Fast Time-Dependent Isosurface Extraction And Rendering

Benjamin Vrolijk Charl P. Botha Frits H. Post

Data Visualisation Group
Delft University of Technology, The Netherlands
http://visualisation.tudelft.nl/

Abstract

For the visualisation of time-dependent data sets, interactive isosurface extraction and rendering is desirable. It allows the user to study the development of a surface shape in time, such as a moving front or an evolving object shape. For this purpose, the user must be able to interactively specify an isovalue, and a sequence of isosurfaces must be visualised, starting from any time step, in forward or backward direction in time. In this paper, we describe efficient and tightly coupled techniques for time-dependent isosurface extraction and rendering at interactive frame rates. In preprocessing, we create data structures from a time-dependent data set that allow real-time extraction of all isovalue-spanning cells, achieving rates of several hundreds of frames per second. These isovalued cells are then passed to a fast hardware-assisted direct point rendering algorithm for display, thus avoiding time expensive surface construction by triangulation. This algorithm makes effective use of the available graphics hardware.

CR Categories: I.3.m [Computing Methodologies]: Computer Graphics—Miscellaneous

Keywords: Visualisation, Isosurface extraction, Volume rendering, Time-varying data sets

1 Introduction

Interactive exploration of large, time-dependent 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 investigatig these large data sets require 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, such as in 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. 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 [Shen 1998]. It is specialised, because it does not allow for other types of visualisation, but 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 [Botha and Post 2003], 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 [Lorensen and Cline 1987]. 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.

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 explain the data structures we have used in Sections 3 and 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) [Sutton and Hansen 1999]. This is an extension to the original Branch-on-Need Octree (BONO), described by Wilhems and Van Gelder [Wilhelms and Gelder 1992]. 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 [Shen et al. 1999]. 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 makes multi-resolution access possible for any dimension, it also means a huge storage overhead.

Shen describes another data structure for isosurface extraction from time-varying fields, called the Temporal Hierarchical Index Tree [Shen 1998]. The idea behind this structure is to store voxels that remain (more or less) constant throughout a certain time span only once for that entire time span. For our purposes, we decided to use and extend the latter. 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.

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 [Westover 1989] and shell rendering [Udupa and Odhner 1993]. 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 QS-plat [Rusinkiewicz and Levoy 2000] and object space EWA surface splatting [Ren et al. 2002].

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 Data structures

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.

The data structure we used consists of three elements: a binary tree representing time, and the Span Space and Interval Tree data structures for making an efficient interval search possible. We will discuss each of these structures in the following Sections, before describing the Temporal Hierarchical Index Tree in Section 4.

We will use the terms *voxel* and *cell* alternately throughout this paper. Also, in this context, the term *interval* refers to the representation we use for cells or voxels.

3.1 Binary Time Tree

An important aspect of the Temporal Hierarchical Index Tree (or THITree), is the use of temporal coherence of cells. Instead of storing all the data set's cells for each time step, cells that remain more or less 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 the THITree is a Binary Time Tree, dividing the entire range of time steps of the data set recursively into smaller and smaller ranges. The nodes at one level of the binary tree represent a single time step of the data set at a certain temporal resolution. The temporal

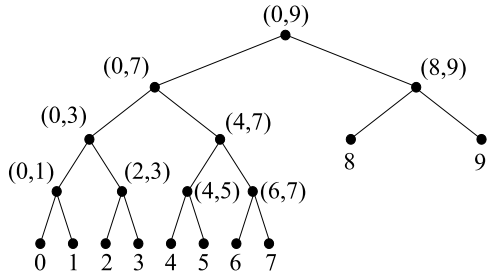

Figure 1: An example of a Binary Time Tree for 10 time steps.

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 more or less 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 possibly 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 more or less constant throughout the entire time range. 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.

We still need a way to classify the variance of cells — we need a way to define "more or less constant" — to determine where the cells should be stored in the Binary Time Tree. Furthermore, 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. Both these problems will be addressed next, when we discuss the Span Space.

3.2 Span Space

As stated above, cells are stored in the THITree as intervals $[min_i, max_i]$, and isosurface cells are simply those cells for which the interval contains the isovalue. The *Span Space*, as described by Livnat et al. [Livnat et al. 1996], is used to represent intervals $[min_i, max_i]$ as points (min_i, max_i) in 2D. The x-coordinate of a point represents the minimum value, or left extreme, of the interval, and the y-coordinate of the point represents the maximum value, or right extreme of the interval. See Figure 2a.

For a time-dependent data set, each cell corresponds to multiple points in Span Space, one for each time step. The amount of temporal variation of a cell can be quantified by the amount of variation of the corresponding points in Span Space. For this, it is useful to define a grid in the Span Space, for example using a lattice subdivision scheme [Shen et al. 1996] (see below). As a measure for the temporal variation of a cell, we use the number of grid elements that the correponding points in Span Space occupy. For example, if all points for a cell during a certain time interval are located within 2×2 lattice elements, we classify the cell as one of low temporal variation for that interval, and therefore, the cell has to be stored only once for that time interval, in the corresponding node of the THITree. We use the parameter MaxVariation for this; in Sections 4.1 and 6 we will discuss the influence of this parameter on the accuracy and size of the THITree.

The lattice subdivision scheme used, works as follows. A sorted list is created of all distinct extreme values of all cells from all time steps. From this list, $L+1$ scalar values are found that divide the list into L equal length sublists. These $L+1$ scalar values can then be used to draw the $L+1$ vertical and horizontal lines in Span Space to form the lattice.

The Span Space is not only used for quantifying the amount of temporal variation of the cells, but also for storing the cells in each node of the THITree. We store one Span Space per node of the tree. Because non-leaf nodes of the THITree represent time spans, instead of single time steps, the cells that are stored in the Span Spaces in these nodes have to be represented by their temporal extremes: a single cell, changing over a number of time steps, corresponds to a number of points in Span Space (one for each time step), but will always be represented by a single point, representing the temporal extreme values.

The points that are stored in Span Space are organised per row of the Span Space. For each row, two lists of points are maintained, one sorted on the minimum value in ascending order, and one sorted on the maximum value in descending order. These lists do not contain the points from the lattice element on the diagonal, because this element requires a min-max search. Instead, these points are stored in a separate data structure, an *Interval Tree* [Cignoni et al. 1997]. This structure will be discussed in Section 3.3.

When the Span Space needs to be searched for isosurface cells, first, the lattice element $[I, I]$ is located that contains the isovalue V_{iso}, represented by the point (V_{iso}, V_{iso}). See Figure 2b.

1. For each Span Space Row $R, R > I$, we search the list that was sorted according to the *minimum* values. We collect the cells from the beginning of the list until the first cell is found with a minimum value

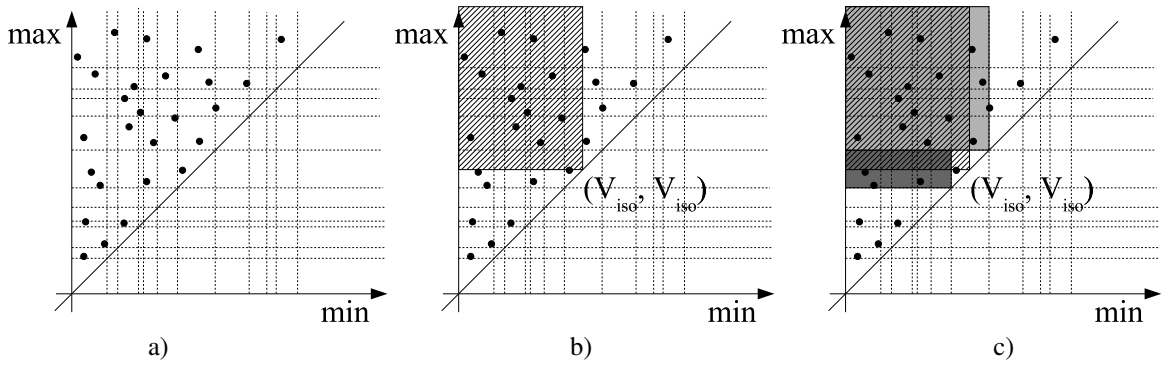

Figure 2: **a**) Intervals represented as points in Span Space. **b**) The intervals spanning a given isovalue V_{iso} are located in the upper left corner from the point (V_{iso}, V_{iso}). **c**) The search for intervals spanning a given isovalue V_{iso} is done in three steps, corresponding to three regions in Span Space.

greater than the isovalue. Because $R > I$, we know that the maximum values are larger than the isovalue, therefore, all cells found are guaranteed to contain the isovalue.

2. For the Span Space Row I, we search the list that was sorted according to the *maximum* values. We collect the cells from the beginning of the list until the first cell is found with a maximum value less than the isovalue. Note that, because we have left out the cells from the diagonal element, all cells in this list have a minimum value less than the isovalue, and therefore, all cells found are guaranteed to contain the isovalue. This is the reason why the cells from the diagonal element are stored in a separate data structure.

3. For the same Span Space Row I, we search this data structure, the Interval Tree, to find the cells from the lattice element $[I, I]$.

In Figure 2c, these three cases are illustrated. The first case corresponds to the light gray region. The second case corresponds to the dark gray region, which is the row containing the isovalue. The third case corresponds to the lattice element on the diagonal, for which only the striped part contains isosurface cells; the white parts have either a too large minimum, or a too small maximum value.

3.3 Interval Tree

The Interval Tree is a data structure that was proposed by Edelsbrunner [Edelsbrunner 1980] to retrieve from a set of intervals those that contain a certain query value. It has an optimal efficiency of $O(\log n)$. We use the Interval Tree to search for intervals (meaning cells) that span a given isovalue [Cignoni et al. 1997].

An Interval Tree is created as follows. Given a set $I = \{I_1, \ldots, I_m\}$ of intervals $[a_i, b_i]$, we create a sorted sequence of distinct extremes $X = (x_1, \ldots, x_h)$, that is, each a_i or b_i is equal to some x_j. The Interval Tree consists of a balanced binary tree, whose nodes correspond to values of X, plus two lists of intervals appended to each non-leaf node of the tree. In Figure 3 is a simple example of a small Interval Tree.

The root of the tree is assigned the "halfway" value $\delta_r = x_{\lceil \frac{h}{2} \rceil}$. The set I is partitioned into three subsets:

- $I_l = \{I_i \in I | b_i < \delta_r\}$; the intervals that are entirely to the left of δ_r;
- $I_r = \{I_i \in I | a_i > \delta_r\}$; the intervals that are entirely to the right of δ_r;
- $I_{\delta_r} = \{I_i \in I | a_i \leq \delta_r \leq b_i\}$; the intervals that contain or overlap δ_r.

The intervals in I_{δ_r} are stored in the root node, arranged into two lists: one containing all intervals sorted according to their left extremes a_i, in ascending order (AL), and one containing all intervals sorted according to their right extremes b_i, in descending order (DR).

The left and right subtrees are defined recursively, by considering the interval sets I_l and I_r, and the sequences $(x_1, \ldots, x_{\lceil \frac{h}{2} \rceil - 1})$ and $(x_{\lceil \frac{h}{2} \rceil + 1}, \ldots, x_h)$, respectively.

When searching the tree for a given isovalue V, the tree is traversed as follows, starting at the root:

- if $V < \delta_r$ then list AL is scanned until an interval I_i is found such that $a_i > V$; all scanned intervals are returned and the left subtree is traversed recursively;
- if $V > \delta_r$ then list DR is scanned until an interval I_i is found such that $b_i < V$; all scanned intervals are returned and the right subtree is traversed recursively;
- if $V = \delta_r$ then list AL is returned.

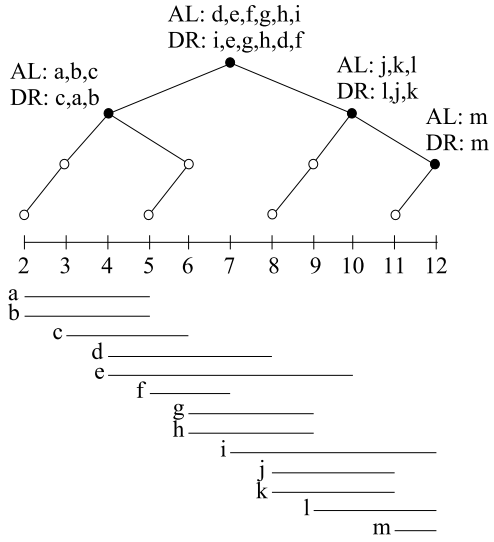

Figure 3: An example of a simple Interval Tree for a small number of intervals.

4 THI Tree

We now have all the tools to construct the Temporal Hierarchical Index Tree. We do all this in a preprocessing step.

First we classify all cells according to their variance over time, using the Span Space, in order to determine their locations in the Binary Time Tree. For each cell, an interval is determined, which is represented as a single point in Span Space. The variation over time is quantified by the number of grid elements in Span Space that are occupied by the points corresponding to that cell in each time step. Cells with a low temporal variation over a long time span are placed high up in the tree. 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 $[0,5]$.

Next, we store all cells for a certain node of the tree in a single Span Space, arranging the cells per row of the Span Space into two lists plus an Interval Tree. We use the same Span Space, meaning the same lattice subdivision, for every Span Space in the THITree. The list of cells for a single Span Space is divided into sublists using this lattice subdivision. Each sublist contains the cells for one row of the Span Space. The cells that belong to the lattice element on the diagonal, are stored in an Interval Tree, and removed from the sublist. The remaining cells are stored in two separate lists, the one sorted according to ascending minimum value, the other according to descending maximum value.

4.1 Isosurface cell query

The Temporal Hierarchical Index Tree can be queried for any isovalue at any time step. First of all, we determine the Span Space lattice element that contains the isovalue, because all Span Spaces used in the THITree use the same subdivision. Next, the tree is traversed from top to bottom, selecting the correct nodes depending on the requested time step. In each node of the tree, the corresponding Span Space is searched, as described above in Section 3.2. The cells returned by every search contribute to the final result, which will be complete when the leaf nodes of the THITree have been reached. The list of cells we have obtained now 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 THITree, 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 do so too. This means that the resulting list of cells contains 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. There are two parameters to control the accuracy (and therefore the memory space) of the THITree. First, the Span Space grid size can be adjusted (the parameter L, we discussed in Section 3.2); smaller grid elements result in fewer false positives. Next, another parameter (MaxVariation) defines which cells are considered as "more or less constant" over time. This parameter corresponds to the number of grid elements that a single cell, varying over time, may occupy in Span Space, and still be called constant. Stated otherwise, this parameter defines the maximum allowed variation of a "constant" cell. Increasing this parameter obviously increases the number of false positives, but reduces the memory size of the resulting THITree. Depending on the setting of these two parameters, the number of false positives ranges from about 0.1% with the largest tree size, up to 5% with the smallest tree size in our test application. Using the default settings, we get approximately 0.5% false positives.

4.2 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, the tree does not need to be searched fully. 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 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 $V[d]$ we 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 performance gain of a factor 3 in our test application, when we search 50 consecutive time steps incrementally, as opposed to 50 full searches. In Table 1 the exact numbers are given (under "Speed up") for several different settings of the parameters.

5 Point-based rendering

Making use of traditional triangulation and surface rendering techniques for visualisation would almost negate the advantages of the fast isosurface cell extraction. At worst, it would entail that the original data would have to be read from disc for all selected voxels and that surface interpolation would have to be performed with for example the Marching Cubes algorithm [Lorensen and Cline 1987].

For us the logical answer was to make use of a point-based direct rendering technique. We further optimised our *ShellSplatting* rendering algorithm [Botha and Post 2003], 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.

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 the orthogonal projection case. Strictly speaking, in the perspective projection case each rendered polygon should be orthogonal to the viewing ray that intersects it. However, for efficiency reasons, we make use of slightly larger screen-aligned polygons [Kilthau and Möller 2001]. 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.

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 its orthogonally projected outline (an ellipse) on the projection plane, the projected 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, however, 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 $\mathbf{M}$ with which we can move between the projection and voxel spaces.

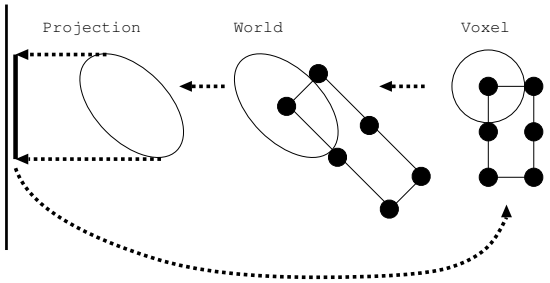

Figure 4: Illustration of the calculation of the voxel sphere in voxel space, transformation to world space and projection space and the subsequent "flattening" and transformation back to voxel space.

A quadric surface, of which an ellipsoid is an example, can be represented in matrix form as follows:

$$\mathbf{P}^T\mathbf{Q}\mathbf{P} = 0$$

where

$$\mathbf{Q} = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix}$$

contains the coefficients of the implicit function defining the quadric and

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Such a surface can be transformed with a $4x4$ homogeneous transformation matrix $\mathbf{M}$ as follows:

$$\mathbf{Q}' = (\mathbf{M}^{-1})^T\mathbf{Q}\mathbf{M}^{-1} \quad (1)$$

A voxel bounding sphere in quadric form $\mathbf{Q}$ is constructed in voxel space. Remember that this is identical to constructing a potentially non-spherical bounding *ellipsoid* in world space. In this way anisotropically sampled volumes are elegantly accommodated.

This sphere is transformed to projection space by making use of Equation 1. The two-dimensional image of a three-dimensional quadric of the form

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix}$$

as seen from a normalised projective camera is a conic $\mathbf{C}$ described by $\mathbf{C} = c\mathbf{A} - \mathbf{b}\mathbf{b}^T$ [Stenger et al. 2001]. In projection space, $\mathbf{C}$ represents the two-dimensional projection of $\mathbf{Q}$ on the projection plane.

An eigendecomposition $\mathbf{C}\mathbf{X} = \mathbf{X}\lambda$ can be written as

$$\mathbf{C} = (\mathbf{X}^{-1})^T\lambda\mathbf{X}^{-1}$$

which is identical to Equation 1. The diagonal matrix λ is a representation of the conic $\mathbf{C}$ in the subspace spanned by the first two eigenvectors in

$$\mathbf{X} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where R and t represent the rotation and translation submatrices respectively. The conic's principal axes are collinear with these first two eigenvectors.

In other words, we have the orientation and length of the projected ellipse's principal axes which correspond to the principal axes of a voxel bounding sphere that has been projected from voxel space onto the projection plane. Finally, these axes are transformed back into voxel space with $\mathbf{M}^{-1}$ and used to construct the rectangles that will be used to render the voxels.

The list of cells extracted from the THITree 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.

Figure 5 shows a single time step of a sample data set rendered with the ShellSplatter and the fast point-based renderer. The ShellSplatted rendering on the left shows the typical fuzziness often associated with splatting-based rendering methods whilst the fast point-based rendering on the right appears slightly jagged due to the use of flat-shaded quads.

6 Results

We have used two data sets for testing the performance of the THITree and the renderer. The first is a 96^3 data set of an air bubble rising in water and breaking through the surface [1]. This data set consists of 50 time steps ("bubble"). The second data set is obtained from a fluid dynamics simulation, and contains turbulent vortex structures [2]. The size of this data set is $64^3 \times 100$ time steps ("vorticity").

The extraction and rendering performance was measured on a 2.4 GHz Pentium 4 with 1 GB of memory and a 128 MB GeForce 4 Ti4600 graphics card.

6.1 THITree size

The memory size of the Temporal Hierarchical Index Tree for the 50 time steps of the bubble data set is about

[1] Data courtesy S. P. van der Pijl of Delft University of Technology.
[2] Data courtesy D. Silver and X. Wang of Rutgers University.

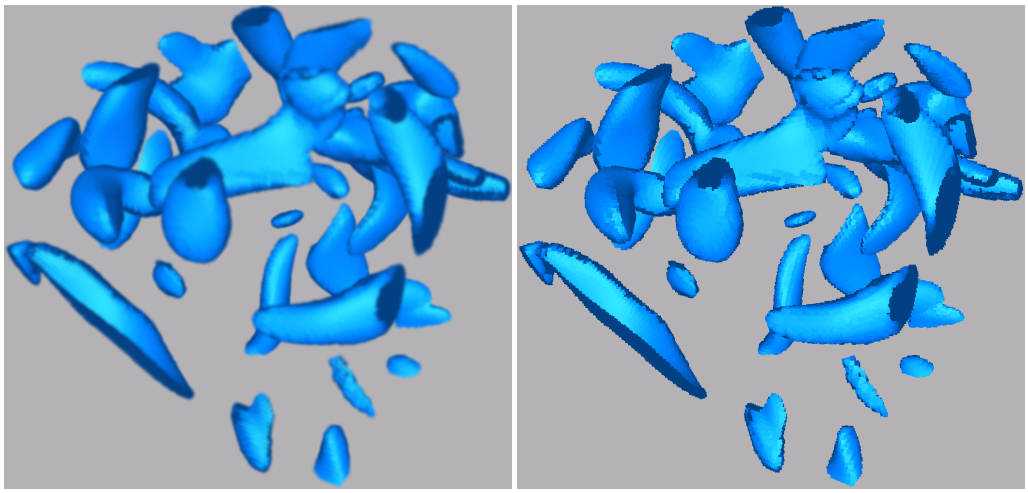

Figure 5: Example renderings of a single time step. On the left the high quality ShellSplatting is shown, on the right the faster simple point-based renderer output is shown.

132 Megabytes. The vorticity data set results in a tree size of about 556 Megabytes. This huge difference has to do with the variability of the data and can be illustrated by examining the number of cells in each of the nodes of the tree.

There are two parameters that influence the size of the data structure, and thereby of course, also performance and accuracy.

First, the size of the Span Space can be changed, that is, the number of rows or columns in the Span Space. This affects the number of cells in each row, the number of Interval Trees in the Span Space (one for each row), and the number of cells that has to be stored in each Interval Tree.

However, the total number of cells in the Span Space is not affected, therefore, the memory size of the Span Space will hardly change. Only the vector representing the Span Space boundaries is affected by this parameter, but this vector is stored only once for the entire THITree. But if the Span Space contains fewer grid elements, meaning that the grid elements are larger, then cells will sooner be considered constant for a longer time span, and therefore these cells will be stored higher up in the THITree, thus reducing the overall size of the data structure. The downside is that more false positives will be found. Accuracy is traded off for memory size.

The same applies to the MaxVariation parameter, which indicates how many grid elements cells may span, and still be considered constant. Thus, without changing the size of each Span Space, we can control the level at which the cells will be stored in the THITree. This way we are able to reduce the total memory size of the tree, but at the cost of increasing the number of false positives that will be found.

In Table 1 a few performance characteristics of the Temporal Hierarchical Index Tree are shown. We have used the bubble data set (see Figure 6) for determining the influence of the two parameters discussed above. We created THITrees with 3 variants of each of the two parameters: for the Span Space size, we used values of 32, 64 and 128, and for the MaxVariation we used 1, 2 and 3 grid elements.

Cells in our data structure are represented by a cell id, a minimum and maximum value, and a gradient. We compared the size of the THITree to the raw data size, meaning simply the number of time steps $\times$ the number of cells $\times$ the memory size of one cell.

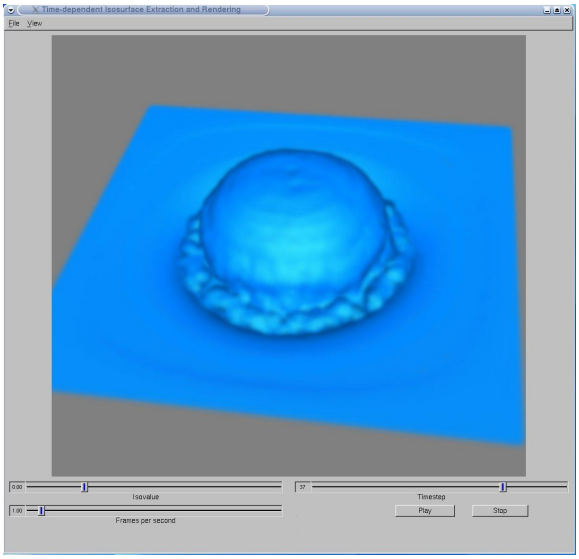

Figure 6: A single time step from the bubble data set. The GUI contains sliders for interactively changing the isovalue and the current time step.

Bubble data set, size: $96^3 \times 50$ time steps Raw size of cell data: 981 MB						
SpanSpaceSize	32	64	128	64	64	64
MaxVariation	2	2	2	1	2	3
THITree size (MB)	89.2	132.6	193.4	355.3	132.6	96.7
% of raw data	9.1%	13.5%	19.7%	36.2%	13.5%	9.9%
Search (ms)	8.2	8.4	9.0	9.0	8.4	8.1
SearchIncr (ms)	2.0	2.5	2.9	2.7	2.5	2.1
Speed up	4.1	3.4	3.1	3.3	3.4	3.9

Table 1: Time and space performance of the THITree for different values of the parameters SpanSpaceSize and MaxVariation.

Rendering mode	Bubble	Vorticity
High quality	11.1	20.4
Fast	67.4	135.7

Table 2: Average rendering frame rates (in FPS) for the two data sets, both in high quality and in fast rendering mode, for 512×512 images.

6.2 Surface cell extraction

The THITree data structure provides a very quick way to search for isosurface cells. In our bubble data set the time for extraction of the isosurface cells for a single time step takes on average approximately 8.4 milliseconds. When we use the incremental search algorithm we can achieve even higher rates: incrementally searching the isosurface cells in 50 consecutive time steps costs about 126 milliseconds. This corresponds to 395 frames per second, or 2.5 milliseconds per frame. In the vorticity data set the average rate of extraction for the 100 time steps is 1186 frames per second.

Referring to Table 1, the row "Search (ms)" displays the average extraction time (in milliseconds) of the isosurface cells from a single time step. The next row shows the same, but with the use of our incremental search routine. The last row shows the speed-up of the incremental search, compared to the normal search. In comparison, the average isosurface extraction time for a single time step, using the VTK implementation of Marching Cubes is about 134 milliseconds. This number can be compared to the 8.4 milliseconds for our normal search, or the 2.5 milliseconds for our incremental search.

6.3 Rendering performance

We tested the two renderers, both the high quality ShellSplatter and the lower quality fast point-based renderer, with the two data sets. The average frame rates for the total pipeline of extraction and rendering (of 512×512 images) are shown in Table 2.

Compared to the extraction times, the rendering is the processing bottleneck. The numbers in this Table are rates for combined extraction and rendering, but 80% to 98% of the time is used in the rendering step, depending on the type of renderer used. For the rendering, the number of isosurface cells is the most important factor. The average number of isosurface cells extracted from the bubble data set for the chosen isovalue is 16291; for the vorticity data set, the number of isosurface cells is on average 7617 per time step.

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 pixel 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 96^3 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 [Bordoloi and Shen 2003].

However, in order for our technique to be truly scalable to very large data sets, out-of-core functionality is required. We are currently working on an out-of-core version of the THITree in which a limited number of time steps remain in memory, and new time steps are loaded on demand. This completely overcomes the huge memory requirements and makes it possible to visualise very large trees also on systems with only a small amount of memory.

There are two possible sources of error in the display of the isosurfaces that must be investigated further. Although this did not show up in the test images, the rendering of false positive cells may cause artifacts. Also, the surface normals are stored only once over a time interval that is considered "more or less 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".

This research was partly supported by the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

References

BORDOLOI, U. D., AND SHEN, H.-W. 2003. Space efficient fast isosurface extraction for large datasets. In *Proc. IEEE Visualization '03*, 201–208.

BOTHA, C. P., AND POST, F. H. 2003. ShellSplatting: Interactive rendering of anisotropic volumes. In *Data Visualization 2003 (Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization)*, ACM SIGGRAPH, G.-P. Bonneau, S. Hahmann, and C. D. Hansen, Eds.

CIGNONI, P., MARINO, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. 1997. Speeding up isosurface extraction using interval trees. *IEEE TVCG 3*, 2 (Apr.–June), 158–170.

EDELSBRUNNER, H. 1980. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria.

KILTHAU, S., AND MÖLLER, T. 2001. Splatting optimizations. Tech. rep., Simon Fraser University.

LIVNAT, Y., SHEN, H.-W., AND JOHNSON, C. R. 1996. A near optimal isosurface extraction algorithm using the span space. *IEEE TVCG 2*, 1 (Mar.), 73–84.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. SIGGRAPH*, 163–169.

REN, L., PFISTER, H., AND ZWICKER, M. 2002. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum 21*, 3, 461–470.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: A multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH*, K. Akeley, Ed., 343–352.

SHEN, H.-W., HANSEN, C. D., LIVNAT, Y., AND JOHNSON, C. R. 1996. Isosurfacing in span space with utmost efficiency (issue). In *Proc. IEEE Visualization '96*, 287–294.

SHEN, H.-W., CHIANG, L.-J., AND MA, K.-L. 1999. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Proc. IEEE Visualization '99*, 371–377,545.

SHEN, H.-W. 1998. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proc. IEEE Visualization '98*, 159–166.

STENGER, B., MENDONÇA, P. R. S., AND CIPOLLA, R. 2001. Model based 3D tracking of an articulated hand. In *Proc. Conf. Computer Vision and Pattern Recognition*, vol. II, 310–315.

SUTTON, P., AND HANSEN, C. D. 1999. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In *Proc. IEEE Visualization '99*, 147–153,520.

UDUPA, J. K., AND ODHNER, D. 1993. Shell rendering. *IEEE CG&A 13*, 6 (Nov.), 58–67.

WESTOVER, L. 1989. Interactive volume rendering. In *Proceedings of the Chapel Hill workshop on Volume Visualization*, ACM Press, Chapel Hill, NC, USA, 9–16.

WILHELMS, J., AND GELDER, A. V. 1992. Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July), 201–227.