Visibility Computation for Interactive Visualization of Complex Enclosed Environments

Roni Yagel and William Ray

Department of Computer and Information Science
The Ohio State University, Columbus, OH

Abstract

In many visualization applications as well as computer graphics we need to consider large amounts of objects in order to render one image. In many cases rendering can be preceded by a culling phase which employs simple mechanisms to reject most of the objects. As a result, only a very small portion of the model has to actually go through the time consuming process of hidden object removal. We report on such a culling mechanism that is based on regular space subdivision into cells. These cells are classified into opaque and transparent and a cell-to-cell visibility algorithm determines, for a given cell, which other cells are potentially visible. Only the objects in the potentially visible set of cells are actually submitted to the hidden object removal algorithm. We report on the 2D implementation of the algorithm and its performance for walkthrough the interior of capillary blood vessels. We propose several improvements to our methods and suggest a simple way to extend it to 3D.

1. Introduction

Visualization systems provide a visual presentation of data that is too complex or voluminous to be analyzed in its raw format. Commonly, such systems face the task of generating images from a model containing millions of elements [18]. In some cases the mapping of the raw data into geometric objects will generate a very large number of objects [4]. In applications requiring interactive rendering it is often necessary to optimize the renderer to decrease rendering time. An obvious solution is to reduce the amount of data which is rendered per frame, either by *scene-simplification* or by *visibility-culling*. In scene-simplification methods, less important objects (e.g. small, distant, or peripheral objects) or object detail (e.g., texture) are not rendered or replaced by simpler objects [6][10][15]. Our work belongs to the second approach of visibility culling which tries to reject objects that are certainly not visible, by using simple mechanisms. Familiar examples of this approach are *clipping* and *backface culling* [7] which are view dependent and have to be repeatedly computed, on-the-fly, for each image. In some cases it is possible to precompute and store some view-independent visibility-culling information. The method we propose here can support both on-the-fly as well as precomputed visibility culling which can trade memory for run-time rendering speed.

In the general case of rendering a world which is sparsely populated with objects and in which the observer may be freely positioned anywhere in or around the data elements (i.e., a large percentage of the scene is visible from any observer location) it is useless to attempt to pre-determine those subsets which are sufficient to render the scene. There exists, however, a class of data sets for which the subset of visible objects for a freely moving observer is a small part of the complete model. This class consists of, for example, data sets which entirely enclose the observer and for which the observer-object distance to an occluding object, in any direction, is much less than the size of the whole data set. Many such data sets are typified by an interconnected series of passages or *voids*. Some examples of this variety of data set are the interior of: building (Figure 2a), sewer systems, ventilation ducks, submarines, subway tunnels, plant roots, caves (Figure 2b), and the metaphor we use, blood vessels. For any observer location interior to such a data set, the local walls occlude the majority of the data. Even when walking through a seemingly non-occluding environment such as the outdoors, in some cases, such as in hilly regions or a densely built urban area, most of the model is occluded by solid regions (Figure 2c). In such a situation, it becomes feasible to preprocess the data and determine what might be visible from any location. This information can be stored and used to expedite rendering when the scene is explored by the observer (*walkthrough time*).

(a) (b) (c)

FIGURE 1. Occluding environments where solid regions are dark: (a) Axially aligned planar walls (e.g., buildings), (b) arbitrary walls (e.g., caves, blood vessels), and (c) densely occluding outdoors scenery (e.g., mountains, city).

In this paper we present a method for visibility preprocessing in viewer-enclosing environments. This preprocessing allows a rendering algorithm to consider only a small subset of the data and still be able to completely render the scene from any viewpoint. We start by surveying previous solutions (Section 2). We describe our approach (Section 3) and its core component - the cell-cell visibility algorithm (Section 4). Experimental results illustrating memory requirements and the portion of the total data set eliminated from consideration in a microvascular dataset are given in Section 5. Several enhancements to the algorithm aimed at reducing memory consumption, speeding up preprocessing and rendering are provided in Section 6. We also describe there one possible method of extending the algorithm to three dimensions as well as other future plans. Finally, we conclude with a short summary of our achievements.

2. Previous Work

Previous attempts to provide visibility precomputation have followed various avenues. Many rely on some form of spatial subdivision and subsequent calculation of visibility for each region in the subdivision. An object is said to be visible to a region if it visible from at least one point within that region. We restrict our review to this category of solutions to which our solution belongs also.

While it might be feasible, in some applications, to attempt to accurately compute the *potentially visible set (PVS)* of objects from within a region [16], our algorithm's goal is to efficiently compute a superset of (PVS). The goal of this type of algorithms is not to accurately solve the hidden object removal problem for a region, but rather to find an approximate solution - one that will contain the PVS and maybe some negligible number of hidden objects. In walkthrough (rendering) time, this approximate, yet greatly reduced, set is submitted to the renderer for final hidden object removal and image generation.

Because several of the visibility computation techniques may be applied to multiple methods of subdividing the data set, we will examine the spatial subdivision and the visibility computation as separate issues.

2.1 Spatial Subdivision

The goal of the spatial subdivision is to divide, or discretize, the data into regions, also called *cells*, so that visibility may be calculated from each region to every other region. The assignment of the data elements to the cells is called the *discretization* of the model. An ideal subdivision would produce a discretization such that any point interior to any region is visible from any other point interior to the same region. This would require that all opaque elements of the data set fall along the region boundaries. This is not always practical with some subdivision techniques.

We distinguish between two main types of subdivision: *regular* and *data-driven*. The regular subdivision embeds the data in a regular grid which cells are axially-aligned parallelopipes (Figure 2a, b). The data-driven subdivision divides objects space along the planes defined by the object's surfaces (Figure 2c) [8]. In the regular approach we distinguish between *uniform subdivision* in which all cells are of the same size [9] (Figure 2a) and *nonuniform subdivision* (Figure 2b).

Regular Uniform Grid. (Figure 2a) This type of grid has the advantages of ease of application, easy neighbor finding, and low storage overhead. Its disadvantages include the inability to optimize region size to data element size for the entire data set, and the inability to produce an ideal subdivision of the data.

Regular Non-Uniform. (Figure 2b) One example of this type of grid is the octree subdivision [14] which does a better job in optimizing region size to fit data behavior at the cost of additional storage and processing overhead.

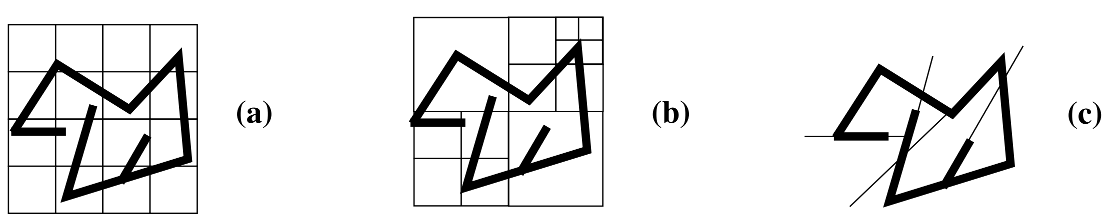

FIGURE 2. Space subdivisions schemes: (a) regular uniform, (b) regular non-uniform, and (c) data-driven.

<u>Data-Driven.</u> (Figure 2c) This method solves the problem of requiring infinite subdivision to place opaque data on region boundaries by choosing its subdivision boundaries to be along opaque elements in the data. One familiar example of this partitioning method is the Binary Space Subdivision Tree (BSP Tree) [8]. Unfortunately there is no convenient way to trace rays through the structure of the discretization, providing an obstacle to some visibility determination methods.

2.2 Visibility Precomputation

Visibility precomputation methods range from non-existent to too computationally complex to be implemented or used in practice. At the simple end of the spectrum, the discretization of the data into a grid structure can be considered to be a sort of visibility preprocessing. The grid structure can be efficiently intersected with the viewer's cone of sight, and only those data elements interior to the cone of sight are rendered [11]. This of course suffers from the problem that it may report as visible an arbitrarily large portion of the data, when in fact, the entire rendered image may consist of one element of the data set which is very close to the observer.

The stochastic ray casting approach performs discrete sampling by casting a finite number of "feelers" from the boundary of the cell. Objects hit by these rays are included in the cell's potentially visible set (PVS) [1]. In practice, this method requires that an impractically large number of rays be cast from every cell. Another method of visibility precomputation treats areas of the data set as light sources, and calculates the regions of the data set which are "illuminated" by these light sources [1]. This method is computationally very expensive, and does not appear to have been usefully implemented [1][2].

The *portal stabbing* method uses a data-driven space partition to create ideal regions with opaque elements occurring only on region boundaries [16][17] (e.g., walls). It finds the non-opaque boundary segments between cells, called *portals* (e.g. doors, windows). It then computes what sequence of these portals must be traversed by a sight-ray for one region to be visible from another. It then computes cell to cell visibility by trying to stab the sequence of portals with a sight ray, failure of which indicating that the two cells are mutually occluded. This method suffers from several disadvantages [12]: it requires expensive precomputation and a great deal of memory to store its results. It is specifically tailored to architectural models and does not support non-planar walls, for example, and, in its current implementation, it requires axially aligned walls. In visualization applications it is especially important to support arbitrary shaped walls. Finally, although the method can be extended to 3D [17] it seems that its implementation is extremely complex and its computational complexity is even higher.

In this paper we present a new approach that is based on regular uniform space subdivision by a cartesian grid, also called the *embedding raster*. As a result of discretization some cells will be assigned data elements which are stored in a list called *data list* (DL), while the rest of the cells will remain empty.

For each of the cells, a subset of the data which is sufficient to completely render the scene from anywhere inside that cell will be calculated (the *potentially visible set (PVS))*. To conserve memory, instead of storing, in each cell, the entire list of data elements which can be seen from the cell, we can store just the coordinates of the cells which contain them. These are stored in a list called the *visible* *cell list (VCL)*. At render time, the VCL is traversed, and the list of data elements (DL) in those cells is submitted to the rendering process. We now turn to describe our approach in more detail.

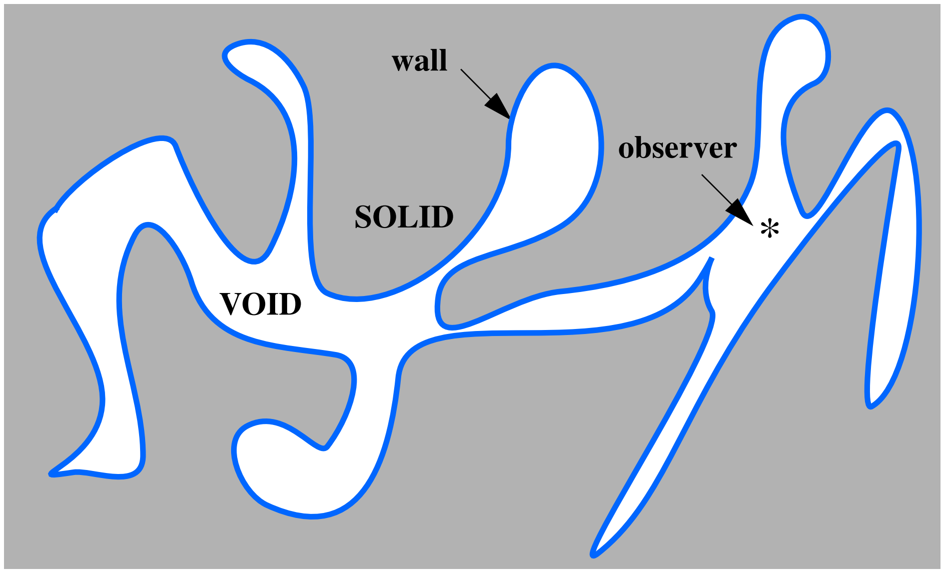

FIGURE 3. A possible complex void with passage-like structure.

3. The Raster Approach to Visibility Preprocessing

We consider an observer which exists in a *void*, or area of space which contains no opaque data. Figure 3 shows a common example of such a void that consists of a series of interconnected passages, where visibility from one passage to another is not possible in most cases. Anything which cannot be seen through, and, in general, the area exterior to the void, will be considered to be *solid* and opaque. For the purposes of this paper, the void boundaries will be exemplified by line segments. Although void regions or passageways are not necessarily connected, for optimal performance the walls (e.g., the line segments) must be well connected such that there are no gaps allowing the void to contact the solid. In the general case walls are not restricted to line segments, but may be any variety of completely connected data elements. For example, when scanning an image of a user enclosing environment, walls are discrete curves[1]. Although our approach is useful in many applications we use the metaphor of blood vessels which emphasizes the irregular nature of the surrounding environment. In this context, data elements stand for the walls of the blood vessels and we will use these terms interchangeably.

The visibility preprocessing algorithm can be described as follows:

1. **embed** the world-space in a regular *embedding grid*.
2. **assign** data elements to cells (discretize) **and store** them in the *data list* (DL).
3. **classify** cells into *void-cells*, *solid-cells*, and *data-cells*.

---

1. Discrete curves have to be 4-connected, that is, be defined by a sequence of adjacent pixels such that every pair of consecutive pixels share one common coordinate (face adjacent). Therefore, (x, y) is 4-connected to $(x \pm 1, y)$ and $(x, y \pm 1)$, and 8-connected to $(x \pm 1, y \pm 1)$. For more information on discrete topology see [5].

4. **for** each void-cell **or** data-cell V **do**
5. **for** each data-cell U **do**
6. **if** U is visible from V **then**
7. **add** U to the visible-cell-list (VCL) associated with V

During walkthrough, when the observer is in cell V:

8. **for** each cell U in the visible-cell-list (VCL) associated with V **do**
9. **for** each data element D in the data-list (DL) associated with U **do**
10. **render** D

In the following sub-sections we describe each of these steps in more detail. Section 4 is devoted to the core operation of our algorithm, that is, the cell-to-cell visibility determination which implements step 6.

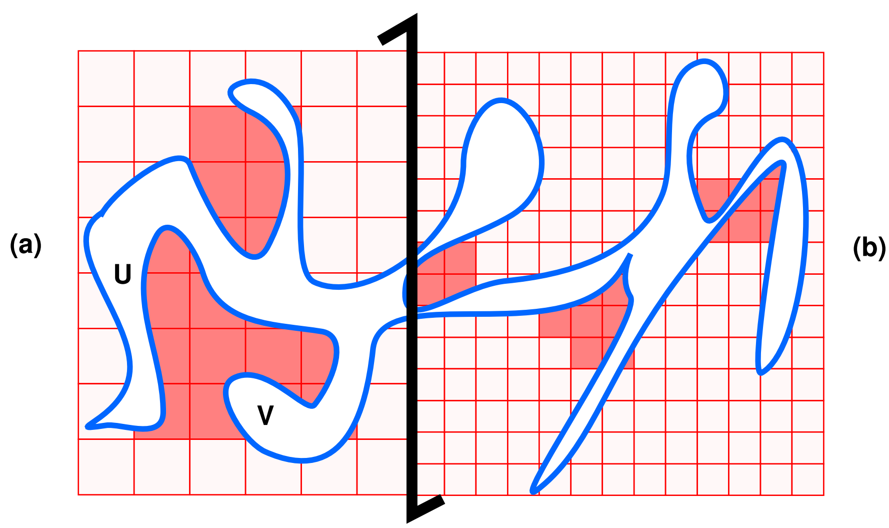

FIGURE 4. Appropriately sized cells. Some of the wall cells are painted darker. In (a) mutually invisible (U,V) passages are connected by wall cells while in (b) most passage ways are separated by solid cells.

3.1 Creating Regions

The first step in processing the data is to create regular regions, or *cells* for which we calculate visibility. In the two dimensional case this results in covering the data set with a grid of squares (see Figure 3). The grid must be at least as large as the bounding rectangle of the data set. The optimal cell size varies with the data set. As the cells grow larger, so does the degree to which any cell's visible list overestimates the amount of data required for any point in the cell (see Figure 3a). With cells sized very small the visible list for each cell can be determined with little overestimation (see Figure 3b). However, the number of necessary visible lists increases, and the difference between the visible lists for any two neighboring cells becomes smaller.

If the size of a cell is such that it is larger than the average inter-passage distance (Figure 3b) it is likely that many cells will span two separate passages. The visible list for any such cell will include data from both passages and this cell will function as a "window" between the passages, thereby reducing the efficacy of the preprocessing.

The memory constraints and rendering speed requirements for the system will define the optimal cell size, where the lower bound on the cell side length is defined by the requirement for sufficient memory to hold all the cell visible lists, and the upper bound on the cell side length is defined by extraneous non-viewable data in a visible list causing walkthrough rendering time to exceed some threshold.

At the end of this step object space is embedded in an optimal regular grid. We now need to assign data elements to the cells they intersect and classify cells according to their role in the algorithm.

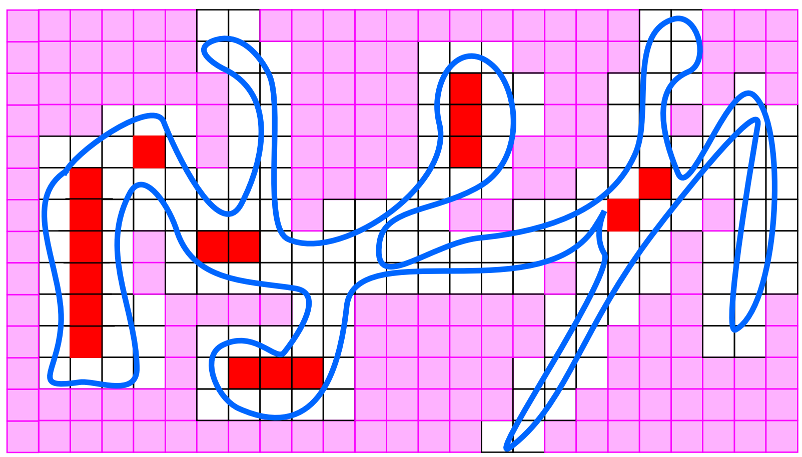

FIGURE 5. Classification of the cells. Note that all wall cells ( ) are 4-connected and prevent a line of sight from the interior to pass through a corner and reach the solid cells ( ). The interior cells ( ) are not necessarily connected.

3.2 Distributing the Data to Cells and Cell Classification

Once an appropriate space subdivision has been chosen, the data must be placed into the cell structure. This is accomplished by marking each cell which holds any part of any data element as a *data cell* (also called *wall cell)* (the cells in Figure 5). Each element of the data set is classified as belonging to every cell into which any portion of it falls, therefore, large objects will be duplicated in many cells. In the case of geometrically defined walls (e.g., lines) the discretization can be driven by a 4-connected scan conversion algorithm. The wall cells define the interface between the void and the solid in the cell space and contain all the visible data in the data set. They may also contain empty space, so the observer may be positioned inside a wall cell. Wall cells are also required to be well-connected so that a sight ray from the interior region will not be able to reach the solid without intersecting first the walls.

Cells are classified as *void cells* if they do not contain data elements but may contain the observer (e.g, the vessel's interior space, the cells in Figure 5) or as *solid cells*, which are those regions where the user can not walk (e.g., the rock, the cells in Figure 5).

3.3 Determining Cell-to-Cell Visibility

The final task of the preprocessing is to determine the *visible cell lists (VCL)* for each cell which an observer may visit (i.e, void and wall cells). For point A to be visible from point B, a ray of sight must be able to pass from point A to point B. A-to-B visibility implies that a straight line must be

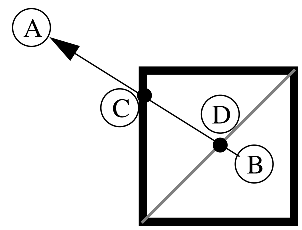

FIGURE 6. Visibility at cell edges. If point A is visible from point B interior to cell V, then point A must also be visible from some point C

able to pass from A to B without intersecting any opaque walls. Since the task of determining point-to-point visibility by an exhaustive search is impractical, a simplification must be found. The first observation in this direction is that if a given point in space is visible from within a cell, that point must be visible from at least one of the sides of the cell and the diagonal (see Figure 6).

For any point, we incur the overhead of rendering some walls which are not visible by virtue of the discrete nature of the visible lists. At the cost of increased overestimation in the visible lists, a simplified approach may be taken to the determination of visibility. Instead of determining, for each non-solid cell, visibility of every line segment in the data set, we may calculate the visibility of every wall cell. Since wall cells contain all data, if a data element is visible, an edge of the wall cell which contains it must be visible (see Figure 6). Further, instead of determining obstruction precisely by using the data in the data set, we may simply consider the solid cells to be opaque, and all other cells to be completely transparent. In this new space, we calculate the visibility of each wall cell on every other cell. If any one of its sides is visible, we consider any data elements which it contains to be visible.

We now turn to describe in the details of the core operation of our algorithm, namely, the cell-to-cell visibility determination.

4. Cell-to-Cell Visibility Determination

The *cell-to-cell visibility* problem is the problem of determining the existence or non-existence of a line segment intersecting both cells without intersecting any opaque (solid) cells in between. At first glance it may appear that an approach using a Bresenham line [7] to step from cell observer cell to target cell could check all possible intervening cells. However this effectively only determines center to center visibility for the two cells in question. A correct approach must check the possibility of *any* line between two cells.

In some cases a single solid cell C can block sight between cells A and B, for example, when A is at (x,y), B at $(x+i, y+i)$ and C in $(x+j, y+j)$, $j<i$. In the general case, however, no algorithm which stops at a single solid cell line can be sufficient to determine cell-to-cell visibility. Any such algorithm fails

because the fact that one discrete line has been blocked does not imply that the line of sight would be blocked from any other point. It is possible that an infinitely small movement to one side would produce a different line, along which the target cell would be visible (see Figure 8a). The question is then how to determine, with a finite set of lines, that every possible line of sight from the observer's cell to the target cell is blocked. The approach taken to solving this problem is to use a somewhat modified concept of line of sight, namely the *sight corridor*.

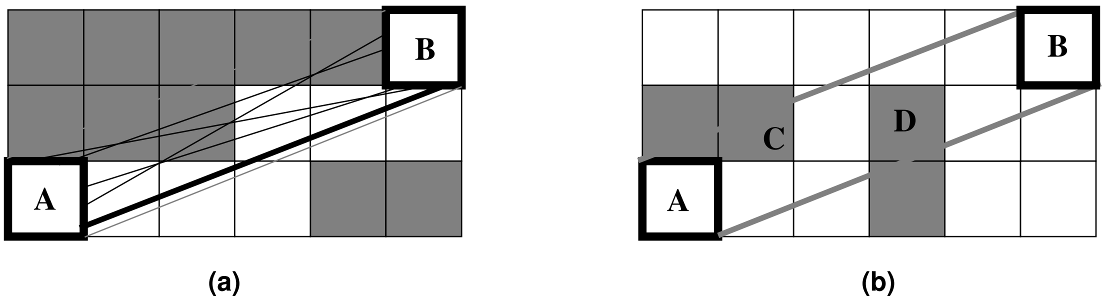

FIGURE 7. The corridor between A and B is shown with the dashed lines. (a) The fact the a very large number of rays are blocked still does not preclude visibility (bottom thick line).(b) Although the corridor is blocked, that is, A and B are mutually invisible, since blockage is caused by disconnected components C and D, A and B will be suspected as visible

We define the *corridor* of cell A and cell B as the convex hull defined by the eight vertices of cells A and B (see Figure 8). Any line segment connecting any point in A to any point in B must then lie in the corridor between them. Therefore we can test if this *sight corridor* is blocked to determine if the cells are mutually occluded.

We observe that it is valid to consider one cell to be invisible from another if the corridor between them is entirely blocked by one or more *connected* opaque cells. Using this definition of visibility the potentially visible set (PVS) is again overestimated. It is possible that if A is invisible from B due to multiple *disconnected* obstructions of the corridor, it will nevertheless be declared to be visible due to the lack of a connected blockage (see Figure 8b).

Visibility may now be determined by using a Bresenham line algorithm [7] as a base. We observe that as the Bresenham line steps from the observer's cell to the target cell it always steps in cells for which some portion is in the corridor. Implementation of this algorithm then becomes a simple matter of checking to see, at each solid cell encountered, if this solid cell completely blocks the corridor, or if there are other connected solid cells such that this connected group of cells completely blocks the corridor. If either of these conditions is met, then the corridor must be blocked, and the visibility check will fail at this point. If these conditions are not met, we continue stepping along the Bresenham line.

4.1 Approximating Connected Blockage by Column Blockage

We said that a corridor is considered blocked only by a connected set of solid cells (although in reality it might be blocked by a disconnected set). We further simplify our computations by looking for a simpler set of solid cells to block the corridor. As the Bresenham line steps from one cell to another, it has a *major direction* of travel; that along which it moves via the counter variable, and a *minor* *direction* of travel, which is perpendicular to the major direction of travel (see Figure 8). We define the *column* containing a cell as the collection of cells which can be reached from the cell by steps in the minor direction of travel. We define as the *row* containing a cell the collection of cells which can be reached from the cell by steps in the major direction of travel. It is observed that for each step in the major direction of travel, there are at most three cells which could possibly contain any portion of the corridor which lies in the same column as the current cell: the current cell, the cell which is edge connected to this cell in the minor direction of travel, and the cell which is edge-connected to this cell in the direction opposite the minor direction of travel. We will call these cells the *possible corridor cells*.

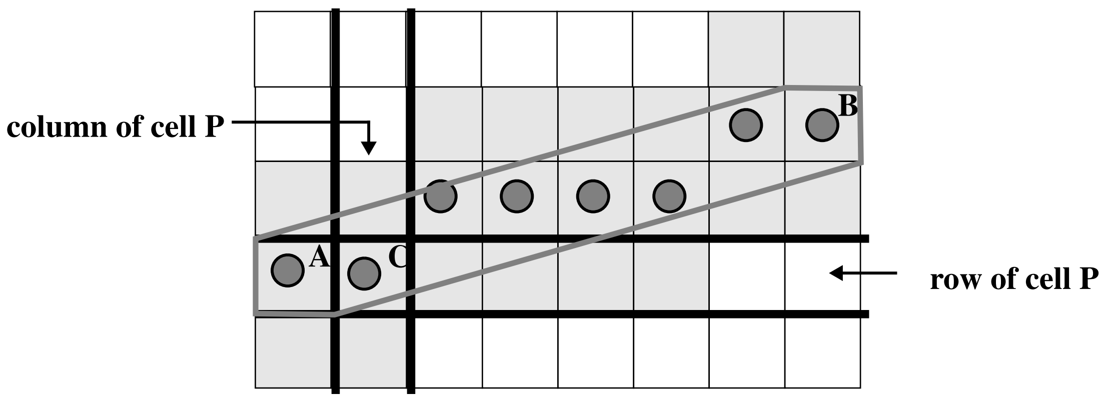

FIGURE 8. Along the line from A to B, the major direction of travel is from left to right, and the minor direction is from bottom to top. Therefore, the possible corridor cells (filled squares) for cell C are those positioned directly above and below it.

Given the possible corridor cells, the algorithm determines whether there is any way of traversing an 8-connected series of cells from the observer's cell to the target cell which remains at each step inside the set of all possible corridor cells (see Figure 8). If such a series exists, then we claim that the target is potentially visible from the observer's cell. As wall cells are encountered during the visibility checks, they are added to a linked list of data cells which are visible from this cell.

5. Results

Experimental results were gathered for the 2D implementation using a dataset consisting of 19,993 line segments (Figure 9). The data gathered is shown in Table 1 while memory overhead and algorithm's efficiency are contrasted in the graph in Figure 10. This case is a relatively simple void depicting a capillary loop in a section of a human tongue [3]. In a more general case, it would be expected that there would be many more passages. As the resolution of the space subdivision increases, so does the number of cells in the grid (columns II, III) and processing time (column VI), while the number of data elements per data cell decreases (column IV). As expected from the algorithm description (Section 3) preprocessing time is a linear function of the number of void and wall cells (column II). Processing time per cell (column VII) shows that we can still perform visibility precomputation on-the-fly in a rate of approximately 10 per second.

TABLE 1. Data for a the dataset in Figure 9 showing the dependency of various attributes and the space subdivision resolution.

I Grid Size: NxN	II # void+wall cells	III # wall cells	IV # walls per wall-cell	V Length of visible list	VI Preprocess time (sec.)	VII Time per cell (sec.)
66	578	578	34.6	121	36	.06
83	807	800	25.0	133	51	.06
100	1078	1054	19.0	150	113	.1
142	1856	1668	12.0	190	167	.09
200	3220	2518	8.0	222	365	.11
333	7766	4648	4.3	323	1688	.21

The most encouraging numbers are the average number of data elements that are actually rendered at walkthrough time. This quantity, which is approximately the product of columns IV and V, is shown by the dashed line in Figure 10. It is clear that we rapidly converge to an exact solution and that at moderate space subdivision (140^2)and modest memory expense (2MB) the approximated PVS is very close to the true solution. Even in this simple case and in a low resolution subdivision (66^2) our preprocessing eliminates 79% of the data. It is also clear from this graph that, for the case in Figure 9,

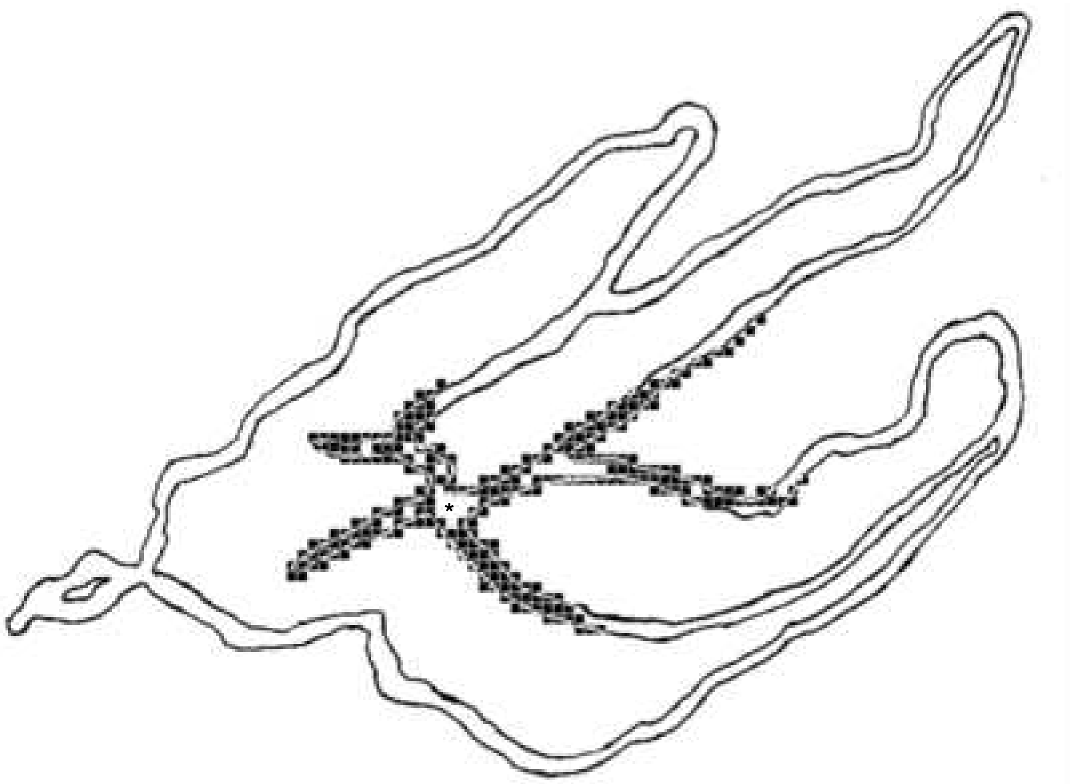

FIGURE 9. A complex void of a capillary loop from a section of a human tongue, with 19,993 walls. The observer (denoted by *) can see only those objects that are in the dark cells in this 125^2 subdivision.

the marginal improvement in the list length (from 11.4% to 6.9%) is not worth the memory expense (6MB), above approximately 150^2 subdivision.

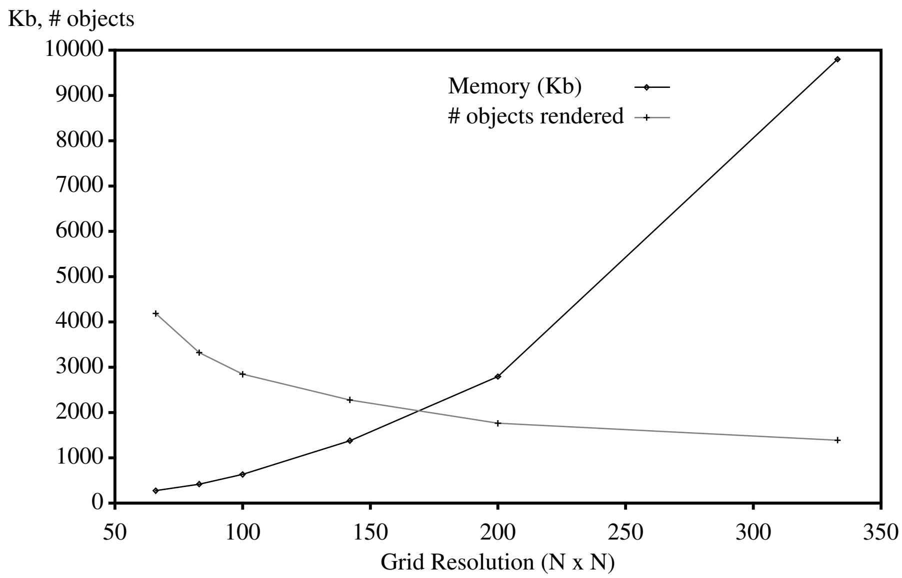

FIGURE 10. Memory requirements and length of the list in each cell as a function of grid resolution for the data set in Figure 9. Increasing the resolution to more than 150^2 is clearly unjustified.

6. Enhancements and Extensions

Storage Reduction. The current implementation requires the storage of a full visible list (VCL) for every cell in which the observer could be positioned. One way to reduce memory usage of the program would be to eliminate the duplication of data between the cells. This can be accomplished by storing, for each cell, a *difference list* from its neighbors, instead of storing a complete visible list. Preliminary results of the implementation of this idea [19] show a reduction of 75% in list length and consequently in rendering time.

Another way to reduce storage consumption is to join cells into larger cells in areas where the difference lists are short. That is, after computing the visible cell list we will run an algorithm that will join adjacent cells (in an octree fashion, for example) if the difference between their lists is not more than ε. Giving the user the control over this similarity parameter ε will provide him with a simple mechanism to control the memory-performance trade-off.

Speeding the Walkthrough Stage. We enhanced the functionality and speed of the final walkthrough algorithm at little cost to the preprocessing by simply modifying the storage algorithm for the visible lists (VCL). The visible list is divided into, for example, eight lists, each containing the cell in one octant of the space. At walkthrough time we clip the space octants with the view cone. Only those octants that are (or partially are) inside the view cone are submitted to the renderer.

So that depth culling will be easily and efficiently performed at run time, the visible list is sorted in a front-to-back order (i.e., increasing distance from the cell). Data can then simply be rendered out of the visible lists in a front-to-back order until a cell is found which is beyond the maximum depth limit. The rest of the list is simply ignored. We note also that being able to depth sort in front-to-back order opens the door for additional rendering optimizations [12].

Extension to Three Dimensions. Perhaps the most useful enhancement to the algorithm would be the extension into three dimensions. This would allow our current implementation to move from being a test bed for modifications and enhancements of the algorithm to being a useful piece of code which could be included in real world rendering applications. Fortunately, the extension of the algorithm to three dimensions appears to be relatively simple.

Instead of discretizing data into a planar grid of cells as we did in the two dimensional case, we must now discretize data into a three dimensional volume of cubes, or *voxels*. The data itself now must consist of some form of primitives with non-zero surface area, which again define the interface between the solid medium and the void. The data need not be polygonal, the only requirement being that it can be discretized into voxel space in the form of 6-connected surfaces with no holes [5]. The voxels making up these 6-connected surfaces are then equivalent to the two-dimensional wall cells and will be called the *wall voxels*.

After categorizing all wall voxels, the solid medium may be categorized into the voxel structure in the same manner as in the two dimensional case. We define a voxel-to-voxel visibility *prism* in the same manner as the cell-to-cell corridor, by connecting the corresponding corners of the voxels to produce a set of mutually parallel lines. It is obvious that if the prism from voxel A to voxel B is blocked, then B is not visible from A. Finally, the column blockage approximation is extended into *slice blockage* in 3D, that is, the prism will be considered blocked only if all voxels in the 2D slice along the minor direction of the 3D Bresenham line [5] are solid.

7. Conclusion

We have presented a new method for visibility precomputation in a user enclosing environment. Our method is based on regular and uniform space subdivision, classification of cells into solid, data, and void cells, and simplified cell-to-cell visibility determination. The method we proposed estimates the exact solution by having some small amount of data which is not visible from anywhere interior to a region included in the visible list for that region. Our method approximates the exact solution due to its discrete nature and the simplified mechanism for cell-to-cell visibility determination it employs. Nevertheless, despite these approximations, our algorithm demonstrates rapid convergence to the exact solution. At a moderate subdivision resolution and small memory overhead the algorithm closely approximates the exact solution.

Our approach suffers from few disadvantages, such as the need to discretize and store the scene in a regular space subdivision. It also assumes that there is a notion of “solid” while in some applications there will be no such notion or the solid will be too thin to be captured with a reasonable subdivision resolution. On the other hand, our approach has several major advantages; unlike some existing methods, ours is simple to understand and easy to implement. It can support the rendition of any type of objects and is not limited to polyhedral environments. Moreover, it can be easily extended to three dimensions (Section 6). It is fast enough to be applied on-the-fly as well as be used in a preprocessing phase. Finally, it provides a user controllable trade-off between memory consumption and rendering speed.

Acknowledgments

This project was supported by the National Science Foundation under grant CCR-9211288. We thank Kim Ciula, Yair Kurzion, Raghu Machiraju, and Ed Swan for proofreading the manuscript and making useful suggestions.

8. Bibliography

1. Airey J.M., *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculation*. Ph.D thesis, UNC Chapel Hill, 1990. Appeared also as Technical Report TR90-27, Department of Computer Science, UNC Chapel Hill, 1990.
2. Airey J.M., Rohlf J.H., Brooks F.P., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics, Computer Graphics,* 24, 2, pp. 41-50.
3. Arms K. and Camp P. "*Biology*", Sounders College Publishing, Third Edition, 1987, pp. 685.
4. Cline H.E, Lorensen W.E, Ludke S., Crawford C.R., and Teeter B.C., "Two Algorithms for the Three-Dimensional Construction of Tomograms", *Medical Physics,* 15, 3, (May/June 1988), pp. 320-327.
5. Cohen D. and Kaufman A., "Scan Conversion Algorithms for Linear and Quadratic Objects", in *Volume Visualization*, A. Kaufman (ed.), IEEE CS Press, 1991, pp. 280-301.
6. DeHaemer M.J. Jr. and Zyda M.J. "Simplification of Objects Rendered by Polygonal Approximations", *Computer & Graphics,* 15, 2, 175-184, 1991.
7. Foley J.D., van Dam, A., Feiner S.K., Hughes J.F., *"Computer Graphics Principles and Practice"*, Addison Wesley, Second Edition.1990.
8. Fuchs H., Kedem Z.M, and Naylor B.F. "On Visible Surface Generation by A Priori Tree Structure", *Proceedings of SIGGRAPH'80*, pp. 124-133.
9. Fujimoto A., Tanaka T., and Iwata K., "ARTS: Accelerated Ray Tracing System", *IEEE Computer Graphics and Applications*, 6, 4, April 1986, pp. 16-26.
10. Funkhousser T.A., Sequin C.H., "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments", *Proceedings of SIGGRAPH'93*. In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 247-254.
11. Garlick B., Baum D., and Winget J.M., "Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations", *SIGGRAPH'90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation),* 1990.

12. Greene N., Kass M., Miller G. “Hierarchical Z-Buffer Visibility”, *Proceedings of SIGGARPH’93*. In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 231-238.

13. Ray, W. “Preprocessing for Visibility Determination for the Efficient Rendering of Complex Passage Systems“, Masters Thesis, Department of Computer and Information Science, The Ohio State University, April 1993.

14. Samet H., “*The Design and Analysis of Spatial Data Structures*”, Addison-Wesley, 1990.

15. Schroder W.J., Zarge J.A., and Lorensen W.E., “Decimation of Triangle Meshes”, *Computer Graphics*, 26, 2, July 1992, pp. 65-70.

16. Teller S.J. and Sequin C. H. “Visibility Preprocessing for Interactive Walkthrough”, *Computer Graphics*, 25, 4, 62-69, 1991.

17. Teller S.J., “*Visibility Computations in Densely Occluded Polyhedral Environments*”, Ph.D thesis, University of California, Berkeley, 1992. Appeared also as Report No. UBC/CSD-92-708, Department of Computer Science, University of California, Berkeley, 1992.

18. Woodward P.R., “Interactive Scientific Visualization of Fluid Flow”, *IEEE Computer*, 26, 10, October 1993, pp. 13-25.

19. Yagel R. and Chandrasekhar R. “Rendering Passageways”, *Israel Computer Graphics Forum*, Tel Aviv University, Israel, September 1992.