Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 30 (2006) 265–276
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Technical section

Interactive out-of-core isosurface visualisation in time-varying data sets

Benjamin Vrolijk*, Frits H. Post

Delft University of Technology, Faculty EEMCS, Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

We present a combination of techniques for interactive out-of-core visualisation of isosurfaces from large time-dependent data sets. We make use of an index tree, computed in a pre-processing stage, which effectively captures temporal coherence in the data set. This tree data structure enables fast extraction of all isovalue-spanning cells from any time step and for any isovalue. For very large time-dependent data sets, such as those resulting from CFD simulations, this data structure can easily become too large to fit in main memory. Therefore, we have adapted the generation of the data structure, as well as the data structure itself for out-of-core application. During generation, the data set is spatially divided into several regions, each resulting in a separate tree. For visualisation, the application uses all these trees simultaneously, but will use only part of each of the trees. Only a user-specified time window will be kept in main memory and other parts of the tree will be read and released on demand. Finally, to avoid time-consuming triangulation and surface reconstruction, we have used a hardware-assisted direct point rendering algorithm for displaying the isosurfaces. These combined techniques allow interactive exploration and visualisation of very large time-varying data sets on a normal PC.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Scientific visualization; Large data handling; Isosurfaces; Out-of-core techniques; Time-dependent data

1. Introduction

One of the greatest challenges in visualisation today, is the interactive exploration of large, time-varying data sets. Especially in areas such as flow visualisation, time-dependent simulations are becoming common practice, and can produce high resolution grid data sets with many thousands of time steps. In spite of the huge size, scientists investigating these data sets need interactive visualisation techniques with which they can browse through the data in both space and time.

Flexible, general-purpose visualisation techniques such as particle tracing, volume rendering, or isosurface extraction are in general not fast enough for time-dependent exploration, or for interactive control of the visualisation parameters. For example, when using isosurface extraction for a time-varying data set, it is desirable to interactively change the isovalue, and watch the development of the surface shape over time. However, extracting and rendering a new isosurface for each time step is generally too slow for interactive exploration.

Our approach to this challenge is to use a specialised data structure allowing very fast access and data retrieval for answering a specific type of visualisation query. We used a number of criteria in choosing such a

*Corresponding author. Fax: +31 15 278 7141.
E-mail address: B.Vrolijk@ewi.tudelft.nl (B. Vrolijk).

0097-8493/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2006.01.032

data structure. First, it should do fast isosurface extraction for any isovalue. Second, it should be suitable for time-dependent data sets. Combining these two, it should be possible to do time-dependent or ‘‘incremental’’ surface extraction, or to determine the differences between successive time steps. This means the data structure should exploit temporal coherence in the data. Of course, it should be much faster than straightforward isosurface extraction from every time step separately. Finally, the results of the extraction should be directly passed to a fast rendering algorithm for display.

We have employed a data structure for fast isosurface extraction from time-dependent data sets [1]. To make our system achieve interactive frame rates in browsing a data set, we have directly linked the output of our isosurface extraction with a fast, hardware-supported direct rendering algorithm [2], resulting in interactive isosurface extraction and visualisation from time-varying data sets. The direct rendering avoids the time-consuming construction of polygonal surfaces using a Marching Cubes-type of algorithm [3]. By combining these two methods, and capitalising on temporal coherence, the user can specify an arbitrary isovalue and time step, and the development of the isosurface can be dynamically visualised in forward or backward time direction (see Fig. 1).

However, the tree data structure used may become too large to fit in main memory. We have overcome the huge memory requirements for creation and use of this data structure. For this, we have adapted the data structure for out-of-core application. We designed and

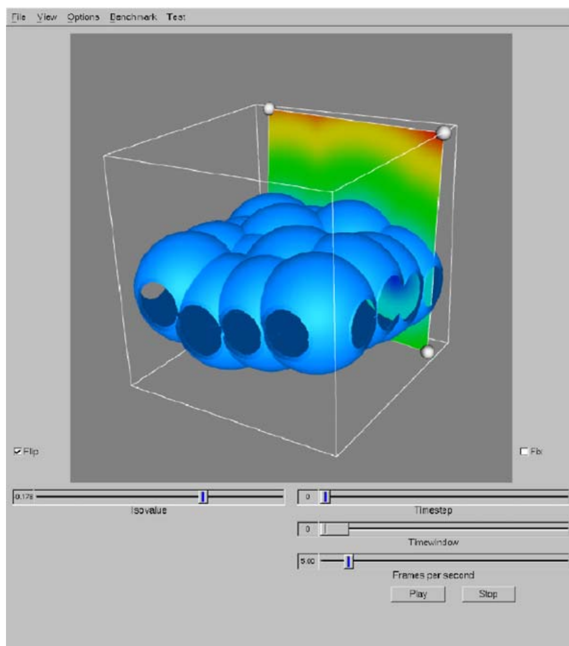

Fig. 1. A 256^3 data set of air bubbles rising in water.

implemented an intelligent paging scheme to enable interactive out-of-core isosurface extraction and rendering on a regular pc.

This paper is organised as follows. In Section 2, we discuss related work in isosurface extraction techniques from time-dependent data, suitable rendering techniques to display the isosurface, and out-of-core techniques. Then we will briefly explain the data structures we have used in Section 3, together with the out-of-core algorithms and adaptations in Sections 4 and 5. The results will be discussed in Section 6, and we will give our conclusions and directions for future work in Section 7.

2. Related work

Many techniques for fast isosurface extraction are based on tree representations. Sutton and Hansen introduced the Temporal Branch-on-Need Tree (T-BON) [4]. This is an extension to the original Branch-on-Need Octree (BONO), described by Wilhelms and Gelder [5]. The T-BON is a version for time-dependent data sets, but it does not make use of temporal coherence. The data structure is suitable for fast isosurface extraction.

Shen presents an algorithm for fast volume rendering of time-varying data sets, using a new data structure, called the Time-Space Partition (TSP) Tree [6]. This structure could also be adapted for fast isosurface extraction. The TSP tree is capable of capturing both spatial and temporal coherence in a time-dependent field. Both the spatial and temporal domain are represented hierarchically in the TSP tree: each node of the octree representing space, contains a full bintree representing time. Although this allows multi-resolution access in any dimension, it involves a huge storage overhead.

Shen describes another data structure for isosurface extraction from time-varying fields, called the Temporal Hierarchical Index Tree [1]. The idea behind this structure is to store voxels that remain approximately constant throughout a certain time span only once for that entire time span. Within this data structure, two other data structures are used. First, the Span Space representation, as introduced by Livnat et al. [7], is used to store intervals in a two-dimensional space. Second, Interval Trees, described by Cignoni et al. [8], provide an optimal interval search algorithm.

Bordoloi and Shen [9] presented an algorithm for storing intervals more efficiently than in the Span Space, using transform coding.

Recently, Gregorski et al. [10] presented a technique for progressive isosurface extraction with adaptive refinement from compressed, time-dependent data sets. However, they are restricted to playing forward and

backward in time. The vertex programming capabilities of modern graphics hardware are used to speed up the surface extraction.

Pascucci also uses the vertex programming capabilities of modern graphics hardware [11]. In his approach, the workload is distributed between the CPU and the graphics card. A tetrahedral decomposition of the domain is used. The application draws one quad per tetrahedron; the vertex program on the graphics card does the interpolation to find the position of the vertices of the isosurface, and computes the normal of the isosurface.

For our purposes, we decided to use and extend the Temporal Hierarchical Index Tree by Shen [1]. We will describe this structure in more detail in the following sections.

We have made an implementation of this data structure with optimisations for space efficiency. We have created search routines for retrieving the isosurface-spanning cells for any isovalue and from any time step, and specialised *incremental* search routines that allow an even faster cell search from any time step, given the previous results from another time step [12].

We wanted to overcome the huge memory requirements both during creation of the data structure and in interactive visualisation, when the data structure is used. Therefore, we have designed a paging scheme for this tree data structure that makes out-of-core tree building and extraction possible for very large data sets. In the application program, this data structure is suitable for paging per time step, unlike for example the TSP tree. Recently, Chiang presented a technique for out-of-core isosurface extraction from time-varying fields [13], which uses as the basic data structure a time tree similar to the one described here. The underlying structures of his technique are however optimised for I/O and out-of-core computation. We have focused on both fast extraction and rendering and afterwards adapted the data structure and added the paging scheme. We did not try to create I/O-optimal data structures and algorithms; we used this scheme because it suits our data structure. For an overview of out-of-core algorithms for computer graphics and visualisation, we refer to the survey by Silva et al. [14].

For visualisation we developed two separate point-based rendering techniques. The first, ShellSplatting, is a hardware-accelerated direct volume rendering method that is based on a combination of splatting [15] and shell rendering [16]. The second is a much faster, but lower quality, point-based volume rendering method that was created specifically for the isosurface extraction documented in this paper. The points are displayed as opaque, flat-shaded polygons that are parallel with the viewing plane. This is an extreme simplification of systems like QSplat [17] and object space EWA surface splatting [18].

Both rendering techniques have been tightly coupled with the extraction technique. The cells that result from the search routines are fed directly into the rendering algorithm, without the need for retrieving the raw data or having to perform interpolation or triangulation. This high level of integration between extraction and rendering is an important advantage of our technique.

3. Temporal index tree

Isosurface extraction involves searching the cells that are intersected by the isosurface, which means that they contain the isovalue. Therefore, each of these cells must be enclosed by vertices of which at least one has a scalar values lower and at least one has a scalar value higher than the isovalue. To check if a cell is intersected by the isosurface, it is sufficient to store the extreme values of the cell.

It is the main idea for the data structure we will describe next, that for each cell only an interval $[\min_i, \max_i]$ is stored. To check if a cell is an isosurface cell, we check if the isovalue is contained in that interval.

We have used and modified the Temporal Hierarchical Index Tree data structure [1]. This data structure makes use of temporal coherence in the data set by storing cells that remain (approximately) constant over a certain time span, only once for that time span.

The basic structure of our index tree is a (Branch-on-Need) binary time tree in which each node represents a time range—the root node represents the data set's entire time range, the leaf nodes represent the individual time steps. (See Fig. 2.) To retrieve the data for a particular time step, we will need to traverse the tree from root to leaf nodes and collect the data found in each node.

In each of the nodes of this binary tree, cells are stored that remain (approximately) constant for that time span. This implies that those cells need not be stored anywhere below that node. Any descendant of a node represents a

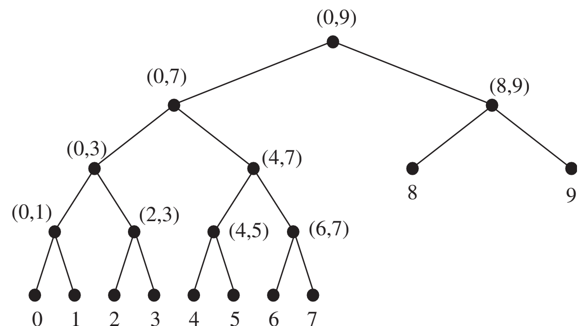

Fig. 2. An example of a binary time tree for 10 time steps.

sub-span of that node's time span, so there is no need to store the cell in the descendant. This is the cause of the potentially large data reduction that can be achieved with this tree structure. Of course, it very much depends on the amount of temporal coherence in a data set. In the worst case, there will be no coherence between successive time steps and all the data will be stored in the leaf nodes of the tree. No data reduction can be achieved in that case. On the other hand, the best possible compression will be achieved when all time steps are similar—all the data will be stored in the root node, and the total amount of data will be equal to the amount of one time step.

3.1. Tolerance

We need a tolerance criterion to determine when a cell is considered constant. We have implemented two different criteria, which we call *absolute* and *relative*, from which the user can choose. However, more can be devised and easily implemented. In both cases, the user can specify a tolerance percentage.

In the absolute criterion the tolerance used will be the given percentage of the entire data set's min–max-range. More precisely, we compute the tolerance as follows:

$$T = t \times (\max_g - \min_g), \tag{1}$$

where t is the user-provided tolerance percentage; $\max_g$ and $\min_g$ are the data set's global minimum and maximum. This criterion can be used when the data values will be more or less evenly distributed; for every data value, the same tolerance will be used.

In the relative criterion, the given percentage will be taken relative to the current value. For example, a relative tolerance of 1% means that a value of 1.0 may deviate by $\pm$ 0.01, but a value of 10.0 may deviate by $\pm$ 0.1. This type of criterion can be used if the data can be expected to be distributed around a certain value, or if one data value is likely to be picked for the isovalue.

The value of the tolerance parameter influences the construction of an index tree. The results (the size of the data structure, the accuracy of the data structure, and with that the accuracy of the rendering, and the speed of the isosurface cell search) strongly depend on the value of this parameter. But even if the connection between the parameter and the results is intuitively clear, it is not obvious to state the best possible value.

The parameter determines the amount of variance a cell is allowed to have over time and yet be called constant. Obviously, this value should not be too large, otherwise important variations will not be apparent in the results. The value should also not be too low, because then little or no temporal coherence will be found, which would undermine the whole purpose of the binary time tree.

On the other hand, the amount of temporal coherence is a property of the data set. In a very "turbulent" data set, there will be hardly any coherence, but in a "stable" data set, there will be very much.

Ideally, we would like to quantify the overall amount of coherence in a data set and automatically determine the best tolerance parameter for that amount.

Even if we could do this, the question remains what is best. There are several criteria to choose from. For example, the best case with respect to the size of the data structure would be to choose a very high tolerance. This will cause many cells to be called constant and therefore result in all cells being stored in the root node of the binary tree. We get the best possible compression, but the worst possible accuracy. On the other hand, the best case with respect to accuracy would be to have the tolerance set to 0. This will result in most of the cells being stored in the leaf nodes. The compression will be almost 0 and the speed will be very low.

For our purposes, the optimal case will be somewhere in between, having a good amount of compression, but also still a good accuracy. In fact, the highest search speed for a single time step will be obtained when all tree nodes contain about the same number of cells. This is what we will aim for.

3.2. Index tree building

When creating an index tree, we start by building the structure of the binary time tree. Note that this structure is determined a priori, only by the number of time steps. Therefore, the time ranges which are represented by each node of the tree are fixed.

From the structure of this tree, and the possible time ranges that can be stored in each node of the tree, we can start classifying the cells in the data set. For each cell, we generate a time-vector v containing the cell's values for each time step. Given this 0D time-dependent data set, (being the temporal evolution of one cell) we start traversing the binary time tree. Referring to Fig. 2, we start at the root node, with time range $[0, 9]$, and check if the current cell remains (approximately) constant for this range. For this we need the cell's time-vector $v[0]$ to $v[9]$. If the cell satisfies the tolerance criterion for this time range, we store the cell in the current binary tree node. If not, we recursively descend the tree and check the child nodes $[0, 7]$ and $[8, 9]$.

To determine if a cell remains constant over a certain time range, we compute its *temporal* extreme values. Because a cell itself contains a minimum and a maximum value, we compute temporal extremes for both the minimum and the maximum. These will be treated similarly. Let us call the minimum L, for *left extreme*, and the maximum R, for *right extreme*. Then,

using the absolute criterion, we would check if

$$(\overset{t}{\max} L_i - \overset{t}{\min} L_i) < T,$$

$$(\overset{t}{\max} R_i - \overset{t}{\min} R_i) < T,$$

where $\overset{t}{\max}$ and $\overset{t}{\min}$ mean the temporal extremes, and L_i and R_i mean the two spatial extreme values stored in cell i.

For the relative criterion, we would check if

$$(\overset{t}{\max} L_i - \overset{t}{\min} L_i) < T \cdot \|\min^t L_i\|,$$

$$(\overset{t}{\max} R_i - \overset{t}{\min} R_i) < T \cdot \|\min^t R_i\|.$$

Note that if the cell is stored, for example, in node [8, 9], meaning that the above criteria return true, its value is called constant for time steps 8 and 9, and there is no need to store the cell in either of the nodes [8] and [9].

Note also that a cell that remains constant for the time range [0, 5], for example, will be stored in the two nodes [0, 3] and [4, 5], because there is no node for the range [0, 5]. Moreover, we would not even know the cell remains constant for that time range, because that range will never be checked. Only the ranges that are represented by binary tree nodes will be checked.

If a cell is stored in a node of the binary time tree, we store the cell's ID and extreme values. For non-leaf nodes, representing a time range instead of a single time step, this means we have to store the *temporal* extreme values of the cell. For rendering, we also need a normal (or normalised gradient) for each cell. Non-leaf nodes represent a time range, meaning there are several normals to choose from. Currently, we pick one, for example the middle one; an average normal might be better, although we have not noticed any artefacts with our method.

After all cells have been classified and stored in the binary time tree, they will be reorganised per tree node. In each of the tree nodes, we have maintained a vector of cells. For efficient interval searching, these cells will be rearranged into an *Interval Tree* [8]. This is where we deviate from the original data structure as described by Shen [1]. He uses another data structure, called the *Span Space*, in which intervals are stored as 2D points. We have also implemented this data structure, but eventually rejected it, the biggest disadvantage being that it is very unintuitive. Furthermore, the Span Space in combination with the Interval Trees, as in the original article, does not result in much higher search speed or more efficient storage, than just using Interval Trees.

In our structure, one interval tree will be created in each binary tree node. This structure and its construction are fairly straightforward and described in detail by Cignoni et al. [8] and in our previous work [12]. Therefore, we will not discuss it any further in this article.

4. Out-of-core tree building

During the creation of the index tree, we have to iterate through all cells in the data set, determine a time-vector for each cell and store each cell in the right node(s) in the index tree. We need a time-vector for each cell, because we want to determine the time spans in which the cell remains constant. This poses a number of problems.

4.1. XYT files

From a practical point of view, the usual way in which time-dependent data sets are stored, is not very well-suited for this purpose. Normally, for each time step a field is stored in a separate file, so the same grid point in different time steps can be located at the same offset in different files. When we need to construct a time-vector for a certain cell, we have to open and search in all files. Either we have to keep all files opened simultaneously, or open and close all files for each cell.

We decided to transpose the entire data set in pre-processing. Instead of storing (x, y, z) data in each file and a separate file for each time step, we transformed the data set to files with (x, y, t) data and with each file representing a different z. Here we assume a regular, Cartesian grid. Of course, other subdivisions are possible, as long as the temporal data for one grid point can be obtained from a single file.

4.2. Multiple trees

There is another obstacle when constructing the index tree. Each cell's time-vector is split into a number of time ranges over which the cell remains constant. Then, the cell is stored in the index tree nodes corresponding to those time ranges. In each of these nodes, the cell is appended to a (potentially very large) list of cells for that node. Because each cell may be stored in a number of index tree nodes, it is essential, for efficiency reasons, that we should try to keep the entire index tree in main memory during construction.

Of course, it could be possible to store each index tree node separately on disk during construction, but that would involve a lot of extra disk I/O for every single cell.

Instead we decided to keep as much as possible in main memory, but split the entire data set into several trees. Because of the file layout just described, we decided to split the data set in the z direction. For example, for a 256^3 data set, we have 256 files, each containing (x, y, t) information for a different z value. We could then create 16 index trees, each for a layer of 16 z slices thick. Or 32 trees of 8 slices, or 4 trees of 64 slices. Each of these trees separately can be kept in main memory, so there is no unnecessary disk I/O during construction. When a layer has been completed, the

current index tree can be written to disk and cleared from memory, and the next z slice will start a new index tree.

This approach does not give any noticeable space overhead compared to using a single tree. The total disk space needed for all trees is equal to the space needed for a single full tree. In the application program almost no extra processing is required to use multiple trees compared to using a single large tree. In fact, the application just iterates over the array of trees and performs the same function for each individual tree. For an isosurface query, the cells returned by the queries on the individual trees are then concatenated.

Another advantage of using multiple trees, is that we can easily add trees that represent other quantities, meaning we can render several isosurfaces simultaneously. As an extension to this concept, we can also reuse a single tree, enabling us to render several isosurfaces with different isovalues from a single quantity. To the program, these concepts are almost identical. In all cases, we iterate over the array of trees, and for each tree we perform an isosurface cell search. In one case, the isovalues will be the same for each tree, but every tree represents another piece of the data set. In another case, the trees will actually represent the same data, but we will search for different isovalues (see Fig. 3) and the last case is that we have different trees, each representing another quantity, and each having its own isovalue.

A fourth case is possible, when we have trees representing the same data set and the same quantity, but a different time range. Of course, this is sub-optimal. We should always try to use the data set's entire time range, in order to find the most temporal coherence. If, somehow, it would not be possible to create index trees for the entire time range, for example because the simulation is still running, it is possible in this way to visualise a time range using several trees. The time ranges for the individual trees will be joined to form one large time range. The difference with the other cases is that at any time only a single index tree will match the current time step, whereas in the other cases all index trees represent the entire time range and thus all trees will always match the current time step. Again, not using the entire time range is sub-optimal. For example, if a data set consisting of 100 time steps is divided into 10 index trees of 10 time steps, it is not possible to find any temporal coherence with a length of more than 10 time steps. This will obviously have a negative impact on the compression ratio.

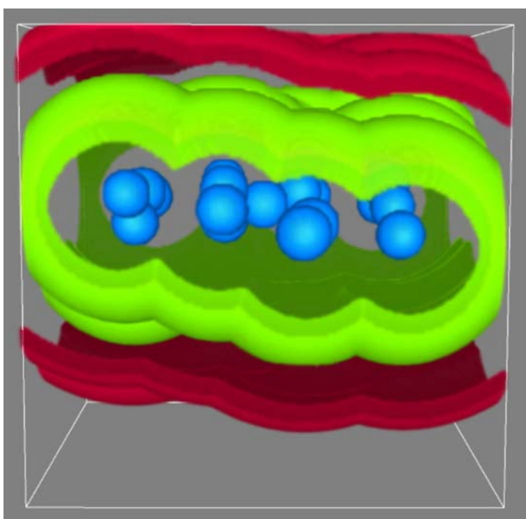

Fig. 3. Several trees can be loaded simultaneously, either representing different quantities, different time ranges, or—as shown in this image—different isovalues.

5. Out-of-core visualisation

5.1. Time window

During visualisation, we may have a different memory problem. Although we can split the data set into layers during construction of the index tree, we cannot do this during visualisation, as we would like to see the data set's entire spatial extent at once. Therefore, we will have to read all constructed index trees simultaneously. However, we do not need to have all time steps in memory at the same time. We created an intelligent paging scheme that allows us to read only a limited number of consecutive time steps from the index trees into main memory. The structure of the index tree enables us to incrementally read new time steps and remove old time steps from memory. This is the basic idea of our sliding time window concept. We assume this corresponds very well to the way scientists will browse through the data set. We assume the user will normally play through the time steps coherently, either forward or backward. This corresponds to our incremental reading of new time steps. Sometimes, the user may browse a few frames backward and forward, within the time window. If a large jump in time is requested, this will cause a delay, because a completely new time window will have to be read. We assume this is a minor problem, because random jumps in time will probably not happen very often.

As an example, let us take the same binary time tree as in Fig. 2, representing a data set with 10 time steps. Assume we have a time window containing the 5 time steps [3, 7]. In Fig. 4 we have depicted which nodes of the tree will be in main memory. If the time window is shifted one time step to the right, the nodes [8, 9] and [8] will have to be read from disk. The node [0, 9] does not have to be read from disk because it is already in main

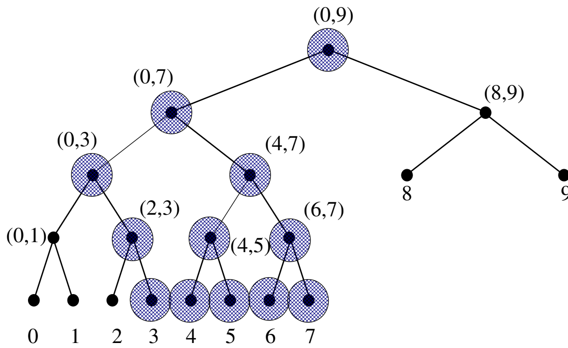

Fig. 4. A binary time tree with a time window [3, 7]. Only the colored nodes will be kept in main memory.

memory. In fact, the root node will always be in main memory, because it is needed for all time steps. While we need to read extra tree nodes from disk on the one side, we can remove nodes from memory on the other side: time step 3 is no longer needed, therefore, we can delete the nodes [0, 3], [2, 3] and [3] from main memory.

Knowing this, we can define the memory requirements for our algorithm. Evidently, the minimal amount of memory needed is just the space that is needed to store a single time step.

5.2. Adaptations to the data structure

To make the time window possible, a number of changes have been made to the index tree data structure. The skeleton of the binary time tree will always be kept in main memory. This means that the tree *structure* will always be available, together with the time span information in each node. Using this structure, we can easily identify which tree nodes have to be traversed to obtain the data for a certain time step. Only the cell data, which is stored in the interval trees in each of the nodes of the index tree, will be eligible for paging to and from disk.

Each index tree is stored as one large binary file on disk. When we have to read the interval tree for a specific index tree node from disk, we need the file offset and the number of bytes to read from our disk file. These two numbers are stored with each node of the index tree.

In fact, the number of bytes is not necessary for reading an interval tree from disk. The entire index tree, but also the individual interval trees will be reconstructed in memory on the fly, while reading from disk. Therefore, we do not have to know in advance the number of bytes to read. However, the number *is* necessary when the interval tree is *not* read; when only the structure of the index tree is read, without the data in the interval trees, it is necessary to know how many bytes to skip to find the file offset for the next tree node to read.

This way, the *structure* of the index tree can be read entirely from disk, without reading any *data*. The tree then occupies only a few hundred bytes in memory. Next, whenever a time step is requested, the appropriate tree nodes will be read from disk.

When a tree node has to be freed, the interval tree for that node is simply removed from memory.

Of course, we must keep track of which tree nodes currently are in main memory. To this end, we have added a single boolean variable to each tree node. We say a tree node is in memory if the interval tree for that tree node is in memory. Again, the index tree *structure* remains in memory at all times, and therefore, all index tree nodes also exist in memory permanently. Only the interval tree in an index tree node can be paged in and out of memory.

5.3. GUI feedback

To provide the user with feedback about the time window, we have designed a GUI element to show a bar from the first to the last time step of the window, with an indicator at the current time step. (See Fig. 5.)

Because reading new time steps will certainly be slower than visualising them, the visualisation will, in the end, catch up with the last time step of the time window. This is of course dependent on the size of the time window and on the frame rate of the player. It can happen that the user will notice a delay. Therefore, it is desirable for the user to get feedback. He will see that the visualisation is catching up with the reading of new time steps and be prepared that he will have to wait. Or, he could slow down the visualisation by lowering the frame rate. Finally, he could also increase the time window if memory size allows this.

5.4. Multi-threading

In order to let the visualisation run independently from the reading of time steps from disk, to prevent unacceptable delays, we have decided to use a multi-threaded design for our program. The main thread of the program is concerned with the visualisation. When a certain time step is selected by the user this thread ensures that the part of the index tree containing that time step is resident in memory. If necessary, it will read

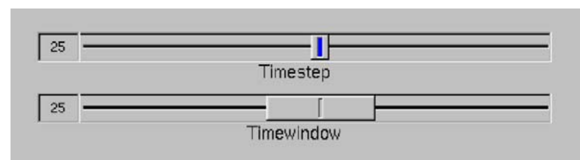

Fig. 5. The GUI element that shows the time window and current time step.

the corresponding tree nodes from disk. Next, it will perform an isosurface cell search and visualise the result. In the mean time, the second thread is awakened and this thread will start reading new time steps, from the current time step in both time directions, until the requested number of time steps (specified as the window size) has been read in from disk. This could take some time, especially if all time steps in the time window have to be read, but as it happens in a separate thread, the user might not notice anything, while he is investigating the current time step. Of course, when the user simply plays through the data set, only one time step will have to be read at a time, in which case interactive browsing is quite feasible. A third thread has been designed to perform the task of cleaning up unused time steps. This thread will remove all nodes in the tree that are not needed for the current time window.

These last two threads both consist of an infinite loop in which they are suspended while waiting for a signal. The main thread broadcasts the signals whenever a new time step is selected.

5.5. Point-based direct rendering

After extracting the cells intersected by the isosurface it would be possible to construct a polygonal mesh for each frame and visualise this using polygon rendering. However, this would take away the advantage of the fast access data structure, as the original data would have to be read from disk in order to perform surface reconstruction using for example the Marching Cubes algorithm [3].

To avoid this, we have used a point-based direct rendering algorithm [12]. We further optimised our *ShellSplatting* rendering algorithm [2], a combination of shell rendering and splatting, to take advantage of the a priori knowledge that the voxels we are dealing with are completely opaque and together constitute an isosurface. ShellSplatting makes use of special data structures that enable fast implicit space leaping and back-to-front or front-to-back traversal from any viewing angle. This ordering is very important as the technique makes use of Gaussian textured polygons that are composited and scaled by graphics hardware.

The ShellSplatting technique yields high quality renderings of the extracted isosurfaces. However, due to the nature of the data structures used, the voxels have to be ordered in at least the fastest-changing dimension and this slows down the data conversion stage. We wished to provide a second, much higher speed rendering option.

By opting to use flat-shaded rectangular polygons instead of Gaussian-textured ones, the ordering constraint could be ignored. In return, the rendering quality would be slightly lower. In this second method, the polygon that is to be used for rendering the cells is calculated in the same way as for ShellSplatting.

The polygon is constructed to be parallel to the viewing plane. This is correct for parallel projection. Strictly speaking, in the perspective projection case each rendered polygon should be perpendicular to the viewing ray that intersects it. However, for efficiency reasons, we make use of slightly larger screen-aligned polygons [19]. The details of the construction are described in our previous work [12].

6. Results

We have tested our application on two large data sets. The first data set is of a multi-phase flow simulation of a number of air bubbles rising in water. Five double-precision floating point values are computed per grid point: the pressure, the level set value and the three components of the velocity. We use only one scalar to create the index tree, being the level set value; this leaves us with 128 MB of data per time step. See Fig. 6.

Another data set we used is of a Large Eddy Simulation of cumulus clouds, with one vector and three scalar quantities: the air velocity vector, meteorological temperature, liquid water and total water. For the creation of an index tree, we only used the temperature. See Fig. 6.

For each of these data sets, we created two sets of index trees; the details are in Table 1.

6.1. Benchmarks

For two of the data sets (the second and fourth from Table 1), we ran a couple of benchmarks. First we ran a rendering benchmark, both with the ShellSplat renderer and the Fast Point-Based renderer, for different isovalues, meaning different numbers of cells to render. In the other two benchmarks we measured the speed at which we could play through the data set. This involves both extraction and rendering for each time step. This was done for a (worst case) time window of 1, meaning that each time step has to be read from disk before extraction can be done, and for a very large time window. In the latter case, all data is kept in main memory and no data transfers from disk are needed. This is done to test the speed of the extraction algorithm. When we use the ShellSplat renderer, a sorting step is needed for each time step. To see the influence of this sorting, we have performed the last benchmark with both renderers.

We ran the benchmarks on a modern computer with an Intel Pentium 4 processor, running at 3.0 GHz, and 1 GB of main memory. The graphics card is a NVidia Quadro FX 1300 with 128 MB of memory on a PCI Express graphics bus.

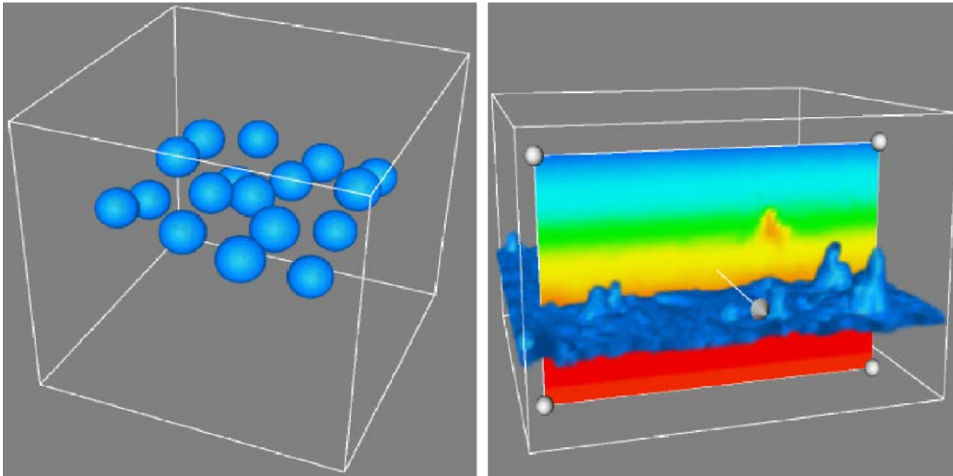

Fig. 6. Scenes from the two data sets. On the left is the bubble data set, on the right is the cloud data set.

Table 1
Details of the two data sets and of the four generated index trees

Data set	Bubbles		Clouds	
Resolution	$256 \times 256 \times 256$		$128 \times 128 \times 80$	
# Time steps	39		600	
Raw data size	4992 *MB*		3000 MB	
# Index trees	16	8	6	8
xy-resolution	256×256	256×256	128×128	128×128
z-resolution	16	32	80	10
# Time steps	39	39	100	600
Total size	3170 MB	1630 MB	824 MB	750 MB

The results of the rendering benchmarks are shown in Fig. 7. It is clear that interactive rendering is possible with the Fast Point-based Renderer, even for over 400,000 cells. Also the Shell Renderer can achieve interactive frame rates up to about 100,000 cells. Because of the texturing and compositing, the Shell Renderer is much slower than the Fast Point-based Renderer.

Next, we timed at which rate we could play through the entire data set. This involves extraction and rendering for every time step, using the same isovalue. With a time window of 1, only a small amount of main memory is needed, but for every frame, we have to read a new time step from disk into main memory and delete the previous time step from memory. The speed is therefore very much dependent on the amount of data that is to be read per time step. The cloud data set, consisting of 600 time steps, occupies a total of 750 MB on disk, or on average 1.25 MB per time step. We can play through the entire range of 600 time steps at an average rate of 7.8 to 9.5 frames per second, depending on the number of cells to render.

The bubble data set, on the other hand, with only 39 time steps and occupying 1.6 GB on disk, has an average of almost 42 MB per time step. Playing this data set with a time window of 1 is not really interactive, with an average frame rate of about 0.46 FPS.

However, if there is more memory available, it should obviously be used. Therefore, we also tested the speed at which we could play through the data within a large time window. We used a fixed time window which could be completely stored in main memory; no disk transfers were needed whatsoever. Because the rendering again depends on the number of cells, we ran the benchmarks with different isovalues. The results are shown in Fig. 8.

As discussed in Section 5.1, playing randomly in time is also possible, although naturally slower than playing consecutive time steps. This is, however, difficult to

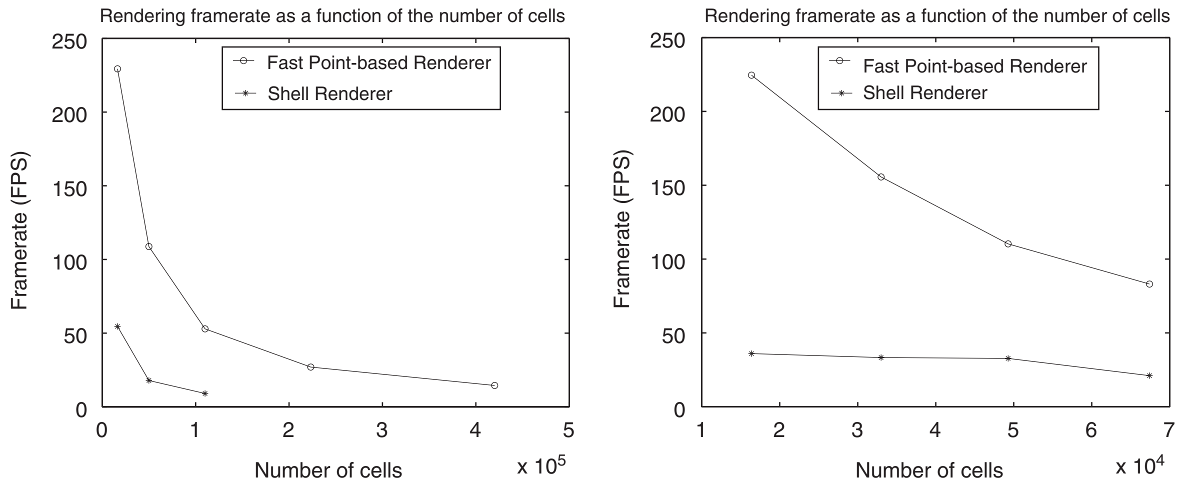

Fig. 7. The results of the rendering benchmarks. On the left is the bubble data set, on the right is the cloud data set.

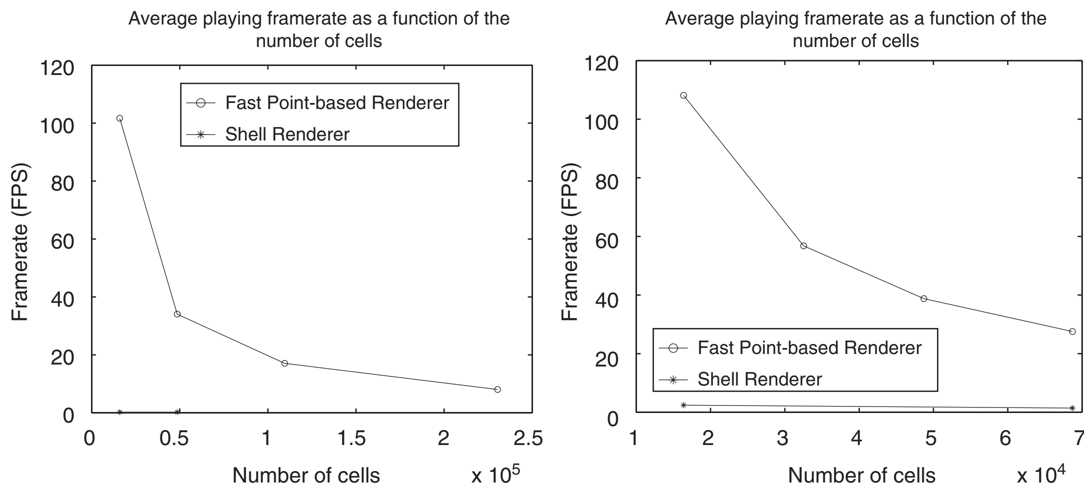

Fig. 8. The results of the play benchmarks. Playing involves extraction and rendering through (part of) the time range. On the left is the bubble data set, on the right is the cloud data set.

benchmark, as the speed will depend upon *which* random time step is selected, and whether or not this time step is within the time window. The size of the time window determines the chance of a "hit" or "miss". As long as a time step within the time window is chosen, visualisation will be interactive. In fact, the isosurface cell search will be significantly slower, because of our incremental search algorithm [12], but the rendering will remain the bottleneck. The achieved playing frame rate will, therefore, be only slightly slower (about 10 or 15%). As an example, the *extraction* frame rate for the 600 time step data set can be more than 4000 FPS, searching through consecutive time steps. The *playing* frame rate will be about 35 FPS. When browsing randomly through these 600 time steps, the extraction frame rate will drop to about 180 FPS. However, the rendering speed will not be influenced, therefore, the playing frame rate will only fall to about 30 FPS.

When a time step outside the time window is selected, disc access will be required. The amount of data that has to be read will determine the update speed. In turn, this amount of data will depend on the data set and the temporal coherence in the data and between the current and the newly selected time step. The update rates will be comparable (but slightly lower) than those discussed above, for playing consecutive time steps with a time window of size 1.

Extraction of the isovalue-spanning cells can be done extremely fast. Rendering is also very fast, as long as we do not need the sorting step to create the shell data structure for the Shell Renderer. Construction of this data structure takes so much time that it is not really suitable for interactive use. Once you have made the shell data structure, it is suitable for interactive rendering, but every time you change the isovalue or the time step, the shell structure has to be regenerated. The recommended use would therefore be to switch to the Fast Point-based Renderer when browsing through time or searching an interesting isovalue. When a particular isosurface in a certain time step has been found and needs to be explored, the Shell Renderer can very well be used interactively.

7. Conclusions and future work

We have presented a combination of techniques to allow interactive isosurface extraction and visualisation from large time-dependent data sets. In pre-processing we create a tree data structure that is designed for fast extraction of all isovalue-spanning cells for any isovalue and from any time step. This data structure makes effective use of temporal coherence in the data by storing values that remain approximately constant over a time range only once for that time range. The cells that are extracted can be quickly rendered using a hardware-assisted direct rendering algorithm. For this direct rendering no interpolation and triangulation for surface reconstruction is needed.

The data structure is in principle limited to isosurface extraction, however, as was shown in Fig. 6, it is possible to make an approximate reconstruction of the original data, providing the ability to perform other algorithms such as slicing.

We have overcome the huge memory requirements for creation of the data structure by spatially sub-dividing the data set and building a separate tree for each subspace. The memory requirements are therefore equal to the amount needed to hold a single tree.

During visualisation the separate trees can be combined to reconstruct the entire spatial domain. To overcome the memory requirements for this stage, we have designed and implemented a paging scheme, based on a time window paradigm, that will keep only those parts of the trees in memory that are required to visualise the time steps within a user-specified time window. Paging is done per tree node. Each node will be kept in main memory just as long as is needed. The memory requirements for visualisation are therefore equal to the amount of memory needed to hold the time window, which can be as little as one time step.

Within the time window, interactive frame rates can be achieved, for the entire pipeline of extraction and rendering, both for varying isovalue and varying time step. Outside the time window, paging will slow down the frame rates. The amount of data to be transferred from disk will determine the speed. This amount depends not only on the size of the data set but also on the amount of temporal coherence.

Optimisations might be possible by making the data structure more I/O efficient. This data structure was designed to do fast isosurface cell extraction and later adapted for out-of-core functionality. It can perhaps be optimised for I/O, however, that will be at the cost of in-core performance.

Acknowledgements

This project was partly supported by the Netherlands Organisation for Scientific Research (NWO) on the NWO-EW Computational Science Project “Direct Numerical Simulation of Oil/Water Mixtures Using Front Capturing Techniques”.

Thanks to Dr. Ir. B. J. Boersma for the bubble data set and Dr. H. Jonker for the cloud data set.

References

[1] Shen H-W. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In: Proceedings of IEEE Visualization ’98; 1998. p. 159–66.

[2] Botha CP, Post FH. ShellSplatting: interactive rendering of anisotropic volumes. In: Data Visualization 2003. Proceedings of VisSym’03; 2003. p. 105–12.

[3] Lorensen WE, Cline HE. Marching cubes: a high resolution 3D surfaces construction algorithm. In: Proceedings of SIGGRAPH 1987. p. 163–9.

[4] Sutton PM, Hansen CD. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In: Proceedings of IEEE Visualization ’99; 1999. p. 147–53, 520.

[5] Wilhelms J, Van Gelder A. Octrees for faster isosurface generation. ACM Transactions on Graphics 1992;11(3): 201–27.

[6] Shen H-W, Chiang L-J, Ma K-L. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In: Proceedings of IEEE Visualization ’99; 1999. p. 371–7, 545.

[7] Livnat Y, Shen H-W, Johnson CR. A near optimal isosurface extraction algorithm using the span space. IEEE Transactions on Visualization and Computer Graphics 1996;2(1):73–84.

[8] Cignoni P, Marino P, Montani C, Puppo E, Scopigno R. Speeding up isosurface extraction using interval trees. IEEE Transactions on Visualization and Computer Graphics 1997;3(2):158–70.

[9] Bordoloi UD, Shen H-W. Space efficient fast isosurface extraction for large datasets. In: Proceedings of IEEE Visualization '03; 2003. p. 201–8.
[10] Gregorski B, Senecal J, Duchaineau MA, Joy KI. Adaptive extraction of time-varying isosurfaces. IEEE Transactions on Visualization and Computer Graphics 2004;10(6):683–94.
[11] Pascucci V. Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. In: Data Visualization 2004. Proceedings of VisSym'04; 2004. p. 293–300.
[12] Vrolijk B, Botha CP, Post FH. Fast time-dependent isosurface extraction and rendering. In: Proceedings of Spring Conference on Computer Graphics; 2004. p. 45–54.
[13] Chiang Y-J. Out-of-core isosurface extraction of time-varying fields over irregular grids. In: Proceedings of IEEE Visualization '03; 2003. p. 217–24.
[14] Silva C, Chiang Y-J, El-Sana J, Lindstrom P. Out-of-core algorithms for scientific Visualization and computer graphics. Tutorial course notes. IEEE Visualization; 2002.
[15] Westover L. Interactive volume rendering. In: Proceedings of Chapel Hill workshop on Volume Visualization; 1989. p. 9–16.
[16] Udupa JK, Odhner D. Shell rendering. IEEE Computer Graphics and Applications 1993;13(6):58–67.
[17] Rusinkiewicz S, Levoy M. QSplat: A multiresolution point rendering system for large meshes. In: Proceedings of SIGGRAPH 2000. p. 343–52.
[18] Ren L, Pfister H, Zwicker M. Object space EWA surface splatting: a hardware accelerated approach to high quality point rendering. Computer Graphics Forum 2002; 21(3):461–70.
[19] Kilthau S, Möller T. Splatting optimizations. Technical report, Simon Fraser University; 2001.