EUROGRAPHICS - IEEE VGTC Symposium on Visualization (2005)
K. W. Brodlie, D. J. Duke, K. I. Joy (Editors)

Isosurface Extraction Using Fixed-Sized Buckets

Kenneth W. Waters, Christopher S. Co, and Kenneth I. Joy

Institute for Data Analysis and Visualization
University of California, Davis
{waters,co,joy}@cs.ucdavis.edu

Abstract

We present a simple and output optimal algorithm for accelerated isosurface extraction from volumetric data sets. Output optimal extraction algorithms perform an amount of work dominated by the size of the (output) isosurface rather than the size of the (input) data set. While several optimal methods have been proposed to accelerate isosurface extraction, these algorithms are relatively complicated to implement or require quantized values as input. Our method is based on a straightforward array data structure that only requires an auxiliary sorting routine for construction. The method works equally well for floating point data as it does for quantized data sets. We demonstrate how the data structure can exploit coherence between isosurfaces by performing searches incrementally. We show results for real application data validating the method's optimality.

1. Introduction

Isosurface extraction has been greatly accelerated by the development of algorithms that determine active cells—cells intersected by a given isosurface. Various data structures and algorithms have been proposed, many of which are output optimal, meaning that the amount of work to identify active cells is proportional to the size of the resulting isosurface. A major impediment to these optimal algorithms is the relative complexity of the implementation. While a few optimal algorithms exist that are relatively simple to implement, they are limited to quantized data. These methods cannot be applied to scalar fields consisting of floating point values, such as data resulting from computational simulations.

We describe an average-case optimal method for active cell lookup based on commonly used data structures that can be implemented with reasonably little effort. The construction of our data structure, discussed in Section 3, involves two sorting passes over the data. Our method for identifying active cells, described in Section 4, involves a straightforward array traversal of this data structure. We show how incremental extraction of isosurface geometry can be achieved using our data structure. The time and space efficiency achieved by our technique coupled with its relative simplicity make it an attractive solution to isosurface extraction.

2. Related Work

Early techniques to accelerate the extraction of isosurface geometry focused on determining active cells spatially. Marching methods [LC87] were enhanced by the use of octree search methods [WG92] to accelerate the identification of active regions. Surface growing techniques [vKvOB*97] that follow the isosurface from a starting point cell outward were also investigated. These methods rely on knowledge of a sufficient starting set of cells, since growing a surface from a single cell can only capture one component of an isosurface that consists of many components.

Near-optimal and optimal techniques were obtained by posing the problem of active cell determination in *span space* [LSJ96]. In 2D span space, each cell corresponds to a point classified by two coordinates: the minimum intensity and maximum intensity values of the cell. These values represent the interval of isovalues contained in a cell, or alternatively the range of surfaces to which the cell contributes geometry. This pioneering observation reduces the problem of active cell determination to the problem of range finding in 2D, see Figure 1a.

Range-based methods are attractive because they are applicable to a wide variety of data sets consisting of different cell representations. The NOISE method [LSJ96] accomplishes this range search with the aid of a kd-tree in

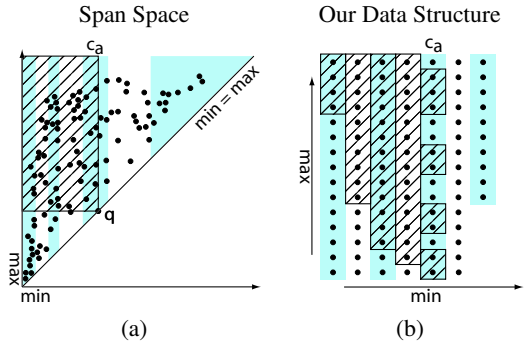

Figure 1: *Illustration of the correspondence between span space and our data structure. (a) The problem of active cell determination becomes a range finding problem in span space. An isovalue q defines a 2D active region. (b) Our method efficiently implements this range search using a straightforward 2D array representation of the span space. Cells are rendered as black points and active regions are shaded with diagonal hatches. Alternating colors are used to show the correspondence between columns (buckets) of our 2D array structure and span space. The region in span space corresponding to the last active bucket c_a is split by the vertical line induced by query q resulting in non-contiguous active cell locations in that bucket.*

near-optimal running time, $O(K+\sqrt{N})$. The ISSUE method [SHLJ96] also achieves near-optimal results $O(K+\log \frac{N}{L}+\frac{\sqrt{N}}{L})$ with the use of an $L \times L$ lattice superimposed on the span space. Cignoni et al. [CMM*97] applied interval trees accomplishing search in $O(K+\log N)$ time. The span-triangle [vRLHJ*04] is an array-based technique that achieves an optimal search time of $O(K+D)$. The span-triangle method uses the inherent quantization of many existing data sets to create an array of buckets to store cells of the volume. The quantization allows the portions of the array that contain active cells to be determined efficiently. Bordoloi et al. [BS03] use standard lossy compression and transform coding techniques to greatly reduce the memory footprint of their algorithm. The cost of this compression is error in the form of false positives in the search and increased complexity of implementation. These extra cells have to be removed at geometry creation time, increasing total extraction time.

Range-trees, kd-trees and interval trees are often non-trivial to implement. Due to the necessary traversal of internal leaf nodes, range-trees and kd-trees achieve a running time proportional to the input data size and not the output. The ISSUE technique contains several special cases, each requiring separate attention. The span-triangle technique, while relatively easy to implement, is limited to quantized data, and it is not clear how to extend the approach to floating point data sets.

Using the span space classification to represent cells of a volume data set, we accomplish optimal active cell determination using a range finding approach. Our data structure uses an array-based memory layout, similar to but distinct from the span-triangle. The data structure enables average-case optimal extraction from scalar fields consisting of floating point data as well as quantized data. Information is stored in a coherent fashion such that large blocks of active cells are stored close to one another.

3. Data Structure Construction

We organize the N cells of the data set into a logical 2D array. This array is constructed in two sorting passes. In the first sorting pass, all cells of the data set are sorted by their minimum value, forming a 1D array of sorted cells. This 1D array of cells is divided into columns, or buckets, of size B, a user-defined *bucket size*. The last cell in each bucket—the bucket cell with largest minimum value—is placed into a *min-dictionary*, which aids in the extraction process. In the second sorting pass, the cells of each bucket are resorted by their maximum intensity value. What results is a 2D array where the first elements of the rows are in increasing order based on minimum intensity value, and each column consists of elements in decreasing order based on maximum intensity value. In this way, the span space is decomposed into zones such that each zone contains at most B points. Min-dictionary points are the points closest to the right-hand border of the zone. Within each zone, the points are ranked from top to bottom. Figure 2 illustrates how our data structure is built.

The first sorting pass requires $O(N \log N)$ time, while the second sorting pass requires $O(B \log B)$ time for each of the $\frac{N}{B}+1$ buckets. Since B is a user-defined constant, the preprocessing is dominated by the first sorting pass. Therefore the construction of this data structure requires $O(N \log N)$ time. This is comparable with the preprocessing required by current state-of-the-art algorithms for active cell identification, see Table 1.

4. Extraction

The range of a cell is given by its minimum and maximum intensity values. Given an isovalue q pertaining to an isosurface of interest, a cell is defined to be *active* if

$$cell.min \geq q \quad (1)$$

$$cell.max \leq q. \quad (2)$$

We refer to (1) as the *min-criterion* and (2) as the *max-criterion*. Brute force algorithms for active cell determination traverse every cell of a data set to check when the above two criteria are true. Acceleration data structures organize the data in a preprocessing step so that cells can be trivially identified as active using few explicit comparisons.

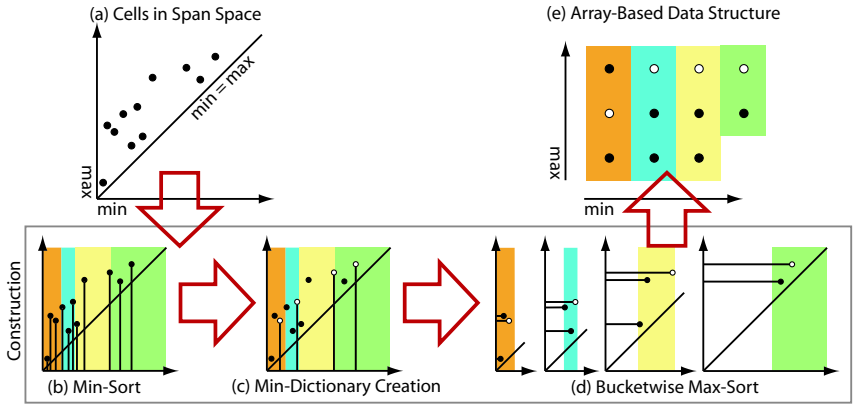

Figure 2: *Illustration of the construction of our data structure. (a) Input cells are classified by their min- and max-values in the span space. (b) The cells are sorted globally by their minimum intensity value and then placed into buckets that contain up to B cells. ($B = 3$ in this example.) Each bucket is shown in a different color. Note that the last bucket may contain less than B cells. (c) Min-dictionary cells, shown here as hollow circles, are identified. (d) The cells of each bucket are sorted according to maximum intensity value. (e) The resulting data structure is a logical 2D array.*

4.1. Full Extraction

After initialization, a *full extraction* is performed. The algorithm determines active cells using a nested loop traversal of the data structure. In the outer loop, *active buckets* (columns) of the array are traversed. In the inner loop, active cells inside the bucket are collected.

The min-dictionary aids in the identification of active buckets for the outer loop. The min-dictionary value for a given bucket is the largest minimum value of the cells of that bucket. If the min-dictionary value satisfies the min-criterion, all cells in that bucket trivially satisfy the min-criterion. Let c_0 be the first bucket of the data structure and c_a be the final active bucket. The active buckets whose cells trivially satisfy the min-criterion are buckets $c_0, \ldots, c_{a-1}$. The final active bucket c_a fails the min-criterion and is treated specially.

In the inner loop for buckets $c_0, \ldots, c_{a-1}$, we traverse the elements of each bucket and collect cells until a cell fails the max-criterion. Since the bucket is stored in decreasing order by maximum value, a contiguous block of cells within the bucket are identified as active. The rest of the cells in the bucket fail the max-criterion and thus are skipped.

The last active bucket c_a requires special processing. Since the bucket's min-dictionary value does not satisfy the min-criterion, cells inside this bucket do not trivially satisfy the min-criterion. In the span space, we can think of the *vertical* edge of the query range dividing the last active bucket into two partitions based on *minimum* values, see Figure 1a. This phenomenon creates non-contiguous blocks of active cells in the bucket c_a, since the bucket is organized according to *maximum* value, see Figure 1b. Thus, for each cell of the last active bucket, it is necessary to check that cells satisfy both the min-criterion and the max-criterion. The number of cells where both criteria must be checked is upper-bounded by B, since the bucket contains at most B cells.

4.2. Incremental Extraction

When a new isovalue q' is specified by the user, existing cell information from the previous extraction can often be retained—some cells become inactive and must be deactivated, while others become active and must be activated in the data structure. To accomplish such an *incremental extraction*, additional cell markers are maintained. These markers keep track of the last cell visited in each bucket during active cell extraction. Our extraction algorithm is enhanced to support incremental extraction by addressing two cases: when the isovalue increases and when the isovalue decreases.

4.2.1. Increasing Isovalue

When $q' > q$, let $c_0, \ldots, c_a, \ldots, c_{a'}$ be the new set of active buckets. The buckets that must be visited fall into three categories. First, buckets $c_0, \ldots, c_{a-1}$ remain active but contain contiguous blocks of cells that must be deactivated. Second, bucket c_a remains active but contains non-contiguous blocks of cells that must be activated and deactivated. Third, buckets $c_{a+1}, \ldots, c_{a'}$ have become active and must have active cells identified. Figure 3 illustrates these regions in the span space and our data structure.

The first category of buckets correspond to buckets $c_0, \ldots, c_{a-1}$. These buckets remain active, but some of the cells have become inactive and must be deactivated. The marker for each bucket is marched from its current position *up* until the marker points to the first inactive cell. Cells

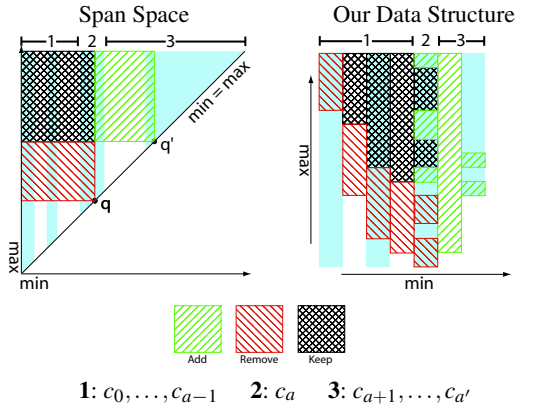

Figure 3: *Incremental extraction for increasing isovalue. Red regions contain cells to be deactivated, and green regions contain cells that must be activated. Three categories of buckets exist: (1) buckets that remain active where markers must be pushed upward, deactivating cells passed along the way, (2) the former last active bucket c_a where active and inactive cells are interleaved, and (3) additional buckets that can be traversed using a procedure similar to full extraction.*

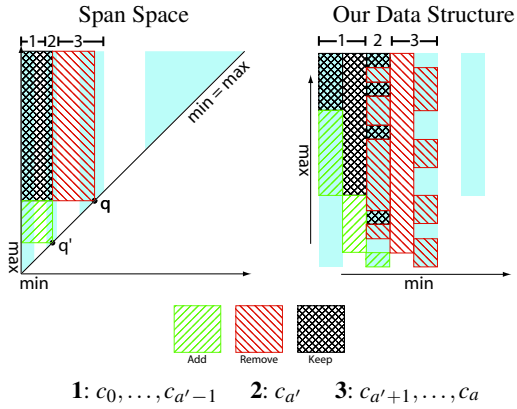

Figure 4: *Incremental extraction for decreasing isovalue. Red regions contain cells to be deactivated, and green regions contain cells that must be activated. Three categories of buckets exist: (1) buckets that remain active where markers must be pushed downward, adding cells passed along the way, (2) the new last active bucket $c_{a'}$ where active and inactive cells are interleaved, and (3) inactive buckets that can be deactivated using a procedure similar to full extraction.*

passed by the marker are no longer active and are deactivated. Since the bucket satisfies the min-criterion, only the max-criterion is checked in updating the markers.

The second category consists of a single bucket, the last active bucket from the previous query c_a. In the previous extraction, bucket c_a contained active cells that were located in scattered locations within the bucket. Bucket c_a now consists of a single contiguous block of active cells and a single contiguous block of inactive cells. Cells of this bucket must be traversed one by one to determine if a given cell needs to be activated or deactivated.

In the third category, buckets $c_{a+1}, \ldots, c_{a'}$ have just become active, and the full extraction method described in Section 4.1 is applied. The only difference is that traversal starts at bucket c_{a+1} rather than bucket c_0.

4.2.2. Decreasing Isovalue

When $q' < q$, let $c_0, \ldots, c_{a'}$ be the new set of active buckets. The buckets that must be visited again fall into three categories. First, buckets $c_0, \ldots, c_{a'-1}$ remain active but contain contiguous blocks of cells that must be *activated*. Second, bucket $c_{a'}$ is an active bucket from the previous extraction that now contains non-contiguous blocks of cells that must be activated and deactivated. Third, buckets $c_{a'+1}, \ldots, c_a$ have become inactive and cells in these buckets must be deactivated. Figure 4 illustrates these regions in the span space and our data structure.

The first category of buckets correspond to buckets $c_0, \ldots, c_{a'-1}$. These buckets remain active, but additional cells in these buckets are now active and must be activated. This corresponds to pushing the markers *down* to include cells that have become active. As before, since the bucket satisfies the min-criterion, only the max-criterion is checked in updating the markers.

The second category consists of a single bucket $c_{a'}$. In the previous extraction, bucket $c_{a'}$ contained a single contiguous block of active cells. Cells of $c_{a'}$ no longer trivially satisfy the min-criterion and must be traversed to determine if a given cell must be activated or deactivated.

The third category, buckets $c_{a'+1}, \ldots, c_a$, have become inactive, and the full extraction method described in Section 4.1 is again applied. Two modifications must be made. First, traversal starts at bucket $c_{a'+1}$ rather than bucket c_0. Second, rather than activate cells traversed by the method we must *deactivate* the visited cells.

5. Analysis and Results

Our algorithm achieves average-case optimal running time with relatively little memory usage. The data structure requires N records (one for each cell) and $\frac{N}{B}+1$ min-dictionary values (one per bucket). If incremental extraction is incorporated, $\frac{N}{B}+1$ marker cells are also necessary. Thus the memory required by our data structure is $O(N+\frac{2N}{B}+2) = O(N)$. In our current implementation, the constant factor increase in memory consumption is $3+\frac{1}{B}$ without incremental extraction markers and $3+\frac{2}{B}$ with incremental extraction markers.

In the average case, where at least one cell per bucket is

Method	Preprocessing Time	Space Complexity	Average-Case Time Complexity
NOISE (kd-tree)	$O(N \log N)$	$O(N)$	$O(K+\sqrt{N})$
ISSUE (lattice)	$O(N \log N)$	$O(N)$	$O(K+\log N+\sqrt{N})$
Interval tree	$O(M \log M)$	$O(H+M)$	$O(K+\log H)$
Span-triangle†	$O(N)$	$O(N)$	$O(K+D)$
Our method	$O(N \log N)$	$O(N)$	$O(K+B)$

N = # of cells K = # of active cells H = # of interval tree nodes
M = # of distinct intervals in data D = # of discrete values in data B = # of cells per bucket
† Only applies to quantized data.

Table 1: *Taxonomy of range-based isosurface extraction methods. Our method is optimal in the average case and has preprocessing time and space complexity comparable with state-of-the-art techniques.*

collected, our method has a runtime complexity of $O(K+B)$. This can be seen by noting that no more than three comparisons per cell collected from a normal bucket is performed. One max-criterion comparison is performed for each active cell in a bucket. For each bucket, one min-dictionary check and one failed max-criterion check are performed. When only one cell is extracted for a given bucket, the extraction cost for that cell is three comparisons. Thus, at most three comparisons per cell in a normal bucket are performed. In the last active bucket, the min- and max-criterion must be checked for every cell, and the number of cells to check is bounded by B. Thus no more than two comparisons are performed for every cell in the last active bucket. This analysis leads to a runtime complexity of $O(3K+2B) = O(K+B)$. Our experimental results support this average-case optimality, see Figure 5.

In the worst case, it is possible to traverse many buckets that do not contain active cells. Consider, for example, a pathological scenario where only one cell in a given input data set is active, and this cell is located at the bottom of the last bucket of the array. The search algorithm would traverse $\frac{N}{B}$ buckets and at most B inactive cells until arriving at the result. The complexity of the method in this pathological case is $O(\frac{N}{B}+B) = O(N)$. We note that this worst-case scenario is extremely rare in real application data. We further note that if we set $B = \sqrt{N}$, the average and worst case extraction is upper-bounded to $O(K+\sqrt{N})$, a near-optimal result.

We implemented an interval tree to compare our technique against existing methods. Two timing experiments were run for two data sets, one per data set. For each data set, 1000 full extraction queries evenly spaced over the value range of the data were performed. For each of these 1000 query isovalues, the average search time for five queries was recorded. We plotted the average search time against the number of active cells, see Figure 5. The results show that our method is competitive with the interval tree method, an optimal technique. These results also demonstrate that our algorithm on average achieves optimal performance for real data sets en-

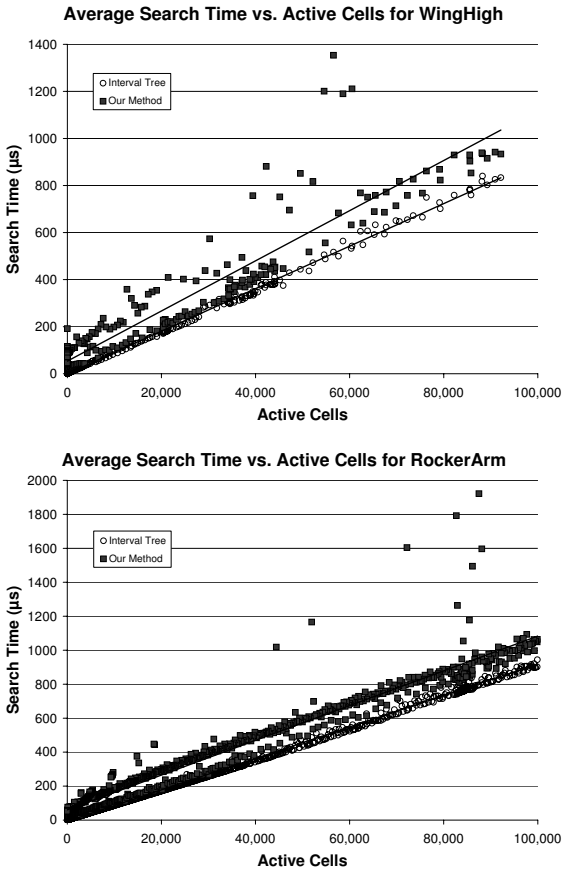

Figure 5: *Timing comparison between an interval tree implementation and our approach shown as a plot of average search time versus the number of active cells. The plots show that the running time of our method is competitive with an interval tree, an optimal technique. The regression lines show that our method achieves $O(K)$ performance on average.*

Data Set	# of Cells	Description
WingLow	287,962	tetrahedral mesh, float
WingHigh	2,000,034	tetrahedral mesh, float
Kyle	1,720,432	tetrahedral mesh, float
Engine	8,258,175	hexahedral grid, byte
Klein128	2,048,383	hexahedral grid, float
RockerArm	2,048,383	hexahedral grid, float

Table 2: *Summary of data sets used in our experiments.*

Data Set	Full	Incremental	Coherence
RockerArm	0.967 ms	0.444 ms	63 %
Engine	2.285 ms	0.432 ms	88 %

Table 3: *Comparison of full versus incremental search. The timing results reflect average search time for 100 isosurfaces over a range of values. The impact of incremental search on performance depends on data set coherence, which is measured as the average percentage of cells that remain active between queries.*

countered in practice. An idea for how our method compares to other methods can be obtained by examining Table 1, which provides a concise summary of the complexity of several existing range-based search algorithms.

We used data sets from a variety of applications, most consisting of single-precision floating point values. We used three data sets resulting from flow simulations computed over unstructured tetrahedral meshes. The *Kyle* data set can be seen in Figure 7. The *WingLow* and *WingHigh* data sets are two different resolutions of the same logical simulation. The *RockerArm* data set is a $128 \times 128 \times 128$ grid volume representing an unsigned distance field constructed around a mechanized part. The *Klein128* data set is a $128 \times 128 \times 128$ grid volume sampled from an implicit function representing a Klein bottle surface. We also applied our technique to a quantized *Engine* data set consisting of $256 \times 256 \times 128$ unsigned byte data values. Table 2 provides a summary of the data sets we used.

Incremental extraction performance depends on the coherence of the data. Table 3 shows timing results comparing our full extraction method to our incremental search algorithm. In the timing experiment, 100 queries evenly spaced in a predefined range were made with increasing isovalue. The same sequence of 100 queries were then made in decreasing order. The search times for a full extraction and an incremental extraction were measured for these 200 queries and then averaged. We used 100 values over the range $[-5, 10]$ for the RockerArm and $[25, 255]$ for the Engine.

In our experience, an appropriate choice for the bucket

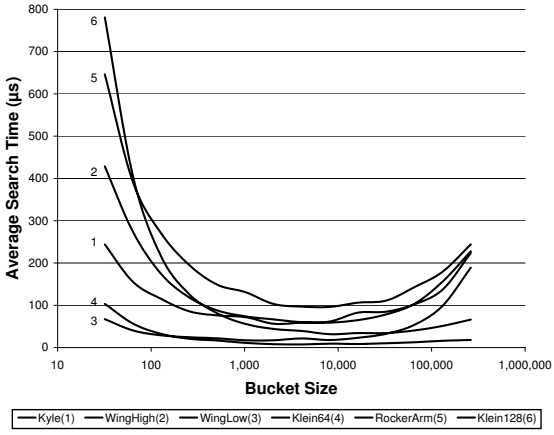

Figure 6: *Log-linear plot of average search time versus bucket size. These results indicate that a reasonable bucket size B is around 4096, regardless of the input data size.*

size B is 4096. This empirical result was obtained by extracting multiple isosurfaces from several data sets of various sizes using different values for B. For this experiment, 1000 isosurfaces were extracted for the six data sets described in Table 2. We plotted the average extraction latency against bucket size, see Figure 6. In our experiments, average active cell lookup time decreases toward 4096 and begins to increase past 8192. The consistency of this result seems to indicate that the optimal choice of B is not related to the input size of the data.

All results were generated on a dual Intel Xeon 3.06 GHz machine with 4 GB of main memory.

6. Conclusions and Future Work

The work we have presented shows that output optimal isosurface extraction is achievable using simple data structures. The ease of implementation, relatively small memory footprint, and average-case optimality of the method make it extremely attractive for active cell determination. The method is flexible enough to incorporate incremental extraction, which can be advantageous in large-scale visualization systems.

The core algorithm is relatively simple to implement. We provide a sample implementation of the construction of the data structure and the full extraction algorithm in Appendix A.

The method is competitive with state-of-the-art approaches. Like most existing methods, our technique requires initial sorting taking $O(N \log N)$ time. Our current implementation incurs only a $3 + \frac{1}{B}$ factor increase in memory consumption, $3 + \frac{2}{B}$ if incremental search is desired. For large enough B, this factor is effectively 3. Although our

method has a worst case performance of $O(N)$, it achieves an average case runtime complexity of $O(K)$, the optimal result. Our experiments with real application data support this analysis.

We hope to address several issues in our future work. The common availability of high-resolution data necessitates the development of methods capable of operating outside main memory. Our method is amenable to such an *out-of-core* implementation. Out-of-core sorting algorithms [CSS98] have been developed and can be incorporated into our technique without loss of generality. At runtime, the necessary information to maintain in main memory is the min-dictionary. Out-of-core and even parallel methods for traversing the buckets in an efficient and memory sensitive fashion can be developed using techniques borrowed from virtual memory systems. We plan to investigate these and other issues related to extending our method.

Acknowledgments

This work was supported by the National Science Foundation under contracts ACR 9982251 and ACR 0222909, the Lawrence Livermore National Laboratory under contract B523818, and by Lawrence Berkeley National Laboratory. We thank the members of the Visualization and Graphics Group of the Institute for Data Analysis and Visualization (IDAV) at UC Davis. The Engine data set was obtained from http://www.volvis.org/. We are grateful to Serban Porumbescu of IDAV for supplying the RockerArm data set. We thank Dr. Kyle Anderson of the NASA Langley Research Center and Dr. Dimitri Mavriplis of the University of Wyoming for providing unstructured data sets.

References

[BS03] BORDOLOI U. D., SHEN H.-W.: Space efficient fast isosurface extraction for large datasets. In *IEEE Visualization 2003* (Oct. 19–24 2003), Turk G., van Wijk J. J.,, Moorhead R., (Eds.), IEEE, pp. 201–208. 2

[CMM*97] CIGNONI P., MARINO P., MONTANI C., PUPPO E., SCOPIGNO R.: Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (Apr. 1997), 158–170. 2

[CSS98] CHIANG Y. J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *IEEE Visualization '98* (1998), Ebert D., Hagen H.,, Rushmeier H., (Eds.), IEEE, pp. 167–174. 7

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A high resolution 3D surface reconstruction algorithm. In *Siggraph 1987, Computer Graphics Proceedings* (July 1987), Stone M. C., (Ed.), vol. 21, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 163–169. 1

[LSJ96] LIVNAT Y., SHEN H., JOHNSON C. R.: A Near Optimal IsoSurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84. 1

[SHLJ96] SHEN H. W., HANSEN C. D., LIVNAT Y., JOHNSON C. R.: Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96* (1996), pp. 287–294. 2

[vKvOB*97] VAN KREVELD M. J., VAN OOSTRUM R., BAJAJ C. L., PASCUCCI V., SCHIKORE D.: Contour trees and small seed sets for isosurface traversal. In *Symposium on Computational Geometry* (1997), pp. 212–220. 1

[vRLHJ*04] VON RYMON-LIPINSKI B., HANSSEN N., JANSEN T., RITTER L., KEEVE E.: Efficient Point-Based Isosurface Exploration Using the Span-Triangle. In *IEEE Visualization 2004* (Oct. 10–15 2004), Rushmeier H., Turk G.,, van Wijk J. J., (Eds.), IEEE, pp. 441–448. 2

[WG92] WILHELMS J., GELDER A. V.: Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July 1992), 201–227. 1

Appendix A: Pseudo-C++ Implementation

We provide pseudocode resembling C++ to facilitate the implementation of our method. The algorithms for construction and full extraction are shown here. Additionally, a complete implementation is available online at http://graphics.cs.ucdavis.edu/~genwitt/eurovis05.html.

A C++ class named `SpanQuad` implements our data structure. In it, a record represents a cell by its minimum and maximum intensity value. The record also contains a handle to the cell the record represents. In practice, this handle can be implemented as an index into a cell array or a pointer to cell information. The record is implemented by the following code.

```
struct Record {
  CellHandle cell;
  Type min, max;
};
```

The value of `Type` depends on the data being examined. The SpanQuad class contains the following information necessary for performing active cell lookups.

```
Record* m_cells ;       // Cell records
Type*   m_min_dict ;    // Min-dict. values
int     m_last_bkt ;    // Last bucket location
int     m_last_bkt_len ;// Last bucket length
```

Construction

The following routine takes as input an array R of N cell records. The constant B represents the bucket size. We assume the routines `sortByMin()` and `sortByMax()` are

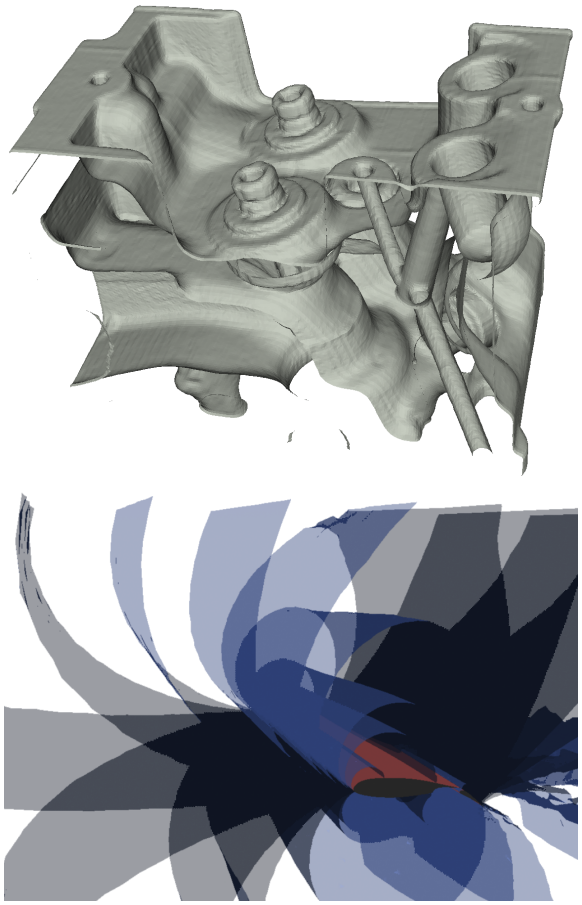

Figure 7: *Isosurfaces extracted using our method.* Above*: Single surface extracted from the Engine data set.* Below*: Multiple isosurfaces from the Kyle unstructured flow simulation data set.*

provided. The logical 2D array is maintained as a 1D array using the same memory occupied by the input array.

```
void SpanQuad::construct( Record* R, int N )
{
  // Keep pointer to cell records
  m_cells = R;

  // Sort all cells by min
  sortByMin( R[0], N );

  // Create buckets
  int num_buckets = N / B + 1;
  m_last_bkt = num_buckets - 1;
  m_last_bkt_len = N - m_last_bkt * B;

  // Create min-dictionary
  m_min_dict = new Type[ num_buckets ];
  for( int i = 0; i < m_last_bkt; i++ )
    m_min_dict[i] = m_cells[(i+1)*B-1].min;
  m_min_dict[m_last_bkt] = m_cells[N-1].min;

  // Sort each bucket by max
  for( int i = 0; i < m_last_bkt; i++ )
    sortByMax( m_cells[i*B], B );
  sortByMax( m_cells[m_last_bkt*B], m_last_bkt_len );
}
```

Full Extraction

The following routine takes as input an isovalue q and accumulates active cells. As before, the constant B represents the bucket size. The routine `addCell()` is a user-defined callback provided to accumulate active cells as they are identified by the algorithm.

```
void SpanQuad::search( Type q )
{
  int cb;  // current bucket
  int rec; // current record to examine
  int lim; // end of last active bucket

  // Normal active buckets
  for( cb = 0;
       cb != m_last_bkt && q >= m_min_dict[cb];
       cb++ )
  {
    rec = 0;
    for( Record* i = &m_cells[cb*B];
         i->max >= q && rec < B;
         i++ )
    {
      addCell( i->cell );
      rec++;
    }
  }

  // Last active bucket
  lim = (cb == m_last_bkt ? m_last_bkt_len : B);
  rec = 0;
  for( Record* i = &m_cells[cb*B];
       i->max >= q && rec < lim;
       i++ )
  {
    if( q >= i->min )
      addCell( i->cell );
    rec++;
  }
}
```