pass specified as a projective texture. Alpha blending is set up in such a way that the alpha values from this pass are subtracted from the values in the frame buffer. Now, the frame buffer contains alpha values of 0 at each pixel that is lit by the light source, and > 0 for pixels that are shadowed. A final rendering pass of the geometry computes the illumination for the lit pixels.

Figure 4 shows an engine block where the shadow regions have been determined using this approach. Since the scene is rendered at least three times for every frame (four times if the light source or any of the objects move), the rendering times for this method strongly depend on the complexity of the visible geometry in every frame, but not at all on the complexity of the geometry casting the shadows. Scenes of moderate complexity can be rendered at high frame rates even on low end systems. The images in Figure 4 are actually the results of texture-based volume rendering using 3D texturing hardware (see [56] for the details of the illumination process).

If the hardware supports multiple textures, the second and third rendering passes can be combined into a single pass with two different textures.

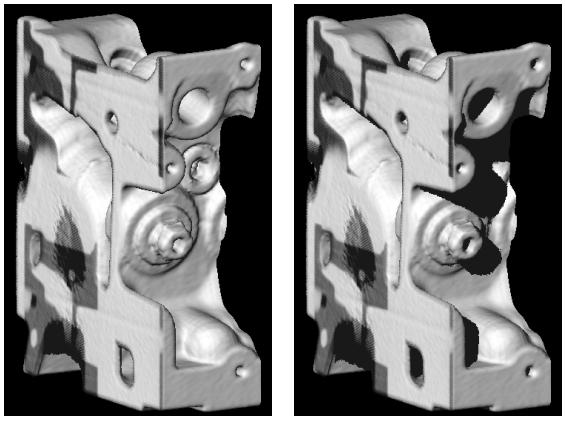


Figure 4: *An engine block generated from a volume data set with and without shadows. The shadows have been computed with our algorithm for alpha-coded shadow maps. The Phong reflection model is used for the unshadowed parts.*

Visualizing Global Illumination with Environment Maps

Environment maps are two-dimensional textures that represent the incoming illumination from all directions at one particular point in space. This information can be efficiently used to render reflections in objects, provided that they are relatively small and distant from the objects that reflect in them (the environment).

View-independent Environment Maps

The first step for using environment maps is the choice of an appropriate parameterization. The spherical parameterization [20] used in OpenGL today is based on the simple analogy of a infinitely small, perfectly mirroring ball centered around the object. The environment map is the image that an orthographic camera sees when looking at this ball along the negative z-axis. The sampling rate of this map is maximal for directions opposing the viewing direction (that is, objects behind the viewer), and goes towards zero for directions close to the viewing direction. Moreover, there is a singularity in the viewing direction, since all points where the viewing vector is tangential to the sphere show the same point of the environment.

With these properties, it is clear that this parameterization is not suitable for viewing directions other than the original one. Maps using this parameterization have to be regenerated for each change of the view point, even if the environment is otherwise static. The major reason why spherical maps are used anyway, is that the lookup can be computed efficiently with simple operations in hardware. In OpenGL, there is an automatic texture coordinate generation mode for this type of environment map.

We use a different parameterization that is both view in-

dependent and easy to implement on current and future hardware, and that we first described in [26] and [27]. A detailed description of its properties can be found in [25]. The parameterization is based on an analogy similar to the one used to describe spherical maps. Assume that the reflecting object lies in the origin, and that the viewing direction is along the negative z axis. The image seen by an orthographic camera when looking at the paraboloid

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2+y^2), \quad x^2+y^2 \leq 1 \tag{2}$$

contains the information about the hemisphere facing towards the viewer (see Figure 5). The complete environment is stored in two separate textures, each containing the information of one hemisphere.

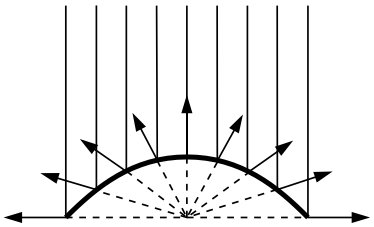


Figure 5: *The reflections off a paraboloid can be used to parameterize the incoming light from a hemisphere of directions.*

One useful property of this parameterization is a very uniform sampling of the hemisphere of directions. Another big advantage of this parameterization is that it can be used very efficiently in hardware implementations. As we have shown in [26], [25], and [27], the texture coordinates can be computed from the reflected viewing ray r_v in eye space via a perspective transformation:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = P \cdot S \cdot M^{-1} \cdot \begin{bmatrix} r_{v,x} \\ r_{v,y} \\ r_{v,z} \\ 1 \end{bmatrix}, \tag{3}$$

where M is a linear transformation mapping the environment map space into eye space. The environment map space is the space in which the environment is defined, that is, the one, in which the paraboloid is given by Equation 2. The inverse of M thus maps the r_v back into this space. Then the matrix S adds the vector $[0,0,1,0]^T$ to compute the halfway vector h, and finally P copies the z-component into the homogeneous component w to perform the perspective division.

In order to specify r_v as the initial set of texture coordinates, this vector has to be computed per vertex. This can be achieved either in software, or by a hardware extension allowing for the automatic computation of these texture coordinates, which we proposed in [26].

What remains to be done is to combine front facing and back facing regions of the environment into a single image. To this end, we mark those pixels inside the circle $x^2+y^2 \leq 1$ of one of the two maps with an alpha value of 1, the others with 0. The second map does not need to contain an alpha channel. Then, with either a single rendering pass and two texture maps, or two separate rendering passes, the object is rendered with the two different maps applied, and the alpha channel of the first map is used to select which map should be visible for each pixel.

Figure 6 shows the two images comprising an environment map in this parameterization, as well as two images rendered with these maps under different viewing directions. The environment maps were generated using a ray-caster. The marked circular regions contain the information for the two hemispheres of directions. The regions outside the circles are, strictly speaking, not part of the environment map, but are useful for avoiding seams between the two hemispheres of directions, as well as for generating mipmaps of the environment. These parts have been generated by extending the paraboloid from Equation 2 to the domain $[-1,1]^2$.

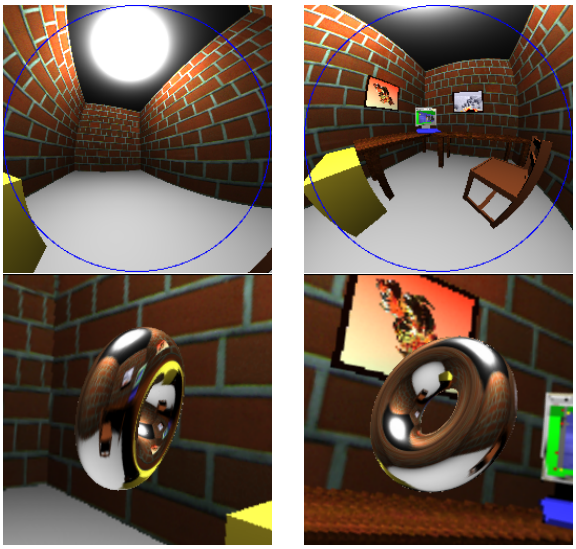


Figure 6: *Top: two parabolic images comprising one environment map. Bottom: rendering of a torus using this environment map.*

Mirror and Diffuse Terms with Environment Maps

Once an environment map is given in the parabolic parameterization, it can be used to add a mirror reflection term to an object. Due to the view-independent nature of this parameterization, one map suffices for all possible viewing positions and directions. Using multi-pass rendering and either alpha blending or an accumulation buffer [22], it is possible to add

a diffuse global illumination term through the use of a precomputed texture. Two methods exist for the generation of such a texture. One way is, that a global illumination algorithm such as Radiosity is used to compute the diffuse global illumination in every surface point.

The second approach is purely image-based, and was proposed by Greene [18]. The environment map used for the mirror term contains information about the incoming radiance $L_i(Px, l)$, where Px is the point for which the environment map is valid, and l the direction of the incoming light. This information can be used to prefilter the environment map to represent the diffuse reflection of an object for all possible surface normals. Like regular environment maps, this texture is only valid for one point in space, but can be used as an approximation for nearby points.

Fresnel Term

A regular environment map without prefiltering describes the incoming illumination in a point in space. If this information is directly used as the outgoing illumination, as with regular environment mapping, only metallic surfaces can be modeled. This is because for metallic surfaces (surfaces with a high index of refraction) the Fresnel term is almost one, independent of the angle between light direction and surface normal. Thus, for a perfectly smooth (i.e. mirroring) surface, incoming light is reflected in the mirror direction with a constant reflectance.

For non-metallic materials (materials with a small index of refraction), however, the reflectance strongly depends on the angle of the incoming light. Mirror reflections on these materials should be weighted by the Fresnel term for the angle between the normal and the viewing direction v.

The Fresnel term $F(\cos\theta)$ for the mirror direction r_v can be stored in a 1-dimensional texture map. This Fresnel term is rendered to the frame buffer's alpha channel in a separate rendering pass. The mirror part is then multiplied with this term in a second pass, and a third pass is used to add the diffuse part. This yields an outgoing radiance of $L_o = F \cdot L_m + L_d$, where L_m is the contribution of the mirror term, while L_d is the contribution due to diffuse reflections.

In addition to simply adding the diffuse part to the Fresnel-weighted mirror reflection, we can also use the Fresnel term for blending between diffuse and specular: $L_o = F \cdot L_m + (1-F)L_d$. This allows us to simulate diffuse surfaces with a transparent coating: the mirror term describes the reflection off the coating. Only light not reflected by the coating hits the underlying surface and is there reflected diffusely.

Figure 7 shows images generated using these two approaches. In the top row, the diffuse term is simply added to the Fresnel-weighted mirror term (the glossy reflection is zero). For a refractive index of 1.5 (left), which approximately corresponds to glass, the object is only specular for grazing viewing angles, while for a high index of refraction (200, right image), which is typical for metals, the whole object is highly specular.

The bottom row of Figure 7 shows two images generated with the second approach. For a low index of refraction, the specular term is again high only for grazing angles, but in contrast to the image above, the diffuse part fades out for these angles. For a high index of refraction, which, as pointed out above, corresponds to metal, the diffuse part is practically zero everywhere, so that the object is a perfect mirror for all directions.

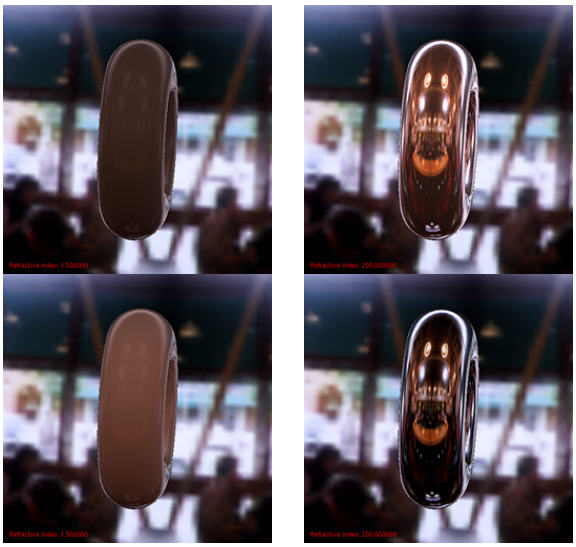


Figure 7: *Mirror and diffuse reflections weighted by a Fresnel term for a varying index of refraction. Top: constant diffuse coefficient, bottom: diffuse reflection fading out with the Fresnel term.*

Precomputed Glossy Reflection and Transmission

It is possible to extend the concept of environment maps from mirror reflections to glossy reflections [27]. The idea is similar to the diffuse prefiltering proposed by Greene [18] and the approach by Voorhies and Foran [52] to use environment maps to generate Phong highlights from directional light sources. These two ideas can be combined to precompute an environment map containing the glossy reflection of an object with a Phong material. With this concept, effects similar to the ones presented by Debevec [12] are possible in real time.

As shown in [36], the Phong BRDF is given by

$$f_r(l \to v) = k_s \cdot \frac{< r_l, v >^{1/r}}{\cos\alpha} = k_s \cdot \frac{< r_v, l >^{1/r}}{\cos\alpha}, \quad (4)$$

where r_l, and r_v are the reflected light- and viewing directions, respectively.

Thus, the specular global illumination using the Phong model is

$$L_o(r_v) = k_s \cdot \int_{\Omega(n)} < r_v, l >^{1/r} L_i(l)\, d\omega(l), \qquad (5)$$

which is only a function of the reflection vector r_v and the environment map containing the *incoming* radiance $L_i(l)$. Therefore, it is possible to take a map containing $L_i(l)$, and generate a filtered map containing the *outgoing* radiance for a glossy Phong material. Since this filtering is relatively expensive, it cannot be redone for every frame in an interactive application. Thus, it is important to use a view-independent parameterization such as the parabolic maps.

Figure 8 shows such a prefiltered glossy map for a Phong exponent of 100, as well as a glossy sphere textured with this map. The same technique can be applied to simulate glossy transmission on thin surfaces. This is also depicted in Figure 8.

Figure 8: *Top: original parabolic map used in this figure and Figure 7, as well as prefiltered map with a roughness of 0.01. Bottom: application of the filtered map to a reflective torus (left) and a transmissive rectangle (right).*

Normal Maps

Bump maps are becoming popular for hardware-accelerated rendering, because they allow us to increase the visual complexity of a scene without requiring excessive amounts of geometric detail.

Normal maps can be used for achieving the same goal, and have the advantage that the expensive operations (computing the local surface normal by transforming the bump into the local coordinate frame) have already been performed in a preprocessing stage. All that remains to be done is to use the precomputed normals for shading each pixel.

We first describe how normal maps can be lit according to the Blinn-Phong illumination model using the *imaging operations*, and afterwards, we discuss how the environment mapping techniques from Section can be used together with normal maps. This part relies on the *pixel texture* extension.

The methods described here assume an orthographic camera and directional light sources. The artifacts introduced by these assumptions are usually barely noticeable for surfaces with bump maps, because the additional detail hides much of them.

Normal Maps with local Blinn-Phong Illumination

Among many other features (see [46] for details), the imaging operations make it possible to apply a 4×4 matrix to each pixel in an image, as it is transferred to or from the frame buffer or to texture RAM. Following this color matrix transformation, a lookup table may be applied to each individual color component.

With these two mechanisms and a given normal map in the form of a color coded image, the map can be lit in two rendering passes. First, a color matrix is specified, which maps the normal from object space into eye space and then computes the diffuse illumination (which is essentially given by the dot product with the light direction). When the normal image is now loaded into texture RAM, the lighting computations are performed. Afterwards the loaded, lit texture is applied to the object using texture mapping.

A similar second rendering pass draws the specular part using the Blinn-Phong reflection model. This time, however, the matrix computes the dot product between normal and the halfway vector $h := (r_l + v)/|r_l + v|$ between reflected light direction $r_l := 2 < l, n > \cdot n - l$ and the viewing direction v. The exponentiation by $1/r$, where r is the surface roughness, is performed by a color lookup table.

Figure 9 shows two images where one polygon is rendered with this technique. On the left side, a simple exponential wave function is used as a normal map. The normal map for the image on the right side has been measured from a piece of wallpaper with the approach presented in [44].

Normal Maps with Environment Maps

The pixel texture extension can be used to apply environment maps to normal mapped surfaces. For orthographic cameras, the x and y components of the unit length normal at each surface point can be directly used as a texture coordinate for the spherical environment map at that pixel. On the other hand, the values n_x/n_z and n_y/n_z, are the texture coordinates for the parabolic environment map, so that both parameterizations can be used.

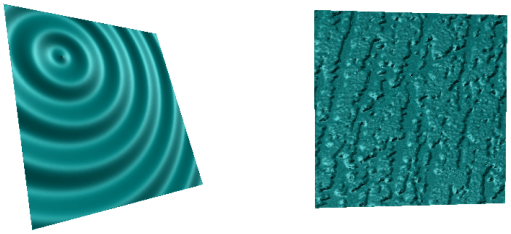


Figure 9: *Two Phong-lit normal maps. The right one has been measured from a piece of wallpaper using the approach presented in* [44].

The rendering then proceeds as follows: first, the object is rendered with the untransformed normal map as a texture. This yields an image containing the texture coordinates into the environment map at each pixel. Then, the frame buffer is copied onto itself, while the environment map is specified as a pixel texture. Special color matrices or lookup tables are not required for this step.

The images in Figure 10 were generated using the pixel texture extension and a single, view-dependent environment map.

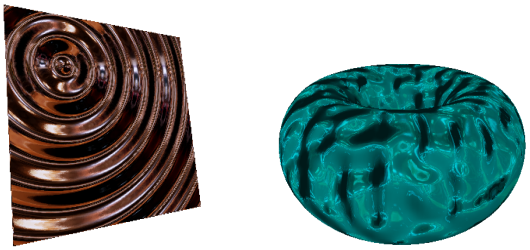


Figure 10: *Combination of environment mapping and normal mapping. Left: environment map only. Right: Local Phong illumination plus environment map.*

Session S5:

Advanced volume rendering techniques

As we have seen in the previous sessions, OpenGL and its extensions provide access to advanced per-pixel operations available in the rasterization stage and in the frame buffer hardware of modern graphics workstations. With these mechanisms, completely new rendering algorithms can be designed and implemented in a very particular way.

Over the past few years workstations with hardware support for the interactive rendering of complex 3D polygonal scenes consisting of directly lit and shaded triangles have become widely available. The last two generations of high-end graphics workstations [2, 41], however, besides providing impressive rates of geometry processing, also introduced new functionality in the rasterization and frame buffer hardware, like texture and environment mapping, fragment tests and manipulation as well as auxiliary buffers. The ability to exploit these features through OpenGL and its extensions allows completely new classes of rendering algorithms to be developed. Anticipating similar trends for the more advanced imaging functionality of todays high-end machines graphics researchers are actively investigating possibilities to accelerate expensive visualization algorithms by using these extensions.

In this session we will summarize various approaches that make extensive use of graphics hardware for the rendering of volumetric data sets. In particular, the goal of this session is to provide participants with dedicated knowledge concerning the application of 3D textures in volume rendering applications and to demonstrate how to exploit the processing power and functionality of the rasterization and texture subsystem of advanced graphics hardware. Although at this time hardware accelerated 3D texture mapping is only supported on a few particular architectures we expect the same functionality to be available on low-end architectures like PCs in the near future thus leading to an increasing need for hardware accelerated algorithms as will be presented.

Hereafter we will first describe the basic concepts of volume rendering via 3D textures thereby focusing on the potential benefits and advantages compared to software based solution. We will further outline extensions that enable flexible and interactive editing and manipulation of large scale volume data. We will introduce the concept of clipping geometries by means of stencil buffer operations and polygon tesselation, and we will review the use of 3D textures for the rendering of lighted and shaded iso-surfaces in real-time without extracting any polygonal representation. Additionally, we will describe novel approaches for the visualization of scalar volume data defined on unstructured grids, where only software solutions exist up to now. The intention here is to streamline new directions for the visualization of unstructured grids without explicit topological information. In particular, the sketched approaches continuously lead to resampling strategies for arbitrary grid structures. In this context we will focus on iso-surface extraction and direct volume rendering techniques, which can be accelerated to new rates of interactivity by simple polygon drawing and frame buffer operations.

Our major concern in this session is to outline techniques for the efficient generation of a visual representation of the information present in volumetric data sets. For scalar-valued volume data two standard techniques, the rendering of iso-surfaces, and the direct volume rendering, have been developed to a high degree of sophistication. However, due to the huge number of volume cells which have to be pro-

cessed and to the variety of different cell types only a few approaches allow parameter modifications and navigation at interactive rates for realistically sized data sets. To overcome these limitations a basis for hardware accelerated interactive visualization of both iso-surfaces and direct volume rendering on arbitrary topologies has been provided in [56].

Direct volume rendering tries to convey a visual impression of the complete 3D data set by taking into account the emission and absorption effects as seen by an outside viewer. The underlying theory of the physics of light transport is simplified to the well known volume rendering integral when scattering and frequency effects are neglected [28, 30, 39, 62]. A few standard algorithms exist for computing the intensity contribution along a ray of sight, enhanced by a wide variety of optimization strategies [35, 39, 34, 11, 33]. But only recently, since hardware supported 3D texture mapping is available, has direct volume rendering become interactively feasible on graphics workstations [5, 10, 63]. This approach has been extended further on with respect to flexible editing options and advanced mapping and rendering techniques.

The major goal is the visualization and manipulation of volumetric data sets of arbitrary data type and grid topology at interactive rates within one application on standard graphics workstations. In this session we focus on scalar-valued volumes and show how to accelerate the rendering process by exploiting features of advanced graphics hardware implementations through standard APIs like OpenGL. The presented approach is pixel oriented, takes advantage of rasterization functionality such as color interpolation, texture mapping, color manipulation in the pixel transfer path, various fragment and stencil tests, and blending operations. In this way it is possible to

- **extend volume rendering via 3D textures** with respect to arbitrary clipping geometries and to improve performance by adaptive rasterization
- **render shaded iso-surfaces** at interactive rates combining 3D textures and fragment operations thus avoiding any polygonal representation
- **accelerate volume visualization of tetrahedral grids** employing polygon rendering of cell faces and fragment operations for both shaded iso-surfaces and direct volume rendering.

Volume rendering via 3D textures

When 3D textures became available on graphics workstations their benefit in volume rendering applications was soon recognized [10, 6]. The basic idea is to interpret the 3D scalar voxel array as a 3D texture defined over $[0,1]^3$ and to understand 3D texture mapping as the trilinear interpolation of the volume data set at an arbitrary point within this domain. The data is re-sampled on clipping planes that are oriented orthogonal to the viewing plane with the plane pixels trilinearly interpolated from the 3D scalar texture. This operation is successively performed for multiple planes that have to be clipped against the parametric texture domain (see Figure 11). These polygons are rendered from front-to-back or back-to-front and the resulting texture slices are blended appropriately into the frame buffer thereby approximating the continuous volume rendering integral.

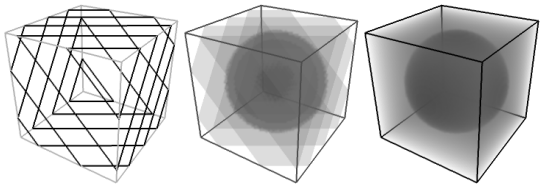


Figure 11: *Volume rendering by 3D texture slicing.*

Dedicated graphics hardware is exploited for trilinearly interpolating within the texture and for blending the generated fragments on a per-pixel basis. However, the real potential of volume rendering via 3D textures just turned out after texture lookup tables became available. Scalar samples that are reconstructed from the 3D texture are converted into RGAα pixels by a lookup-up table prior to their drawing. The possibility to directly manipulate the transfer functions necessary to perform the mapping from scalar values to RGBα values without the need for reloading the entire texture allows the user to interactively find meaningful mappings of material values to visual quantities. In this way arbitrary parts of the data can be highlighted or suppressed and visualized using different colors and transparencies.

Nevertheless, besides interactive frame rates, in many practical applications editing the data in a free and easy way is of particular interest. Although texture lookup tables can be modified in order to extract portions of the data, the use of additional clipping geometries often allows separating the relevant structures in a much more convenient and intuitive way. Planar clipping planes available as core OpenGL mechanisms may be utilized, but from the user's point of view more complex geometries are necessary.

Clipping geometries

Stenciling

A straightforward approach which is implemented quite often is the use of multiple clipping planes to construct more complex geometries. However, notice that even the simple task of clipping an arbitrarily scaled box cannot be realized in this way. More flexibility and ease of manipulation can be achieved by taking advantage of the per-pixel operations provided in the rasterization stage. As will be outline, as long as the object against which the volume is to be clipped is a closed surface represented by a list of triangles it can be efficiently used as the clipping geometry.

The basic idea is to determine for all slicing planes those pixels which are covered by the cross-section between the object and this plane (see Figure 12). Then, these pixels are locked, thus preventing the textured polygon from getting drawn to these locations. The locking mechanism is implemented by exploiting the OpenGL stencil test. It allows pixel updates to be accepted or rejected based on the outcome of a comparison between a user defined reference value and the value of the corresponding entry in the stencil buffer. Before the textured polygon gets rendered the stencil buffer has to be initialized in such a way that all color values written to pixels inside the cross-section will be rejected.

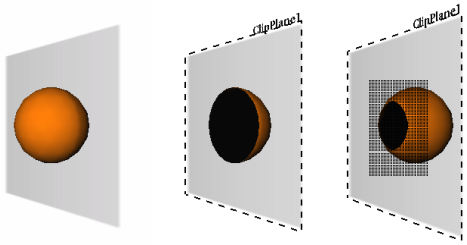


Figure 12: *The use of arbitrary clipping geometries is demonstrated for the case of a sphere. In regions where the object intersects the actual slice the stencil buffer is locked. The intuitive approach of rendering only the back faces might result in the patterned erroneous region.*

In order to determine for a certain plane whether a pixel is covered by a cross-section or not the clipping object is rendered in polygon mode. However, since one is only interested in setting the stencil buffer none of the frame buffer values altered. At first, an additional clipping plane is enabled which has the same orientation and position as the slicing plane. All back faces with respect to the actual viewing direction are drawn, and everything in front of the plane is clipped. Wherever a pixel would have been drawn the stencil buffer is set. Finally, by changing the stencil test appropriately, rendering the textured polygon, now, only affects those pixels where the stencil buffer is unchanged.

In general, however, depending on the clipping geometry this procedure fails in determining the cross-section exactly (see rightmost image in Figure 12). Therefore, before the textured polygon is rendered all stencil buffer entries which are set improperly have to be updated. Notice that in front of a back face which was written erroneously there is always a front face due to the topology of the clipping object. The front faces are thus rendered into those pixels where the stencil buffer is set and the stencil buffer is cleared where a pixel also passes the depth test. Now the stencil buffer is correctly initialized and all further drawing operations are restricted to those pixels where it is set or vice versa. Clearing the stencil buffer each time a new slice is to be rendered can be avoided by using different stencil planes. Then the number of slices that can be processed without clearing the buffer depends on the number of stencil bits provided by the current visual.

Since this approach is independent of the used geometry it allows arbitrary shapes to be specified. In particular it turns out that transformations of the geometry can be handled without any additional overhead, thus providing a flexible tool for carving portions out of the data in an intuitive way.

Reducing Rasterization

Although the stencil buffer approach works quite optimal as long as the rendering of the triangle mesh does not affect or even dominate the overall performance, there is no gain in general concerning the number of rasterization operations that have to be performed. Even if only a small part of the data is to be rendered the number of texture lookups remains the same since texturing is performed prior to the stencil test.

A different alternative that was originally developed to render thin boundary regions [57] circumvents this drawback by balancing the load between the rasterization unit and the processor more equally. The slicing planes are explicitly clipped against the clipping geometry, which results in the generation of a planar contour within each slice.

In general, as long as the clipping geometry is simple and does not consist of a large number of triangles, the computation of the sectional contour can be done very efficiently. At the core of the intersection algorithm the triangle mesh is stored in an edge based data structure. Each edge has references to the triangles that share this edge as well as to the points that define this edge. Based on the distance of each point to the viewing plane all edges can now be sorted accordingly. For each slice an edge has to be determined that intersects this slice, which can then be used to find all other edges that intersect this slice by recursively following the appropriate links.

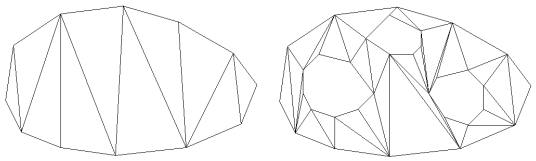
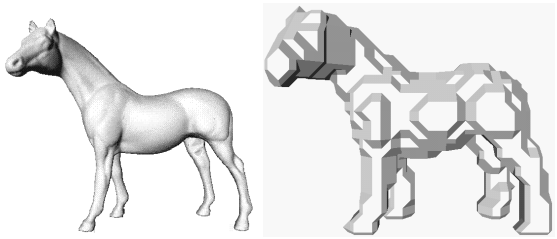


Figure 13: *On the left a simple contour has been tesselated using the OpenGL tesselation utilities. On the right a more complex tesselation is shown consisting of the original contour with three holes.*

Once the sectional polygon between the slicing plane and the clipping geometry has been calculated it needs to be tesselated in turn (see Figure 13). In an OpenGL based setting it just seems to be self-evident to utilize the OpenGL tessellation utilities for computing trapezoidal decompositions of concave polygons. As a result, a number of triangles that

completely cover the sectional region is obtained and can be rendered with the appropriate texture coordinates issued at each vertex. Note that affine transformations of the clipping object within the texture domain can be handled by issuing the current transformation in the texture matrix used to transform texture coordinates before texturing is performed.

In this way the load in the rasterization unit can be considerably reduced at the expense of additional numerical operations necessary to compute and tesselate the sectional contours. In Figure 14a, for example, a triangle mesh portraying a horse was first voxelized and then rendered with a method described below. In Figure 14b the clipping geometry is illustrated that was used to restrict rasterization to the relevant parts. Overall, the number of rasterization operations has been reduced of about 50% resulting in a gain in performance of almost the same amount since the tessellation of the sectional contours was done in approximately 0.07 seconds.



(a) The voxelized horse data set.

(b) The clipping geometry used to visualize the horse.

Figure 14: *Reducing rasterization by adaptive clipping.*

In Figure 15 two images are shown, which should demonstrate the extended functionality of 3D texture based volume rendering. In the first image a simple box was used to mask the interior of a MRI-scan by means of the stencil buffer approach. The second images was generated by explicitly clipping the slicing planes against the box and by tesselating the resulting contours. Note that only the region of interest needs to be textured in this way.

Rendering iso-surfaces via 3D textures

So far we described extensions to texture mapped direct volume rendering that have been introduced in order to define a general hardware accelerated framework for adaptive exploration of volumetric data sets. In practice, however, the display of shaded iso-surfaces has been shown as one of the most dominant visualization options, which is particularly useful to enhance the spatial relationship between structures. Moreover, this kind of representation often meets the physical characteristics of the real object in a more natural way.

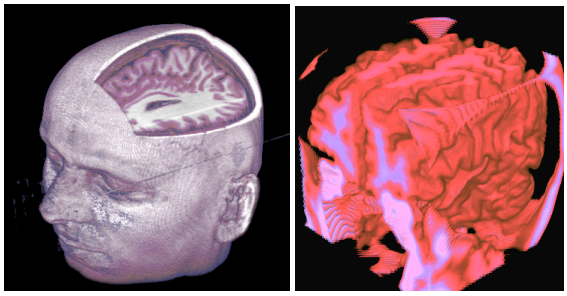


Figure 15: *Box clipping using the stencil test (left) and the OGL tesselation (right).*

Different algorithms have been proposed for efficiently reconstructing polygonal representations of iso-surfaces from scalar volume data [38, 40, 47, 58], but none of these approaches can effectively be used in interactive applications. This is due to the effort that has to be made to fit the surface and also to the enormous amount of triangles produced. For realistically sized data sets interactively manipulating the iso-value seems to be quite impossible, and also rendering the surface at acceptable frame rates can hardly be achieved. In contrast to these polygonal approaches, in [55] an algorithm was designed that completely avoids any polygonal representation by combining 3D texture mapping and advanced pixel transfer operations in a way that allows the iso-surface to be rendered on a per-pixel basis.

Recently, first approaches for combining hardware accelerated volume rendering via 3D texture maps with lighting and shading were presented. In [51] the sum of pre-computed ambient and reflected light components is stored in the texture volume and standard 3D texture composition is performed. On the contrary, in [23] the orientation of voxel gradients is stored together with the volume density as the 3D texture map. Lighting is achieved by indexing into an appropriately replicated color table. The inherent drawbacks to these techniques is the need for reloading the texture memory each time any of the lighting parameters change (including changes in the orientation of the object) [51], and the difficulty to achieve smoothly shaded surfaces due to the limited quantization of the normal orientation and the intrinsic hardware interpolation problems [23].

Basically, the non-polygonal 3D texture based approach is similar to the one used in traditional volume ray-casting for the display of shaded iso-surfaces. Let us consider that the surface is hit if the material values along the ray of sight do exceed the iso-value for the first time. At this location the material gradient is computed, which is then used in the lighting calculations.

By recognizing that texture interpolation is already exploited to re-sample the data, all that needs to be evaluated is how to capture those texture samples above the iso-value

that are nearest to the image plane. Therefore the OpenGL **alpha test** can be employed, which is used to reject pixels based on the outcome of a comparison between their alpha component and a reference value.

Each element of the 3D texture gets assigned the material value as its alpha component. Then, texture mapped volume rendering is performed as usual, but pixel values are only drawn if they pass the depth test and if the alpha value is larger than or equal to the selected iso-value. In any of the affected pixels in the frame buffer, now, the color present at the first surface point is being displayed.

In order to obtain the shaded iso-surface from the pixel values already drawn into the frame buffer two different approaches should be outlined:

- **Gradient shading**: A four component 3D texture is stored which holds in each element the material gradient as well as the material value. Shading is performed in image space by means of matrix multiplication using an appropriately initialized color matrix.
- **Gradientless shading**: Shading is simulated by simple frame buffer arithmetic computing forward differences with respect to the light source direction. Pixel texturing is exploited to encompass multiple rendering passes.

Both approaches account for diffuse shading with respect to a parallel light source positioned at infinity. Then the diffuse term reduces to the scalar product between the surface normal, **N**, and the direction of the light source, **L**, scaled by the material diffuse reflectivity, k_d.

The texture elements in gradient shading each consist of an RGBα quadruple which holds the gradient components in the color channels and the material value in the alpha channel. Before the texture is stored and internally clamped to the range [0,1] the gradient components are being scaled and translated by a factor of 0.5.

By slicing the texture thereby exploiting the alpha test as described the transformed gradients at the surface points are finally displayed in the RGB frame buffer components (see left image in Figure 16). For the surface shading to proceed properly, pixel values have to be scaled and translated back to the range [-1,1]. In order to account for changes in the orientation of the object the normal vectors have to be transformed by the model rotation matrix. Finally, the diffuse shading term is calculated by computing the scalar product between the light source direction and the transformed normals.

All three transformations can be applied simultaneously using one 4x4 matrix. It is stored in the currently selected color matrix which post-multiplies each of the four-component pixel values if pixel data is copied within the active frame buffer. For the color matrix to accomplish the transformations it has to be initialized as follows:

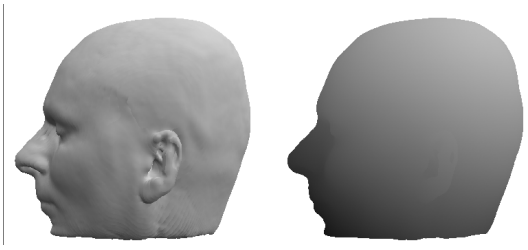


Figure 16: *On the left, for an iso-surface the gradient components are displayed in the RGB pixel values. On the right, for the same iso-surface the coordinates in texture space are displayed in the RGB components.*

$$CM = \begin{pmatrix} L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} M_{rot} \begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By just copying the frame buffer contents onto itself each pixel gets multiplied by the color matrix. In addition, it is scaled and biased in order to account for the material diffuse reflectivity and the ambient term. The resulting pixel values are

$$\begin{bmatrix} I_a \\ I_a \\ I_a \\ 0 \end{bmatrix} + \begin{bmatrix} k_d \\ k_d \\ k_d \\ 1 \end{bmatrix} CM \begin{bmatrix} R \\ G \\ B \\ \alpha \end{bmatrix} = \begin{bmatrix} k_d \langle L, N_{rot} \rangle + I_a \\ k_d \langle L, N_{rot} \rangle + I_a \\ k_d \langle L, N_{rot} \rangle + I_a \\ \alpha \end{bmatrix}$$

where obviously different ambient terms and reflectivities can be specified for each color component.

Figure 17 illustrates the quality of the described rendering technique for shaded iso-surfaces. The surface on the left image was rendered in roughly 9 seconds using a software based ray-caster. 3D texture based gradient shading was run with about 6 frames per second on the next image. The distance between successive slices was chosen to be equal to the sampling intervals used in the software approach. The surface on the right appears somewhat brighter with a little less contrast due to the limited frame buffer precision, but basically there can hardly be seen any differences.

To circumvent the additional amount of memory that is needed to store the gradient texture a second technique can be employed which applies concepts borrowed from [42] but in an essentially different scenario. The diffuse shading term can be simulated by simple frame buffer arithmetic if the surface is assumed to be locally orthogonal to the surface normal and the normal as well as the light source direction are orthonormal vectors.

Notice that the diffuse shading term is then proportional to the directional derivative towards the light source. Thus, it can be simulated by taking forward differences toward the

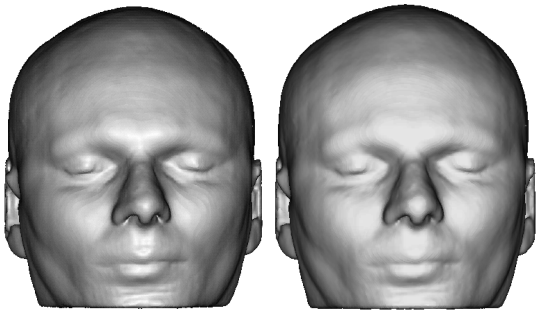


Figure 17: *Iso-surface rendering by direct ray-casting (left) and by using a gradient texture (right).*

light source with respect to the material values:

$$I_d \approx \frac{\partial X}{\partial L} = X(\vec{p}_0) - X(\vec{p}_0 + \triangle \cdot \vec{L})$$

By rendering the scalar material values twice, once those that correspond to the original surface points and then those that correspond to the surface points shifted towards the light source, OpenGL blending operations can be exploited to compute the forward differences.

In order to obtain the coordinates of the surface points it is taken advantage of the alpha test as proposed and pixel textures are applied to re-sample the material values. Therefore it is important to know that each vertex comes with a texture coordinate as well as a color value. Usually the color values provide a base color and opacity in order to modulate the interpolated texture samples.

By considering that to each vertex the computed texture coordinate (u,v,w) is assigned as RGB color value. Texture coordinates are supposed to be within the range [0,1] since they are computed in parametric texture space. Moreover, the color values interpolated during rasterization correspond to the texture space coordinates of points on the slicing plane. As a consequence we now have the position of surface points available in the frame buffer rather than the material gradients.

In order to display the correct color values they must not be modulated by the texture samples. However, remember that in gradientless shading the same texture format is used as in traditional texture slicing. Each element comprises a single-valued color entry which is mapped via a RGBα lookup table. This allows one to temporarily set all RGB values in the lookup table to one thus avoiding any modulation of color values.

At this point, the real strength of pixel textures can be exploited. The RGB entries of the texture lookup table are reset in order to produce the original scalar values. Then, the pixel data is read into main memory and it is drawn twice into the frame buffer with enabled pixel texture. In the second pass pixel values are shifted towards the light source by means of the OpenGL pixel bias. By changing the blending equation appropriately all values get subtracted from those already in the frame buffer thus yielding the approximated diffuse lighting.

In Figure 18 illustrates the difference between gradient shading and gradientless shading. Obviously, surfaces rendered by the latter one exhibit low contrast and even incorrect results are produced especially in regions where the variation of the gradient magnitude across the surface is high. Although the material distribution in the example data is almost iso-metric, at some points the differences can be easily recognized. At these surface points the step size used to compute the forward difference has to be increased, which, of course, can not be realized by the presented approach.

However, only one fourth of the memory needed in gradient shading is used in gradientless shading, and also the rendering times differ insignificantly. The only difference lies in the way the shading is finally computed. In gradient shading the whole frame buffer is copied once. In gradientless shading the pixel data has to be read and written twice with enabled pixel texturing. On the other hand, since the overhead does not depend on the data resolution but on the size of the viewport, its relative contribution to the overall rendering time can be expected to decrease rapidly with increasing data size.

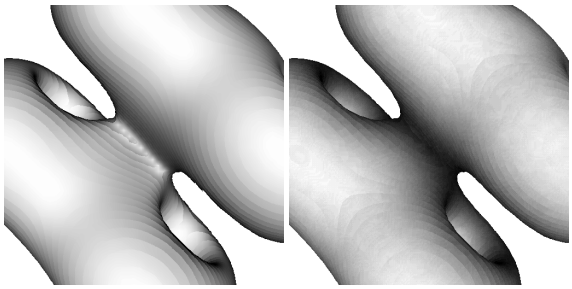


Figure 18: *Comparison of iso-surface rendering using a gradient texture (left) and frame buffer arithmetic (right).*

Volume rendering of unstructured grids

Now we turn our attention to tetrahedral grids most familiar in CFD simulation, which, on the other hand, have also recently shown their importance in adaptive refinement strategies. Since most grid types that provide the data at unevenly spaced sample points can be quite easily converted into this kind of representation, tetrahedra based techniques, in general, are potentially attractive to a wide area of different applications.

Caused by the irregular topology of the grids to be processed the intrinsic problem showing up in direct volume

rendering is to find the correct visibility ordering of the involved primitives. Different ways have been proposed to attack this problem, e.g. by improving sorting algorithms [50, 61, 16], by using space partitioning strategies [59], by taking advantage of hardware assisted polygon rendering [48, 50, 64, 54] and by exploiting the coherence within cutting planes in object space [17, 49]. In [55] a hardware accelerated approach that entirely avoids the sorting of elements was introduced for both direct rendering and the display of iso-surfaces from unstructured volume data. Before we start with a detailed explanation of this approach let us first make same general considerations.

In its interior, each tetrahedron (hereafter termed the volume primitive or cell) exhibits a linear range in the material distribution and therefore a constant gradient. The affine interpolation function $f(x,y,z) = a + bx + cy + dz$ which defines the material distribution within one cell is computed by solving the system of equations

$$\begin{pmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \end{pmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

for the unknowns a, b, c and d. f_i are the function values given at locations (x_i, y_i, z_i).

Now the partial derivatives, b, c and d, provide the gradient components of each cell. Gradients at the vertices are computed by simply averaging all contributions from different cells. These are stored in addition to the vertex coordinates and the scalar material values, the latter ones given as one-component color indices into a RGBα lookup table.

Shaded iso-surfaces

In contrast to volume data defined on Cartesian grids the unstructured data can no longer be stored in a regular 3D texture. However, geometry processing and advanced per-pixel operations can be exploited in a highly efficient way in order to avoid any polygonal representation.

At first, let us consider a ray of sight passing through one tetrahedron thereby re-sampling the material values. Since along the ray the material distribution is linear it suffices to evaluate the data within the appropriate front and back face and to linearly interpolate in between. This, again, can be solved quite efficiently using the graphics hardware. Therefore the material values are issued as the color of each vertex before the smoothly shaded cell faces are rendered. The correctly interpolated samples are then being displayed and can be grabbed from the frame buffer.

Obviously, the same procedure can be applied by choosing an appropriate shading model and by issuing the material gradient as the vertex normal. Then the rendered triangles will be illuminated with respect to the gradients of the volume material.

Nevertheless, since a specific iso-surface showed be rendered those elements have to be find the surface is passing through. But even more difficult, the exact location of the surface within these cells has to be determined in order to compute appropriate interpolation weights that are needed to combine the contributions of the front and back faces, respectively.

The key idea lies in a multi-pass approach:

a: **Faces are rendered having smooth color interpolation in order to compute the interpolation weights.**
b: **Faces are rendered having smooth shading in order to compute illuminated pixels.**
c: **The interpolation weights are used to modulate the results properly.**

In order to compute the interpolation weights the material values given at the vertices are duplicated into RGBα-quadruples. These are used as the current color values. Next, all the back faces are rendered, but only those pixels nearest to the image plane with an alpha value larger than the threshold are maintained by exploiting the alpha test and the depth test. The stencil buffer is set whenever a pixel passes both tests.

Now the alpha test is inverted and the front faces are drawn. Although depth testing will be performed, *glDepthMask(GL_FALSE)* is issued in order to prevent z-values from being altered. Pixel values may be affected where the stencil buffer is set, but in fact, not all color components will be altered in order to retain the previously written results.

Instead of processing all front faces at once one alternatively render each element's back faces again. By setting their alpha values to zero it is guaranteed that they always pass the alpha test. Notice that if pixels were accepted in the first pass the corresponding z-values are still present since the z-values have not been altered. Choosing an appropriate stencil function allows the stencil buffer to be locked whenever a pixel is written with a z-value equal to the stored one. At these locations the frame buffer is locked in order to prevent the correctly drawn pixels from being destroyed. Finally, the pixel data is read into main memory and the interpolation weights are computed and stored into two distinct pixel images I_f and I_b, respectively.

Once again, the entire procedure is repeated, but now the hardware is exploited to render illuminated faces instead of colored ones. The results of the first rendering pass are blended with the pixel image I_f and the modulated pixel data is transfered to the accumulation buffer [22]. Pixel data produced in the second pass is blended with the pixel image I_b and added to the data already stored in the accumulation buffer. During both passes blending ensures that the shaded faces are interpolated correctly in order to produce the sur-

face shading. Finally, the entire image is drawn back into the frame buffer.

However, even without computing the interpolation weights, just by equally blending the lighted front and back faces, this method produces sufficient results (see Figure 19). It is quite evident that sometimes the geometric structure of the cells shows up, but this seems to be tolerable during interactive sessions.

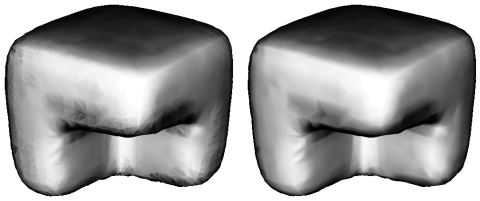


Figure 19: *Iso-surface reconstructed from a tetrahedral grid. Color values of the image were equalized to enhance the effects. The left image was generated without using the interpolation weights necessary to achieve smooth results.*

Directly slicing unstructured grids

It is now easy to derive an algorithm that allows one to reconstruct arbitrary slices out of the data. For each vertex its distance to the image plane is stored in addition and it is temporarily used as the scalar material value. But then, a slice just corresponds to a planar iso-surface defined by an iso-value that is equal to the distance of that slice to the image plane. As a consequence the proposed method for reconstructing shaded iso-surfaces can be applied directly. Even more efficiently, since one is interested in the scalar material values only smooth color interpolation across cell faces has to be issued.

For the method to proceed properly, the scalar values are stored in the RG color components and the distance values in the Bα components issued at each vertex. Again, back faces are rendered first. For a slice at the distance d from the image plane only pixel values with an alpha value larger than or equal to d are accepted. Only RB components will be altered in the frame buffer. As usual, the stencil buffer is set where a pixel passes the depth test and the alpha test. Now the front faces are rendered but only the Gα components are going to be altered. Locking the stencil buffer is done in the same way as described.

Finally, all values necessary to correctly interpolate the scalar values within the actual slice are available in the pixel data. These are read and

$$S = (1 - \frac{d-\alpha}{B-\alpha}) \cdot G + \frac{d-\alpha}{B-\alpha} \cdot R,$$

is calculated for each RGBα pixel value. The scalar values are now written back into the frame buffer thereby mapping each component via a lookup table provided by the graphics hardware. In order to approximate the volume rendering integral the grid is sliced multiple times and the generated images are blended properly.

The remarkable fact is that the topology of the underlying grid never needs to be used explicitly. Sorting is implicitly done by depth testing on a per-pixel basis in the rasterization unit. Polygon drawing is exploited for interpolating scalar values on the cell faces and all values necessary to interpolate within one slice are accessed from the frame buffer. Moreover, adaptive slicing with arbitrary resolution can easily be performed, leading to a powerful method that allows for the re-sampling of arbitrary regions in the desirable resolution.

Obviously, in order to avoid rendering every cell for each slice, a coarse partitioning of the data set according to the distance of cells to the viewing plane has to be performed. Whenever the viewing definition changes, cells are insert into a certain number of slabs parallel to the viewing plane. During rendering only those cells have to be touched which are stored in the slab that contains the current slice. In this way, the overall number of cells to be rendered will still be considerably larger than the original number, but on the other hand, depending on the number of slabs this increase can be controlled arbitrarily.

Figure 20 show some examples of tetrahedral grids that were rendered using the presented approaches. The first image shows an iso-surface from the NASA bluntfin which was converted into 225000 tetrahedra. Direct volume rendering of a finite-element data set is demonstrated by the second example. Notice the adaptive manipulation of the transfer function in order to indicate increasing temperature from blue to yellow. The glowing inner kernel can be clearly distinguished, which might be hard to achieve with cell based projection techniques like the Shirley-Tuchman algorithm [48].

The significant improvement of the described approach is that the expected times do not depend on the grid topology. As a consequence one ends up with constant frame rates for arbitrary topologies but equal number of primitives. This is a major difference to a variety of existing approaches which exploit the connectivity between cells. Then, the grid can be traversed very efficiently by taking advantage of precomputed adjacency information. Finally, we expect the proposed method to be of great relevance in applications where the data is updated frequently, e.g. in numerical simulations. As long as the complex sorting of each new time-step has to be performed there seems to be no chance to run the visualization and the simulation simultaneously. The presented algorithm, on the other hand, entirely avoids any complex pre-processing steps.

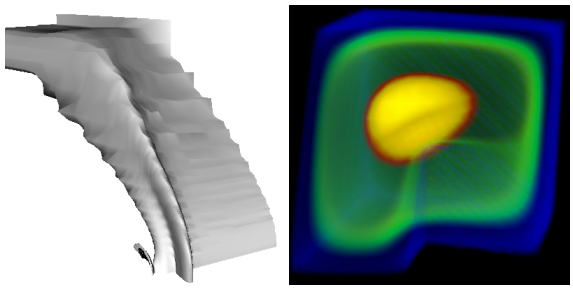


Figure 20: *Two tetrahedral grids that were visualized using the hardware-accelerated methods as described.*

Session S6:

Cosmo3D / OpenGL Optimizer in CAE applications

This session should illustrate the use of ways and means provided by Cosmo3D / OpenGL Optimizer by presenting some visualization examples in the field of crash simulation.

Finite element post-processing has been dominated by software that is tightly integrated with simulation packages. Many of these packages have not kept up with the state-of-the-art developments in graphics technology and visualization techniques. Especially the large and time-dependent data sets resulting from crash-worthiness simulations in the automotive development process demand for new visualization tools which allow interactive manipulation of complex geometries and meaningful mapping of physical properties. In this session we demonstrate that careful design of scene graph structures and extensive use of texture mapping can improve rendering performance and visual appearance for post-processing tasks such as inspecting finite element discretization and analyzing intrusion depth or vector quantities. Furthermore, we describe an iconic visualization method as first presented in [31] and extended in [32] which improves the understanding of cross-section forces and bending moments in longitudinal structures of the car body.

One of the main goals in the development of a new car is the achievement of an optimal ”crash-worthiness” using as many analytical tools as possible and minimizing hardware-prototype testing. During the last few years, the absolute simulation time for modeling, computing and investigating a complete crash model has been reduced significantly. However, we notice a shift of the proportions between the time required for pre-processing, computation and post-processing respectively. The post-processing stage turned out to become the most time consuming activity performed by the simulation engineers. These changes and the rapid development of computer graphics technology during the last few years has increased the need for new visualization techniques to facilitate the analysis of crash-worthiness simulations.

Considering the progress of scientific visualization in various areas during the last decade, it becomes obvious that the application of 3D visualization techniques to finite element analysis has not been a primary focus [15, 29, 66]. Nevertheless, the use of commercial visualization packages is now well established in the automotive industry. In the case of crash analysis, these traditionally employed post-processors have been designed to manage the enormous amount of simulation data on workstations with limited memory by performing animations of wire-frame meshes and polygonal representations of the simulated crash models. However, associated with these design criteria and with wide platform availability is a trade-off which leads to poor graphics performance in terms of available frame rates on high-end graphics subsystems.

Memory efficient scene graph design

Several graphics APIs, such as IRIS Performer or OpenGL Optimizer, have been developed to take advantage of recent progress in compute server and workstation architecture with multiprocessing hardware in mind. Since those APIs are usually scene graph based, one can take advantage of model optimization during scene graph creation and benefit from multiprocessing using view frustum culling and occlusion culling while traversing the scene graph to increase frame and interaction rates (see Figure 21). Since the time-dependent databases of our FE models are very bulky, an efficient scene graph design is very important in order to handle the complex data interdependencies and to achieve high rendering speed.

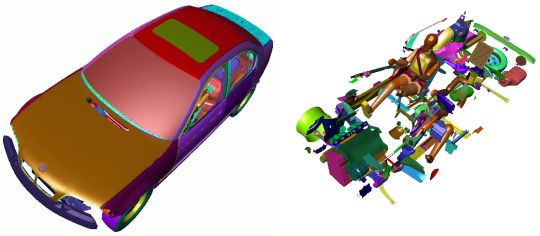


Figure 21: *On the left we see the visible geometry. All the displayed parts on the right are detected by the CPU as occluded – all these triangles will not be sent into the graphics pipeline.*

The goal is to visualize meshes of about 300,000 finite elements with nearly the same number of nodes for each one of 60 time steps. Additionally, the connectivity of the finite elements has to be represented. Storing both coordinates and connectivity for each time step would be a waste of memory resources, since the element topology does not change during the crash. Therefore a much better approach is to use an indexed geometry.

In OpenInventor [53], a widely used object-oriented 3D

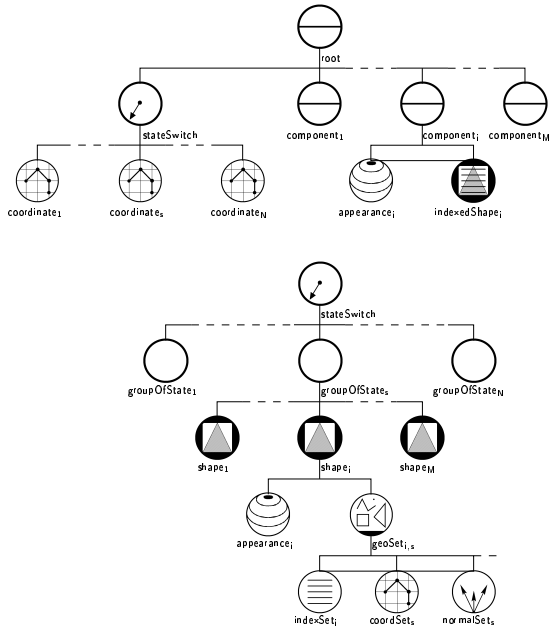


Figure 22: *Comparison of an OpenInventor (top) and a Cosmo3D (bottom) scene graph*

graphics toolkit, we can store the coordinates of each time step under a stateSwitch node and the time-invariant description of the connectivity can be placed to the right side of that node once (see top of Figure 22). For each frame the scene graph is traversed by a render action object which holds the traversal state. One member of the traversal state is the actual set of coordinates which are defined by one of the coordinate_s nodes and which will be referenced by the indexedShape_i nodes. That approach appears to be a very memory efficient representation of our data in a scene graph structure, but it would not allow local scene graph optimizations or multiprocessing of independent subgraphs. This is because objects on the right hand side of the scene graph may depend on settings of the traversal state which have been made by scene graph nodes on the left hand side.

Using SGI's Cosmo3D which forms the underlying scene graph layer for OpenGL Optimizer, an API for large-model visualization, enables the utilization of features such as multiprocessing, occlusion culling, and accelerated hardware-assisted scene manipulation. Cosmo3D provides a scene graph structure which resembles the semantics of the Virtual Reality Modeling Language [1]. It basically differs from that of the OpenInventor scene graph. There is no information inherited horizontally in the Cosmo3D scene graph which is traversed just downward from top to bottom in each branch. Thus, a different scene graph structure has to be chosen (see bottom of Figure 22) which reduces redundant data storage as much as possible by taking advantage of indexed geometries and by shared instancing of scene graph nodes.

That means a coordSet node containing all coordinates of time step s can be assigned to several geoSet nodes. Each of these geoSet nodes only stores a reference to the data which is resident in main memory just once. Therefore, appearance_i and indexSet_i, which represent one and the same car body part across all time steps, can also be shared by the shape_i subgraphs. In analogy each $\text{geoSet}_{i,s}$ in the subgraph of groupOfState_s has a reference to one and the same coordSet_s. If normals should be provided for each vertex the indexSet_i and the coordSet_s will be expanded in a way that the index array can also be used to refer to a normal array held in normalSet_s.

Based on this scene graph design it is now possible to visualize an entire crash data set on a modern desktop multiprocessor graphics workstation at interactive frame rates.

Efficient visualization of physical and structural properties using texture mapping

Using traditional visualization systems, physical properties like plastic strains on FE surfaces are visualized through color coding of the polygons representing the surface elements. Surface areas with nearly equal physical values within predefined ranges are visualized with iso-contouring and color bands. Usually, the intersection points between the contour edges and the polygons representing the elements have to be calculated and additional polygons with different colors have to be created on both sides of the contour line.

In the discussed technique, the physical values serve as entries into a one dimensional texture. As already mentioned in the last session texture mapping is a well established and widely used technique in computer graphics (see the survey by Heckbert [24]) and in scientific visualization [8, 6]. The color of the object, onto which the texture is applied, is modified at each pixel by a corresponding color from the texture image. And hardware support for texture mapping is now widely available from high-end graphics workstations of various vendors down to PCs.

A texture can be thought of not only as an image, but also as a lookup table [20]. In this example we use a color table texture and map the physical values into floats between 0 and 1 serving as coordinates of the one-dimensional texture which are assigned to the vertices. Utilizing a texture with fewer colors and sharp borders between the colors iso-contours are automatically created on the textured model without any calculation of intersection points and without rendering of additional polygons resulting in lower calculation and rendering costs as compared to traditional visualization packages.

Parameter visualization using 1D textures

Similarly, a 1-D texture is used in order to analyze the intrusion of components into the passenger cell in the case

of a frontal or side impact collision. The iso-contours show where in the deformed structure the intrusion of the passenger cell is unacceptable and how far it is away from the acceptable limit.

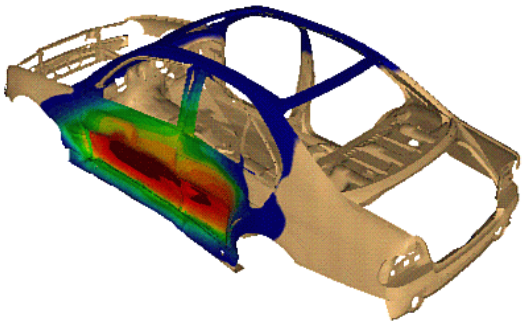


Figure 23: *Intrusion of the car body during a side impact collision. By using 1D-texture mapping we can determine without additional rendering cost, whether the acceptable quantitative intrusion limit is exceeded in some areas.*

The nature and the magnitude of the intrusion are determined by relating the nodes of the FE structure to a reference plane for each time step of the crash simulation. The reference plane is defined within an appropriately chosen coordinate system so that it moves with the car body during the simulated crash. The differences between the distance of the undeformed structure and the distance of the deformed structure from the plane are calculated and scaled to values between 0 and 1, with respect to a predefined range of interest. The scaled values serve as texture coordinates of the vertices of the structure.

If the acceptable limit is changed or the intrusion has to be investigated using a broader or tighter range of interest, no additional computational effort is necessary, only the texture definition has to be adapted. Iso-contour visualization of the intrusion of the car body during a side impact collision is shown in Figure 23.

Model discretization using 2D textures

In many situations the model discretization of the FE structure has to be displayed. There are two alternatives to simultaneously visualize the borders of each quadrilateral FE element together with the shaded polygon representation. Using traditional post-processors, the polygonal model is rendered in a first step, and the lines representing the element borders are drawn in an additional step, resulting in rendering the geometry twice.

Besides the described property mappings, textures can also be used to improve the understanding of the shape of complex structures [43, 38]. In this particular case the grid of the finite element models is visualized by mapping a texture, which paints borders onto each element of the FE model. The main goal is to eliminate the rendering cost induced by the additional drawing of wire-frame lines. A two-dimensional texture, which is represented by a white image with a black border is employed. Figure 24 shows the FE mesh of a dummy model visualized with wire-frame mapping.

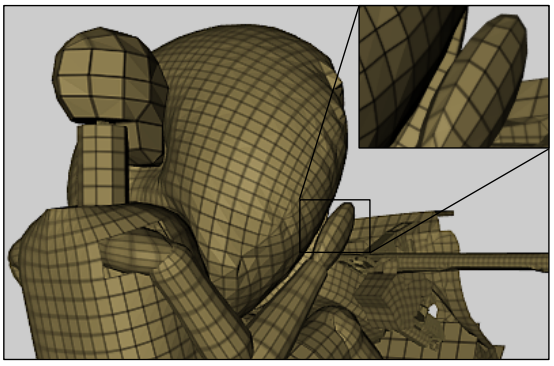


Figure 24: *Visualizing FE model discretization using a 2-D texture.*

In the case of crash simulations, usually 90% of the finite elements are four-sided, 10% of the elements are three-sided. The coordinates of the corners of the texture image are assigned to the corresponding vertices of the polygons to be rendered. If the element is three sided, an additional vertex with the same spatial coordinates and the same normal as the third vertex is created. The fourth texture coordinate of the image is assigned to this new vertex. Since an efficient visualization requires the creation of triangle or quadrilateral strips from the polygonal model, common vertices of adjacent polygons must have the same texture coordinates. Therefore, the texture coordinates are mirrored along the shared edges of adjacent elements.

Visualization of vector data using animated textures

Traditional post-processors visualize vector data like node velocities with thin and opaque lines and arrow heads. Since one car component usually comprises thousands of nodes and finite elements, the same large number of vector arrows is drawn, covering each other and the underlying structure. This makes the analysis of vector data difficult in many cases. The goal is the visualization of the vector data in a way that leaves the underlying structure mostly visible. In [32] we followed an idea of Yamrom, who visualized flow vector fields using animated textures [65], and adapted and extended this method for the visualization of nodal velocities of structural dynamic non-linear FE models.

In contrast to the traditional way lines without arrows are used, but with segments of changing opacity, which move in

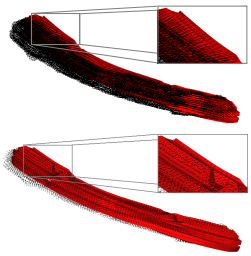


Figure 25: *The upper image shows the visualization of the node acceleration vectors of a FE model of the front bumper structure using a traditional post-processing system. The lines and arrows hide parts of the structure. The same bumper structure can be recognized much better in the lower image, where animated opacity changing textures are used.*

the direction of the vector. The motion is achieved by switching 6 different textures with 16 texels each at each vector line.

The first texture starts with two semi-transparent texels followed by four totally transparent texels, again followed by two texels with opacity values greater than the values of the first two texels and so on. The six textures differ in the position of these ”opacity fields” within the texture. By switching the textures the fields move with growing opacity towards the top of the vectors. That allows it to recognize the direction of the nodal accelerations as well as the structure behind the vectors.

Figure 25 shows the front bumper structure with additional nodal acceleration vectors. The upper image is visualized using a traditional post-processor, the lower image employs the previously described texture animation technique; it shows a snapshot of the animated vectors revealing the structure behind.

Force flux visualization with force tubes

During a car collision, each component of the car body is stressed in a different manner. Some parts absorb very high forces, other parts transfer the forces to the passenger cell. The determination of the structural components, which guide the main forces, enables the engineer to design car components with an optimal crash behavior. Since 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, it is necessary to detect and to understand the force progression within these components.

In order to calculate the forces that act inside a car component, section force calculations are performed. In existing post-processors, first a section plane must be defined and positioned within the component. Next, the section force is calculated. Finally, a diagram is displayed showing the section force as a sum of the forces of the elements influenced by the section plane at this position of the longitudinal structure and its time progression during the crash. For the investigation of the whole component many different sections have to be positioned within the component and a large number of diagrams have to be investigated in a very abstract and time consuming task.

The approach which has been presented first in [31] for the visualization of the force flux is to position an additional tubular element next to the deforming longitudinal structure, whose radius variation visually relates to the local force. The section forces are displayed just like water flow in a flexible tube. Certain parts of the tube are expanding, when the longitudinal force through the corresponding part increases, whereas other parts of the tube are constricting, showing a decrease of force in the structure. Using the tubing method, the behavior of longitudinal structures can be analyzed by investigating their deformation and simultaneously their ability to absorb forces.

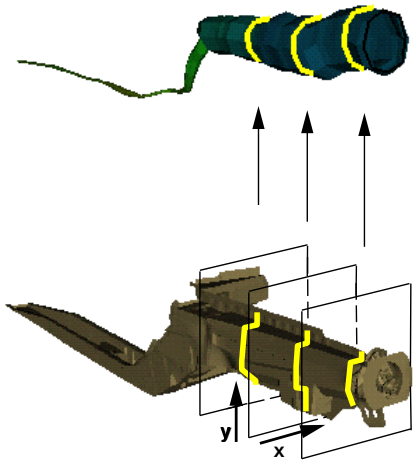


Figure 26: *Force tube generation using section force values.*

The section planes for the computation of the section forces are positioned automatically perpendicular to the structure along a trace line, that follows the shape of the longitudinal structure. Only those elements that are intersected by a plane are taken into account. The forces that affect their element nodes lying on the normal side of the plane are calculate. For each section, the forces are accumulated and the vector component of the accumulated force vector that is parallel to the plane’s normal is computed.

The tube is positioned next to the longitudinal structure with a reasonable spacing and parallel to a line through a number of nodes of the structure. Several rings are positioned around this tubular midpoint-line. Each ring represents one section force. The position of a ring in the tube corresponds to the position of the section in the structure

while the diameter is dynamically related to the value of the scaled section force. A number of points are created around the circle of each ring serving as vertices for polygons that connect the rings and form the tube. The creation of a force tube is outlined in Figure 26.

Visualization of bending moments

The analysis of bending moments in longitudinal structures is very important, because these bending moments can cause high torsion stresses in components, which are connected with the longitudinals. In this part of the session, we describe a method for the visualization of such bending moments based on the force tube approach.

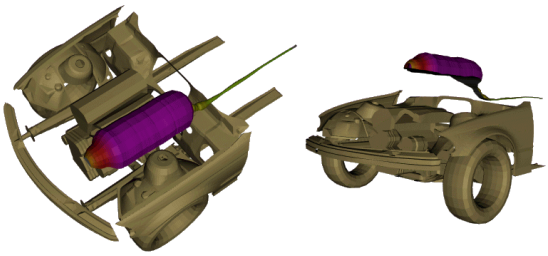


Figure 27: *Moment tube over the left longitudinal structure, displaying bending moments that affect the front part of the longitudinal at the time of 12 ms after the crash. After cutting the tube the upper half tube remains and shows a positive bending of the structure "towards the top", whereas the rear part of the longitudinal is not yet affected by bending stresses.*

The bending moments are calculated through the longitudinal structures – similar to the section forces – from the nodal forces lying on the normal side of the section planes. For each section, two bending moments are computed in relation to two different moment axes (defining the local x-axis and the local y-axis of the section plane). Each axis intersects the center of gravity of the longitudinal structure within the section. The two bending moments are calculated by accumulating the products of the nodal forces and the lever arms defined by the distances between the nodes and the respective moment axis. Each accumulated moment carries a sign defining a bend that is caused by either a positive rotation or a negative rotation around the moment axis.

Similarly to the force tube, a moment tube is created based on the calculated bending moments per section. Each ring of the tube is cut resulting a half tube which shows the sign of the bending moment in that section.

Based on this visualization information about the moment progression can be derived as well as both the magnitude and the sign of the bending moments that affect the longitudinal structure. Figure 27 shows the bending moments in the left longitudinal structure.

References

1. ISO/IEC 14772-1:1997. The virtual reality modeling language. http://www.vrml.org/Specifications/VRML97/, 1997.

2. Kurt Akeley. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, August 1993.

3. James F. Blinn. Jim blinn's corner: Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, January 1988.

4. L. S. Brotman and N. I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):71–81, October 1984.

5. B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, October 1994.

6. B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *ACM Symposium on Volume Visualization '94*, pages 91–98, 1994.

7. B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In A. Kaufman and W. Krüger, editors, *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, 1994.

8. R. A. Crawfis and N. Max. Texture Splats for 3D Scalar and Vector Field Visualization. In G. M. Nielson and Bergeron D., editors, *Visualization 93*, pages 261–265. IEEE Computer Society, 1993.

9. Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, July 1977.

10. T.J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

11. J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, pages 91–98, 1992.

12. Paul E. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 189–198, July 1998.

13. P. J. Diefenbach. *Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-Pass Rendering.* PhD thesis, University of Pennsylvania, June 1996.

14. P. J. Diefenbach and N. Badler. Pipeline Rendering: Interactive refractions, reflections and shadows. *Displays: Special Issue on Interactive Computer Graphics*, 15(3):173–180, 1994.

15. R. Gallagher, R. Haber, G. Ferguson, D. Parker, W. Stillman, and J. Winget. Applying 3D Visualization Techniques to Finite Element Analysis. In *IEEE Visualization 1991*, pages 330–335. IEEE Computer Society Press, 1991.

16. M. Garrity. Ray Tracing Irregular Volume Data. In *ACM Workshop on Volume Visualization '90*, pages 35–40, 1990.

17. C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *Computer Graphics and Applications*, 12(2):40–48, 1992.

18. Ned Greene. Applications of world projections. In *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

19. P. Haeberli and K. Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '90*, pages 309–318, July 1990.

20. P. Haeberli and M. Segal. Texture mapping as A fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993.

21. P. Haeberli and M. Segal. Texture Mapping as a Fundamental Drawing Primitive. In M. Cohen, C. Puech, and F. Sillion, editors, *Proc. of the Fourth Eurographics Workshop on Rendering, Paris, France, June, 1993*, pages 259–266. Springer, Vienna, New York, 1993.

22. P. E. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *SIGGRAPH '90 Proceedings*, pages 309–318, August 1990.

23. M. Haubner, Ch. Krapichler, A. Lösch, K.-H. Englmeier, and van Eimeren W. Virtual Reality in Medicine - Computer Graphics and Interaction Techiques. *IEEE Transactions on Information Technology in Biomedicine*, 1996.

24. P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

25. W. Heidrich. *High-quality Shading and Lighting for Hardware-accelerated Rendering.* PhD thesis, University of Erlangen-Nürnberg, April 1999.

26. W. Heidrich and H.-P. Seidel. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

27. W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *SIGGRAPH '99 Proceedings*, August 1999. See http://www.mpi-sb.mpg.de/~heidrich.

28. J. T. Kajiya and B. P. Von Herzen. Ray Tracing Volume Densities. *ACM Computer Graphics, Proc. SIGGRAPH '84*, 18(3):165–174, July 1984.

29. G. Kerlick and E. Kirby. Towards Interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations. In *IEEE Visualization 1993*, pages 374–377, 1993.

30. W. Krüger. The Application of Transport Theory to the Visualization of 3-D Scalar Data Fields. In *IEEE Visualization '90*, pages 273–280, 1990.

31. S. Kuschfeldt, T. Ertl, and M. Holzner. Efficient visualization of physical and structural properties in crash-worthiness simulations. In R. Yagel and H. Hagen, editors, *IEEE Visualization '97*, pages 487–490. IEEE Computer Society Press, Oct 1997.

32. S. Kuschfeldt, O. Sommer, and T. Ertl. "efficient visualization of crash-worthiness simulations". *IEEE Computer Graphics and Applications*, 18(4):60–65, July/August 1998. ISSN 0272-1716.

33. P. Lacroute and M Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. *Computer Graphics, Proc. SIGGRAPH '94*, 28(4):451–458, 1994.

34. D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '93*, 25(4):285–288, July 1991.

35. M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

36. Robert R. Lewis. Making shaders more physically plausible. In *Fourth Eurographics Workshop on Rendering*, pages 47–62, June 1993.

37. W. Lorensen. Geometric Clipping Using Boolean Textures. In *IEEE Visualization 1993*, pages 268–274, 1993.

38. W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics, Proc. SIGGRAPH '87*, 21(4):163–169, 1987.

39. N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. In *ACM Workshop on Volume Visualization '91*, pages 27–33, 1991.

40. C. Montani, R. Scateni, and R. Scopigno. Discretized Marching Cubes. In *IEEE Visualization'94*, pages 281–287, 1994.

41. John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 293–302, August 1997.

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

43. P. Rheingans. Opacity-modulating Triangular Textures for Irregular Surfaces. In *IEEE Visualization 96*, pages 219–225, 1996.

44. Holly Rushmeier, Gabriel Taubin, and André Guéziec. Applying shape from lighting variation to bump map capture. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 35–44, June 1997.

45. Marc Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

46. Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

47. H. Shen and C. Johnson. Sweeping Simplices: A Fast Iso-Surface Axtraction Algorithm for Unstructured Grids. In *IEEE Visualization '95*, pages 143–150, 1995.

48. P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '90*, 24(5):63–70, 1990.

49. C. Silva and J. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *Transactions on Visualization and Computer Graphics*, 4(2), June 1997.

50. C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *ACM Symposium on Volume Visualization '94*, pages 83–90, 1994.

51. A. Van Gelder and K. Kwansik. Direct Volume Rendering with Shading via Three-Dimensional Textures. In R. Crawfis and Ch. Hansen, editors, *ACM Symposium on Volume Visualization '96*, pages 23–30, 1996.

52. D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

53. Josie Wernecke. *The Inventor Mentor*. Addision-Wesley, 1994.

54. R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing. In *IEEE Visualization '97*, pages 35–43, 1997.

55. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *Computer Graphics, Proc. SIGGRAPH '98*, pages 293–303, July 1998.

56. R. Westermann and Th. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98 Proceedings*, pages 169–178, July 1998.

57. R. Westermann, O. Sommer, and T. Ertl. Decoupling polygon rendering from geometry using rasterization hardware. In *Eurographics Rendering Workshop '99, Workshop Proceedings*, pages 53–65. Eurographics, June 1999.

58. J. Wilhelms and A. Van Gelder. Octrees for faster Iso-Surface Generation. *ACM Transactions on Graphics*, 11(3):201–297, July 1992.

59. J. Wilhelms, A. van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *IEEE Visualization 1996*, pages 57–65, 1996.

60. Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

61. P. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):102–126, 1992.

62. P. Williams and N. Max. A Volume Density Optical Model. In *ACM Workshop on Volume Visualization '92*, pages 61–69, 1992.

63. O. Wilson, A. Van Gelder, and J. Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

64. R. Yagel, D. Reed, A. Law, P. Shih, and N. Shareef. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In *ACM Symposium on Volume Visualization '96*, pages 55–63, 1996.

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

66. W. Ye and J. Vance. Visualization of Structural Impact Problems in a Virtual Environment. In A. Tentner, editor, *High Performance Computing 1997*, pages 325–330, P. O. Box 17900, San Diego, CA 92177, U.S.A., April 1997. Society for Computer Simulation International. ISBN: 1-56555-122-2.