Portholes and Planes: Faster Dynamic Evaluation of Potentially Visible Sets

Timothy Roden, Ian Parberry
Department of Computer Science & Engineering,
University of North Texas, P.O. Box 311366, Denton, Texas 76230
http://www.cs.unt.edu

***Abstract* – We describe a simple and efficient dynamic occlusion culling algorithm for computing potentially visible sets (PVS) in densely occluded virtual environments. Our method is an optimization of a widely used technique in which a 3D environment is divided into cells and portals. Our algorithm computes the PVS in approximately half the time of previous portal methods at the expense of producing a slightly relaxed PVS. In addition, our algorithm enables fast culling of objects within cells using inexpensive object space methods by using a lookup table to compute the diminished object space view frustum. The algorithm takes advantage of temporal coherence, is easy to implement, and is particularly well suited for applications that need to compute a PVS for use in non-rendering tasks such as AI.**

I. INTRODUCTION

Many applications of interactive 3D computer graphics render scenes in which there is a high degree of occlusion. Examples include architectural environments and game mazes. It is desirable to determine, as efficiently as possible, only those objects that are visible or are potentially visible to the viewer. Occlusion culling techniques have been developed to cull from consideration large sections of an environment. What is left is assumed to be at least potentially visible. This potentially visible set (PVS) can then be used in a variety of ways. Typically, the PVS is used in rendering with exact visibility determined using standard rendering procedures such as view volume clipping and z-buffering. Besides rendering, a PVS can be used in a variety of ways. For example, real-time agents navigating a 3D virtual environment may want to compute what objects are visible for AI-related processing tasks that do not involve rendering.

In order to facilitate efficient occlusion culling for architectural and similar environments, we borrow from prior work the idea of cells and portals [1, 5, 7, 11]. A cell is a polyhedral volume of 3D space while a portal is simply a transparent partition that divides two adjacent cells.

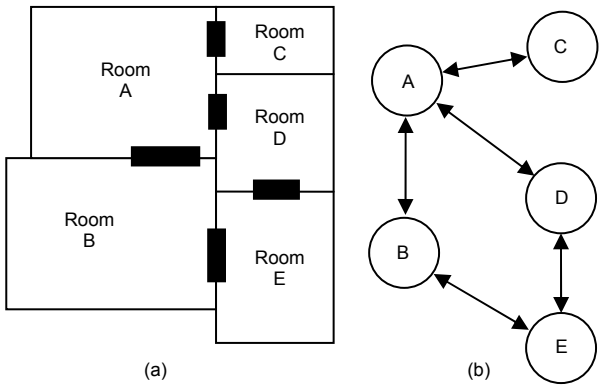

Fig. 1. (a) Top view of a building divided into cells (rooms) and portals (doors). (b) Adjacency graph representation with room nodes and portal edges.

Typically, cells are visible by the viewer only if the viewer is located inside a cell or has a line of sight, through a portal, to another cell. As an example application of cells and portals, Figure 1(a) shows the geometry for a building divided into one cell for each room or hallway and one portal for each door or window.

Partitioning an environment into cells and portals is typically done as a manual preprocess when creating a 3D model of an environment. For example, a modeler, using a 3D modeling program, may explicitly or implicitly 'tag' geometry as belonging to a particular cell and also create special 'portal' polygons connecting cells. These tagged models are typically imported into a rendering engine at execution time and the data structures are generated to enable the division of geometry into cells and portals. These data structures essentially create a directed adjacency graph with cells as the nodes of the graph and portals as the edges, as in Figure 1(b).

A related problem in portal schemes is how to handle dynamic objects within cells. Inside a potentially visible cell a number of dynamic objects may be present such as non-static (i.e. movable) furniture or people. The geometry for these objects would likely be kept separate from the geometry of the cell itself. After determining that a cell is potentially visible, the next step is to decide if any in-cell objects are visible. A naïve approach would be to assume

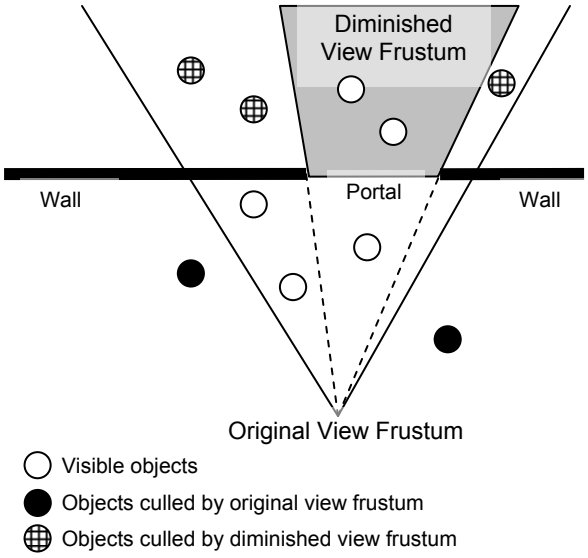

Fig. 2. Objects inside cells connected via portals should be culled with the view frustum diminished by the portal.

all objects in the cell are potentially visible. A more efficient technique involves culling in-cell objects against the view frustum that is diminished by the portal admitting a view of the cell, as shown in Figure 2.

The most efficient algorithm for dynamic PVS evaluation appears to be [7] but we note that screen space projection for culling of bounding volumes is not generally as efficient as compared to the corresponding object space methods. In particular, several popular bounding volumes used to enclose objects in real-time applications include bounding spheres, axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs). Fast algorithms have been developed to cull these volumes to an object space view frustum consisting of six planes where each plane is stored as four values corresponding to the four coefficients of the equation of a plane, (Equation 1). An efficient sphere-frustum test is given in [10], while methods for testing AABBs and OBBs to frustum planes are given in [2, 4]. An extensive survey of algorithms for these and other bounding volumes is given in [8]. We therefore decided to find ways to modify the [7] algorithm to gain greater efficiency with a primary goal of culling as much geometry as possible in object space as opposed to screen space.

Similar to [7], our algorithm does not compute visibility as a preprocess but rather determines the PVS on-demand at execution time. We are not concerned with computing exact visibility but only the set of cells that are potentially visible. Our PVS is a conservative estimate of the exact visibility in that it overestimates the set of visible cells. It is a slightly larger set than [7] but our algorithm has the advantage of running in approximately half the time. For in-cell object culling, our algorithm works in object space as compared to the less efficient method of culling in screen space proposed by [7]. The algorithm has been implemented on a Pentium 4 PC and tested with randomly generated environments consisting of hundreds of cells. The tests showed a significant speedup in calculating the PVS with only a fractional increase in the average size of the PVS.

II. Related work

The idea of dividing an environment into convex polyhedral cells and convex polygonal portals to implement occlusion culling was proposed by Jones [5]. His algorithm renders a scene by first drawing the geometry of the cell in which the viewer is located. Each portal in the cell is then projected into screen space and clipped against the screen space view plane. The resulting clipped screen space polygon acts as a clipping mask. The cell connected via the portal is then recursively rendered using the clipping mask to clip any geometry before it is rendered. If adjacent cells contain any additional portals, the new portals are clipped to the clipping mask and their geometry is clipped to the updated mask and rendered. The effect is a depth-first traversal and rendering of cells visible from the starting cell with only the visible geometry being drawn.

Later work has built on Jones' algorithm by using the cells and portals approach but eliminating the need to compute exact visibility, instead relying on a z-buffer. The idea is to compute, as conservatively as possible, a PVS of cells that may be visible from the viewer. Several approaches have described computationally expensive methods to compute a conservative PVS as a preprocess with applications, including, but not limited to, architecture [1, 11].

Luebke and Georges advanced the cells and portals idea with a dynamic algorithm that requires only a small amount of preprocessing [7]. Their algorithm requires an adjacency graph for the cells and portals but does not pre-compute any PVS information, instead computing it at runtime on a frame by frame basis. Like Jones' algorithm, they project each polygonal portal into screen space after which they compute the axial 2D bounding box of the resulting points. They refer to this as the cull box. The cull box is a conservative bound of the portal since any objects whose screen space projection falls outside the box are guaranteed to be hidden from view. Also, they follow the more contemporary practice of leaving exact visibility determination to a renderer as compared to Jones' manual clipping of geometry. Their algorithm recursively considers cells that are viewable through any sequence of portals, maintaining an aggregate cull box along the way. In the same manner in which portals are handled, their algorithm projects the bounding volumes of in-cell objects into screen space where they are compared against the current cull box to determine visibility.

It has been suggested that portal culling and in-cell culling be done entirely in object space by testing against actual frustum planes as opposed to using a screen space projection method, although an efficient method for computing new frustum planes was not described [3]. In his explanation of portal systems, Lengyel provides a formula for computing a diminished frustum plane given two vertices in world space. Despite its simplicity the calculation requires computing the cross product of the two vertices and the magnitude, requiring a square root to be taken. In addition, this needs to be done for each diminished plane in the frustum [6]. Compared to the technique proposed by [7] which maintains an implicit representation of the view frustum as a screen space rectangle and computes the diminished view frustum by comparing two rectangles, Lengyel's calculation of an explicit frustum is costly.

A thorough survey of visibility algorithms for architectural applications, including descriptions of portal rendering methods, is given in [12].

III. Required Preprocessing

Our method requires a small amount of preprocessing. The environment to be rendered must have its geometry broken up into cells and portals. We use the adjacency graph representation shown in Figure 2(b). We enclose the vertices of each portal inside a sphere and store portals as spheres instead of maintaining portals as a set of vertices, as shown in Figure 3(a). We are currently using a method to compute a near-optimal bounding sphere from a set of vertices, [9]. Using spheres overestimates the actual geometry of the portal but it allows us to compute the PVS much faster at the expense of generating only a slightly larger PVS. For each portal we also store a vector normal to the plane of the original portal. In some environments the normal vector can be useful. For example, Figure 3(b) shows an architectural layout in which the view frustum can be behind a portal. In this case we do not want to add the cell connected via the portal to the PVS.

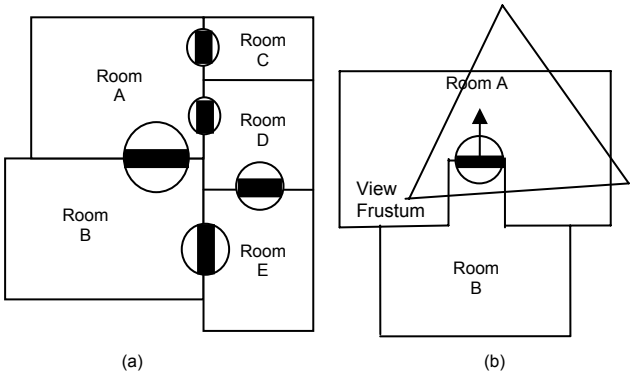

Fig. 3. (a) Portals enclosed in bounding spheres. (b) Normal vector of portal used to determine that view frustum is behind the portal.

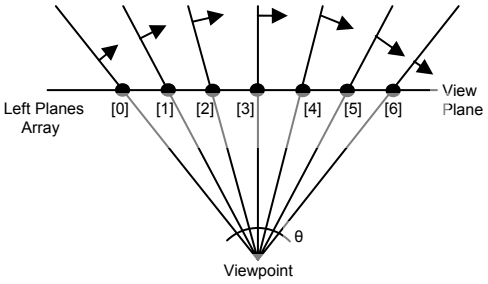

Fig. 4. Array of left planes for the view frustum, pre-computed at even intervals along the view plane, with normal directions shown. θ = maximum horizontal field of view to be used during execution.

We also pre-compute an array of all possible planes for the left, right, top and bottom planes of the view frustum. These are used to quickly select the new planes of a diminished view frustum. We use the maximum field of view that will be used at execution time and compute a set of planes at even intervals on the view plane, as shown in Figure 4. The size of the interval depends on the level of accuracy desired. We use a smaller interval to compute a larger set of planes in order to enable the computation of a tighter frustum at the expense of a larger array. A larger interval can be used to reduce the size of the array at the expense of computing looser diminished frustums. Each plane is stored as a vector – the normal of the plane. We do not store the distance of the plane from the origin. This is the D value in the equation of a plane and is always zero for the left, right, top and bottom frustum planes, where the equation of the plane is given as:

$$Ax + By + Cz + D = 0 \qquad (1)$$

If memory space is a consideration then only the left and top planes need be pre-computed since their opposites, the right and bottom planes, respectively, can be easily computed by inverting a portion of the normal vector. For example, in a left-handed coordinate system, a right plane can be extracted from a left plane by inverting the x and z components of the plane normal.

IV. PVS Calculation

We keep track of the PVS using a list. To calculate the PVS we begin by placing the cell containing the viewpoint in the list. We refer to this as the starting cell. We then perform a depth-first traversal of the adjacency graph, beginning with the starting cell. For each portal we project the center and radius of the sphere onto the view plane of the current view frustum and compute the axial 2D bounding box of the projected sphere in view space coordinates. This becomes the 2D cull box, which is a conservative bound for the portal, and is used in manner similar to the cull box of [7]. Each successive portal is projected onto the view plane of the view frustum for the

current cell and tested against the current cull box. If an intersection is detected the adjacent cell is added to the PVS, the intersection becomes the new cull box and the cell connected by the portal is recursively checked for visibility. Since a cell may be visible through multiple portals, we flag each cell added to the PVS to keep it from being added more than once.

In addition to diminishing the size of the cull box for each cell added to the PVS, we also compute a new diminished object space view frustum for the cell. The view frustum consists of a rectangular view plane and a set of six frustum planes. The near and far planes of the diminished view frustum are the same as the original frustum. In order to calculate the other four planes (left, right, top, bottom), we keep track of which edges of the view plane are intersected by the projection of the portal. The intersected edges represent planes in the original view frustum that can be simply copied to the new diminished view frustum. For any of the other four planes the portal is effectively inside the frustum so these planes need to be adjusted to enclose the portal as tightly as possible. This is done by scaling the portal's projection on the view plane to a discrete value used as an index into the pre-computed array of frustum planes, effectively reducing the calculation of the diminished object space view frustum to a table lookup, Figure 5.

As we recursively calculate the PVS, potentially visible cells are added to the PVS list, as shown in Figure 6. Each entry in the list contains a pointer to a data structure describing the cell, including both its static geometry and any dynamic objects contained in the cell. In-cell objects contain, at least, the geometry of the object and a bounding volume. In addition, each entry in the list also contains a diminished view frustum consisting of a 2D view plane and 6 frustum planes (near, far, left, right, top, bottom). Each frustum plane is comprised of a plane normal and distance to the viewpoint origin. The distance will be zero for all planes except the near and far planes. We also store in each entry a distance from the starting cell computed as the number of cells traversed.

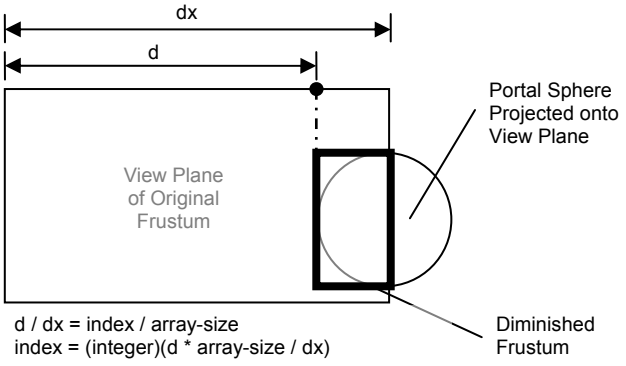

Fig. 5. Calculating the index into the array of left frustum planes.

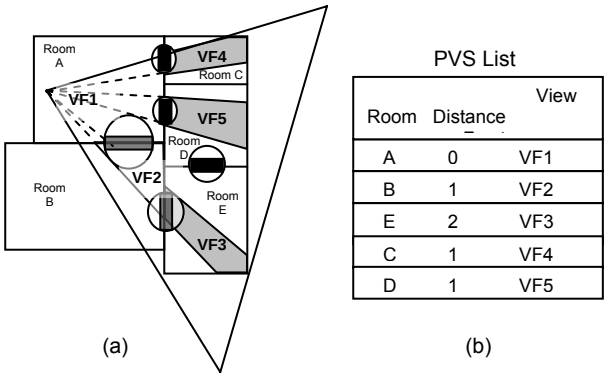

Fig. 6. (a) Top view of building showing original view frustum diminished by portals. (b) The resulting PVS, stored as a list.

Once the PVS list is created we examine each cell in the list. We clip each in-cell object's bounding volume against the cell's diminished view frustum to determine if the in-cell object is potentially visible. Since these tests are carried out in object space we take advantage of fast methods to perform the clipping [2, 4, 10]. We can also take advantage of temporal coherence by not re-computing the PVS in the event the view position has not changed.

V. Implementation and Results

We have implemented our approach as a C++ program using a Pentium 4 Extreme Edition 3.2GHz PC. We tested the algorithm using a program that randomly generates 3D mazes consisting of rooms, corridors and stairs. The static geometry of the maze is built on a grid with portals generated automatically where geometry crosses grid lines. A typical maze consists of several hundred cells and portals. We compared several versions of our algorithm against the algorithm of [7] by running all the algorithms in parallel as we navigated the maze.

We wanted a machine-independent metric for measuring the amount of work done. We measured only significant arithmetic operations and comparisons. After running timing tests on several PC computers we came up with the relative costs for operations shown in Table 1, all based on multiplication as the basic operation.

TABLE 1
Cost of Operations

Operation	Cost
Addition	1
Subtraction	1
Multiply	1
Divide	20
Comparison	4
Square Root	230 (using C library sqrt() function)

TABLE 2
EXAMPLE EXECUTION

Algorithm	Cost	|Portals|	|PVS|	Add/Sub	Mul	Div	Comp
Luebeke	1540	6	2.2	282	196	13	196
Porthole1	629	6.6	2.6	114	60	3	95
Porthole2	798	6.6	2.6	119	70	8	106

Luebeke = algorithm from [Luebke98]
Porthole1 = our algorithm without object space view frustum calculation
Porthole2 = our algorithm with object space view frustum calculation
Cost = operation cost per frame
|Portals| = average number of portals examined per frame
|PVS| = average size of PVS, in cells per frame
Add/Sub = average number of (additions + subtractions)
Mul = average number of multiplies per frame
Div = average number of divisions per frame
Comp = average number of comparisons per frame

We did not measure any preprocessing or culling of in-cell objects, instead focusing on the cost of calculating the PVS. We calculated a new PVS every frame. The tests showed our algorithm for PVS calculation, averaged less than half the cost of the [7] algorithm at the expense of a slightly larger PVS, which, in all of our tests, amounted to less than one cell. We tested two different versions of our algorithm – one version that calculated the diminished object space view frustum for each PVS cell and another version that did not. View frustum calculation added approximately 25% to the per-frame cost of calculating the PVS using our method. A typical test produced the results shown in Table 2.

We also tested the efficiency of our algorithm in culling in-cell objects using the diminished object space view frustum as compared to a screen space projection method. Object space culling was more efficient. We do not present those results here since we believe it is widely agreed that object space culling is superior [2, 4, 8, 10].

VI. CONCLUSION

Obtaining good results using a dynamic cells and portals approach to visibility determination can be highly dependent on the geometry of the environment. Our results show our approach is a good choice. Using spheres for portals gives a significant speedup when calculating the PVS. The size of the PVS was only marginally larger in our tests. This was expected since spheres can overestimate the size of portals more than a bounding rectangle.

We were concerned about degenerate cases in which the view frustum, positioned at a right angle to a portal, grazed the portal's bounding sphere without intersecting the actual geometry of the portal. In these cases the algorithm will assume the portal has been intersected and add the connected cell to the PVS. Despite this anomaly, the problem appears to have little chance for further negative effects since successive portals are unlikely to be in the path of the diminished frustum. This is because the diminished frustum will likely be projecting at a right angle to the adjacent cell and therefore unlikely to intersect any subsequent portals of the adjacent cell.

There may be implementations where using portal spheres is not desirable. For example, when computing a PVS for use in rendering, the cost savings of using portal spheres may be negated by the higher cost of rendering a larger PVS. In this case it may be desirable to use flat polygonal portals. Calculating the diminished view frustum can still be done using our method since the calculation is based on the 2D axial aligned rectangular projection of the portal on the view plane. This compromise would allow for a tighter PVS while enabling fast culling of in-cell objects.

REFERENCES

[1] John Airey. Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations. Ph.D. thesis, UNC-CH CS Department TR #90-027 (July 1990).

[2] Ulf Assarsson and Tomas Moller. "Optimized View Frustum Culling Algorithms for Bounding Boxes", *Journal of Graphics Tools*, vol. 5, no. 1, pp. 9-22, 2000.

[3] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, Michael Shantz, "Designing a PC Game Engine", *IEEE Computer Graphics and Applications*, vol. 18, no. 1, pp. 46-53, 1998.

[4] Ned Greene. "Detecting Intersection of a Rectangular Solid and a Convex Polyhedron", *Graphics Gems IV, Heckbert*, pp. 74-82, 1994.

[5] C. B. Jones. A New Approach to the 'Hidden Line' Problem. *The Computer Journal*, vol. 14 no. 3 (August 1971), pp 232-237.

[6] Eric Lengyel. *Mathematics for 3D Game Programming & Computer Graphics*, 2002.

[7] David Luebke and Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potential Visible Sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pp. 105-106. ACM SIGGRAPH, April 1995.

[8] Tomas Moller and Eric Haines. *Real-Time Rendering, Second Edition*, 2002.

[9] Jack Ritter. "An Efficient Bounding Sphere", *Graphics Gems, Glassner*, pp. 301-303.

[10] Tim Round, "Object Occlusion Culling", *Game Programming Gems, DeLoura*, pp. 421-431, 2000.

[11] Seth Teller. Visibility Computation in Densely Occluded Polyhedral Environments. Ph.D. thesis, UC Berkeley CS Department, TR #92/708 (1992).

[12] Daniel Cohen-Or, Yiorgos Chrysanthou, Claudio Silva, Fredo Durand, "A Survey of Visibility for Walkthrough Applications", *IEEE Computer Graphics and Applications*, vol. 9, no. 3, pp. 412-431, 2003.