Computing and Visualization in Science Manuscript-Nr.
(will be inserted by hand later)

Volume Visualization on Sparse Grids

Texture Hardware Based Volume Rendering on Compressed Data Sets

Christian Teitzel[1], Matthias Hopf[1], Roberto Grosso[2], and Thomas Ertl[1]

[1] Computer Graphics Group,University of Erlangen, Am Weichselgarten 9, 91058 Erlangen, Germany
URL: http://www9.informatik.uni-erlangen.de/eng/research/vis/sparse/
[2] AEA Technology GmbH, Staudenfeldweg 12, 83624 Otterfing, Germany

Received: 13 August 1998 / Accepted: 15 December 1998

Abstract. Volume rendering is an important technique of displaying volumetric three-dimensional scalar data sets resulting from measurement or simulation. Additionally, sparse grids are of increasing interest in numerical simulations. Based upon hierarchical tensor product bases, the sparse grid approach is a very efficient one improving the ratio of invested storage and computing time to the achieved accuracy for many problems in the area of numerical solution of partial differential equations, for instance in numerical fluid mechanics. The volume visualization algorithms that are available so far cannot cope with sparse grids. We present an approach that directly works on these grids. As a second aspect in this paper, we suggest to use sparse grids as a data compression method in order to visualize huge data sets even on workstations with low main memory. Because the size of data sets used in numerical simulations is still growing, this feature makes it possible that workstations can continue to handle these data sets. In addition to the standard sparse grid interpolation algorithm and the so-called combination approach, we have developed a new sparse grid interpolation method, which harnesses the texture-mapping hardware of Silicon Graphics workstations for acceleration purposes. Therefore, hardware based volume rendering becomes possible on compressed data sets at interactive frame rates. This is not possible if other compression methods like wavelet or fractal compression are used.

Key words: Volume Visualization – Sparse Grids – Data Compression – Using Texture Hardware – Volume Ray Casting – Maximum Intensity Projection – Iso-Surface Extraction

1 Introduction

The idea of the sparse grid technique was developed in the 1960s by Babenko [1] and Smolyak [17]. A special tensor product technique of constructing higher dimensional quadrature formulas and approximation operators from corresponding one-dimensional objects leads to almostoptimal error rates.

In 1990 sparse grids were introduced to numerical computation by Zenger [23]. By means of these grids, it is possible to reduce the total amount of data points or the number of unknowns in discrete partial differential equations. Due to these benefits, sparse grids are more and more used in numerical simulations nowadays [2, 3, 8, 9].

On the other hand, it is rather difficult to visualize the results of the simulation process directly on sparse grids, since evaluation and interpolation of function values is quite complicated. Because of this, the results of numerical simulations on sparse grids are usually interpolated to the associated full grid. Then all known visualization algorithms on full grids can be performed, e.g. particle tracing, iso-surface extraction, volume rendering, etc.. However, a major drawback of this procedure is the fact that the advantage of low memory consumption of sparse grids comes to nothing using the associated full grid for the visualization step.

Therefore, visualization tools working directly on sparse grids are going to be an important topic of research. Recently, Heußer and Rumpf presented an algorithm for iso-surface extraction on sparse grids [11]. In a previous work, we introduced particle tracing on uniform sparse grids [19]. The first aim of this paper is to present volume ray casting directly on sparse grids. Furthermore, a second aspect of our work is the idea that sparse grids can be used for data compression in order to visualize huge data sets even on workstations with a limited amount of main memory. Moreover, using the combination technique, which is presented below, it is possible to decompose a sparse grid into a certain number of uniform full grids of low resolution. Because of this, Silicon Graphics' texture hardware can be deployed for the necessary function interpolation. Hence, we are able to perform volume visualization methods on compressed data sets at interactive frame rates. This is not possible if other compression methods like wavelet

Correspondence to: teitzel@informatik.uni-erlangen.de

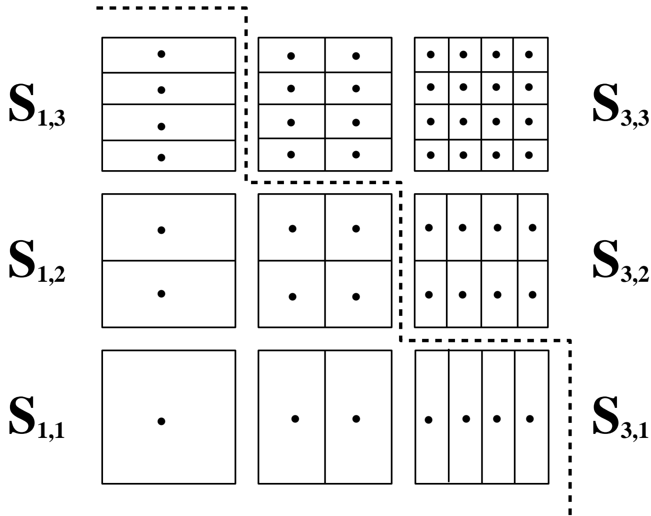


Fig. 1. Two-dimensional hierarchical subspace decomposition.

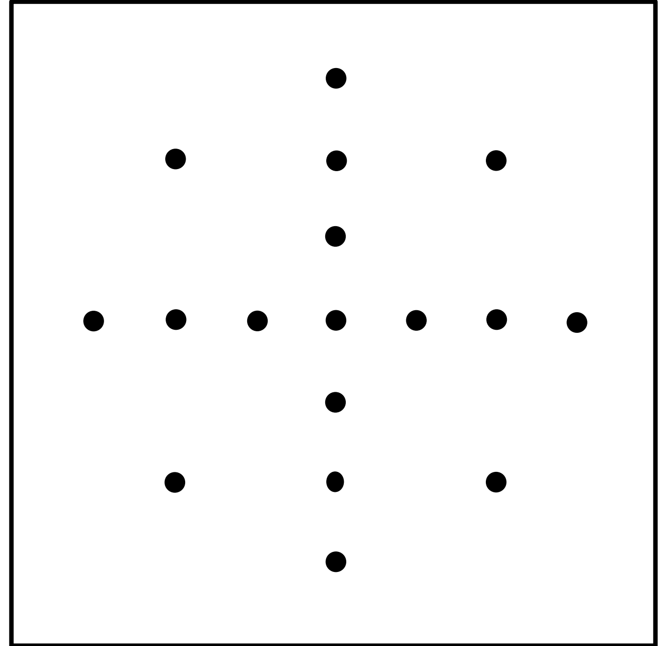


Fig. 2. Two-dimensional sparse grid of level 3.

or fractal compression are used. However, we are able to handle sparse grids of level 11, which correspond to volumes of 2049^3 voxels.

This paper is organized as follows. In Sect. 2 we give a brief introduction to sparse grids and in Sect. 3 the combination technique is described. In order to introduce volume ray casting on sparse grids, an object-oriented framework has been developed. The implementation of our volume ray caster and the special class hierarchy for the different sparse grid interpolation approaches are described in Sects. 4 and 5. Thereafter, in Sect. 6 we describe how to exploit the texture hardware in order to accelerate the sparse grid interpolation process. Finally, the results of memory, time, and error analyses are listed in detail in Sect. 7.

2 Basics of Sparse Grids

In this section a brief summary of the basics of sparse grids is given. For a detailed survey of sparse grids we refer to [2, 23]. In order to make this overview easy to understand and to reduce the number of indices, we describe only three-dimensional grids, whereas the sketches reveal the two-dimensional situation.

Let $f : [0,1]^3 \longrightarrow \mathbf{R}$ be a smooth function defined on the unit cube in $\mathbf{R}^3$ with values in $\mathbf{R}$. Furthermore, f should vanish on the boundary of the cube. This condition is not a strong restriction but is just helpful for an elegant description. Of course, our program can handle three-dimensional functions and even vector fields without zero boundary conditions. If such a function f is stored in the computer memory, then function values at certain positions on a spatial grid are stored in an array. The simplest mesh is a uniform one. Now let G_{i_1,i_2,i_3} be a uniform grid with respective mesh widths $h_{i_j} = 2^{-i_j}$, $j = 1,2,3$. On these grids we can introduce the following partial ordering relation: G_{i_1,i_2,i_3} is a refinement of G_{k_1,k_2,k_3} if and only if $k_j \leq i_j$, $j = 1,2,3$ and $k_1 + k_2 + k_3 < i_1 + i_2 + i_3$. Thus we obtain a hierarchy of meshes.

Now let $\hat{L}_n$ be the function space of the piecewise tri-linear functions defined on $G_{n,n,n}$ and vanishing on the boundary. Additionally, consider the subspaces S_{i_1,i_2,i_3} of $\hat{L}_n$ with $1 \leq i_j \leq n$, $j = 1,2,3$, which consist of the piecewise tri-linear functions defined on G_{i_1,i_2,i_3} and vanishing on the grid points of all coarser grids. Apparently, the hierarchy of grids naturally introduces a hierarchy of subspaces and it follows that

$$\hat{L}_n = \bigoplus_{i_1=1}^{n} \bigoplus_{i_2=1}^{n} \bigoplus_{i_3=1}^{n} S_{i_1,i_2,i_3} \quad . \tag{1}$$

Hence, we have found a hierarchical basis decomposition of the function space $\hat{L}_n$. Piecewise tri-linear finite elements are used as basis functions in each subspace S_{i_1,i_2,i_3}. Then we define the basis functions of the subspace S_{i_1,i_2,i_3} of $\hat{L}_n$:

$$b_{k_1,k_2,k_3}^{(i_1,i_2,i_3)}(x_1,x_2,x_3) := \prod_{j=1}^{3} w_{i_j}(x_j - m_{k_j}^{(i_j)}) \tag{2}$$

$$\text{with} \quad m_{k_j}^{(i_j)} = (2k_j - 1) \cdot h_{i_j} \quad , \quad 1 \leq k_j \leq 2^{i_j - 1} \quad ,$$

$$\text{and} \quad w_i(x) := \begin{cases} \frac{h_i + x}{h_i} & : \ -h_i \leq x \leq 0 \\ \frac{h_i - x}{h_i} & : \ 0 \leq x \leq h_i \\ 0 & : \ \text{else} \end{cases} \quad .$$

Now we are interested in some estimations of the interpolation error. Hence, let $\hat{f}_n \in \hat{L}_n$ be the interpolated function on the grid $G_{n,\cdots,n}$. Then $\hat{f}_n$ is given by

$$\hat{f}_n = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \sum_{i_3=1}^{n} f_{i_1,i_2,i_3} \tag{3}$$

where $f_{i_1,i_2,i_3} = \sum_{k_1=1}^{2^{i_1-1}} \sum_{k_2=1}^{2^{i_2-1}} \sum_{k_3=1}^{2^{i_3-1}} c_{k_1,k_2,k_3}^{(i_1,i_2,i_3)} \cdot b_{k_1,k_2,k_3}^{(i_1,i_2,i_3)}$.

The values $c_{k_1,k_2,k_3}^{(i_1,i_2,i_3)}$ are called contribution coefficients and $f_{i_1,i_2,i_3} \in S_{i_1,i_2,i_3}$ is a linear combination of the basis functions of the appropriate subspace. It can be shown that the following estimations hold with regard to the L^2 or L^∞ norms (compare [2, pp. 13]):

$$\|f_{i_1,i_2,i_3}\| \leq const \cdot \left\| \frac{\partial^6 f}{\partial x_1^2 \partial x_2^2 \partial x_3^2} \right\| \cdot h_{i_1}^2 h_{i_2}^2 h_{i_3}^2, \quad (4)$$

$$\left\| f - \hat{f}_n \right\| \leq O\left(h_n^2\right). \quad (5)$$

So far we have just dealt with regular uniform meshes, which are named full grids. Now let us turn to sparse grids. Consider the subspaces S_{i_1,i_2,i_3} with $i_1 + i_2 + i_3 = const$. Equation (4) shows that $\|f_{i_1,i_2,i_3}\|$ has a contribution of the same order of magnitude, namely $O(2^{-2 \cdot const})$ for all subspaces with $i_1 + i_2 + i_3 = const$. Additionally, these subspaces have the same number of basis functions, namely $2^{const-3}$. Since the number of basis functions is equivalent to the number of stored grid points and because of the contribution argument as well, it seems to be a good idea to define a sparse grid space $\tilde{L}_n$ as follows:

$$\tilde{L}_n := \bigoplus_{i_1+i_2+i_3 \leq n+2} S_{i_1,i_2,i_3}. \quad (6)$$

Now the interpolated function $\tilde{f}_n \in \tilde{L}_n$ is given by

$$\tilde{f}_n = \sum_{i_1+i_2+i_3 \leq n+2} f_{i_1,i_2,i_3} \quad (7)$$

and the interpolation error with regard to the L^2 or L^∞ norm is given by (compare [2, pp. 23])

$$\left\| f - \tilde{f}_n \right\| \leq O\left(h_n^2 \left(\log_2\left(h_n^{-1}\right)\right)^2\right). \quad (8)$$

This estimation shows that the sparse grid interpolated function $\tilde{f}_n$ is nearly as good as the full grid interpolated function $\hat{f}_n$ (compare Eq. (5) with Eq. (8)).

Now we consider the dimensions of the function spaces $\hat{L}_n$ and $\tilde{L}_n$, which correspond to the number of nodes of the underlying grids. Obviously, the dimension of the full grid space is given by

$$dim\left(\hat{L}_n\right) = O\left(2^{3n}\right) = O\left(h_n^{-3}\right). \quad (9)$$

For the sparse grid the following equation holds:

$$dim\left(\tilde{L}_n\right) = O\left(2^n \cdot n^2\right) = O\left(h_n^{-1}\left(\log_2\left(h_n^{-1}\right)\right)^2\right). \quad (10)$$

Therefore, a tremendous amount of memory is saved if sparse grids are used instead of full grids.

If the function f is given and a certain accuracy is required, then it is possible to use $\hat{f}_n \in \hat{L}_n$ or $\tilde{f}_m \in \tilde{L}_m$ where m is just slightly greater than n. Due to the very low memory consumption of sparse grids, it is better to use the function $\tilde{f}_m$. On the other hand the function f is often given in discrete form as data set on a full grid. In this case it is not possible to reach a better accuracy with the sparse grid approach than with the original full grid data. However, Eqs. (8) and (10) show that a very small loss of accuracy is rewarded with a huge amount of saved storage.

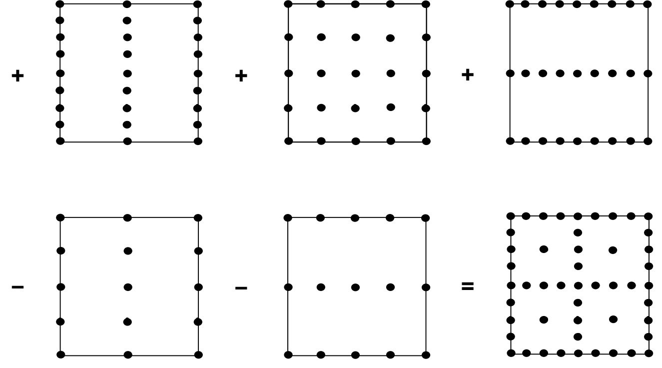


Fig. 3. A two-dimensional sparse grid of level 3 can be reconstructed by linear combination of five full grids of low resolution.

Finally, recall that the sparse grid space $\tilde{L}_n$ is the direct sum of all subspaces $S_{i,j,k}$ with $i + j + k \leq n + 2$. Now we define the *level of a subspace* as the number $n = i + j + k - 2$. Moreover, we define a *level of the sparse grid space* as the direct sum of all subspaces of the same level of subspaces. Hence, $\tilde{L}_n$ is the direct sum of its first n levels and is called a *sparse grid of level n*.

3 Combination Technique

Since the described sparse grid interpolation of function values is quite complicated and rather time consuming, we have implemented the so-called combination technique. This method was introduced by Griebel, Schneider, and Zenger in 1992 [9]. Actually, the combination method has been used in numerical simulations in order to combine partial solutions computed on smaller, suitable full grids to the desired sparse grid solution. However, we start with a data set given on a sparse grid and decompose the grid so that the data set is represented on certain uniform full grids of low resolution. Now the quick and easy tri-linear interpolation can be performed on each of these full grids. The resulting value is computed by linear combination of the tri-linear interpolated full grid results.

Going into details, it can be proofed that the three-dimensional interpolated function $\tilde{f}_n \in \tilde{L}_n$ is given by

$$\tilde{f}_n = \sum_{i_1+i_2+i_3=n+2} f_{i_1,i_2,i_3}^c - 2 \cdot \sum_{i_1+i_2+i_3=n+1} f_{i_1,i_2,i_3}^c + \sum_{i_1+i_2+i_3=n} f_{i_1,i_2,i_3}^c \quad (11)$$

where f_{i_1,i_2,i_3}^c denotes the tri-linear interpolation of function values on the respective full grid. Figure 3 reveals the two-dimensional situation. Notice that the used full

grids consist of the same nodes as the corresponding sparse grid.

Now let us turn to the benefit of the combination technique. The total number of summands of the standard sparse grid interpolation on a three-dimensional sparse grid of level n is given by

$$\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{1}{6} n(n+1)(n+2) \tag{12}$$

(compare Eq. (7)), whereas the total number of tri-linear interpolations of the combination method adds up to

$$\sum_{i=n-2}^{n} \frac{i(i+1)}{2} = \frac{3}{2} n(n-1) + 1 \tag{13}$$

in the three-dimensional case (see Eq. (11)).

However, the main advantage of the combination technique is the fact that uniform full grids are used. Thus, it is possible to exploit the texture hardware of Silicon Graphics workstations for the interpolation of function values (see Sect. 6). Hereby, we are able to use volume ray casting on sparse grids interactively. In the next section the lighting models and transfer functions used in our volume rendering implementation are presented.

4 Volume Rendering

Volumetric three-dimensional scalar data sets resulting from measurement or numerical simulation can be displayed very well by volume rendering techniques. In this section the used volume visualization algorithms are described briefly. For a detailed presentation of volume rendering techniques we refer to [5, 7, 10, 12, 13, 14, 22].

The idea behind volume ray casting is that rays are traced back from each pixel of the desired image into the volume. The integral form of the equation of transfer is given by

$$I = I_0 \, e^{-\tau(x_0,x)} + \int_{s_0}^{s} \eta \, e^{-\tau(x',x)} ds' \tag{14}$$

where I_0 denotes the specific intensity at the boundary at x_0, $\tau(x_1, x_2)$ the optical depth, and η the total emission coefficient [10]. While tracing a ray from a pixel, contribution values of hit voxels are integrated or summed up in order to compute the intensity of this pixel.

We have implemented three different illumination models into our program. In the X-ray model, which provides X-ray like images, the intensity of a pixel is calculated by evaluating the line integral of function values along the ray.

Another algorithm we have implemented is the so-called maximum intensity projection method. Here, the intensity of a pixel is the maximum function value occurring on the corresponding ray.

Extracting iso-surfaces is a third technique we have implemented in our tool so far. To find an iso-surface to a given iso-value, each ray is traced and a patch is displayed if the difference of current function value and iso-value changes the sign. In order to use Lambertian or diffuse reflection for illumination of the resulting surface, the normal of the surface is needed. Recall that the gradient of the function can be used because the gradient is normal to all iso-surfaces.

5 Software Based Interpolation

In this section the different ways of implementing the interpolation routine needed for volume ray casting are described. Since the employment of texture hardware should be depicted in detail, this important technique is presented in a separate section (see Sect. 6).

The graphical user interface of our program was developed using RapidApp, an interactive tool for creating applications [16]. RapidApp uses the ViewKit class library based on the standard X11 toolkit Motif. ViewKit facilitates the use of OpenGL and Open Inventor.

Our new interpolation routines on sparse grids and the adapted volume visualization methods have been implemented in an object-oriented manner as C++ classes.

5.1 Interpolation on Sparse Grids

All volume ray casting methods have in common that they must evaluate the function f at certain positions, which are in general not at grid points. Therefore, the value of f at such a position has to be interpolated. As mentioned above, this interpolation on sparse grids is different from that one on full grids.

In contrast to the tri-linear full grid interpolation, the sparse grid interpolation does not operate locally because one basis function in every subspace contributes to the function value. Since the complicated sparse grid interpolation is one of the most time consuming operations during the volume rendering process on sparse grids, it is important to perform the interpolation as fast as possible.

5.2 Using Improved Sparse Grid Interpolation

Equation (7) specifies the sparse grid interpolation, which has to be implemented. Now we have to choose the appropriate data structure for an efficient implementation of the sparse grid interpolation.

The contribution coefficients of the sparse grid are usually stored in a binary tree [2, 3, 11]. Then a recursive tree traversal has to be performed in order to interpolate the function value. This tree traversal is very slow. Although caching strategies can increase the efficiency of the traversal [11], the computation of the values remains rather time consuming.

In order to avoid the tree traversal and to accelerate the access to the contribution coefficients, we have developed a new, very efficient data structure based upon arrays. Therefore, we have implemented a particular C++

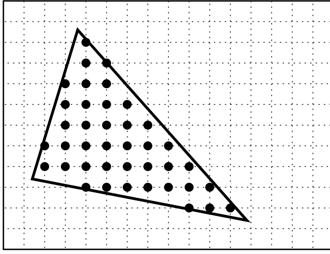


Fig. 4. Rasterization of a triangle

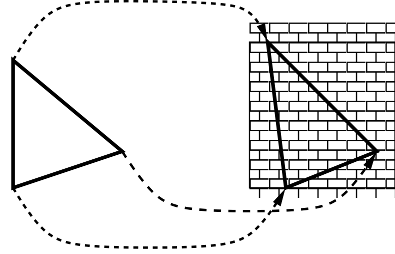


Fig. 5. Mapping of a texture onto a triangle by texture coordinates

class hierarchy. Here we can just give a very brief idea of the classes.

Besides abstract base classes, classes for input, and other auxiliary classes, the classes of interest are named `hbSparseGrid`, `hbLevel`, and `hbSubspace`. The class `hbSparseGrid` principally contains a stack of n levels of class `hbLevel`. Furthermore, `hbLevel` comprises the respective number of subspaces ($(n+1)n/2$), denoted `hbSubspace`. The class `hbSubspace` contains an array of the size 2^{n-1} times data dimension, where the contribution coefficients are stored. The function value at an arbitrary position is computed by means of formula (7). In order to compute a function value, the class `hbSparseGrid` contains a method `calcValue(...)`. This method sends a '`calcValue()`' to each `hbLevel` to accumulate the contributions to the resulting value. Then the method `hbLevel::calcValue(...)` performs a loop over all subspaces of the current level. In this loop, the required basis function is determined by means of the coordinates of the current position. Recall that only one basis function per subspace is unequal to zero at a certain position because all basis functions are hat-functions with disjunct supports. Hence, we know the required contribution value. Now the 'height' over the current position in the tri-linear hat-function is determined and multiplied with the contribution value. Thus, we obtain the total contribution of this subspace to the function value. Additionally, we compute the gradient, which is needed for the illumination of iso-surfaces, in this loop by looking up the correct 'height' of the derivative of the hat-function, a simple box-function.

5.3 Using the Combination Technique

Remember that the idea of the combination method is combining full grids of low resolution to a resulting sparse grid. Then, Eq. (11) determines the interpolation process using the combination technique.

Since the low resolution full grids needed by the combination method are uniform grids, the function values can be stored in three-dimensional arrays. Thus, the `calcValue(...)` method of the according derived class `hbCombinationSparseGrid` can address the necessary function values in a tight loop. This fact and the smaller number of required basic arithmetic operations makes the combination technique an order of magnitude faster than the previously described sparse grid interpolation even for low levels.

In the following section we are going to describe a method that accelerates the combination technique by using texture hardware.

6 Employment of Texture Hardware

When interpolating data using the combination technique the processor spends most of its time on the tri-linear interpolation of the full grids. In order to significantly reduce the computation time, we decided to take advantage of the texturing hardware of modern Silicon Graphics workstations. As the reader may not be familiar with the standard rendering pipeline and especially texture mapping, we will address this topic first.

Most 3D graphics hardware nowadays works by projecting triangles from 3D space onto the user's viewing plane called *viewport*, lighting them with some derivate of a Phong model, rastering, shading, and texturing them, and finally blending them with the image calculated so far, after some sort of depth test has succeeded. Projection, lighting, shading, and depth test are irrelevant for our system and thus no more mentioned here.

As all modern graphics systems use raster displays as final output system, triangles have to be rastered. In this step, a set of pixels is created for the triangle according to Fig. 4, which represent the discrete points lying inside the triangle.

While texturing a triangle, the so-called *texture coordinate* is assigned to each point of the triangle. The texture coordinate describes which part of the texture should be mapped onto the triangle according to Fig. 5. In the rasterization process these coordinates are used to look up texture values for each pixel of the triangle. Because textures are given as discrete picture maps, the texture pixels, also called *texels*, have to be interpolated. In general, bilinear interpolation is used.

If the drawn triangles are getting small relative to the size of the texture, noticeable artifacts can occur, because several texels are just missed in the rasterization process. In order to minimize these artifacts, a technique called *MIP mapping* was introduced. With this technique several versions of the texture are used, one original texture and several down-sampled variants. In order to reduce visible artifacts even more, two of these texture versions are used for each texel lookup and the

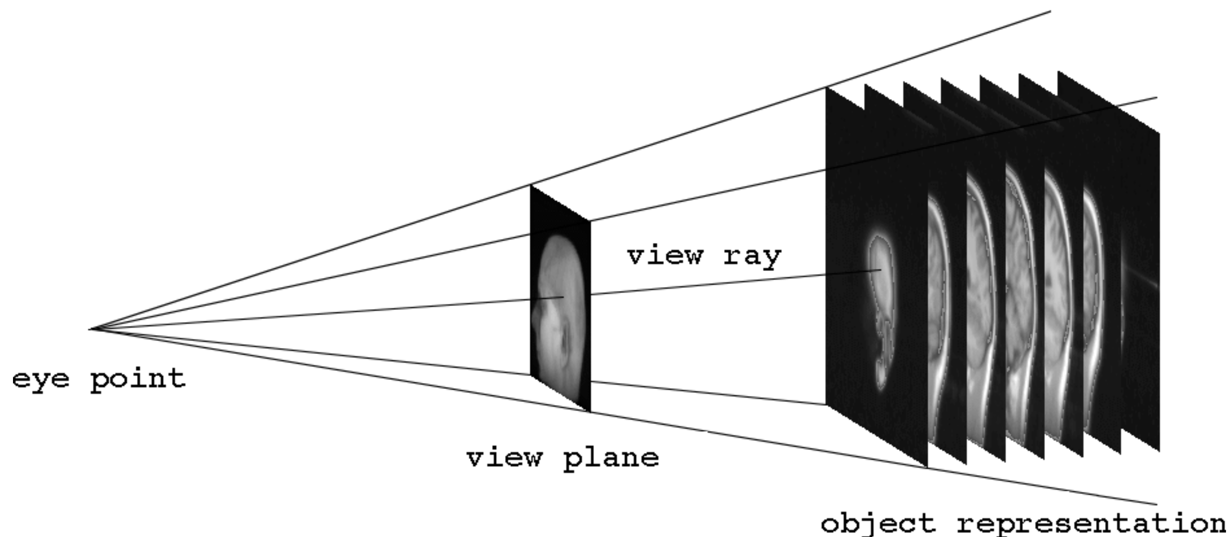


Fig. 6. Rendering volumes using textured polygons

two resulting values are linear interpolated. For more information about texturing have a look at [15].

High performance graphics engines have hardware based support for MIP mapped texturing. As some variant of tri-linear interpolation is used in this process, it can as well be used for real tri-linear interpolation of volume textures. Thus, it is possible to draw polygons textured with a planar slice of the volume. With this feature in mind ray casting algorithms can be rewritten so that they render all rays simultaneously. According to Fig. 6 the pictures are composited by blending several planes textured with the volume data. In these planes the sample points of the ray caster are represented by the pixels of the rendered polygons.

Remember how interpolation is done on sparse grids with the combination method (see Eq. (11)). Here we need tri-linear interpolation of several volume textures, adding and subtracting the interpolation results. Now we can load three-dimensional textures with the values of the full grids $f^c_{i_1,i_2,i_3}$ and employ the hardware.

After rasterization, the pixels resulting from the triangles are drawn into the *frame buffer*, the final output system of the graphics hardware. But graphics systems do not just overwrite the former values of the frame buffer. It depends on the graphics interface which operations can be performed in this last so-called *blending* step. According to Eq. (11), we first need the new values to be added to and later to be subtracted from the former values in the frame buffer. These are no standard rendering operations, but fortunately on SGI's hardware some extensions exist that allow these operations.

As our system was developed on Silicon Graphics workstations, we used the graphics system *OpenGL*. It is an interface standard introduced by Silicon Graphics and now maintained by the OpenGL Architecture Review Board (ARB). OpenGL contains routines for texture mapping, using special graphics hardware whenever possible. Several vendors added some extensions for three-dimensional volume textures. By using these extensions our class `hbOglSparseGrid` is able to exploit the hardware in SGI's *Reality Engine* and *Maximum Impact* graphics systems, which are able to perform the necessary tri-linear interpolation for volume texturing in hardware. For further information about the extensions and Silicon Graphics hardware see [15].

Because the interpolation of single points is rather inefficient — several functions have to be called, the hardware has to be initialized, results have to be fetched back — the ray caster was rewritten so that it renders all rays simultaneously, just as explained above. Now the texture hardware can be utilized to draw a complete plane textured with the volume data. Remember that several values have to be added and subtracted for each sparse grid interpolation. For each of these interpolation functions, a plane is drawn with a certain volume texture representing the function. Then the occurring planes are composed during the blending step using the `GL_FUNC_ADD_EXT` and `GL_FUNC_REVERSE_SUBTRACT_EXT` blending extensions of OpenGL.

The pseudo code on page 8 shows the two main functions, *createTextures()* for creating and loading the volume textures and *draw()* for drawing a view of the function. *createTextures()* has to be called only once, because the textures are stored in the graphics hardware and do not need to be changed for different views. Note that the data function *d()* is no longer needed in *draw()*. These main functions need two help functions that are described on page 9.

In order to switch quickly between the different volume textures, they all have to fit into texture memory at the same time. By using the *texture-name* extension of OpenGL, pre-loaded textures can be selected for rendering almost instantly. Recall that the combination technique uses full grids of size $(2^i+1)\times(2^j+1)\times(2^k+1)$. On the other hand, volume textures have to be of size $2^p \times 2^q \times 2^r$. In order to fit the full grids completely

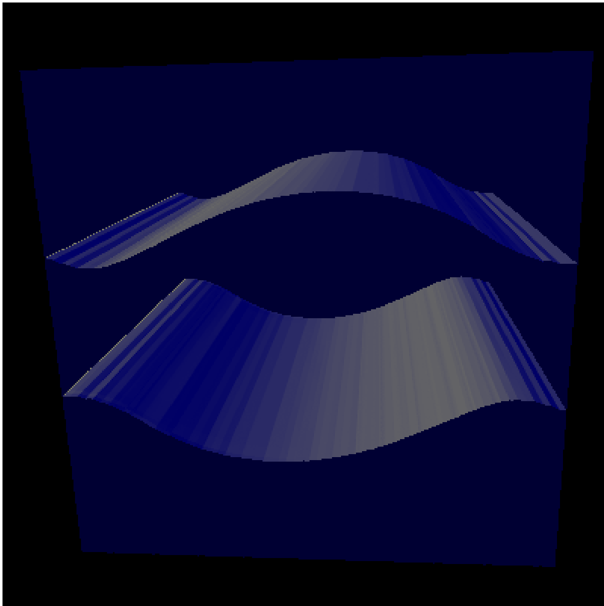


Fig. 7. Iso-surface of pressure in a cavity flow computed by the combination technique.

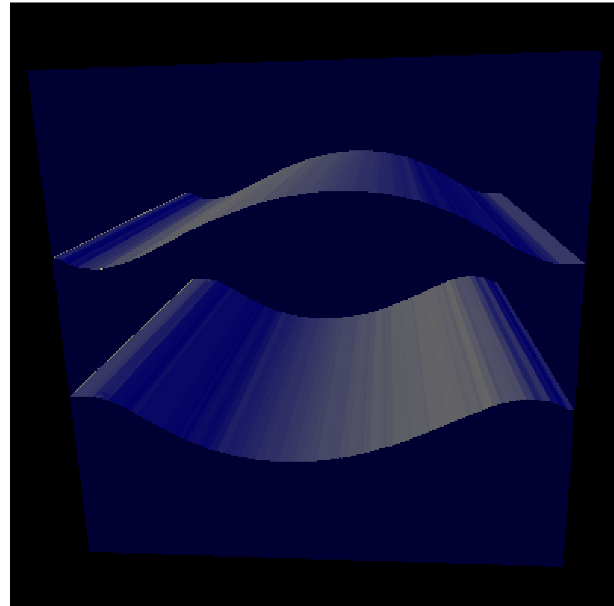


Fig. 8. Iso-surface of pressure in a cavity flow computed on a full grid.

into volume textures, we have to allocate textures of size $2^{i+1} \times 2^{j+1} \times 2^{k+1}$. Hence, quite a lot of texture memory is wasted, although it is a scarce resource.

In principle, OpenGL supports volume textures featuring a so called border of size 2 in every direction so that the mentioned full grids would fit almost seamlessly. Unfortunately, these texture borders are not implemented in todays hardware.

When using hardware for mathematical computations, accuracy can be quite a problem. On Silicon Graphics machines the blending operation can currently be performed in the frame buffer with 12 bits at most. We could use the accumulation buffer for higher accuracy, but several tests revealed that using this feature slows down the calculation process significantly. Since pixel values are automatically clamped to values in the interval $[0, 1]$, all texture elements have to be scaled down by the number of functions contributing positive values to Eq. (11). For a level 10 sparse grid, 91 of 136 functions contribute positive data, which means a loss of almost 7 bits, resulting in only 5 bits of accurate information. Since we have at best 12 bits of accurate information in the frame buffer, it is sufficient to use only 2 bytes of texture memory per voxel. If future hardware will incorporate larger frame buffers with higher pixel accuracy, this will automatically enhance the image quality of our algorithm. Although visible artifacts are remarkably small, some can be seen in the color plates (compare Figs. 9, 10, and 11).

7 Results

In order to appraise the presented approaches, we compare the results of our sparse grid visualization tool with results obtained by using uniform full grids. The tests were performed on several data sets. Two of them are cavity flow data sets, given on a full grid of level 6, i.e. 64^3 nodes. These data sets are the result of a numerical flow simulation and contain pressure and temperature distributions of the flow. Another data set is an analytical test function given on a full grid of level 6 as well. The forth data set, given on a full grid of level 8 (256^3 nodes), contains a spherical harmonic (Legendre's function), which displays a solution of the Schrödinger equation of a hydrogen atom.

Figures 9, 10, and 11 show X-ray images of the pressure values in the mentioned cavity flow. These images have been rendered in order to reveal differences between the three implemented interpolation algorithms. In the OpenGL picture, small flaws can be detected, whereas the combination technique and the sparse grid method render nearly identical images. The artifacts in the OpenGL picture occur because of the already mentioned loss of accuracy using the hardware for mathematical operations. However, the small loss of accuracy is rewarded with a huge saving of computing time.

In the other figures, sparse and full grid results are compared using different lighting models and data sets. Figures 7 and 8 display iso-surfaces of the pressure in the cavity flow data set. In Figs. 12 and 15, the same data set is visualized by means of the maximum intensity projection lighting model. In further maximum intensity projection images (see Figs. 13 and 16), the data set of a spherical harmonic function is visualized. Then, the temperature distribution in the cavity flow is depicted in Figs. 14 and 17 by using an X-ray lighting model. Finally, Figs. 18, 19, and 20 show that the smoothness of extracted iso-surfaces depends on the used grid level.

Here the functions that are used to visualize any scalar volumetric data function $d : [0,1]^3 \to [0,1]$ *are described using pseudo code. The function* createTextures() *is called once to create the necessary textures and to transport them into the graphics hardware; the function* draw() *is the actual volume raycaster. Remember that the interpolation function is given by*

$$\tilde{f}_n = \sum_{i_1+i_2+i_3=n+2} f^c_{i_1,i_2,i_3} \; - \; 2 \cdot \sum_{i_1+i_2+i_3=n+1} f^c_{i_1,i_2,i_3} \; + \sum_{i_1+i_2+i_3=n} f^c_{i_1,i_2,i_3} \; . \tag{15}$$

The functions dataToTexture() *and* calcValueRect() *are subroutines and splitted for clarity and code reuse.*
This version uses the frame buffer for accumulating the function values, not the accumulation buffer. The later can be used for greater accuracy, but it slows down the calculation process significantly.

global $g_x[\cdot]$, $g_y[\cdot]$, $g_z[\cdot]$: *Level in direction of the three axes of all textures*
n_0: *Total number of textures*
n_+: *Number of textures that have a positive contribution to Eq. (15)*

global function **createTextures** *Create textures for a given function* $\mathbf{d}$ *and level* l
in: l: *Level*
$\mathbf{d}$: *Volume data* $[0,1]^3 \to [0,1]$

{
glXMakeCurrent $(\ldots)$ *Make the graphics hardware realize that we need it*
$n_0 := 0, \quad n_+ := 0$
for $(i,j) = [1 \ldots l] \times [1 \ldots l]$ *Count number of textures that have a positive contribution to the function value*
if $(i+j+1 \le l+2)$ *This number is needed in dataToTexture(), thus it has to be calculated first*
$n_+ := n_+ + 1$
for $(i,j) = [1 \ldots l] \times [1 \ldots l]$
if $(i+j+1 \le l)$
$n_+ := n_+ + 1$

for $(i,j) = [1 \ldots l] \times [1 \ldots l]$ *Now create the positive textures*
if $(i+j+1 \le l+2)$
dataToTexture $(n_0,\, \mathbf{d},\, i,\, j,\, l+2-i-j,\, 1)$, $n_0 := n_0 + 1$
for $(i,j) = [1 \ldots l] \times [1 \ldots l]$
if $(i+j+1 \le l)$
dataToTexture $(n_0,\, \mathbf{d},\, i,\, j,\, l-i-j,\, 1)$, $n_0 := n_0 + 1$
for $(i,j) = [1 \ldots l] \times [1 \ldots l]$ *Now create the negative textures, scaled with* 2
if $(i+j+1 \le l+1)$
dataToTexture $(n_0,\, \mathbf{d},\, i,\, j,\, l+1-i-j,\, 2)$, $n_0 := n_0 + 1$
} n_0 *now holds the total number of textures*

global function **draw** *Draw a view of the volume*
in: $h_x,\, h_y$: *Bitmap size*
h_z: *Number of planes to be sampled*
$b[\cdot][\cdot]$: *Bitmap to draw into*
$\mathbf{s}$: *Vector to lower left of back-most plane (starting point)*
$\Delta\mathbf{x},\, \Delta\mathbf{y}$: *Difference vectors to lower right / upper left of back-most plane*
$\Delta_{\mathbf{s}}$: *Difference vectors from back-most to front-most plane for starting point*
$\Delta_{\Delta\mathbf{x}},\, \Delta_{\Delta\mathbf{y}}$: *Difference vectors from back-most to front-most plane for perspective correction*
shade: *Shading function*

{
glXMakeCurrent $(\ldots)$ *Make the graphics hardware realize that we need it*
for $z = [0 \ldots h_z]$ *Render plane by plane, back to front*
calcValueRect $(\mathbf{s},\, \Delta\mathbf{x},\, \Delta\mathbf{y},\, h_x,\, h_y,\, t)$ *Interpolate data into intermediate bitmap* t
$\mathbf{s} := \mathbf{s} + \frac{1}{h_z}\Delta_{\mathbf{s}}$, $\Delta\mathbf{x} := \Delta\mathbf{x} + \frac{1}{h_z}\Delta_{\Delta\mathbf{x}}$, $\Delta\mathbf{y} := \Delta\mathbf{y} + \frac{1}{h_z}\Delta_{\Delta\mathbf{y}}$ *Increment plane vectors*
shade $(t,\, b,\, h_x,\, h_y)$ *Shade interpolated data to bitmap*
}

function **dataToTexture** — *Create one texture and load it into the graphics hardware*
in: n: *Texture number*
$\mathbf{d}$: *Volume data* $[0,1]^3 \to [0,1]$
l_x, l_y, l_z: *Level in direction of the three axes*
f: *Data scaling factor*

{
glBindTextureEXT (GL_TEXTURE_3D_EXT, n) — *Select texture to be loaded*
glTexParameteri (...) — *Set minor texture parameters*
glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE) — *Set modulation mode*

$g_x[n] := l_x$, $g_y[n] := l_y$, $g_z[n] := l_z$ — *Save size for use in calcValueRect()*
$s_x := 2 \cdot 2^{l_x}$, $s_y := 2 \cdot 2^{l_y}$, $s_z := 2 \cdot 2^{l_z}$ — *Textures have to be of size* 2^i *in each direction (see text)*
$t[\cdot] := 0$
for $(i,j,k) = [0 \ldots 2^{l_z}] \times [0 \ldots 2^{l_y}] \times [0 \ldots 2^{l_x}]$ — *Sample and reformat data for TexImage3DEXT()*
  $t\left[k + s_x(j + s_y i)\right] := \frac{f}{n_+} \cdot \mathbf{d}\left(\frac{k}{2^{l_x}}, \frac{j}{2^{l_y}}, \frac{i}{2^{l_z}}\right)$ — *The data has to be scaled by* $\frac{1}{n_+}$ *(see text)*
Now copy data to texture memory
glTexImage3DEXT (GL_TEXTURE_3D_EXT, 0, GL_LUMINANCE16_EXT, s_x, s_y, s_z, 0, GL_LUMINANCE, GL_FLOAT, t)
}

function **calcValueRect** — *Interpolate values in one rectangular area*
in: $\mathbf{s}$: *Vector to lower left of current plane*
$\Delta\mathbf{x}$, $\Delta\mathbf{y}$: *Difference vectors to lower right / upper left of current plane*
h_x, h_y: *Bitmap size*
$t[\cdot][\cdot]$: *Bitmap to be written in*

{
glViewport (0, 0, h_x, h_y) — *Set rendering area*
glClear (GL_COLOR_BUFFER_BIT) — *Clear the frame buffer*
glBlendEquationEXT (GL_FUNC_ADD_EXT) — *Add all textures in the frame buffer when blending*

for $i = [0 \ldots n_0 - 1]$
  $b_x := \frac{1}{4 \cdot 2^{g_x[i]}}$, $b_y := \frac{1}{4 \cdot 2^{g_y[i]}}$, $b_z := \frac{1}{4 \cdot 2^{g_z[i]}}$ — *Calculate the correct texture coordinate bias*
  glBindTextureEXT (GL_TEXTURE_3D_EXT, i) — *Select correct preloaded texture*
  if $(i = n_+)$ — *As soon as all the positive contributing textures are done, start subtracting*
    glBlendEquationEXT (GL_FUNC_REVERSE_SUBTRACT_EXT)

  glBegin (GL_TRIANGLE_STRIP) — *Draw a texture mapped rectangle*
  glTexCoord3f $\left(\frac{s_x}{2} + b_x, \frac{s_y}{2} + b_y, \frac{s_z}{2} + b_z\right)$, **glVertex2f** (0, 0)
  glTexCoord3f $\left(\frac{s_x + \Delta\mathbf{x}_x}{2} + b_x, \frac{s_y + \Delta\mathbf{x}_y}{2} + b_y, \frac{s_z + \Delta\mathbf{x}_z}{2} + b_z\right)$, **glVertex2f** (1, 0)
  glTexCoord3f $\left(\frac{s_x + \Delta\mathbf{y}_x}{2} + b_x, \frac{s_y + \Delta\mathbf{y}_y}{2} + b_y, \frac{s_z + \Delta\mathbf{y}_z}{2} + b_z\right)$, **glVertex2f** (0, 1)
  glTexCoord3f $\left(\frac{s_x + \Delta\mathbf{x}_x + \Delta\mathbf{y}_x}{2} + b_x, \frac{s_y + \Delta\mathbf{x}_y + \Delta\mathbf{y}_y}{2} + b_y, \frac{s_z + \Delta\mathbf{x}_z + \Delta\mathbf{y}_z}{2} + b_z\right)$, **glVertex2f** (1, 1)
  glEnd ()

glFlush () — *Tell graphics engine, we are done*
glReadPixels (0, 0, h_x, h_y, GL_RED, GL_FLOAT, t) — *Fetch result from graphics engine*
for $(i,j) = [0 \ldots h_y - 1] \times [0 \ldots h_x - 1]$
  $t[i][j] := n_+ \cdot t[i][j]$ — *Scale final result back to correct range*
}

Now let us consider the performance of our volume visualization program. Since the time consumption of the X-ray and maximum intensity projection algorithms is data independent and exactly the same, we used the X-ray method on the first data set, the pressure distribution in a cavity flow, to determine the computing times. The iso-surface extraction takes about two times longer than the other methods. Furthermore, the total time of the iso-surface computation depends on the size of the extracted surface. However, our sparse grid algorithms consume the same amount of memory whether they are used for extracting iso-surfaces or for ray casting.

Table 1. Computing times

Sparse grid and combination: CPU-seconds on an SGI O2 with a 195 MHz R10000.
OpenGL: real-time seconds on an SGI Onyx with RealityEngineII graphics pipe,
16 MB texture memory, and 196 MHz R10000.

level	5	6	7	8	9	10	11
sparse grid	755 s	1040 s	1380 s	1935 s	2750 s	3910 s*	5400 s*
combination	83 s	124 s	173 s	233 s	309 s	454 s	726 s
OpenGL	3.6 s	4.5 s	5.5 s	6.8 s	8.4 s	10.3 s	12.5 s
full grid extraction	4.93 s	53.9 s	593 s	1.8 h	20 h*	9.5 d*	105 d*

* estimated

Table 2. Memory consumption

level	5	6	7	8	9	10	11
points of full grid	33^3	65^3	129^3	257^3	513^3	1025^3	2049^3
full grid	128 kB	1 MB	8 MB	64 MB	512 MB	4 GB	32 GB
sparse grid	6 kB	15 kB	35 kB	83 kB	200 kB	450 kB	1 MB
combination	22 kB	59 kB	152 kB	377 kB	914 kB	2.1 MB	5 MB
OpenGL	43 kB	124 kB	338 kB	884 kB	2.2 MB	5.4 MB	13.1 MB
num. low res. full grids	31	46	64	85	109	136	166

In order to estimate the times listed in Table 1, images of resolution 200×200 pixels were computed and 128 sample points per ray were used for the calculation. The pictures in the color plate are rendered in the same resolution as well. In our visualization tool, it is possible to scale such images at nearly no cost and display for instance an image of size 500×500 on screen.

The time measurements of the sparse grid and software combination method were performed on a Silicon Graphics O2 with a 195 MHz R10000 processor. The performance of the hardware based combination technique (OpenGL) was measured on an SGI Onyx with RealityEngineII graphics pipe and 16 MB texture memory. Additionally, we tested this method on an SGI Indigo^2 Maximum Impact with 4 MB texture memory. This machine is slightly slower than the Onyx. Moreover, due to the smaller amount of texture memory, the Indigo^2 can just handle grids up to level 8 in hardware, whereas the Onyx works even with grids of level 11.

The times of the software based algorithms are given in CPU-seconds. However, the times of the hardware based method are measured in real-time seconds because the CPU was not working to its capacity but was waiting until the graphics hardware had finished its computations.

Table 1 shows that rendering times of both software implementations approximately increase by a factor of 1.4 every time the used level rises by one, whereas the measured times of the hardware based implementation increase by a factor of around 1.2 if the level rises by one. This table also reveals the speed up factors of the different algorithms. The combination method is between seven and ten times faster than the sparse grid algorithm. Furthermore, the OpenGL hardware method is between 25 and 60 times faster than the software combination technique. This results in a speed up factor between 200 and 450 from the sparse grid to the hardware based method.

The great benefit of the sparse grid technique is the low number of required grid points. Table 2 shows the memory consumption of a typical data set resulting from a numerical simulation. Assume that a floating point value is given at each grid node. Thus, we obtain the results listed in Table 2.

Table 2 shows that sparse grids are very suitable for compressing huge data sets. Due to this, it is possible to visualize such data even on small workstations.

The last row of Table 2 contains the number of full grids that are combined to a specific sparse grid. In case of the OpenGL implementation, this number is equivalent to the number of used textures.

The combination technique requires more storage than the actual sparse grid algorithm since some of the needed nodes are stored several times. The OpenGL version of the combination methods consumes about twice as much memory as the software version because each of the used textures has to have dimensions that can be written as two to the i-th power. Nevertheless, compared with the original full grid data set, both implementations of the combination technique require a negligible amount of memory.

In this paragraph, we correlate sparse grid compression to the well-known wavelet compression [4, 6]. The advantage of wavelet compression is the fact that the wavelet decomposition is data dependent. That is, the resulting compression is also adapted to the underlying data set. Now let us compare the decomposition and reconstruction processes of wavelets and sparse grids. The wavelet decomposition is rather difficult, whereas the sparse grid decomposition is very easy. On the other hand, the wavelet reconstruction is quite simple, whereas the sparse grid reconstruction, the sparse grid interpo-

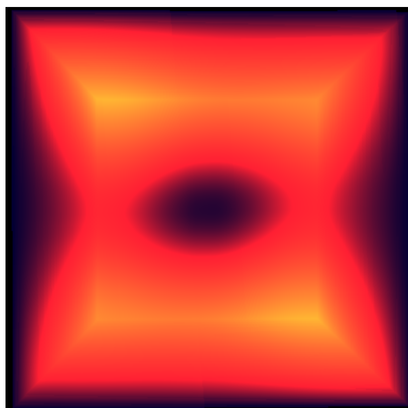


Fig. 9. Sparse grid algorithm

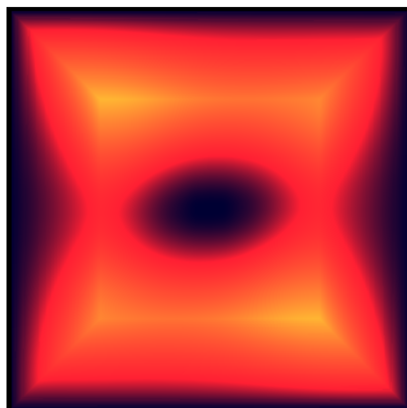


Fig. 10. Combination technique

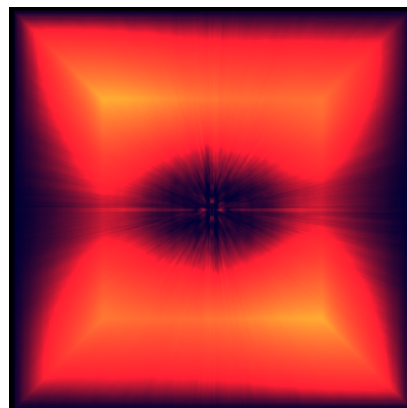


Fig. 11. OpenGL method

lation, is rather complicated and very time consuming. However, the basis functions used in the case of sparse grids are so simple (compact support and piecewise trilinear) that the texture hardware can perform the reconstruction in connection with the combination method. Hence, in the end it turns out that the hardware assisted sparse grid volume visualization is much faster than visualization methods working on other compressed data sets.

Finally, we are going to describe two scenarios where the visualization process can take advantage of sparse grid methods. Firstly, assume that a sparse grid data set resulting from a numerical simulation is given. Then, there are two possibilities to visualize the data set. The traditional approach is to interpolate the data sets once to a full grid. This interpolation process once only takes the time listed in Table 1 (full grid extraction) and generates a huge full grid data set (see Table 2). If the resulting full grid data set squeezes into the machine's memory, fast full grid volume visualization methods can be performed. In contrast to this way, our strategy is to use the OpenGL algorithm in order to get the first images of the data long before the traditional interpolation process will be finished. If the full grid data set does not fit into the main memory, a direct sparse grid visualization method has to be performed anyway.

As a second scenario, assume that a huge full grid data set should be visualized, which results from a numerical simulation running on a mainframe or from extensive measurements. Now, the sparse grid method can be used to compress the huge data set so that it will fit into the main memory of a workstation. Then, it is possible to visualize the compressed data using the techniques presented in this article.

8 Conclusion

We have introduced volume ray casting on sparse grids. This allows to carry out volume visualization directly on sparse grids without transforming the results of numerical simulations on sparse grids to the associated full grids. Note that in real applications it is often impossible to load such full grids of more than 512^3 nodes into the main memory of a workstation.

Secondly, the sparse grid approach can be used as a compression method in order to realize volume rendering in huge data sets on workstations with a small amount of main memory.

By dint of using texture hardware, we have been able to overcome the tardiness of sparse grid interpolation. Therefore, it is possible to use volume visualization on sparse grids interactively, in contrast to other compression approaches.

9 Future Work

There are several directions of future work. The first aim is to implement further transfer functions and lighting models into our visualization program, for instance an emission-absorption model. As a second goal, we intend to use OpenGL in order to accelerate the surface and volume illumination as well. This approach is already used in case of volume rendering on full grids as described in [18, 20, 21].

10 Acknowledgments

We are grateful to S. H. Enger from the Lehrstuhl für Strömungsmechanik of the University of Erlangen for making the cavity data set available to us. Additionally, we would like to thank K. Frank from the High-Performance Computing-Center Stuttgart for helpful discussion. Finally, we wish to thank our colleague R. Westermann for his valuable remarks about OpenGL programming.

References

1. K. I. Babenko. Approximation by trigonometric polynomials in a certain class of periodic functions of several variables. *Soviet Mathematics*, 1:672–675, 1960. Translation of Doklady Akademii Nauk SSSR.

2. H.-J. Bungartz. *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung.* PhD thesis, TU Munich, 1992.
3. H.-J. Bungartz and T. Dornseifer. Sparse grids: Recent developments for elliptic partial differential equations. Technical report, TU Munich, 1997.
4. Chui, C. K. *An Introduction to Wavelets.* Academic Press, Inc., San Diego, 1992.
5. J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. *Transactions Workshop on Volume Visualization*, pages 91–98, 1992.
6. Daubechies, I. *Ten Lectures on Wavelets.* Number 61 in CBMS-NSF Series in Applied Mathematics. SIAM, Philadelphia, 1992.
7. B. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. *Computer Graphics*, 22(4):65–74, August 1988.
8. M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. The combination technique for parallel sparse-grid-preconditioning or -solution of pde's on multiprocessor machines and workstation networks. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Second Joint International Conference on Vector and Parallel Processing*, pages 217–228, Berlin, 1992. CONPAR/VAPP, Springer-Verlag.
9. M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In P. de Groen and R. Beauwens, editors, *International Symposium on Iterative Methods in Linear Algebra*, pages 263–281, Amsterdam, 1992. IMACS, Elsevier.
10. H.-C. Hege, T. Höllerer, and Stalling D. Volume Rendering. Technical Report 93-7, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1993.
11. N. Heußer and M. Rumpf. Efficient visualization of data on sparse grids. In H.-C. Hege and K. Polthier, editors, *Visualization and Mathematics*, Berlin. Springer-Verlag. In preparation.
12. A. Kaufman. *Volume Visualization.* IEEE Computer Society Press, 1991.
13. M. Levoy. Display of Surfaces from Volume Data. *Computer Graphics & Applications*, 8(3):29–37, May 1988.
14. M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.
15. Silicon Graphics Inc., Mountain View, California. *OpenGL on Silicon Graphics Systems*, 1996.
16. Silicon Graphics Inc., Mountain View, California. *RapidApp User's Guide*, 1996.
17. S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics*, 4:240–243, 1963. Translation of Doklady Akademii Nauk SSSR.
18. O. Sommer, A. Dietz, R. Westermann, and T. Ertl. An Interactive Visualization and Navigation Tool for Medical Volume Data. In N. M. Thalmann and V. Skala, editors, *WSCG '98, The Sixth International Conference in Central Europe on Computer Graphics and Visualization '98*, volume II, pages 362–371, Plzen, Czech Republic, February 1998. University of West Bohemia Press.
19. C. Teitzel, R. Grosso, and T. Ertl. Particle Tracing on Sparse Grids. In D. Bartz, editor, *Visualization in Scientific Computing '98*, Wien, April 1998. Springer-Verlag. Proceedings of the Eurographics Workshop in Blaubeuren, Germany.
20. R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing. In R. Yagel and H. Hagen, editors, *Visualization '97*, pages 35–43. IEEE Computer Society, IEEE Computer Society Press, 1997.
21. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics Proceedings*, Annual Conference Series, pages 169–177, Orlando, Florida, 1998. ACM SIGGRAPH.
22. J. Wilhelms and A. van Geldern. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics*, 25(4):275–284, July 1991.
23. C. Zenger. Sparse grids. In *Parallel Algorithms for Partial Differential Equations: Proceedings of the Sixth GAMM-Seminar*, Kiel, 1990.

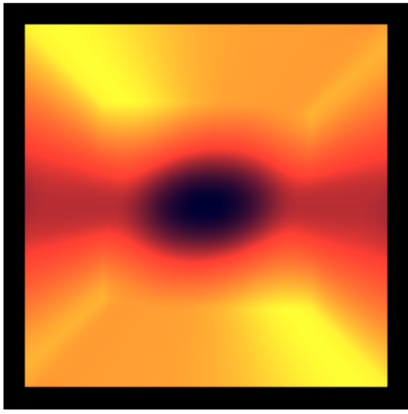


Fig. 12. MIP image of a cavity flow on a sparse grid.

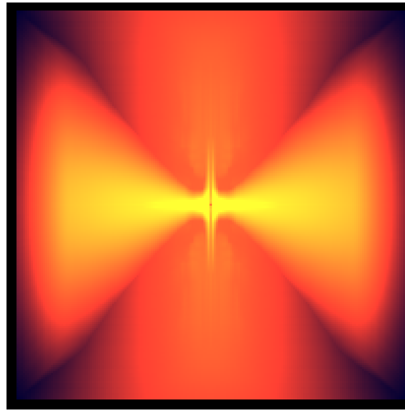


Fig. 13. MIP image of a hydrogen atom on a sparse grid.

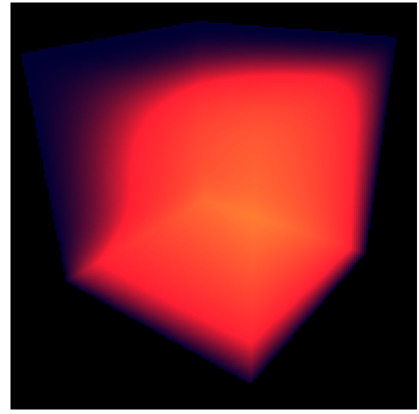


Fig. 14. X-ray image on a sparse grid data set.

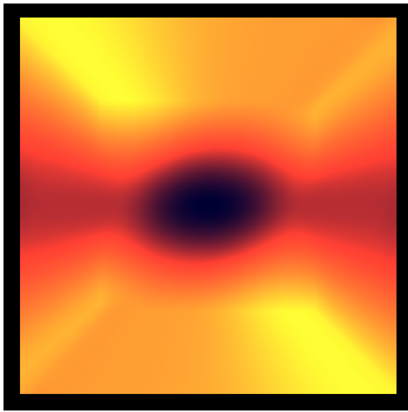


Fig. 15. For comparison: MIP image of a cavity flow on a full grid.

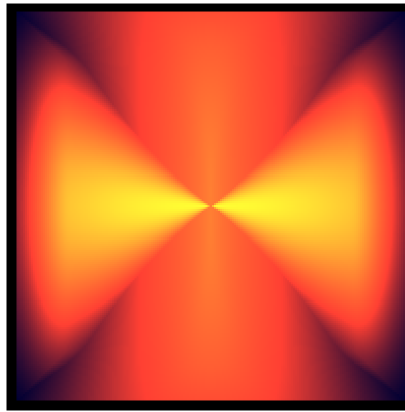


Fig. 16. For comparison: MIP image of a hydrogen atom on a full grid.

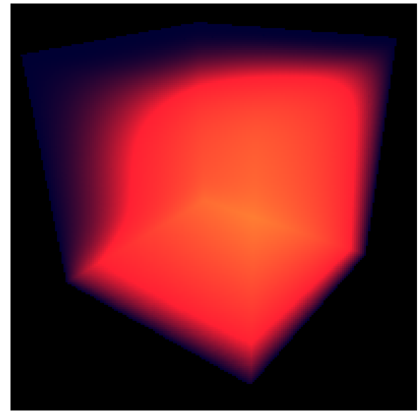


Fig. 17. For comparison: X-Ray image on a full grid data set.

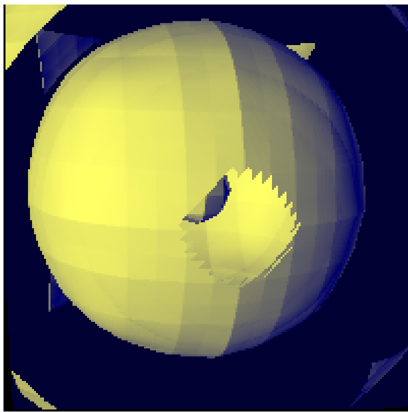


Fig. 18. Iso-surface of an analytical function given on a sparse grid of level 4.

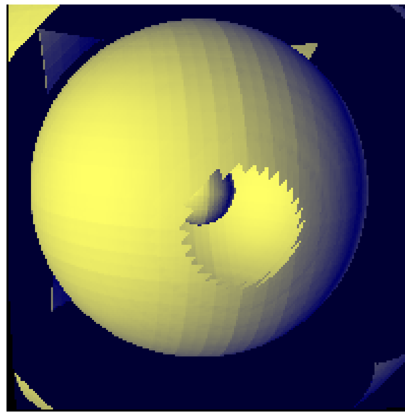


Fig. 19. Iso-surface of an analytical function given on a sparse grid of level 6.

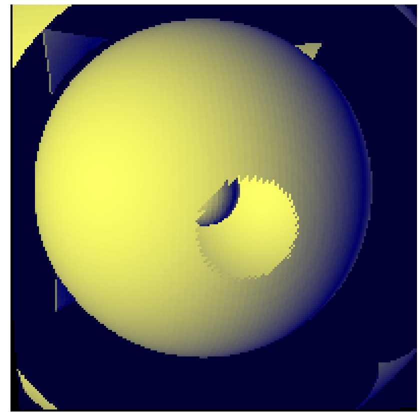
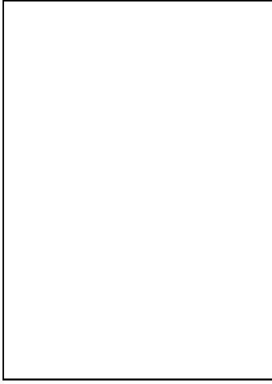
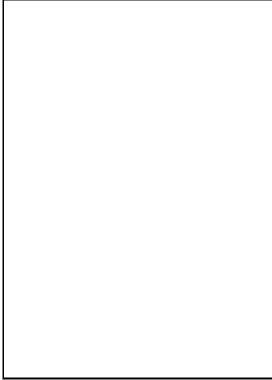


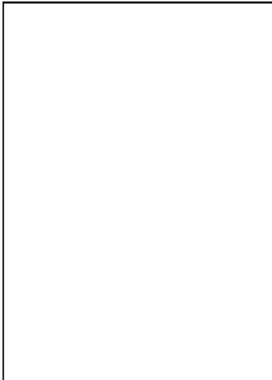
Fig. 20. For comparison: Iso-surface of an analytical function given on a full grid.



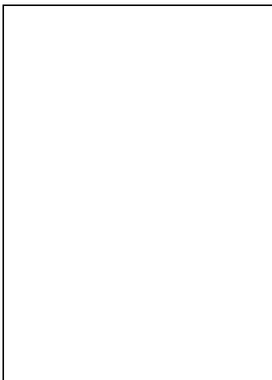
Christian Teitzel is currently a PhD student of computer science at the computer graphics group of the University of Erlangen, Germany. He joined the department in 1996. His research interests cover scientific visualization, hierarchical data structures, adaptive algorithms, computational steering, and interactive three-dimensional flow visualization. In 1995 he received his MS in theoretical mathematics from the University of Bonn, Germany.



Matthias Hopf is a PhD student at the computer graphics group of the University of Erlangen. His research interests include volume rendering, hardware accelerated graphics, and large data sets. He received his MS in computer science in 1998 from the University of Erlangen.



Roberto Grosso is an employee at AEA Technology GmbH in Munich, Germany. Until April 1998 he was a post-doctoral scientist at the computer graphics group of the University of Erlangen. His research interests cover shape design and optimization in CFD, mesh generation, and adaptive mesh refinement. He received his MS in physics in March 1986 from the University of Cordoba, Argentina, and the PhD in theoretical physics in November 1990 from the University of Jena, Germany.



Thomas Ertl is a professor of computer graphics and visualization in the computer science department of the University of Erlangen where he leads the scientific visualization group. His research interests include volume rendering, flow visualization, multi-resolution analysis, parallel and hardware accelerated graphics, large data sets and interactive steering. He received an MS in computer science from the University of Colorado at Boulder, and a PhD in theoretical astrophysics from the University of Tübingen, Germany.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.