Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

Hardware-assisted View-dependent Isosurface Extraction using Spherical Partition

Jinzhu Gao and Han-Wei Shen

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
E-mail: gao@cis.ohio-state.edu and hwshen@cis.ohio-state.edu

Abstract

Extracting only the visible portion of an isosurface can improve both the computation efficiency and the rendering speed. However, the visibility test overhead can be quite high for large scale data sets. In this paper, we present a view-dependent isosurface extraction algorithm utilizing occlusion query hardware to accelerate visible isosurface extraction. A spherical partition scheme is proposed to traverse the data blocks in a layered front-to-back order. Such traversal order helps our algorithm to identify the visible isosurface blocks more quickly with fewer visibility queries. Our algorithm can compute a more complete isosurface in a smaller amount of time, and thus is suitable for time-critical visualization applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.7 [Computer Graphics]: visible line/surface algorithms

1. Introduction

Visualizing isosurfaces is an effective method to analyze three-dimensional scalar datasets. To compute isosurfaces, the Marching Cubes algorithm [1] is typically used. One bottleneck for the Marching Cubes algorithm is that the number of triangles generated from a large dataset can be enormous. Although researchers have proposed various techniques[2, 3, 4, 5, 6, 7, 8] to accelerate the process, it is still a major undertaking to compute, store, and render a large number of triangles at an interactive speed. To alleviate the problem, view-dependent methods were proposed [9, 10, 11, 12, 13]. In essence, the view-dependent methods minimize the computation and rendering overhead by extracting and rendering only the visible portion of the isosurface.

Since computing the visibility for every triangle in an isosurface can be very expensive, the visibility determination is typically done at the block level. Livnat and Hansen [9] used an octree to partition the data into small blocks, and visit the isosurface blocks in a front-to-back order. To determine the visibility of an isosurface block, the projection of the block's bounding box is compared against the screen coverage of the already computed isosurface patches. A block is invisible if its bounding box is completely occluded. Otherwise, it is visible, and the isosurface patches within are extracted. This process is applied to every block that contains the isosurface.

Although the view-dependent isosurface extraction algorithms can reduce the surface extraction and rendering cost, the visibility determination itself can become a performance bottleneck since the occlusion tests are mostly done in software. The algorithms are also inherently sequential, that is, the visibility of an isosurface block needs to be determined before the triangles within can be extracted. This makes it difficult to incorporate occlusion culling into parallel isosurface extraction algorithms.

Recently, the commercial graphics hardware such as nVidia's GeForce4 cards provide efficient visibility query functionality [17]. The hardware also allows multiple visibility queries to be issued at once, so that the visibility tests performed by GPU can be overlapped with the CPU com-

putation. In this paper, we propose a new view-dependent isosurface extraction algorithm which can take advantage of those features to reduce the occlusion culling overhead. Our algorithm consists of two phases. In the first phase, the algorithm identifies the front-most isosurface blocks, and then extracts the isosurface patches within those blocks. In the second phase, we use the partial isosurface generated in the first phase as the occluder to cull away the invisible blocks, and then extract the remaining isosurface patches.

We use the hardware visibility query features supported by nVIDIA's GeForce 4 graphics cards to identify the front-most blocks, as well as to cull away the invisible blocks. To efficiently find the front-most blocks to construct an occluder, a front-to-back traversal of the isosurface blocks is performed. Traditionally, an octree is used for this purpose. When using an octree, however, the front-most isosurface blocks can not be identified without traversing the entire tree. This is because the traditional octree traversal algorithm visits each octree node in a depth-first search order. It only ensures that a local front-to-back traversal order is not violated, but does not guarantee globally that the nodes closer to the eye will always be visited before the nodes that are farther away. Traversing the entire dataset introduces additional overhead, which can be quite significant for a large dataset. To amend this problem, we propose an alternative space partition scheme, called spherical partition, to locate the front-most blocks more efficiently. Our new scheme partitions the whole volume based on the spherical coordinate system and organizes the data blocks in a binary partition tree. Using the spherical partition scheme, we can traverse the isosurface blocks in a layered front-to-back order, so that a larger occluder can be constructed more rapidly. The algorithm also allows for an efficient update of the front-to-back order when the view changes, as well as a quick elimination of non-isosurface blocks.

In the following, related work is first discussed. We then briefly overview the occlusion culling features supported by existing graphics hardware. Section 4 gives an overview of our algorithm, followed by the details in section 5 and 6. The results are presented in section 7, and the conclusion of the paper and the future work are given in section 8.

2. Related Work

The Marching Cubes algorithm[1] was first introduced to provide a simple and robust way for isosurface extraction. Subsequently many algorithms[2, 3, 4, 5, 6, 7, 8] were proposed to improve the performance of isosurface cell search process. For a large dataset, however, the number of triangles extracted by the Marching Cubes algorithm can be huge.

Surface simplification methods [14, 15, 16] are very effective in reducing the size of the surface geometry. However, they are usually used for post-processing and thus cannot be used for the applications that require interactive isosurface extraction and rendering.

Occlusion culling is another way to reduce the size of geometry. The culling can be done in either image space or object space. Hierarchical z-buffer method [18] performs the occlusion test in image space by comparing the bounding box of the object with a hierarchical representation of the depth buffer. The method takes advantage of occluder fusion, that is, the cumulative occlusion formed by multiple occluders. Zhang *et al.*[13] proposed the Hierarchical Occlusion Map algorithm which only utilize graphics hardware to perform occluder fusion. Among object space algorithms, Coorg *et al.*[19] introduced a visibility determination algorithm based on the shadow volume or shadow frustum defined by large convex occluders. A set of visual events is always maintained for the algorithm to utilize temporal coherence. However, occluder fusion is not considered in their algorithm. Another method, aspect graph[20, 21], provides a new way for visibility determination by encoding visibility information for all possible views of the object and managing the visibility changes through visual events. All the above algorithms have successfully shown that the occlusion culling can effectively improve the rendering performance.

Occlusion culling has been used to reduce the size of isosurface geometry. Different view-dependent techniques[9, 10] have been proposed to reduce the extraction and rendering time by only extracting and rendering the visible isosurface. Contour propagation and ray casting technique are also proposed to progressively extract view-dependent isosurfaces [12]. To further reduce the rendering time, parallel rendering algorithms [22, 23] can be used. Recently, some efforts were put to parallelize both the extraction and rendering of the visible isosurfaces[11, 13].

Many occlusion culling algorithms[18, 24, 13] have already utilized graphics hardware to speed up the visibility query. However, some of the expensive operations such as bounding box projections and depth-buffer read-back are still the bottleneck. Recently, graphics cards such as HP's FX6 and nVidia's GeForce4 started to support hardware-based visibility queries. It would benefit the view-dependent isosurface extraction algorithms if such hardware capability can be fully utilized. We will briefly discuss the hardware occlusion culling feature in the next section.

3. Hardware Occlusion Culling

Hardware occlusion culling extensions are available in HP's FX6 and nVidia's GeForce 4 graphics cards. Both extensions aim to provide a simple and efficient way to rapidly determine the visibility status of a given object.

The HP occlusion extensions operate in a "stop-and-wait" manner. An occlusion culling algorithm that utilizes the extension typically works as follows. Firstly, it issues the visibility query for the bounding box of a target geometry. Then the hardware renders the bounding box and returns whether the depth buffer is modified. If the depth buffer is updated,

the bounding box will be considered visible and the geometry inside will be rendered. Otherwise, the geometry can be skipped since it is not visible. The method provides a simple way to query an object's visibility. However, it doesn't return the number of pixels that pass the test. Furthermore, the "stop-and-wait" model prevents any potential overlap of GPU and CPU computations.

To solve the problems in the HP occlusion extensions, nVidia occlusion query mechanism[17] is proposed. It returns the number of pixels passing the test and also allows multiple queries to be issued at once before checking any query result. In this way, the occlusion culling algorithm can overlap the GPU and the CPU calculations to gain better performance.

4. Algorithm Overview

The purpose of our algorithm is to extract the visible portion of an isosurface efficiently. Our algorithm consists of two parts: pre-processing and run-time view-dependent isosurface extraction.

At the preprocessing stage, the dataset is partitioned based on the spherical coordinate system. The smallest partition unit is called a "sector". All the data blocks, each of which consists of $n \times n \times n$ voxels, are bucketized into those sectors. A partition tree is used to store the partition information, which is to enable efficient front-to-back traversal to assist visibility determination.

At run time, the view-dependent isosurface extraction algorithm is performed with the hardware occlusion query support. There are two major phases in this stage: occluder construction and occlusion culling. The goal of the occluder construction is to identify the front-most isosurface blocks efficiently without extracting any isosurface patches. To achieve this goal, we traverse the spherical partition tree in a layered front-to-back order and query the visibility of the isosurface blocks' bounding boxes using the graphics hardware. We take advantage of the nVIDIA GeForce 4 graphics card's capability to overlap the CPU and GPU computation by continuously sending the queries without waiting for the results to come back, until the first several non-empty layers of the isosurface blocks are traversed. After that, our algorithm begins to check the query result for each of the blocks. If the bounding box of an isosurface block is determined to be visible, we extract the isosurface patches within. Otherwise, its visibility status is undetermined, and will require another test in the second phase due to the approximate nature of the bounding-box-based visibility test. The isosurface triangles extracted at this phase will be rendered and used as the occluder. In the second phase of the algorithm, the occluder is used to test the visibility of the remaining isosurface blocks. This is also performed by the hardware. An isosurface block can be culled away if its bounding box is completely occluded by the occluder. Otherwise, isosurface patches are extracted from the block.

Since an isosurface block needs to be tested again if it does not pass the visibility test in the first phase, it is important to only traverse the blocks that are the most likely visible when constructing the occluder to reduce the visibility query overhead. The proposed spherical partition method can achieve this goal because the front-to-back layered traversal allows a quicker construction of the occluder without traversing through the entire data set. For a dataset with high depth complexity, for example, at least half of the isosurface is invisible. Therefore, the visibility query in the occluder construction step can stop after traversing half of the isosurface blocks. To get a more complete occluder, our algorithm also extracts the triangles from the direct neighbor blocks of the visible blocks to minimize the holes in the occluder. Because the data blocks are organized in the spherical coordinate system, when the view change is small, only a small adjustment is needed to give us the new front-to-back traversal order. In the following sections, we discuss our algorithm in detail.

5. Preprocessing

The spherical partition and block bucketization need to be done only once for a dataset and can be reused when the isovalue or the viewing parameters are changed. The spherical partition subdivides the whole volume in the spherical coordinate system and the block bucketization sorts the blocks into different sectors. The run-time front-to-back traversal and visibility determination are based on this pre-calculated information.

5.1. Spherical Partition

Figure 1 shows a 2D example of the spherical partition. An example of the optimal partition for a front-to-back order traversal is shown in Figure 1(a), where the partition is always perpendicular to the eye direction. However, prepartitioning the data in this way is not feasible in practice since it is view-dependent. If the partition is done in the spherical coordinate system as shown in Figure 1(b), the layered traversal similar to Figure 1(a) can be easily achieved for any given view.

Figure 2 illustrates the spherical coordinate system. To define spherical coordinates, we take an axis (the polar axis) and a perpendicular plane (the equatorial plane), on which we choose a ray (the initial ray) originating at the intersection of the plane and the axis (the origin O). In this system, the coordinates of a point P are: the distance r from P to the origin O; the angle ϕ between the line OP and the positive polar axis (Z axis); and the angle θ between the initial ray (X axis) and the projection of OP to the equatorial plane (XOY plane).

Many data structures can be used to store the spherical partition results. The simplest one is to utilize an octree

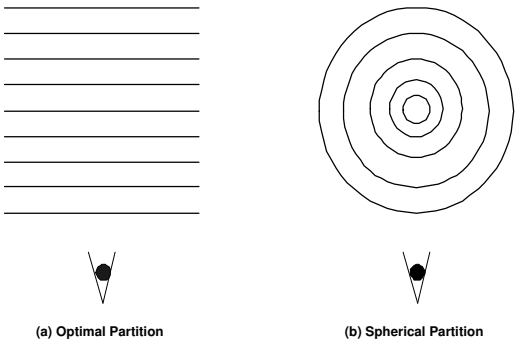


Figure 1: *A 2D example of two partition methods.*

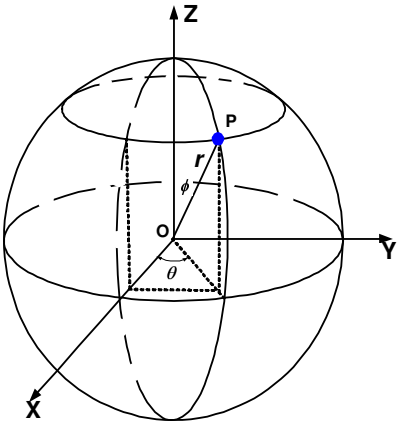


Figure 2: *The spherical coordinate system.*

except that the partition is defined in the spherical coordinate system instead of the Cartesian coordinate system. This method is easy to implement. However, it is difficult to traverse such tree in a front-to-back order.

Our algorithm partitions the volume in two steps to facilitate both data structure construction and efficient front-to-back traversal. During the first step, the partition is done along the sphere radius (r dimension), to create a layered structure as shown in Figure 1(b). This allows the traversal of the tree to be done in a front-to-back layered manner at run time. The algorithm visits the data blocks in the inner layer only after it finishes the outer layers. In the second step, the partition will be done in ϕ and θ dimensions alternatively. This partition ensures the front-to-back traversal within each layer. We will use "distance partition" to represent the partition performed in the first step and "angle partition" to represent the partition in the second step. A binary tree structure is used to represent the partition and facilitate the front-to-back traversal of isosurface blocks. The tree traversal method will be introduced in detail in section 6.1.

A 2D example of the partition and the corresponding bi-

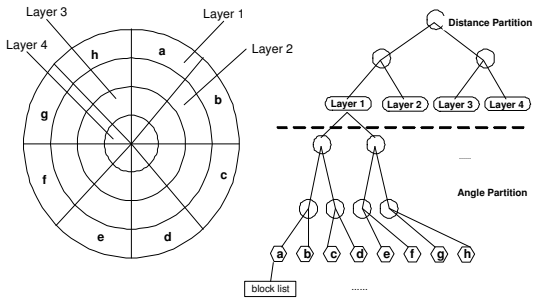


Figure 3: *An example of 2D spherical partition and its binary tree.*

nary tree structure is shown in Figure 3. The binary tree shows only the first layer's angle partition. The angle partitions in other layers are similar. At run time, the tree is traversed from the outer layer (layer 1) to the inner layer (layer 4). Within each layer, a block bucketization is performed to further sort the blocks into sectors.

5.2. Block Bucketization

In our algorithm, the whole volume is subdivided into a group of blocks. Each block consists of $n \times n \times n$ voxels. The voxels are grouped into blocks so that we can skip the empty voxels (voxels that contain no isosurface) efficiently. A block is also the basic unit for which our algorithm determines the visibility. In essence, the goal of the block bucketization in our algorithm is to sort each block to the corresponding sectors based on the following rules: A block that overlaps with two or more layers is assigned to the sector in the outer layer. Within each layer, a block is assigned to a sector if it is completely inside of the sector. If a block overlaps with two or more sectors, the block (the pointer to the actual data) will be replicated and assigned to those sectors. However, only the first one encountered in the traversal will be processed at run time. Each sector will sort and link together all the blocks assigned to it in the order of the r value of the block center. The pre-sorted block list will be used for an efficient front-to-back traversal.

After the bucketization, our algorithm needs to calculate the minimal and maximal values for each sector from the data in its blocks. The values are used for a faster search of the isosurface blocks. The sector also needs to calculate a center position which is the average of the center positions of all its blocks. The center will be used to sort the sectors.

6. Hardware-assisted View-Dependent Isosurface Extraction

With the help of the occlusion extension provided by the nVidia GeForce 4 graphics card, our algorithm can find all the visible isosurface blocks in two phases. The first

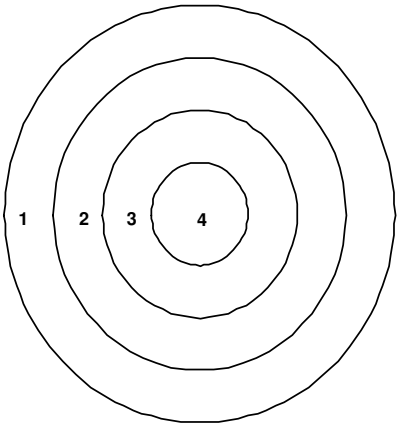


Figure 4: *A 2D example of the layer traversal: the increasing order of the numbers represents the traversal order.*

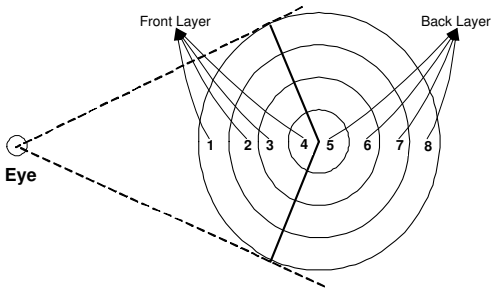


Figure 5: *A 2D example of front and back layer traversal: the increasing order of the numbers represents the traversal order.*

phase constructs an occluder from the front-most isosurface blocks, and the second phase culls away the blocks that are occluded by the occluder. The two phases work together to generate the visible portion of the isosurfaces.

6.1. Front-to-back Traversal

To quickly identify the front-most isosurface blocks, our algorithm traverses the spherical partition tree in a front-to-back order at three different levels: layers, sectors, and blocks. Starting from the outermost layer, we move toward the sphere center and traverse the layers along the way. In each layer, the isosurface sectors (the sectors that contain isosurface blocks) are traversed in an order determined by their orientations. Within each sector, isosurface blocks are visited in an order based on their distances to the sphere center. In the following, we discuss each traversal level in detail.

6.1.1. Layer Traversal

The layer traversal order is decided by the distance (radius) partition portion of the spherical partition tree. A 2D example is shown in Figure 4. The layers are traversed from the outermost layer to the innermost layer. During the traversal, the layers that contain no isosurface are skipped.

6.1.2. Sector Traversal

By traversing the angle partition portion of the spherical partition tree, the sectors at each layer will be visited and the sectors containing no isosurface will be skipped.

Each layer can be partitioned into two parts: front layer and back layer. The front layer is the portion that faces toward the eye, and the other portion is the back layer. To achieve a correct front-to-back traversal, the algorithm traverses the front layers from the outermost layer inwards, and then traverses the back layers from the innermost layer outwards, as shown in Figure 5. To build such a layer partition, the sectors in the same layer are classified into front sectors, middle sectors and back sectors. The front sectors are those which are completely inside a front layer, the back sectors are those which are completely inside a back layer, and the middle sectors are those which overlap with both the front and the back layers. To classify the sectors, we define *eye vector* as the vector from the sector center to the eye, *center vector* as the vector from the sphere center to the sector center, and *angle spread* as the largest angle difference in ϕ and θ dimensions of the spherical coordinates in a sector. A sector is classified based on the angle between *eye vector* and *center vector*. If the angle is less than $90 - 0.5 \times$ *angle spread*, the sector is classified as a front sector. If the angle is greater than $90 + 0.5 \times$ *angle spread*, the sector is classified as a back sector. Otherwise, the sector is a middle sector.

In our algorithm, the front sectors of all layers are traversed before the middle sectors of all layers, and the middle sectors of all layers are traversed before the back sectors of all layers. Among all the front sectors in each layer, the traversal order is determined based on the angle between each sector's eye vector and center vector. The sectors with smaller angles are visited first since they are closer to the eye. A 2D example of traversing the front sectors at one layer is shown in Figure 6. The same criterion is applied to the back sectors in each layer. No ordering is done for the middle sectors at this level. The front-to-back ordering of blocks inside the middle sectors is taken care of in the block traversal level.

6.1.3. Block Traversal

The goal of the block traversal is to traverse the isosurface blocks inside a sector in a front-to-back order. During the block bucketization stage, the blocks in each sector are sorted according to the block radius, which indicates the distance to the eye. To perform a front-to-back traversal, the isosurface blocks in a front sector are traversed in a decreasing

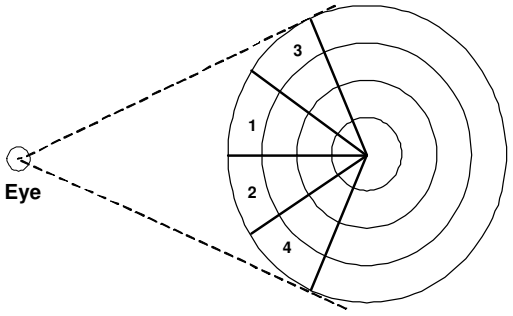


Figure 6: *A 2D example of sector traversal in a front layer: the increasing order of the numbers represents the traversal order.*

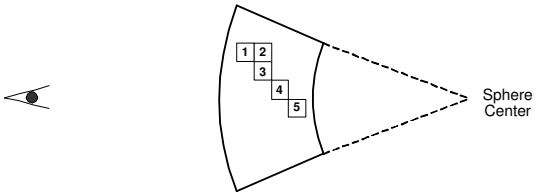


Figure 7: *A 2D example of front-to-back traversal of the blocks in the front sector: the increasing order of the numbers represents the traversal order.*

radius order(Shown in figure 7), while the isosurface blocks in a back sector are traversed in an increasing radius order. For the blocks in the middle sectors, the traversal order is determined by the angle between the eye vector and the center vector. Traversing those blocks in an increasing order of such angles ensures a correct front-to-back traversal.

If a block crosses several sectors, it is replicated and assigned to those sectors. Note that we do not replicate the data, but only the pointers to the data. In addition, the isosurfaces within the block will be extracted only once when it is first encountered.

6.2. Hardware-Assisted Occlusion Culling

After sorting the isosurface blocks in a front-to-back order by traversing the spherical partition tree, we begin to process the blocks in that order to extract the visible isosurfaces. Our occlusion culling algorithm consists of two phases: occluder construction and occlusion culling. To construct the occluder, we compare the bounding boxes of the isosurface blocks in the first several non-empty layers and select those whose bounding boxes are visible. The isosurface patches are then extracted from those blocks to form an occluder. This occluder is used to cull away the invisible isosurface blocks in the second phase.

Recently, nVidia GeForce 4 graphics cards provide hardware support for occlusion query. The programmer can use the occlusion query extension to test an object's visibility. It also allows multiple queries to be issued at once to overlap the GPU and CPU calculations. Our algorithm takes advantage of such hardware support when performing both the occluder construction and the occlusion culling.

6.2.1. Occluder Construction

In this phase, we want to identify the visible front-most isosurface blocks to construct an occluder. To achieve this, both the depth buffer write and the depth test need to be enabled first. We then render the bounding boxes of the isosurface blocks in the first several non-empty layers and test their visibility. The following is the pseudo code that utilizes the nVidia's occlusion culling extension:

1. Issuing queries for each block. Assuming there are N isosurface blocks to be tested:

```
for (i = 0; i < N; i++)
  Begin the query
    Render the bounding faces of the ith block
  End the query
```

2. Check the query result:

```
for (i = 0; i < N; i++)
  Count = number of pixels that pass
  the test for the ith block
  if (count > 0) then
    The block is visible
    Isosurface patches inside will be extracted
  endif
```

After the visible blocks are found, isosurface patches are extracted. To reduce holes in the occluder, we also extract isosurface from those isosurface blocks next to the visible blocks since they are most likely visible as well.

6.2.2. Occlusion Culling

The second phase is to cull away the isosurface blocks occluded by the occluder. After this, the rest of the isosurface blocks will be classified as visible, from which the isosurface patches will be extracted.

To identify the invisible isosurface blocks, we use the nVidia hardware in a way similar to the first phase, except that the depth buffer write should be disabled before issuing the queries. In this way, only the occluder, not the bounding boxes, is used to cull away the invisible blocks.

It is worth mentioning that such visibility test is conservative. That is, the final isosurface extracted by our algorithm (as well as the algorithms in [9, 11]) will include the true visible portion of the isosurface, plus a small portion of the invisible isosurface. However, the correctness of the final image is not compromised.

7. Results

We tested our algorithm on a PC with a 2.0 GHz P4 processor, 2 GB of memory and a GeForce 4 graphics card. Our

Iso-value	visible triangles	total triangles	%
25.5	266084	1056866	25.2
38.25	281103	1282374	21.9
63.75	312555	1291010	24.2

Table 1: *The comparison between the number of triangles extracted by our algorithm and the number of triangles in the whole isosurface.*

Number of queries	stop-and-wait	spherical partition
24020	0.190	0.090
112279	0.741	0.390

Table 2: *The visibility test time(in seconds) used by the "stop-and-wait" algorithm and the spherical partition algorithm when issuing different number of queries. The test is done on the UNC brain dataset with iso-value 25.5 and the number of queries is controlled by the data block size.*

	Octree partition	spherical partition
UNC brain	0.230222	0.105389
Leg	0.109903	0.050750

Table 3: *The visibility test time(in seconds) for the UNC brain dataset (iso-value: 25.5) and Leg dataset (iso-value: 700.0) by the octree partition algorithm and the spherical partition algorithm.*

	Octree partition	spherical partition
traverse	0.017917	0.000556
Visibility	0.230222	0.105389
Extraction	0.013917	0.012000
Rendering	0.036708	0.025042
Total	0.298764	0.142986

Table 4: *The average time(in seconds) used by the octree partition algorithm and the spherical partition algorithm for the UNC brain dataset (iso-value: 25.5).*

test datasets included a $256 \times 256 \times 145$ UNC brain dataset and a $256 \times 256 \times 308$ Leg dataset. In our tests, unless stated otherwise, a data block contained $4 \times 4 \times 4$ voxels.

Our algorithm culls away invisible isosurfaces effectively with the help of nVidia occlusion extensions. Table 1 compares the size of the visible isosurfaces extracted by our algorithm with the size of the whole isosurfaces for the UNC brain dataset. Three test iso-values were used. From the results, it can be seen that only about 25 percent of the whole isosurface was extracted.

Table 2 compares the visibility test time between the "stop-and-wait" method and the spherical partition algorithm. In the "stop-and-wait" method, after a query is issued, the program waits for the query result, extracts the isosurface if the block is visible, and then processes the next block. This is similar to the algorithm proposed in [9]. Our algorithm sends multiple queries for the isosurface blocks at once before checking the query results and extracting isosurface inside the visible blocks. Both methods were tested using the UNC brain dataset with iso-value 25.5. We used two different data block sizes, $4 \times 4 \times 4$ voxels and $2 \times 2 \times 2$ voxels. The numbers of isosurface blocks were 24020 and 112279, respectively. From the results, it can be seen that our method had smaller visibility query overhead because the "stop-and-wait" method cannot take advantage of the overlap between CPU and GPU calculations.

Our new algorithm takes advantage of the spherical partition so that a larger occluder can be constructed more quickly. Figure 8 compares the rendering images of the occluders obtained by the octree front-to-back traversal and the spherical traversal. In our test, both methods visited the first 50 percent of the isosurface blocks and queried their visibility. The octree traversal algorithm found only 2441 visible blocks (35 % of the total visible blocks), while the spherical partition algorithm found 6403 visible blocks (90 % of the visible blocks). In order to get an occluder similar to what obtained by the spherical partition algorithm, the octree-based algorithm had to visit almost all the isosurface blocks and query their viability. Table 3 compares the visibility query time between the octree partition algorithm and the spherical partition algorithm for the UNC brain dataset and the Leg dataset. It can be seen that the spherical partition algorithm spent less time in visibility query. In our test, we found that visiting about 50 % of the front-most isosurface blocks using the spherical partition can give us a pretty good occluder.

Table 4 and Table 5 compare the breakdown of the average visible isosurface extraction time between the octree partition algorithm and the spherical partition algorithm. The tests were performed by rotating the datasets about the Y axis continuously. Visible isosurfaces were generated from 72 evenly spaced view positions. The isosurface was extracted incrementally when the view changed. That is, only newly visible isosurface triangles were extracted at each frame. Except the initial view, the isosurface extraction

	Octree partition	spherical partition
traverse	0.014597	0.000556
Visibility	0.109903	0.050750
Extraction	0.004875	0.003347
Rendering	0.024319	0.025194
Total	0.153694	0.079847

Table 5: *The average time(in seconds) used by the octree partition algorithm and the spherical partition algorithm for the Leg dataset (iso-value: 700.0).*

time for the subsequent views became smaller. The visibility query time, however, was almost constant for each view. From the table, it can be seen that the spherical partition algorithm had better performance in visibility query and front-to-back traversal, and thus a better overall performance.

Since our algorithm can get a majority of the visible isosurface more quickly, it can be used for time-critical visualization applications. When the time budget is limited, our algorithm can render a more complete isosurface with a smaller amount of time. Figure 9 shows the images after 20%, 35%, and 50% of isosurface blocks were visited for the Leg dataset. Images (a) to (c) give the results generated by the spherical partition algorithm while images (d) to (f) give the results generated by the octree partition algorithm.

8. Conclusion and Future Work

We present a view-dependent isosurface extraction algorithm utilizing the occlusion query hardware. Our algorithm partitions the volume using the spherical coordinate system, which allows us to efficiently identify the front-most visible isosurface blocks. Those blocks are used to form an occluder, which can cull away the remaining invisible blocks efficiently. Our algorithm can reduce the visibility query overhead, and also extract a more complete visible isosurface in a shorter amount of time.

Future work includes utilizing the spherical partition for out-of-core and parallel isosurface extraction. We will also apply the similar idea to other visualization techniques that can benefit from efficient visibility determination. Volume rendering is one such technique. Finally, we will extend our algorithm for curve-linear or unstructured datasets.

References

1. W. E. Lorensen and H. E. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. *Computer Graphics* **21**(4): 163–169, July 1987.
2. J. Wilhelms and A. Van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*. **11**(3): 210–227, July 1992.
3. H. W. Shen and C. R. Johnson. Sweeping Simplices: A fast isosurface extraction algorithm for unstructured grids. *Proc. of Visualization '95*. pp. 143–151, 1995.
4. Y. Livnat, H.-W. Shen and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*. **2**(1): 73–84, 1996.
5. H.-W. Shen, C. D. Hansen, Y. Livnat and C. R. Johnson. Isosurfacing in Span Space with Utmost Efficiency (ISSUE). *Proc. of Visualization '96*. pp. 287–294, 1996.
6. T. Itoh and K. Koyamada. Isosurface generation by using extreme graphs. *Proc. of Visualization '94*. pp. 77–83, 1994.
7. C. L. Bajaj, V. Pascucci and D. R. Schikore. Fast Isocontouring for Improved Interactivity. *Proc. of Symposium on Volume Visualization '96*. pp. 39–46, 1996.
8. P. Cignoni, P. Marino, C. Montani, E. Puppo and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*. **3**(2): 158–170, 1997.
9. Y. Livnat and C. Hansen. View Dependent Isosurface Extraction. *Proc. of Visualization '98*. pp. 175–180, 1998.
10. S. Parker, P. Shirley, Y. Livnat, C. Hansen and P.-P. Sloan. Interactive Ray Tracing for Isosurface Rendering. *Proc. of Visualization '98*. pp. 233–238, 1998.
11. J. Gao and H.-W. Shen. Parallel View-Dependent Isosurface Extraction Using Multi-Pass Occlusion Culling. *Proc. of 2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics*. pp. 67–74, 2001.
12. Z. Liu, A. Finkelstein and K. Li. Progressive View-Dependent Isosurface Propagation. *Proc. of Vissym'01*. 2001.
13. X. Zhang, C. Bajaj and V. Ramachandran. Parallel and Out-of-core View Dependent Isocontour visualization Using Random Data Distribution. *Proc. of the symposium on Data Visualisation 2002*. 2002.
14. W. J. Schroeder, J. A. Zarge and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics* **26**(2): 65–70, 1992.
15. M. Garland and P. S. Heckbert. Surface Simplification using Quadric Error Metrics. *Proc. of SIGGRAPH '97*. **p**p. 209–216, 1997.
16. R. Shekhar, E. Fayyad, R. Yagel and J.F. Cornhill. Octree-Based Decimation of Marching Cubes Surfaces. *Proc. of Visualization '96*. pp. 335–342, 1996.

17. A. Rege. Occlusion(HP and NV Extensions). *http://developer.nvidia.com/docs/IO/2645/ATT/GDC2002_occlusion.pdf*. 2002.

18. N. Greene. Hierarchical Polygon Tiling with Coverage Masks. *Proc. of SIGGRAPH 96*. pp. 65–74, 1996.

19. S. Coorg and S. Teller. Temporally coherence conservative visibility. *Proc. of Twelfth Annual Symposium On Computational Geometry*. pp. 78–87, 1996.

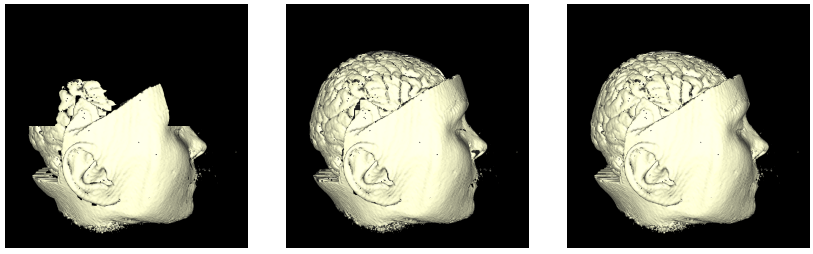
20. J. J. Koenderink and A. J. vanDoorn. The singularities of the visual mapping. *BioCyber*. **24**(1):51–59, 1976.

21. D. Eggert, K. Bowyer and C. Dyer. Aspect graphs: State-of-the-art and applications in digital photogrammetry. *Proc. of the 17th Congress of the International Society for Photogrammetry and Remote Sensing*. pp. 633–645, 1992.

22. S. Molnar, M. Cox, D. Ellsworth and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*. **14**(4): 23–32, 1994.

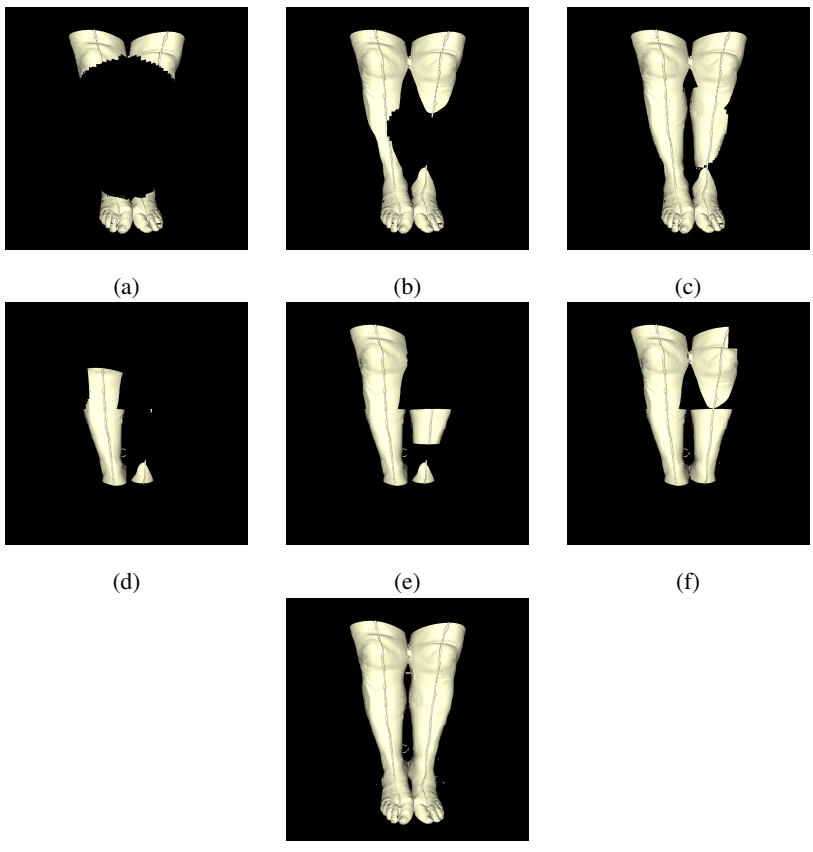
23. R. Samanta, T. Funkhouser, K. Li and J. P. Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with A Cluster of PCs. *Proc. of Eurographics Woskshop on Graphics Hardware*. August 2000.

24. D. Bartz, M. MeiSSner and T. Hüttner. OpenGL-assisted Occlusion Culling of Large Polygonal Models. *Computer and Graphics*. **23**(5), 1999.



(a) (b) (c)

Figure 8: *The occluder constructed by different algorithms after 50 percent of the isosurface blocks were queried. (a) The octree partition algorithm; 2441 visible blocks are found. (b) The spherical partition algorithm; 6403 visible blocks are found. (c) The final image of the isosurface. 6620 visible blocks.*



(a) (b) (c)

(d) (e) (f)

(g)

Figure 9: *The images generated for the Leg dataset after 20 percent, 35 percent and 50 percent of total isosurface blocks are queried during the occluder selection stage. (a)-(c) show the corresponding images generated by the spherical partition algorithm; (d)-(f) show the corresponding images generated by the octree partition algorithm. (g) shows the final image of the isosurface.*