Fast out-of-core isosurface extraction and rendering of time-varying data sets

Benjamin Vrolijk Frits H. Post

Data Visualisation Group
Delft University of Technology, The Netherlands
`http://visualisation.tudelft.nl/`
B.Vrolijk@ewi.tudelft.nl F.H.Post@ewi.tudelft.nl

Keywords: Out-of-core visualisation, Isosurfaces, Time-dependent data sets

Abstract

We present a combination of techniques for interactive out-of-core isosurface extraction and rendering of time-dependent data sets. We make use of an index tree that allows extraction of all isovalue-spanning cells from any time step, and for any isovalue, at rates of several hundreds of frames per second. This data structure is constructed in a pre-processing stage, and effectively uses the temporal coherence in the data set. However, for very large data sets such as those resulting from CFD simulations, this tree structure can easily become too large to fit in main memory. Therefore, we have adapted the data structure for out-of-core application by adding an intelligent paging scheme, which allows interactive exploration of very large data sets on a normal PC. 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. This paging scheme was implemented using multi-threading. Finally, to avoid time-consuming triangulation and surface reconstruction, we have used a hardware-assisted direct point rendering algorithm, achieving interactive rendering frame rates.

1 Introduction

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

When using a flexible, general-purpose visualisation technique such as 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 isosurfaces separately for each time step is generally too slow for interactive exploration.

Our approach to this challenge is to use specialised data structures allowing very fast access and data retrieval for answering a specific type of visualisation query, as required for isosurface extraction. We used a number of criteria in choosing such a 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 incremental surface extraction, or to determine the differences between successive time steps. 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]. It is generic in the sense that any isovalue can be extracted from any time step. 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 incremental surface extraction, 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.

However, the tree data structure used may become too large to fit in main memory. Therefore, we have designed and implemented an intelligent paging scheme to enable interactive out-of-core isosurface extraction and rendering.

This paper is organised as follows. In Section 2, we discuss related work in isosurface extraction techniques from time-dependent data, and suitable rendering techniques to display the isosurface. Then we will shortly explain the data structures we have used in Section 3, together with the paging scheme we have implemented in Section 4, and the modified shell rendering algorithm in Section 5. Some performance results are given in Section 6, and we will reflect on the results and further work in Section 7.

2 Related Work

Most data structures 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 Wilhems and Van 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.

Recently, Gregorski et al. [7] 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 [8]. 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 [9].

We have designed and implemented a paging scheme for this tree data structure that makes out-of-core extraction possible for very large data sets. The data structure presented 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 [10], 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. For an overview of out-of-core algorithms for computer graphics and visualisation, we refer to the survey by Silva et al.[11].

For visualisation we implemented two different point-based rendering techniques. The first, ShellSplatting, is a hardware-accelerated direct volume rendering method that is based on a combination of splatting [12] and shell rendering [13]. 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 [14] and object space EWA surface splatting [15].

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

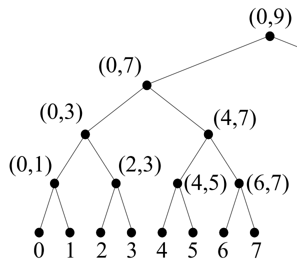

Figure 1: An example of a binary time tree for 10 time steps.

our technique.

3 Temporal index tree

Isosurface extraction involves selection of the voxels, or cells, that are intersected by the isosurface, that is, those cells that contain the isovalue. This means that those cells must have some vertices with scalar values lower and some with values higher than the isovalue. To check if a cell is intersected by the isosurface, it is therefore sufficient to store the extreme values of the cell. It is the main idea for this and other data structures, that each cell is stored as an interval $[min_i, max_i]$, and to check if a cell is an isosurface cell, we simply check if the isovalue is contained in that interval.

We have used and modified the Temporal Hierarchical Index Tree [1]. An important aspect of this data structure is the use of temporal coherence of cells. Instead of storing all the data set's cells for each time step, cells that remain approximately constant (that is, within a certain tolerance) throughout a given time span, are stored only once for that entire time span.

The basic structure of our index tree is a binary time tree, dividing the entire range of time steps of the data set recursively into smaller and smaller ranges. Each node in the tree corresponds to a certain range of time steps. In other words, one level of the binary tree represents the data set at a certain temporal resolution. This resolution doubles with each level of the binary tree. See for a simple example Figure 1. In each node of this binary tree, the cells are stored that remain approximately constant throughout the corresponding time interval. This means that those cells need not be stored anywhere in the tree below the current node. This is the main cause for the potentially large data reduction that can be achieved using this data structure.

The top node of the binary tree represents the entire range of time steps of the data set. The leaf nodes of the tree represent the single time steps at the highest temporal resolution. To retrieve the isosurface cells for a certain time step, the binary tree must be traversed from root to leaf nodes. The cells that are found first, are cells that remain constant throughout the entire time range and therefore must be searched for every time step. The cells that are found in the leaf nodes are those that differ with respect to the neighbouring time steps. Only when the tree has been traversed entirely from root to leaf node, all isosurface cells have been found.

Note that the time tree structure is determined *a priori*, only by the number of time steps. Therefore, the time intervals which are represented by each node of the tree, are fixed. Referring to Figure 1, if a cell remains constant for the time interval $[0,5]$, for example, it will be stored in the two nodes $[0,3]$ and $[4,5]$, because there is no node for the interval $[0,5]$.

We need a way to store a (possibly large) number of cells in each binary tree node efficiently, enabling a quick and efficient search for isosurface cells. For this we use an Interval Tree [16]. A description of this data structure is beyond the scope of this article. We refer to the original article by Cignoni et al. [16] and our previous work [9]. In the current context, it suffices to know that the intervals are stored in interval trees, and that one interval tree is stored in each of the nodes of the index tree.

3.1 Isosurface cell query

The index tree can be queried for any isovalue at any time step. The tree will be traversed from top to bottom, selecting the correct nodes depending on the requested time step. In each node of the tree, the corresponding interval tree is searched. The cells returned by every search contribute to the final result, which will be complete when the leaf nodes of the index tree have been reached. The list of cells we have obtained then contains all cells in the requested time step that span the isovalue, and therefore, all cells that are intersected by the isosurface. However, cells that are found outside the leaf nodes of the index tree, are represented by their temporal extreme values, measured over a certain time interval. The fact that these temporal extreme values span the isovalue does not guarantee that the extreme values for the current time step also span the isovalue. This means that the resulting list of cells will contain a number of false positives.

The number of false positives can be controlled, but a reduction of this number will be at the cost of memory space [9]. In general we can say that with our default setting, we get approximately 0.5% false positives. This will hardly show up in the images.

3.2 Incremental search

The binary tree structure for representing time spans makes it possible to do incremental searching for isosurface cells. Because each node in the tree represents a certain time span, the information that is known in that node can be used for all time steps in that span, that is, for all child nodes of that node. For example, let us assume that a search has been performed for time step 0, and that the resulting isosurface cells are known. When time step 1 is to be searched next (for the same isovalue), there is no need to do a full search of the tree again. Instead, the previous result can be used, because all the cells that have been found from the root of the tree down to the node representing time span $[0,1]$, can be reused. These cells are (by definition) identical for both time step 0 and time step 1. Only the leaf node representing the single time step 1 must be searched. Next, when time step 2 is to be searched, we need to do a little more 'back-tracking', because the last common node for time steps 1 and 2 is the node $[0,3]$.

This can be implemented fairly easily. The search in each node of the tree returns a number of cells. These cells are appended to a single result vector. For the incremental search to work, we save the number of cells found so far, that is, the size of the result vector, in a single vector of integers. This vector is the only space overhead for the incremental search — at most d integers, where d is the maximum depth of the time tree.

For an incremental search of any time step t_n, we pass the result vector of the previous search, the integer vector $V[d]$ just described, and the time step t_o of the previous search. Note that these time steps do not have to be consecutive; any two time steps can be used. The binary tree is then traversed from the root to the leaf node representing t_n. In each node N_i (at depth i), we check whether t_n and t_o are in this node's time span. If so, we simply go to the next node, because we can reuse the first $V[i]$ cells from the result vector. If not, we truncate the result vector after $V[i-1]$ cells, because that is the number of cells that t_n and t_o have in common. The rest of the tree must be searched normally. During this search the result vector and the integer vector V have to be kept up-to-date. While only causing a negligible space overhead, this incremental search routine offers a speed-up of about 35 in our application, when we search 600 consecutive time steps incrementally, as opposed to 600 full searches.

4 Paging scheme

The index tree described in the previous section can easily become too large to fit in main memory. The size of the tree very much depends on the size of the original data set and on the amount of temporal coherence in the data. However, the tree structure hints at an intelligent paging scheme. Such a scheme would remove the constraints on memory size for the index tree and allow arbitrary-sized data sets to be explored. Therefore we adapted the data structure to make it possible to keep only a limited number of time steps in memory, and load other time steps on demand from disk. We have used the idea of a time window, centered around the current time step, to implement this.

4.1 Time window

This concept corresponds very well to the way scientists will work when exploring a data set. Often, one will browse forward or backward through the data, until a certain event is detected. Then the main focus will be on the frames around the event. This time window needs to be in main memory in order to be explored interactively.

A number of adaptations have been made to provide this functionality. First, the user must be able to specify the *time window size*, defining the number of time steps that will be kept in main memory at one time. Alternatively, the user can specify the maximum amount of memory that is to be used, after which the window size can be determined automatically. This way, the number of time steps in memory can be maximised, while always fully using the available amount of memory.

The index tree structure itself has been modified. We have decided that the skeleton of the tree should always be kept in main memory. In each individual node of the tree the range of time steps for that node is stored, the interval tree containing all the intervals, and the pointers to both child nodes. The range of time steps and the pointers to the children will be kept in main memory at all times. Only the interval tree, which takes up most of the memory, can be paged in and out of main memory.

The entire index tree is stored as one (binary) file on disk, so when we want to read the data (that is the interval tree) for a single tree node, we need the file offset and the number of bytes to read. These two values have been added to and are now stored with each node in the index tree. Actually, the number of bytes is not necessary for reading a single index tree node. The entire index tree including all interval trees, will be reconstructed in memory on the fly, while reading from disk. Therefore the number of bytes to be read does not need to be known beforehand. However, this number *is* necessary when the index tree node is *not* read, that is, when the structure of the index tree is read, without the data in nodes. In this case, it is necessary to know how many bytes to skip to find the file offset for the next tree node to be read.

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, tree nodes can be read (meaning the interval trees in the index tree nodes) selectively whenever they are needed.

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, or online, if the interval tree for that tree node is in memory, and offline, otherwise. 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.

As the user can specify the window size, meaning the number of time steps that will be in memory at one time, we can easily keep track of the window of time steps, centered around the current time step. This means we can free all time steps before the first, or after the last time step of the window. When reading a time step, we have to make sure we read all nodes from the top of the tree down to the leaf node representing the given time step. However, when freeing a time step, we must make sure we free all nodes that are no longer needed, but leave those nodes that span or overlap with the time window untouched.

This is easily implemented. Each tree node contains the range of time steps that the node represents. Furthermore, we know the first and last time step of the time window. Therefore, we can check if these two ranges overlap. If not, we can remove the node from memory.

This does have some consequences for the time complexity, both for reading tree nodes and freeing them. (See Figure 2.) If for example the time window starts at $t = 3$, time step 2 can be removed from memory. In this case, that means the single leaf node representing $t = 2$ can be freed. If the time window shifts one time step forward, time step 3 is no longer needed. But now, the three nodes $[0,3]$, $[2,3]$ and 3 can be freed. Similarly, when reading time steps the number of nodes and therefore the amount of data that has to be read will be different with every time step.

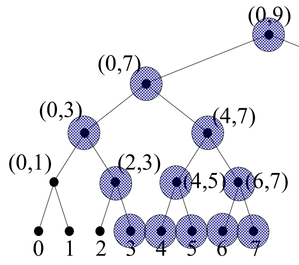

Figure 2: A binary time tree with a time window $[3,7]$. Only the colored nodes will be kept in main memory.

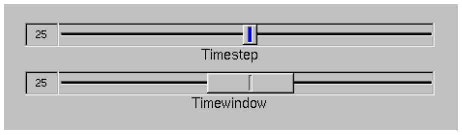

Figure 3: The GUI element that shows the time window and current time step.

4.2 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 Figure 3.

Because reading new time steps will always be slower than visualising them, the visualisation will, in the end, catch up with the last time step of the 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 have to wait (or rather, the user will notice a delay). For this purpose 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.

4.3 Multi-threaded design

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 in memory. If necessary, it will read 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 the background, 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. The two threads wake up: one will read the required tree nodes from disk, the other will remove all other tree nodes from memory.

4.4 Mutexes

A multi-threaded approach requires very careful programming to avoid *race conditions* or *deadlock*. The solution is in the use of *mutex variables*. We need mutexes (short for mutual exclusion objects) to ensure that the individual threads do not read from or write to the same tree node at the same time. The straightforward solution would be to use a single mutex on the entire index tree. However, that would be too restrictive: it should be possible for the main thread to search in one time step, while another thread is writing in another part of the tree. Instead, we decided to use a single mutex for every tree node, as the tree nodes can be seen as the basic units for reading from disk, freeing from memory, or searching for isosurface cells. Using this locking scheme, with one mutex per tree node, it is possible to search in one tree node, while for example, one of its child nodes is being read from disk.

It is very important to consistently check the mutex variable before reading or writing a tree node. If one thread or one function call does not strictly adhere to this principle, the program will show unpredictable and irreproducible behaviour. Therefore both the read and cleanup functions used in the two auxiliary threads, but also the search function in the main thread have to check and lock / unlock the mutex associated with every tree node very consistently.

Summarising, we have adapted the index tree data structure to provide out-of-core functionality. The structure of the tree will remain in memory at all times, but the data at each node can be paged in and out of memory whenever needed. In this manner, it is possible to keep only a certain time window centered around the current time step in memory, while the other time steps remain on disk and will be read on demand. We have chosen a multi-threaded approach, where the main thread is concerned with the visualisation and two auxiliary threads have the tasks of reading new time steps from disk and freeing up the memory of time steps that are no longer needed.

5 Point-based 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 fast access, as the original data would have to be read from disk to perform surface reconstruction using for example the Marching Cubes algorithm [3].

To avoid this, we have used a point-based direct rendering algorithm. We further optimised our *Shell-Splatting* 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 very 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 [9].

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 [17]. The polygon is also constructed so that we can perform all rendering in isotropic voxel space and have the graphics hardware perform necessary anisotropic scaling.

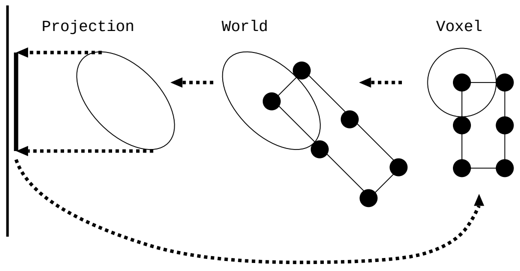

Figure 4: Illustration of the calculation of the voxel sphere in voxel space, transformation to world space and projection space and the subsequent "flattening" and transformation back to voxel space.

To visualise this construction, imagine a three-dimensional ellipsoid bounding a small neighbourhood around a voxel. If we were to project this ellipsoid onto the projection plane and then "flatten" it, i.e. calculate the outline of its orthogonal projection (an ellipse) on the projection plane, the outline would also bound the projected voxel. A rectangle with principal axes identical to those of the projected ellipse, transformed back to the drawing space, is used as the rendering polygon.

Figure 4 illustrates a two-dimensional version of this procedure. In the figure we also show the transformation from voxel space to world space. This extra transformation is performed so that rendering can be done in the isotropically sampled voxel space, even if the volume has been anisotropically sampled. Alternatively stated, the anisotropic volume is warped to be isotropic. The voxel-to-model, model-to-world, world-to-view and projection matrices are concatenated in order to form a single transformation matrix **M** with which we can move between the projection and voxel spaces.

The list of cells extracted from the index tree is uploaded to the graphics pipeline in arbitrary order as a list of view plane parallel polygons. Because all polygons are non-textured and completely opaque, their ordering is not important. As explained above, scaling is done in hardware, so anisotropic volumes are handled correctly.

6 Results

We have tested our application on two large data sets. The first data set is 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 Figure 5, left.

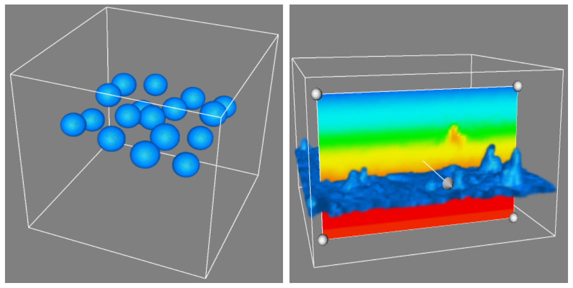

Figure 5: Scenes from the two data sets. On the left is the bubble data set, on the right is the cloud data set.

Data set	Bubbles	Clouds
Resolution	256^3	$128 \times 128 \times 80$
# Time steps	39	600
Raw data size	4 992 MB	3 000 MB
Index tree size	1 630 MB	750 MB

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

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 Figure 5, right.

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

6.1 Benchmarks

For these two data sets, 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.

The results of the rendering benchmarks are shown in Figure 6. 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 framerate 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 Figure 7.

Extraction of the isovalue-spanning cells can be done extremely fast. In the cloud data set, extraction rates of over 5 000 time steps per second can be achieved, using our incremental search. For the larger bubble data set, we get extraction rates of about 180 time steps per second. Rendering is also very fast, as long as we don't 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 described techniques for fast isourface extraction and direct rendering from time-varying data sets. In a preprocessing step, data structures are generated that allow us to retrieve the isovalue-spanning cells at any time step and for any isovalue with high frame rates. Incremental searching uses temporal coherence to further speed up the extraction process. The extracted cells are rendered directly with a fast point-based rendering technique, displaying a shaded quadrangle at each voxel at high frame rates. No visibility ordering is needed in this case, so the overall speed is not reduced by an intermediate data conversion step. A high quality rendering technique based on ShellSplatting does require visibility ordering, but can still achieve interactive frame rates for a 256^3 data set. In an interactive environment, the fast rendering can be used during interaction, while the high quality technique can be automatically invoked when the input queue is empty. We will integrate this in our VR data exploration system.

With this work, our main contributions are the fast incremental search and the integration of the fast isospanning-cell extraction and rendering stages. We have also attempted to further optimise the search data structures for space efficiency. Even more improvement is possible by using compression techniques, as recently proposed by Bordoloi and Shen [18].

We have made our technique truly scalable to very large data sets by introducing the possibility of out-of-core isosurface extraction and rendering in combination with an intelligent paging scheme. We have adapted the data structure such that only part of the tree will be in main memory. Paging is done per time step. A user-specified window of consecutive time steps will be in main memory, while other time steps are read from disk on demand. Within the time window the frame rates will be the same as for the in-core version. Only when playback catches up with the time window, the frame rate will drop and the reading of new time steps will be the bottleneck.

Further optimisations might be possible by making the data structure more I/O efficient. The temporal index tree was designed to do fast isosurface cell extraction and we adapted it for out-of-core functionality. It can probably be optimised for I/O, however, that might be at the cost of in-core performance.

There are two possible sources of error in the display of the isosurfaces that must be investigated further.

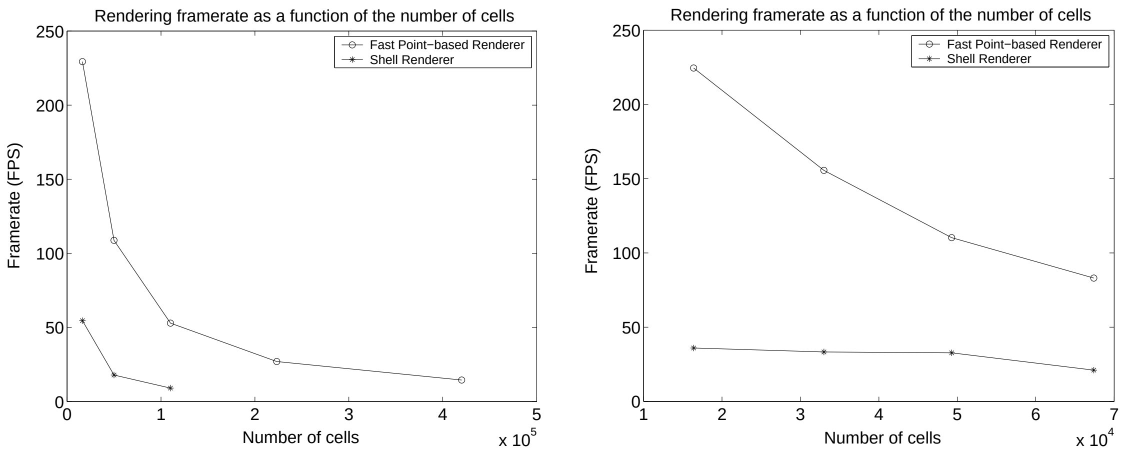

Figure 6: The results of the rendering benchmarks. On the left is the bubble data set, on the right is the cloud data set.

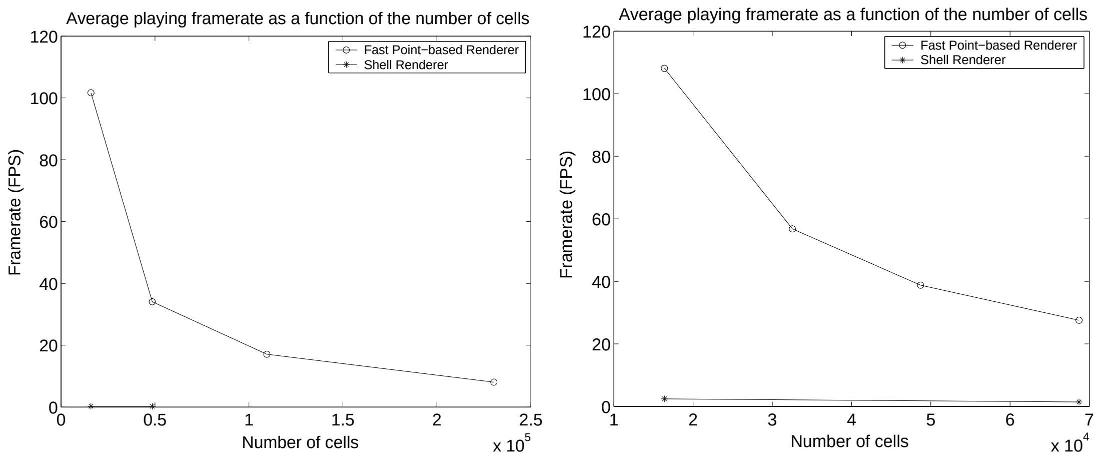

Figure 7: The results of the play benchmarks. Playing involves extraction and rendering through (part of) the time range. On the left is the bubble data set, on the right is the cloud data set.

Although this did not show up in the test images, the rendering of false positive cells may cause artefacts. Also, the surface normals are stored only once over a time interval that is considered “approximately constant”. This also did not have any noticeable effect in the images, but we will analyse the extent of the errors caused.

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] Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proc. IEEE Visualization ’98*, pages 159–166, 1998.

[2] Charl P. Botha and Frits H. Post. ShellSplatting: Interactive rendering of anisotropic volumes. In *Data Visualization 2003. Proc. VisSym’03*, 2003.

[3] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surfaces construction algorithm. In *Proc. SIGGRAPH*, pages 163–169, 1987.

[4] Philip M. Sutton and Charles D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In *Proc. IEEE Visualization ’99*, pages 147–153,520, 1999.

[5] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. on Graphics*, 11(3):201–227, July 1992.

[6] Han-Wei Shen, L.-J. Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In *Proc. IEEE Visualization ’99*, pages 371–377,545, 1999.

[7] Benjamin Gregorski, Joshua Senecal, Mark A. Duchaineau, and Kenneth I. Joy. Adaptive extraction of time-varying isosurfaces. *IEEE Trans. on Visualization & Computer Graphics*, 10(6):683–694, November 2004.

[8] Valerio Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Data Visualization 2004. Proc. VisSym’04*, 2004.

[9] Benjamin Vrolijk, Charl P. Botha, and Frits H. Post. Fast time-dependent isosurface extraction and rendering. In *Proc. Spring Conference on Computer Graphics*, pages 39–48, 2004.

[10] Yi-Jen Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proc. IEEE Visualization ’03*, pages 217–224, 2003.

[11] Claudio Silva, Yi-Jen Chiang, J. El-Sana, and Peter Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. Tutorial Course Notes, IEEE Visualization, 2002.

[12] Lee Westover. Interactive volume rendering. In *Proc. Chapel Hill workshop on Volume visualization*, pages 9–16, 1989.

[13] Jayaram K. Udupa and Dewey Odhner. Shell rendering. *IEEE Computer Graphics and Applications*, 13(6):58–67, November 1993.

[14] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH*, pages 343–352, 2000.

[15] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[16] Paolo Cignoni, P. Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Trans. on Visualization & Computer Graphics*, 3(2):158–170, April 1997.

[17] Steven Kilthau and Torsten Möller. Splatting optimizations. Technical report, Simon Fraser University, 2001.

[18] Udeepta D. Bordoloi and Han-Wei Shen. Space efficient fast isosurface extraction for large datasets. In *Proc. IEEE Visualization ’03*, pages 201–208, 2003.