Implicit Occluders

Sinésio Pesco[†] Peter Lindstrom[‡] Valerio Pascucci[‡] Cláudio T. Silva[†]

[‡] Lawrence Livermore National Laboratory
[†] Scientific Computing and Imaging Institute, University of Utah

Abstract

In this paper we propose a novel visibility-culling technique for optimizing the computation and rendering of opaque isosurfaces. Given a continuous scalar field $f(x)$ over a domain D and an isovalue w, our technique exploits the continuity of f to determine conservative visibility bounds implicitly, i.e., without the need for actually computing the isosurface $f^{-1}(w)$.

We generate *Implicit Occluders* based on the change in sign of $f^*(x) = f(x) - w$, from positive to negative (or vice versa) in the neighborhood of the isosurface. Consider, for example, the sign of f^* along a ray r cast from the current viewpoint. The first change in sign of f^* within D must contain an intersection of r with the isosurface. Any additional intersection of the isosurface with r is not visible.

Implicit Occluders constitute a general concept that can be exploited algorithmically in different ways depending on the framework adopted for visibility computations. In this paper, we propose a simple from-point approach that exploits well-known hardware occlusion queries.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation–Bitmap and Framebuffer Operations—Display Algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms

Keywords: isosurfaces, visibility computations, occlusion culling, marching cubes, computational geometry, volume visualization, graphics hardware algorithms

1 Introduction

Isosurfaces play a central role in the visualization of three dimensional scalar fields. By being able to compute and display isosurfaces interactively, scientists can explore their datasets, study detailed features of interest and obtain insights into the inner workings of real physical phenomena and simulated models. Because of their ubiquitous use in visualization, the computation and rendering of isosurfaces has received great attention from the visualization research community. Starting with the invention of the Marching Cubes algorithm [13], a whole effort towards the development of effective techniques for the computation and visualization of isosurfaces has been in continuous progress for over a decade (see, e.g., [1–5, 11, 12, 14, 15, 17, 19, 20]).

This paper introduces a technique that exploits the continuity of the underlying scalar field to speed up the computation and rendering of isosurfaces by performing visibility computations. The key interesting property of our technique is that it can find occluders without *explicitly* computing the isosurface, thus the term *Implicit Occluders* for our technique. We exploit auxiliary information that is usually stored for speeding up the isosurface computation. Given an octree representation of a volume, a generic cell c usually stores the minimum f_c^{min} and maximum f_c^{max} values of the scalar field in c. The computation of an isosurface with isovalue w is greatly accelerated using a coarse-to-fine traversal of the octree that skips the subtrees for which the range $[f_c^{min}, f_c^{max}]$ does not contain w.

We use this information to generate occluders by observing that if a cell c_i, with range entirely above w, is in front of a cell c_j, with range entirely below w, they together form an occluder. In fact, any ray intersecting both c_i and c_j must also intersect the isosurface in the region between them. In other words, even though c_i and c_j themselves contain no piece of the isosurface, c_i becomes an occluder when covered by c_j, and c_j becomes an occluder when covered by c_i. Moreover, the occluder is valid for any level of resolution and could therefore be used to avoid refining the surface itself.

Using the concept of Implicit Occluders we propose a new efficient from-point visibility algorithm that exploits visibility queries commonly available on graphics hardware to speed up the extraction and rendering of an isosurface by early elimination of large occluded regions. We present practical results obtained with a prototype implementation of our approach.

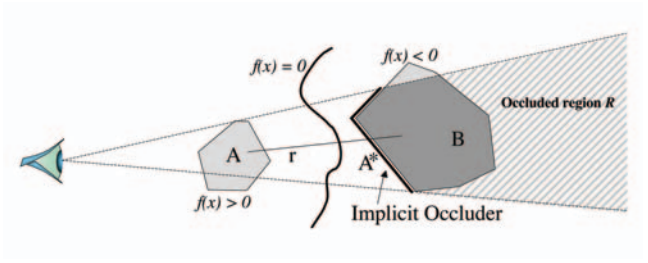


Figure 1: **Overview of Implicit Occluders.** The fundamental idea in *Implicit Occluders* is to exploit the continuity of a scalar field $f(x)$ to define regions of occlusion. In general, any *ray* traveling from a region that is above (or resp. below) the isosurface threshold to one that is below (above) has to intersect the isosurface, thus implicitly (i.e., without the need for computing the surface) defining regions of occlusion.

2 Related Work

In this section, we focus on techniques most related to our work, in particular work related to output-sensitive isosurface extraction and visibility culling algorithms designed for the display of isosurfaces.

Techniques for the efficient computation of isosurfaces started with the Marching Cubes algorithm [13]. Given a voxel grid with n voxels, the number k of active voxels (i.e., voxels that contain the isosurface) is usually only a small percentage of the whole domain. The Marching Cubes algorithm takes $O(n)$ time to compute any isosurface because it scans all the voxels in the domain. Wilhelms and Van Gelder [20] were the first to propose an optimized data-dependent technique. They build an octree on top of the regular voxel array and save for each node the minimum and maximum

IEEE Symposium on Volume Visualization and Graphics 2004
October 11-12, Austin, Texas, USA
0-7803-8781-3/04/$20.00 ©2004 IEEE

values of the scalar field within the corresponding voxels. This allows skipping the traversal of the octree nodes that do not contain the isosurface. Livnat et al. [12,17] define the notion of *span space*, from which they developed a number of efficient isosurface extraction algorithms. A related technique is the work of Cignoni et al. [5] and of Bajaj et al. [1], who independently introduced the first algorithm that is provably optimum for extraction of full resolution isosurfaces. Another set of techniques are based on the use of *seed cells* [1, 2]. These, and others [3, 4], have been extended to work in external memory, making it possible to compute isosurfaces of arbitrarily large data.

A more recent, but no less important research thread is the efficient rendering of isosurfaces. A key idea in this work is that a solution that first computes a complete isosurface and then uses a *generic* visibility culling algorithm [6] is simply too inefficient. Instead, simpler and more efficient approaches can be achieved by custom developing (or modifying) an (existing) algorithm to take visibility into account for computing and rendering a given isosurface. The first such approach was proposed by Livnat and Hansen [11] based on previous work by Greene [8]. The essential idea of their work is to build image-space occluders using hierarchical tiles while incrementally computing the isosurface. Another related technique is the approach of Gao and Shen [7]. They propose a parallel multi-pass technique, where each processor computes a piece of the isosurface, separating visible and invisible parts, until the whole surface is completed.

The main difference between our approach and both [11] and [7] is that we do not need to compute the isosurface to create occluders. The occluders are built using more generic information about the scalar field. This may have both advantages and disadvantages. It is much cheaper to compute and generate occluders that are valid for surfaces at different levels of resolution. At the same time Implicit Occluders provide only a conservative estimate of the shape of an occluding portion of an isosurface. Depending on the input data such an estimate may be too conservative to provide a practical advantage in the isosurface computation and rendering stage.

In an unpublished manuscript, Tsai et al. [18] developed independently a visibility framework based on the same concept of implicit occlusion calculations. Their work provides a nice theoretical characterization of occlusion culling queries based on ray tracing. Moreover, they propose an algorithm for extracting implicit occludes using sign distance functions from closed geometric surfaces. The theoretical foundations of our work can be cast within the same framework while we focus on the computation of implicit occluders for isosurfaces and take advantage of the existing scalar field and do not need to build the auxiliary signed distance function. We propose an efficient algorithm that takes advantage of advanced features of current graphics hardware. Moreover, our synchronization of the multi-resolution representations of the occluders and of the isosurfaces allows providing culling information for isosurfaces of undetermined level of detail.

3 Implicit Occluders

Consider a continuous scalar field $f(x)$ defined on a domain D and a real value w. We study the generation of occluders for the isosurface $f(x) = w$ with isovalue w. For sake of simplicity we assume, without loss of generality, that $w = 0$.[1] Additionally, we assume that the domain D is convex, which is true in our test cases (rectilinear grids). This limitation could also be removed but at the cost of making the discussion and the implementation of the approach much more complicated without providing any additional insight in the presentation of the fundamental concept.

[1]If w is not zero, we can simply define an auxiliary function $f^*(x) = f(x) - w$ and use f^* in place of f.

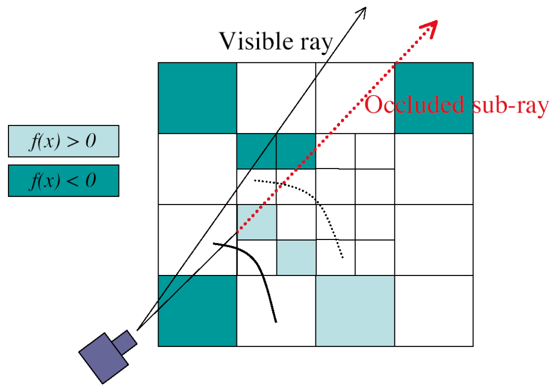


Figure 2: **Implicit Occluders and from-point visibility.** One way to exploit Implicit Occluders for optimizing the rendering and computation of isosurfaces is by using a from-region approach. The key idea here is to classify nodes of the octree to be fully above and/or below the isosurface query value. Then to use this knowledge by computing for each image-space ray the first *sign* change.

Our technique exploits the continuity of f to determine conservative visibility bounds implicitly, i.e., without computing the isosurface $f^{-1}(0)$. In particular, we generate Implicit Occluders based on the change in sign of $f(x)$, from positive to negative (or vice versa) in the neighborhood of the isosurface.

Assume that f is known to have only positive values in a region $A \subset D$ and to have only negative values in a region $B \subset D$. Since D is convex, any ray segment r connecting a point in A to a point in B is entirely contained in D. Moreover, the value of f along r changes continuously from positive to negative and therefore must have a zero crossing, which is an intersection with the isosurface $f^{-1}(0)$.

Consider now a particular viewpoint and the central projection A^* of the region A onto the boundary of B. Any segment connecting the current viewpoint with A^* must also intersect the isosurface $f^{-1}(0)$. Therefore, the region R behind A^* is completely occluded and A^* can be used as an occluder in place of $f^{-1}(0)$. Fig. 1 illustrates the idea.

Note that the construction of the occluder A^* is derived from a pure existence argument and does not require the computation of $f^{-1}(0)$. This means that the occluder A^* is valid for any other scalar function f' such that $f'(A) > 0$ and $f'(B) < 0$ (or vice versa). This can be used in the coarse-to-fine traversal of a multiresolution representation where we generate the occluder A^* as soon as we establish that $f'(A) > 0$ and $f'(B) < 0$. Any further refinement of the field can radically change the position and topology of $f^{-1}(0)$, but not the fact that the region R is not visible.

In order to use this simple concept to develop an algorithm for efficient computation and rendering of isosurfaces, a number of issues must be addressed. First of all, the regions where the function does not change sign must be computed. Then, for a given viewpoint, one must determine which regions are occluded and which regions are visible and potentially contain the isosurface. Finally, the actual isosurface extraction and rendering need to be performed.

3.1 A From-Point Algorithm

Implicit Occluders do not define a specific algorithm to be used for visibility calculations. In this paper, we develop a simple and efficient from-point approach to explore the effectiveness of Implicit Occluders. The general idea is to explore the framework of Wilhelms and Van Gelder [20] with ideas similar to the work of Livnat and Hansen [11]. Our fundamental data structure is an octree

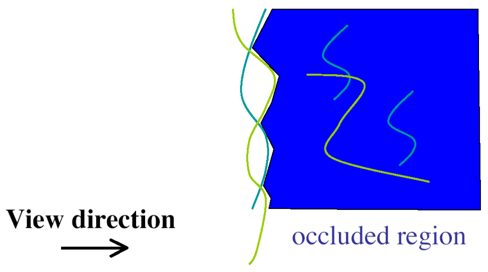


Figure 3: **Occluder envelope.** This figure depicts the occluder as the far envelope of the regions above and below the isosurface query value.

augmented with per node min-max information to allow efficient determination of the areas of the octree which might contain the isosurface (see Fig. 2). We use the nodes of the octree that are completely above and below the isovalue as potential occluders, which are rendered to create screen-space occluders. Visibility computation is achieved by adaptively pruning parts of the octree that are obscured by the per-pixel occluders built from the composition of the octree nodes by using hardware occlusion queries [6].

The complete algorithm can be divided into three main parts:

1. Build screen-space per-pixel occluders.
2. Use hardware occlusion-culling queries to prune invisible parts of the octree.
3. For the remaining (visible) nodes, compute and render the isosurface (optionally using an approach similar to [10]).

Parts (2) and (3) use previously developed techniques, see, e.g., [10, 13]. In the rest of this section, we will describe our technique for building the screen-space per-pixel occluders.

Before presenting (1), we need to review the functionality of the hardware occlusion-culling query available on current graphics boards, and what is needed to use them. The key idea is to use the current snapshot of a partially complete depth buffer to cull away geometry that can not be seen. That is, in order to test if a given object O is visible or not, we would check whether a bounding volume $bv(O)$, usually the bounding box of the object, is visible and only render the object if $bv(O)$ is visible. This query is performed without changing the contents of the depth and color buffers. If $bv(O)$ is determined to be visible, then O is rendered, and O's contents are used to update the relevant buffers. The use of these occlusion-culling queries does not bring much benefit unless some significant amount of visible, occluding geometry has been rendered. In order to achieve this, some visibility algorithms, such as the one described by Klosowski and Silva [10], propose to bootstrap the depth buffer by rendering a *guess* of the visible set [9], which is then refined by rendering more geometry until the picture is done.

In the approach described here, Implicit Occluders are used for generating a set of occluders without the need to compute any visible geometry. The algorithm to build the per-pixel occluders involves finding regions of screen-space overlap between nodes that are above and below the isosurface value, as shown in Fig. 3. Potentially, there are several such regions for a given pixel, but we are interested only in finding the closest occurrence of such a sign change (see the red ray in Fig. 2).

A naive way to compute the depth values for each pixel to serve as such an occluder would be to use a three pass approach. First, the nodes of the octree that are below the isovalue are rendered with the depth function set to keep the closest fragments to the viewer and the stencil buffer is set up to flag where pixels were written (to allow for the pixels to be identified later on). Then the same is performed

```
for each pixel $(i,j)$
    $range(i,j) \leftarrow \infty$
    $incomplete(i,j) \leftarrow$ false
for each node $N$ with $f_N^{max} < w$
    for each scan converted depth value $z(i,j)$ of $bv(N)$
        $incomplete(i,j) \leftarrow$ true
        if $z(i,j) < range(i,j)$ then
            $range(i,j) \leftarrow z(i,j)$
sort nodes front-to-back
for each node $N$ with $f_N^{min} > w$
    for each scan converted depth value $z(i,j)$ of $bv(N)$
        if $incomplete(i,j)$ then
            $incomplete(i,j) \leftarrow$ false
            if $z(i,j) > range(i,j)$ then
                $range(i,j) \leftarrow z(i,j)$
for each pixel $(i,j)$
    if $incomplete(i,j)$ then
        $range(i,j) \leftarrow \infty$
```

Figure 4: **Pseudocode for building an occlusion map using Implicit Occluders.** In the algorithm description, '*range*' is the depth (positive distance) of the occlusion map (implemented as a depth buffer) and '*incomplete*' signifies whether a pixel is covered only by a region *below* the isolevel (implemented using a one-bit stencil buffer). The isovalue is represented by w.

on the nodes that are above the isovalue. Finally, both buffers are traversed on a per-pixel basis. For each location that a pixel was written to both buffers (which can be determined by examining the stencil buffer), we can compute the *farther* depth value and store it in a buffer that will be used as the *occluder depth buffer*. This approach is quite inefficient since it requires maintaining multiple depth and stencil buffers and transferring them between the GPU and CPU back and forth.

Fortunately, there is a more efficient algorithm that does not suffer from the disadvantages cited above. We need both the depth and (one bit of) stencil buffers in the implementation, but require only one instance of each buffer and no read-backs. The scheme is based on a two-pass strategy, where all the nodes with negative function value are drawn in the first pass and the nodes with positive function value are drawn in the second pass. The stencil buffer is used to determine which pixels have been covered in the first pass, that is for which pixels a partial Implicit Occluder has been generated. The second pass is performed front-to-back,[2] and sets the correct depth for the Implicit Occluders that are completed. The depth of the Implicit Occluders that remain incomplete after the second pass is finally set back to infinity. See Fig. 4 for the algorithm. It is possible to map this to OpenGL using depth buffering and stenciling:

```
glClearDepth(1);
glClearStencil(0);
glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
```

This clears the buffers. Then, before the first render pass, we set

```
glDepthFunc(GL_LESS);
glStencilFunc(GL_ALWAYS, 1, 1);
```

Then, before the second pass,

```
glDepthFunc(GL_GREATER);
glStencilFunc(GL_NOTEQUAL, 0, 1);
```

To reset the depth values for incomplete pixels, we render a polygon that lies *behind* the octree by drawing a slightly expanded *root* node.

[2]Note that this only requires traversing the octree structure front-to-back, i.e. no actual sorting is done.

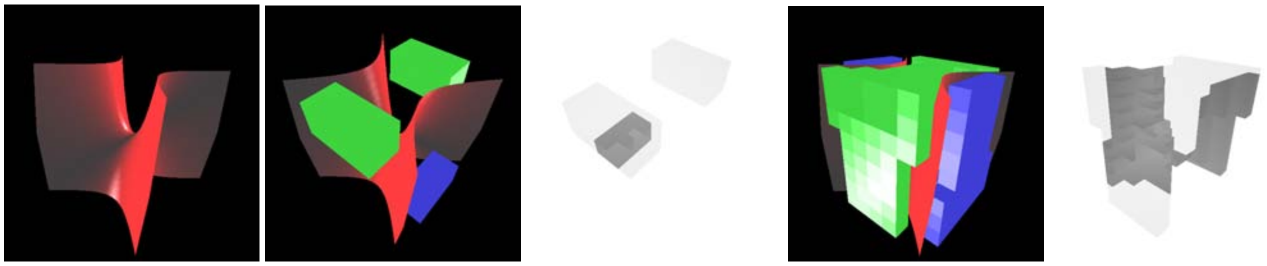


Figure 5: **Occlusion maps from Implicit Occluders.** The figure shows a simple example of occlusion maps obtained from Implicit Occluders to highlight the process. On the left, we show a hyperbolic paraboloid, $z = x^2 - y^2$, which is an isosurface of $f(x,y,z) = x^2 - y^2 - z$. The second and third image from the left show the octree nodes used for occluder generation and the computed occlusion map, respectively. The next two images show the same information for refined occluders. The octree nodes are color coded using the sign of the scalar field.

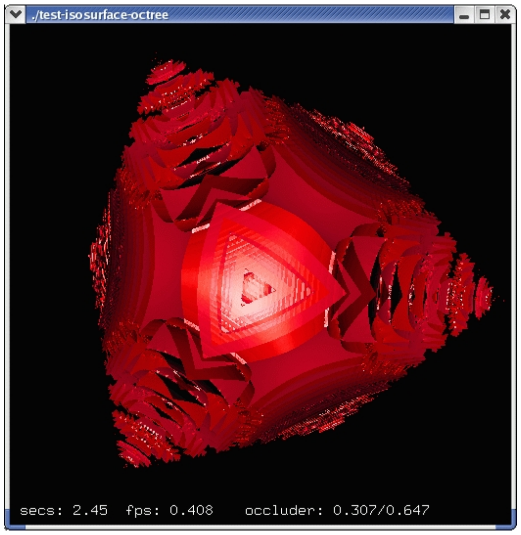
4 Experimental Results

We have implemented the algorithm described above in C++, and we used OpenGL for our graphics library. The implementation is straightforward and closely matches the description of the algorithm given in the previous section. One caveat of the implementation is that we use the Nvidia occlusion-culling query extension. We report our results on a DELL Workstation 360 configured with one Pentium 4 running at 2.8 Ghz, 1.5 GB of RAM, and an Nvidia FX 1000 card. For the reported results, we had RedHat 9 installed on this machine.

For the results reported below, we note that we do not compute and cache the actual polygons that constitute the isosurface. Instead, each time we would like to render an octree node, the isosurface contained in that node is computed and rendered. We use an output-sensitive algorithm for the computation of the isosurface though. First we use the min-max information in the nodes to efficiently determine the nodes of the octree which might contain the isosurface by performing a top-down traversal which avoids touching nodes that do not contain the isosurface. Next we test these nodes for occlusion and compute the isosurface in each visible node using the Marching Cubes algorithm [13]. Our approach is more computationally expensive than caching the polygonal mesh. On the other hand, it reduces memory consumption and allows for the modification of the isovalues interactively. This reported work is part of a larger project to make it possible to extract isosurfaces from very large datasets. In those scenarios, the idea is to use the Implicit Occluders to first determine the data that is potentially visible, and to then incrementally fetch the data from disk and render it. Finally, we note that isosurface computations can be performed directly in programmable hardware [16], thus making caching potentially unnecessary.

As we go through our results, there are several important points that need to be addressed. Possibly the most important is the savings in the number of rendered isosurface triangles obtained by our from-point algorithm. Second is the overall savings in time, which depends on our particular implementation and current hardware limitations, and we believe it is amenable to further optimizations. In our experiments we would like to highlight the costs of our algorithm. In particular, there are costs to: build the implicit occluder; test the octree nodes for occlusion; and finally extract and render the isosurface.

We have run our code on a number of real and synthetic datasets. For all examples we used an octree with a maximum depth of 7. To build the implicit occluders, we may choose to use a level of the octree above the maximum depth. This saves time in constructing the occluder, but potentially lowers its coverage area. For isosurfaces with very fine detail (such as our synthetic example), it becomes quite useful to use more refined occluders. Note that the best results in terms of total time depend on this choice, given the associated costs. In the three reported examples, the best timing results are not obtained at the maximum possible depth. The reason for this is that excessive refinement of the octree can be more expensive than computing and rendering a few additional (invisible) faces of the isosurface. On the other hand, the per-face rendering time for the occlusion map is often significantly lower than the per-face display time for the isosurface. The latter not only includes the CPU and memory overhead of extracting and storing the isosurface, but also involves a higher rendering cost, e.g., for lighting, texturing, smooth shading, etc. In contrast, we can use a very fast rendering path for the occlusion map by disabling lighting, etc., while making efficient use of a single triangle fan for the three visible quads of each octree cell. Indeed, while the total number of rendered triangles after occlusion culling may in some cases even exceed the size of the original isosurface, we are still able to obtain a significant speedup by rendering "cheaper" triangles.



./test-isosurface-octree

```
secs: 2.45   fps: 0.408    occluder: 0.307/0.647
```

Figure 6: **Synthetic scalar field.**

We first report the results for a synthetic dataset: the scalar field $f(x,y,z) = \sin(xyz/8) + \cos((x-2)(y-2)(z-2))$ defined over the interval $[-5,5]^3$. For our tests, we chose the zero isosurface, which has 2,023,877 triangles. The average time to compute and render all these triangles without using occlusion tests is about 7.01 seconds (see Fig. 6).

Fig. 7 compares the time to display (i.e., compute and render) the zero isosurface for each of 31 views. In this example, we used depth 7 to build the implicit occluders and we tested octree nodes up to depth 6 for visibility using an adaptive coarse-to-fine traver-

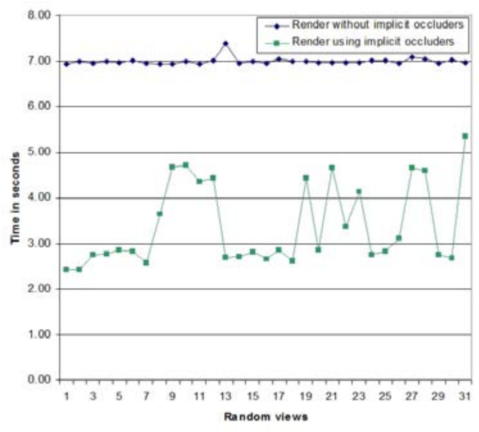


Figure 7: **Display times for the synthetic scalar field.** The green graph shows the total time for computing and rendering the zero isosurface using Implicit Occluders, and the blue graph is the total time without visibility tests.

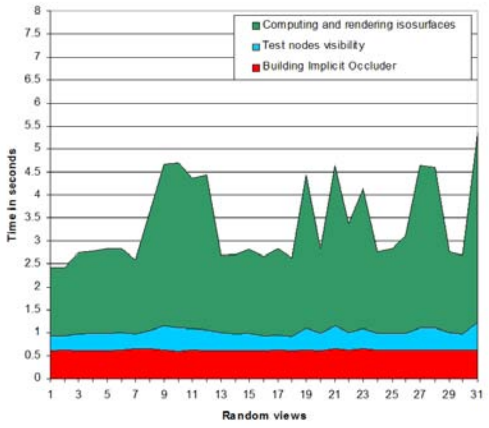


Figure 8: **Breakdown of the total frame time using implicit occluders.** The red region corresponds to the time for building the implicit occluders. The blue region corresponds to the time for testing the octree nodes for visibility. The green region represents the time for computing and rendering the isosurface contained in the nodes that passed the visibility test.

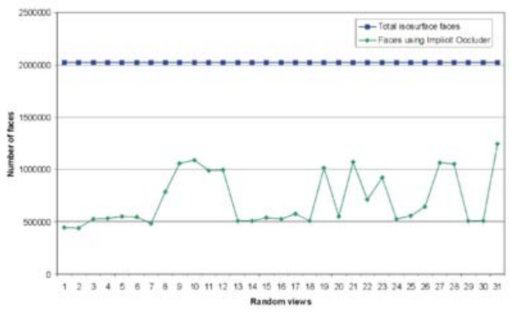


Figure 9: **Number of triangles.** The green graph shows the number of triangles in the isosurface rendered by our algorithm using an octree depth of 6 for visibility tests. The blue graph is the number of triangles without visibility computations (constant at approximately 2 million).

sal. On average, Implicit Occluders are more than twice as fast as not doing visibility computations. We take a closer look at this result in Fig. 8, which shows the times for the different phases of the algorithm. Fig. 9 shows the number of triangles computed and rendered. Observe that the time is proportional to the number of faces computed.

In Table 1, we show detailed statistics (including timings) of variations of implicit-occluder build depth versus occlusion query depth. The best result in terms of total frame time was for the case 7/6 at 2.45 seconds. Observe that in the case 7/7 we get 65,315 fewer isosurface faces than in the case 7/6. Here the additional visibility tests are more expensive than computing and rendering the extra geometry. Notice that we are not able to find any non-trivial occlusion when the build depth is 4.

Table 2 shows detailed timings for one particular view of the MRI head (see Fig. 10). Without visibility computations, the number of isosurface triangles in this example is 493,932, and it takes 1.12 seconds to compute and render this isosurface for the same reference view. Using our technique (with 7/6 build/cull depth), we lower the time to 0.43 seconds, i.e., 38% of the original time. The number of faces rendered is reduced to 69,663, i.e., 14% of the total.

Table 3 lists the results for the neghip, shown in Fig. 11. This isosurface has 246,194 triangles. Without visibility computations, it takes 0.63 seconds to compute and render the triangles. Using our technique, we lower the time to 0.29 seconds and the number of triangles to 80,564.

5 Discussion

Our experiments show that our from-point algorithm is very effective at separating visible and invisible geometry. A nice feature of the implicit framework is that it makes it possible to delay data transfer and other computations to the very end of the pipeline. This leads us to believe that our algorithm might be suited for use in remote and out-of-core visualization.

A nice feature of our from-point algorithm is that it does not require expensive *readbacks* from the GPU to the CPU, which often limit the performance of other algorithms. After we finished our implementation, we determined that one of the slowest operations is the traversal of the octree. (See Table 4.) On average, we spend 50% of the time to compute an Implicit Occluder in the traversal of the octree. A simple way to lower this cost that has been sucessfully employed by other researchers is to use a "shallow" hierarchical data structure, e.g., a two-level grid [15].

We now briefly discuss limitations of our technique. An intrinsic limitation is related to the fact that we need to find sign variations along the rays in the viewing direction. Since this is done by a sampling approach, we need to make sure that we are sampling sufficiently densely to find the sign change. For functions that change sign quite rapidly, this might cause the algorithm to essentially fail, meaning we might need to visit the entire full resolution dataset, in which case Implicit Occluders do not buy us much.

The technique described here is quite general, and it works even for non-convex occluders. There are some caveats, though. One issue has to do with supporting non-convex domains in the context of extracting isosurfaces from unstructured grids. The from-point algorithm described in this paper assumes that the scalar field is defined over a convex domain. For the non-convex domain, one needs to make sure that sign differences that happen along a ray in the viewing direction but in different "connected components" are not considered.

Occluder construction					Cull depth				
depth	time	nodes	map		7	6	5	4	3
7	0.64	533,253		Total time	3.25	**2.45**	2.64	2.96	3.47
				Cull time	1.35	0.31	0.11	0.09	0.10
				Surf. faces	378,906	444,221	542,241	684,733	819,348
				Cull nodes	400,808	78,600	14,176	2,352	392
6	0.11	48,944		Total time		3.01	2.96	3.26	3.62
				Cull time		0.46	0.15	0.12	0.12
				Surf. faces		741,273	814,817	898,944	1,017,575
				Cull nodes		113,880	18,400	2,840	456
5	0.01	3,557		Total time			4.07	4.37	5.03
				Cull time			0.28	0.20	0.18
				Surf. faces			1,111,660	1,233,561	1,423,012
				Cull nodes			25,752	3,760	536
4	0.00	121		Total time				7.01	6.97
				Cull time				0.26	0.24
				Surf. faces				2,023,877	2,023,877
				Cull nodes				4,600	584

Table 1: Detailed statistics for the synthetic dataset using different occluder build and cull depths. The depth, time and number of octree nodes rendered to build the occlusion map for a given level are listed on the left. The cull time and number of octree nodes (Cull nodes) involved in the occlusion queries (i.e., tested for occlusion), as well as the total frame time and number of rendered isosurface triangles are shown on the right for various maximum cull depths.

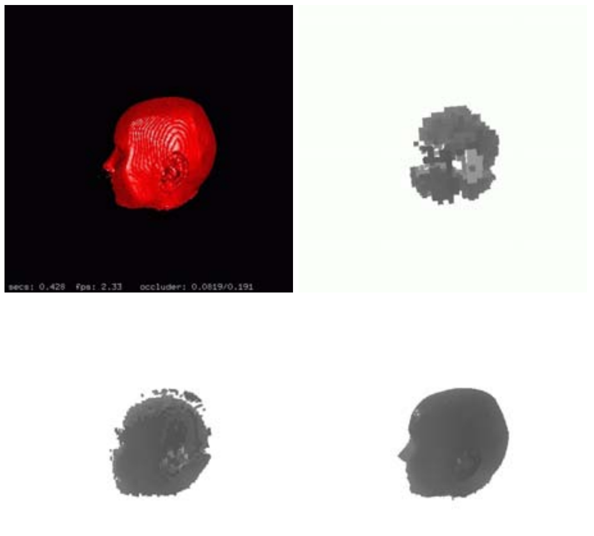


Figure 10: The MRI head and its level 5, 6, and 7 occluders.

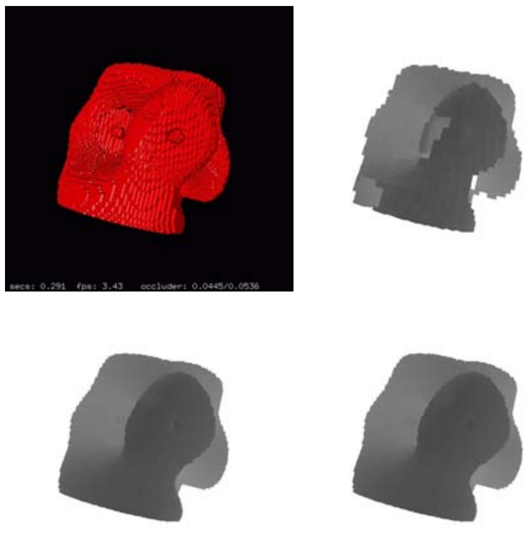


Figure 11: The neghip and its level 5, 6, and 7 occluders.

Occluder construction				Cull depth				
depth	time	nodes		7	6	5	4	3
7	0.19	169,903	Total time	0.58	**0.43**	0.46	0.58	0.77
			Cull time	0.27	0.08	0.03	0.03	0.03
			Surf. faces	55,494	69,663	103,478	167,891	258,007
			Cull nodes	80,616	21,632	5,912	1,656	440
6	0.05	27,382	Total time		0.58	0.57	0.66	0.86
			Cull time		0.14	0.06	0.04	0.04
			Surf. faces		180,783	216,203	272,956	364,565
			Cull nodes		39,936	8,520	2,056	464
5	0.01	4,207	Total time			0.88	1.02	1.05
			Cull time			0.14	0.12	0.07
			Surf. faces			346,430	405,463	460,242
			Cull nodes			11,896	2,424	480
4	0.001	716	Total time				1.13	1.10
			Cull time				0.13	0.08
			Surf. faces				475,274	485,599
			Cull nodes				2,536	480

Table 2: Results for the head dataset (size: $256 \times 256 \times 109$), isolevel $w = 24.58$ with 493,932 triangles. See Table 1 for a detailed description.

Occluder construction				Cull depth				
depth	time	nodes		7	6	5	4	3
7	0.17	165,604	Total time	0.64	0.44	0.39	0.41	0.45
			Cull time	0.34	0.11	0.04	0.02	0.02
			Surf. faces	59,578	68,557	79,178	91,010	108,889
			Cull nodes	98,496	27,512	7,320	1,824	400
6	0.05	38,974	Total time		0.33	**0.29**	0.33	0.35
			Cull time		0.11	0.05	0.02	0.03
			Surf. faces		70,961	80,564	91,576	110,049
			Cull nodes		28,008	7,392	1,840	400
5	0.01	8,372	Total time			0.33	0.35	0.36
			Cull time			0.11	0.09	0.05
			Surf. faces			92,075	104,415	126,284
			Cull nodes			8,368	2,024	408
4	0.002	1,374	Total time				0.50	0.55
			Cull time				0.09	0.06
			Surf. faces				170,669	207,957
			Cull nodes				3,168	568

Table 3: Results for the neghip dataset (size: $66 \times 66 \times 66$), isolevel $w = 1.03$ with 246,194 triangles. See Table 1 for a detailed description.

Dataset	Build depth	Build time	Traversal time	%
Synthetic	7	0.641	0.353	55%
	6	0.110	0.075	68%
	5	0.011	0.001	9%
Head	7	0.194	0.096	49%
	6	0.048	0.022	46%
	5	0.006	0.004	67%
Neghip	7	0.164	0.081	49%
	6	0.054	0.025	46%
	5	0.009	0.005	56%

Table 4: The time spent traversing the octree as a percentage of the total time to build the Implicit Occluder.

6 Conclusion and Future Work

In this paper we introduced the novel concept of Implicit Occluders for isosurfaces of a scalar field. Before extracting the isosurface, we build the Implicit Occluders by exploiting the standard min-max information normally used only for avoiding traversal of empty regions.

We implemented the scheme for the case of rectilinear grids using an octree multi-resolution representation. Our experiments show that the amount of occlusion generated is sufficiently large to save the cost of computing large portions of the isosurface. The main advantage of this scheme is the ability to generate conservative occluders from geometric information that is potentially much coarser than the actual geometry being rendered. Therefore, the entire visibility culling process can be managed with much less space and time resources. In practice, the difference in resolution between the rendered surface and the geometry used to generate the occluders becomes the parameter driving the tradeoff between the quality of the occluders generated and the amount of resources devoted to the visibility culling computation.

We plan to experiment with a more extensive use of the implicit occluders. In particular we plan to develop a tighter coupling of the occlusion culling with the isosurface refinement process to avoid refining invisible regions, similar to how the min-max information avoids refinement in empty regions. Moreover, we plan to experiment with the scalability of the scheme for very large datasets and develop a parallel and out-of-core implementation of our approach.

Acknowledgments

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. Cláudio T. Silva is partially supported by the DOE under the VIEWS program and the MICS office, and the National Science Foundation under grants CCF-0401498, EIA-0323604, and OISE-0405402. Sinésio Pesco is on leave from Pontificial Catholic University (PUC) of Rio de Janeiro. He is partially supported by a post-doctoral fellowship from CAPES-Brazil.

References

[1] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, pages 39–46, 1996.

[2] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Symposium on Parallel Visualization and Graphics*, pages 97–104, 1999.

[3] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *IEEE Visualization '97*, pages 293–300, 1997.

[4] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization '98*, pages 167–174, 1998.

[5] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. *1996 Volume Visualization Symposium*, pages 31–38, October 1996. ISBN 0-89791-741-3.

[6] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[7] J. Gao and H.-W. Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *IEEE Symposium on Parallel and Large Data Visualization and Graphics 2001*, pages 67–74, 2001.

[8] N. Greene. Hierarchical polygon tiling with coverage masks. *Proceedings of SIGGRAPH 96*, pages 65–74, August 1996.

[9] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, April - June 2000. ISSN 1077-2626.

[10] J. T. Klosowski and C. T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[11] Y. Livnat and C. Hansen. View dependent isosurface extraction. *IEEE Visualization '98*, pages 175–180, October 1998. ISBN 0-8186-9176-X.

[12] 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, March 1996. ISSN 1077-2626.

[13] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):163–169, July 1987.

[14] G. M. Nielson and B. Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91*, pages 83–91, 1991.

[15] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, 1998.

[16] V. Pascucci. Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. In *Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym)*, pages 293–300, 2004.

[17] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, pages 287–294, 1996.

[18] R. Tsai, L.-T. Cheng, P. Burchard, S. Osher, and G. Sapiro. Dynamic visibility in an implicit framework. Technical Report UCLA CAM Report 02-06, University of California at Los Angeles, February 2002. ftp://ftp.math.ucla.edu/pub/camreport/cam02-06.ps.gz.

[19] T. Udeshi and C. D. Hansen. Parallel multipipe rendering for very large isosurface visualization. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, 1999.

[20] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992. ISSN 0730-0301.