Occlusion Culling Using Minimum Occluder Set and Opacity Map

Poon Chun Ho Wenping Wang
Department of Computer Science and Information Systems
University of Hong Kong
{ chpoon | wenping } @csis.hku.hk

Abstract

The aim of occlusion culling is to cull away a significant amount of invisible primitives at different viewpoints. We present two algorithms to improve occlusion culling for a highly occluded virtual environment. The first algorithm is used in pre-processing stage. It considers the combined gain and cost of occluders to select an optimal set of occluders, called minimum occluder set, for each occludee. The second algorithm uses the improved opacity map and sparse depth map for efficient run-time overlap tests and depth tests, respectively. Without using pixel-wise comparison, this algorithm uses only three integer operations to perform an overlap test, and carry out a depth comparison sparsely. Both algorithms have been implemented and applied to test a model composed of about three hundred thousand polygons. Significant speedup in walkthroughs of the test model due to our algorithms has been observed.

1 Introduction

In many applications the demand for interactive display of complex geometric environments composed of millions of geometric primitives always outpaces the advance of the high-end graphics technology. These include interactive visualisation of architectural models and walkthrough of outdoor scenes. The models found in these applications normally have high depth complexity. An efficient algorithm for identification of hidden primitives is critical to interactive rendering, while pixel-level culling, such as hardware z-buffering, is no longer enough to determine visibility in real-time. An occlusion culling algorithm makes use of occlusion relation among the primitives of the model, and culls away a significant amount of invisible primitives at different viewpoints quickly, in order to achieve an interactive frame rate.

There are two stages in an occlusion culling algorithm: selection of occluders, which is usually off-line for a static environment, and actual invisible surface culling with occluders, which is a run-time operation for real time rendering.

In this paper, we present two new algorithms:

1. a novel method to select occluders with multiple criteria at pre-processing stage, using the idea of the *minimum occluder set* (MOS). The MOS of an occludee is the minimal set of primitives that occludes the occludee.
2. an efficient occlusion culling algorithm using the *opacity map* (OM) and *sparse depth map* (SDM), which are applied to the spatial hierarchy of the whole model at each frame at run-time.

Though we perform occluder selection using the minimum occluder set, the culling part makes no assumption about the model and occluders, and can therefore be carried out along with occluders selected with any other criteria.

We shall briefly discuss related work in the section 2. In the section 3, we present an overview of our occlusion culling approach. Details of the minimum occluder set algorithm and the occlusion culling algorithm using the opacity map and sparse depth map will be presented in sections 4 and 5, respectively. The result and analysis are given in section 6, and the paper concludes in section 7.

2 Related Work

Hidden surface removal is a fundamental problem of computer graphics. The conventional z-buffer algorithm is implemented in hardware or software [2, 3] that yields exact visibility information by pixel-wise comparison of depth values of every primitive.

The binary space partitioning (BSP) tree algorithm [6, 12], which refines the work in [13], determines visible primitives in a static environment from an arbitrary

viewpoint. After building the BSP tree, one can have a linear query response of visibility sorting for the whole set of primitives.

Based on probabilistic geometry, an efficient and randomized algorithm for hidden surface removal is presented in [11]. Further research in computational geometry on randomized algorithms for maintaining a BSP tree for a dynamic model has been conducted [1, 16], which, however, does not lead to practical results.

The potentially visible set (PVS) [10, 15] is designed for indoor architectural walkthrough systems. It divides the entire model into cells, and computes cell-to-cell visibility at the pre-processing stage. Combined with a view cone, one can obtain a tight bound for the visible primitives (eye-to-cell visibility) at run-time.

For densely occluded scenes, hierarchical z-buffer visibility [7] is exploited to speedup the conventional depth value comparison during rasterization process. With z-pyramid, this method allows quick termination of depth comparison for the nodes of octree hierarchy far away from the viewpoint. It performs efficiently when is implemented in hardware. Hierarchical polygon tiling [8] combines z-pyramid to further reduce the rasterization time with triage coverage masks. It traverses the convex polygons in front-to-back order, and culls off polygons that are covered in image hierarchy.

The occlusion culling algorithm in [4, 5] computes the separating and supporting planes for each pair of occluders and the nodes of the model hierarchy. If the viewpoint is found inside the supporting frustum, then its corresponding node is considered as completely occluded. The algorithm takes the advantage that frustum is constant and has to be computed only once for static models. However, it is relatively computationally consuming, especially with a floating point implementation. Another occlusion culling algorithm [9] applies shadow frusta that are extended from the viewpoint, and uses several large occluders as bases, and then culls off object nodes which are inside the frusta. This approach is limited with the number and the shape of occluders. Later, the same authors proposed a visibility culling algorithm using hierarchical occlusion maps (HOM) [17]. Our approach is closely related to this work. The main innovations of HOM are occluder fusion and efficient usage of conventional hardware acceleration.

Recently, the problem of exact visibility sorting of geometric objects without the help of hardware z-buffer is addressed in [14]. Instead of using conventional 3D rendering, it produces a sequence of layered images from a set of geometric parts, and uses them to compose the final image. This approach does not demand fast 3D graphics hardware, and relies mainly on general computation and 2D image operations.

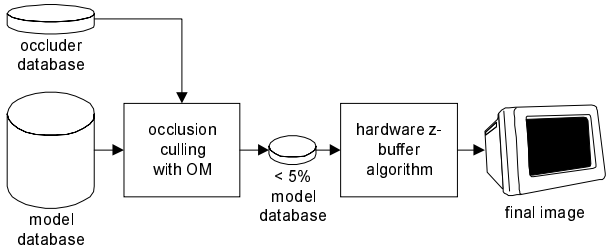


Figure 1: The occlusion culling algorithm using opacity map acts as a fast filter to cull away a large portion of hidden primitives in the model database.

3 Overview

We first divide the entire model into hierarchical bounding volume, by constraining that the leave nodes of the tree contain at most 256 primitives. Our approach makes use of occluders that are selected carefully in the pre-processing stage, to cull away a large portion of hidden nodes of the hierarchical bounding volume tree at run-time. Figure 1 shows the process flow of the rendering pipeline integrating this approach.

At the pre-processing stage, we construct the occluder database for certain grid points of the whole environment, using the minimum occluder set algorithm. The minimum occluder set is a minimal set of primitives that occlude one occludee. Note that an occludee may have several different minimum occluder sets. We compute the minimum occluder sets only for the occludees with more than 20 primitives. After grouping and sorting, the optimal set of occluders can be found out.

At the run-time, the algorithm performs the following tasks at each frame:

1. To query the occluder database, and retrieve the occluder list from the grid point nearest to the current viewpoint.
2. To render the retrieved occluders off-screen by conventional graphics hardware with frame and depth buffers. As we only need the image bitmap and depth value of the occluders, this rendering process is optimised by ignoring light and material setting. The resolution applied can be lower than the final display.
3. The resulting buffer contents are used to construct the opacity map and sparse depth map, respectively.

4. Using the opacity map and sparse depth map, we test recursively occlusion with the rectangular bounding box of the node's projected image. The occlusion culling consists of two dimensional overlap tests and depth comparisons. The two dimensional overlap test is enhanced by using only three integer additions or subtractions, while the depth comparison is carried out sparsely.
5. Finally, the nodes not culled in the occlusion culling step are regarded as conservatively visible and fed into the hardware z-buffer algorithm for exact visibility determination.

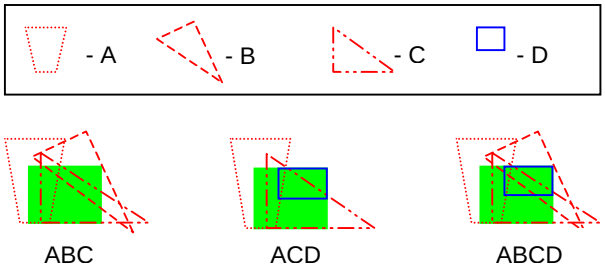


Figure 2: The idea of MOS. The primitives and their labels are shown in the box above. The shaded rectangle is the image of an occludee. The left and middle figures show two MOS (ABC and ACD) of the same occludee, while the one on right shows the wrong selection for MOS, as either B or D is redundant.

4 Minimum Occluder Set Algorithm

The general occluder selection criteria involve four major properties of a primitive. They are the size or projected size, first-hit, redundancy and computation cost. The former two are typically used to determine good occluders. Note that a primitive with a large projected size may have low depth complexity and incomplete coverage, and that first-hit primitives usually form a super set of the optimal set. Moreover, these do not take into account the combined gain with its neighbour occluders.

We define the optimal set of occluders to be the set of primitives giving the maximum ratio of its culling percentage to its computation cost. As any single occluder selection criterion cannot give the combined culling percentage of the whole set of occluders, only a rough approximation can be expected. In contrast, our scheme tries to find out the minimum set of primitives that occlude an occludee, as shown in Figure 2. This means that it chooses a set of primitives at one time, instead of picking only one primitive, which leads to more efficient elimination than occluders with incomplete coverage. With a suitable scoring scheme, we can find out the optimal set of occluders at a given viewpoint. The MOS algorithm has three major components: construction of occluder stack, generation of MOS for each occludee, and calculation of the score for each MOS. We pick the MOS with the highest score, and keep checking on redundancy.

4.1 Construction of Occluder Stack

For each occludee, an occluder stack is constructed to generate MOS for each occludee. It is a three dimensional array, with the rectangular base of the same size as the bounding box of the projected image of the occludee in the screen space. After depth sorting, if a primitive is in front of the occludee and covers some pixels of the occludee's projected image, the identifier of the primitive is pushed into the stack at the location of the covered pixels, as shown in Figure 3. With hardware graphics pipeline, the projected image of occludees and primitives can be found quickly.

Ideally, we would like to construct the occluder stack for all occludees. But it may need too much memory and time. In order to make it practical, the algorithm filters out the less significant occluders and occludees, such as tiny occluders and occludees containing only few primitives.

4.2 Generation of Minimum Occluder Set

After constructing the occluder stack, the algorithm first sorts the pixels of the occludee rectangular base, in ascending order of the number of primitives' identifier it contains, and builds up a table as shown in Figure 3. If there are pixels that are covered by no primitive, this occludee is regarded as visible and the search for this occludee's MOS stops. The first row of the combination table shows the number of primitives that cover different pixels. The first column indicates that there are pixels covered by only one primitive (*A* or *C)*, while the second column indicates that there are some pixels covered by two primitives, and similar for the rest. In other words, one slot represents a group of pixels that are covered by the IDs (primitives) it contains.

A slot will be cancelled if any of its IDs has been picked to be in *intermediate MOS*. For example, if the primitive *A* is picked, all slots containing *A* will be cancelled. If all slots of combination table have been cancelled, the occludee is completely covered by the current MOS, which will be stored into MOS database. Thus, finding one MOS of an occludee is equivalent to finding one combination of primitives that cancel all the

slots - the whole combination table. In order to find all MOSs of an occludee, an exhaustive search is carried out, for all the combination of primitives inside the table.

We run through the table from left to right, as it usually gives early termination. We simply pick the IDs of first column's slots as intermediate MOS, and cancel the associated slots. Then we concatenate the first IDs of the first remaining slot, and cancel the corresponding slots repeatedly. If the whole table is cancelled, we save this intermediate MOS in MOS database. Afterwards, we backtrack to the last concatenated ID's slot, remove the last ID from the intermediate MOS, recover the slots it cancelled, and try the next allowable ID in the same slot, and cancel the corresponding slots repeatedly, until we get another MOS. If there is no next allowable ID in the same slot, we backtrack further to the previous concatenated ID's slot, one step back at a time, until we find out all the MOSs. According to Figure 3, we first collect *A* and *C* as intermediate MOS. Then, only the third slot (*BD*) of second column remains. Hence, the MOSs of this example are *ABC* and *ACD*.

An upper bound on the complexity of an exhaustive search is *O(n!)*, where *n* is the number of different primitives of the table. Though it is run at pre-processing stage, shorter computation time is preferred. In practice, we usually do not need to compute all MOSs of each occludee; only the cheapest (in cost) portion of MOSs for each occludee will be kept. A pruning technique is applied to shorten the exhaustive search. If we find that the intermediate MOS already has higher cost compared with the ones inside the MOS database, we backtrack immediately. This leads to a quicker termination, and is a trade off for efficiency.

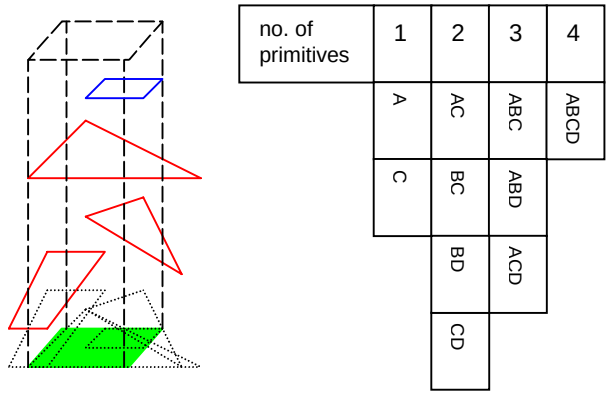


Figure 3: Occluder stack and combination table.

4.3 Scoring and Selecting

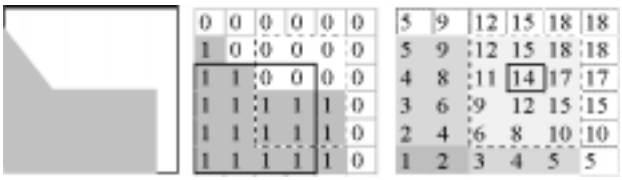
Each MOS has its gain and cost. The gain is the number of occludees it occludes, and the cost is the computation time for processing the MOS during occlusion culling at run-time. The gain is found by grouping the identical MOSs of all occludees together. If an MOS S_1 is the superset of MOS S_2, the algorithm adds the gain of S_2 to S_1. This approach explores the effectiveness of occlusion fusion. The cost of MOS is usually the rendering cost, as the occlusion culling will render all the selected occluders at each frame at run-time. This value is approximated by the number and the total projected sizes of occluders that the MOS contains. The number of occluders increases geometric computation, while their image sizes affect the rasterization time. Combining the gain, cost and user preference, the algorithm assigns a score to each MOS.

After sorting, the algorithm collects the top portion of MOSs up to a user defined limit. In order to remove redundant occluders that are contained in more than one MOS, or even hidden by occluders with higher scores, the algorithm makes use of *ID rendering*, that is, to render the occluders into the frame buffer with their IDs for rasterization, instead of their colours. With ID rendering, the redundant or hidden occluders will not be found in the ID buffer. The algorithm overlays the ID rendering of each MOS to the previous ID buffer, and repeats until the number of selected occluders reaches the limit. The final set of optimal occluders for the whole scene from a fixed viewpoint is then extracted from the ID buffer. This process of selecting MOS is essentially repeated for all representative viewpoints in different directions.

5 Occlusion Culling

The occlusion culling consists of three parts. These are view frustum culling, overlap test with the opacity map, and depth comparison with the sparse depth map. The view frustum culling is the typical one to apply on the hierarchical bounding volume tree at first. It culls away the nodes falling outside the view frustum, but not those hidden by occluders. In the occlusion culling algorithm an occludee is occluded if (*a*) the bounding box of its projected image is completely covered by occluders' image; and (*b*) the nearest depth value of occludee is farther than the depth values of occluders. The overlap test and depth comparison are applied to check these two conditions. If a node passes through both tests, it is hidden by the selected occluders; otherwise, the occlusion culling continues for its children recursively.

The straight forward solution to the overlap test and depth comparison is by a pixel-wise test. But its computation cost is prohibitive for interactive display. In contrast, the opacity map needs only two integer additions and one subtraction to do the overlap test. The sparse depth map further simplifies depth comparison. In this section, the opacity map and sparse depth map, as well as their uses and features will be described.



(a) (b) (c)

Figure 4: (a) The back buffer for rendering the occluders. The grey area is covered by all occluders. (b) The bitmap of the occluders. (c) The opacity map, and the shading showing the usage of opacity function.

5.1 Opacity Map

The opacity map is a two dimensional array of the scaled size of the final image, and stores the *opacity values* at each pixel. The *opacity value* of a pixel is the number of pixels , being covered by occluders and laying inside the rectangular area from lower left corner up to the pixel. In Figure 4, pre-selected occluders are rendered off-screen to produce the bitmap of the occluders' image. The bitmap is generated in the back buffer by graphics hardware. A 1 in the bitmap indicates that the pixel is covered by occluders, with 0 indicating not. The opacity value of the black box in (c) is equal to the number of 1's in the black bordered region in (b). The algorithm uses scan-line conversion to calculate the opacity values at each pixel. A row and column of zeros are added to eliminate the boundary cases during overlap test. For simplicity, we do not show them in the figure. As these zeros do not need to be updated, they are ignored at the construction phase of the opacity map. The resolution of the opacity map used for the model tested in this paper is 128×128, excluding the first row and column of zeros, while the displayed image resolution for the final images are 512×512 or 1024×1024. We feel that this is a good balance between the accuracy and computation time.

5.2 Overlap Test

The aim of the overlap test is to check whether the rectangular area of the projected image of an occludee is completely covered by occluders' images. In other words, it checks if the area of occludee's image is fulfilled by 1s in the bitmap. With the opacity map, this query can be done by *opacity function (OPF)*,

$$OPF(x_1, x_2, y_1, y_2) = Op(x_1,y_1) - Op(x_1,y_2) - Op(x_2,y_1) + Op(x_2,y_2)$$

where $Op(s, t)$ means the opacity value at co-ordinates (s, t) in the opacity map, while the lower left corner of the opacity map has the co-ordinates (1, 1). The OPF calculates the number of 1's in the rectangular region ($x_1<x<=x_2$, $y_1<y<=y_2$) of the bitmap. Figure 4c shows the application of OPF to do overlap test for one occludee. The dash lines border the rectangular region (2<x<=5, 1<y<=5), which is the occludee's projected image. The region has 12 pixels in total. Then, we calculate,

$$OPF(2, 5, 1, 5) = Op(2, 1) - Op(2, 5) - Op(5, 1) + Op(5, 5) = 2 - 9 - 5 + 18 = 6$$

It means that the occluders cover only 6 pixels inside this region. Compared with the region size (12 pixels), the occludee is not occluded by the occluders and therefore fails the overlap test.

Besides the benefit of occluder fusion, the opacity map allows the overlap test of one occludee to be done with only two additions and one subtraction. Moreover, two more modifications can be made to perform approximate overlap tests and adaptive overlap tests.

Approximate Overlap Test: For a highly dense scene composed of many tiny primitives, such as a bottle full of small stones, a certain tolerance can be added to the opacity function. This makes the overlap test ignore some holes of the occluders' image, and regard the almost entirely hidden nodes being occluded. Using the opacity map, this modification is easy to achieve.

Adaptive Overlap Test: In order to balance the computation time of the occlusion culling algorithm and rendering process, a *coverage ratio* threshold is used to trigger a stop signal to the recursive occlusion culling algorithm. The coverage ratio is the ratio of result of the opacity function of one occludee to its rectangular image size. If the occludee has a coverage ratio less than 0.2, the algorithm stops testing its descendants, as in this case the

occluders cover too little area of the occludee and have low chance to completely cover the occludee's descendants. Consequently, those descendants are regarded as conservatively visible. The threshold will be adjusted according to the culling time, and prohibits the extra occlusion culling in the case where the rendering capacity is much larger than the number of primitives falling in the view frustum.

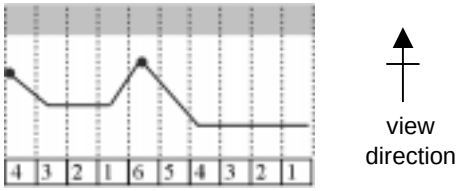


Figure 5: One row segment of the sparse depth map. It is the top view of an occludee (the grey rectangular box), and some occluders (the black lines). The two black dots mark the local farthest pixels (with the locally largest depth values) of this segment, called the peaks.

5.3 Sparse Depth Map

The sparse depth map is an auxiliary data structure of the depth map, which is generated at the same phase of off-screen rendering. The depth map is a two dimensional array recording the depth values (nearest) of the occluders. In a general approach, the depth comparison is carried out for every pixel the occludee covers. But there is depth coherence in the same row, especially in the case of the same occluder. In a row, the depth value varies in three modes, *near-to-far*, *far-to-near* or *still*; and this can be plotted as a line segment chart, where the line segment increases, decreases or keeps flat. With the chart, we locate the local peaks, which has the largest depth values locally, as shown in the Figure 5. The algorithm now only seeks the local peaks of the occluders, instead of every pixel. The sparse depth map is constructed to store the number of pixels apart from the nearest local peak to the right.

To construct the sparse depth map, the algorithm transverses the depth map from the upper right corner to the bottom left, row by row. An integer variable *step* is used to record the number of pixels that can be skipped. Ignoring the border case, it tests two consecutive (named *current* and *last*) pixels. If they are increasing or keeping still, the algorithm adds one to the *step* variable and saves it into *current* pixel of sparse depth map. Otherwise, if the previous test shows increasing and keeping still, the current pixel is the peak. It stores *step* plus one into the peak pixel of the sparse depth map, and then resets the *step* to one.

To reduce the construction time of the sparse depth map, the algorithm does not compute the row of pixels that are covered by no occluders, because those rows will not be used for the depth comparison. As the sparse depth map exploits the pixel coherence, if the depth map varies from near-to-far and far-to-near alternatively each pixel, the sparse depth map will contain all 1s. This means there is no pixel that can be skipped, and the algorithm will test every pixel as the usual depth comparison. In this case, the sparse depth map should be disabled, in order to save the construction time. The resolutions of depth map and sparse depth map used in our tests are the same as the opacity map, i.e. 128×128.

5.4 Depth Comparison

The depth comparison uses both the depth map and sparse depth map. For an occludee, the algorithm finds the nearest depth value of its bounding volume. This simplifies the depth comparison, and also guarantees the correctness of the culling algorithm. The depth comparison is applied to the rectangular projected area of the occludee. It tests the depth from the bottom row to the top of the rectangular area. For one row, it first tests the depth value of the leftmost pixel. If the nearest depth value of the occludee is large than the pixel value of the depth map, it will test the next-jump pixel indicated in the sparse depth map. Otherwise, the occludee is in front of the occluder and the depth test fails and terminates.

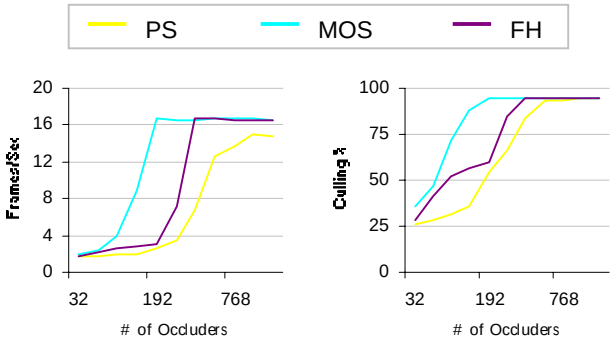
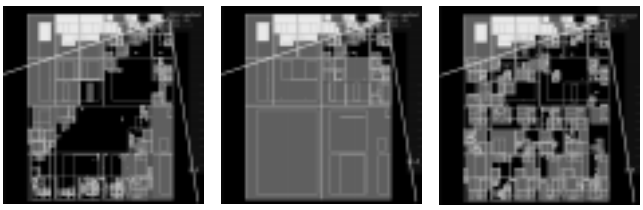


Figure 6: Frame rate and culling percentage of different occluder selection methods. *PS* stands for the criterion of projected size, *FH* stands for the criterion of first-hit.

6 Results and Analysis

We have implemented the above algorithms on a simple walkthrough system, which uses OpenGL and runs on SGI Max IMPACT workstation with R10000 CPU (200MHz) and 192 MB RAM. In this section, we demonstrate the performance of the minimum occluder set algorithm and occlusion culling using the opacity map. The test model is composed of thirty copies of a Chicago city model and contains 300,540 polygons in total. The whole environment uses one light and no texture. An overview of the test model is shown in Figure 11.



(a) (b) (c)

Figure 7: The top view of model. The light grey boxes are outside the view frustum, the dark grey boxes are culled by occluders and the black boxes are conservatively visible. From the left to right, the figures show the cases (a) projected size, (b) *MOS* and (c) first-hit criteria respectively.

6.1 MOS Algorithm

In the following tests, we compare the performances of different occluder selection criteria. They are the projected size, MOS, and first-hit. The experiment is carried out at a certain viewpoint that gives about 400 visible primitives in 512×512 resolution. For the criterion of projected size, we simply pick occluders in the descending order. For the first-hit criterion, we first find out all the visible primitives, and count the number of pixels covered by these primitives. Afterwards, we choose the occluders in the descending order. We record the frame rate and culling percentage, varying the maximum number of occluders used.

Figure 6 shows that the MOS algorithm needs 192 occluders to achieve the optimal culling percentage, about 94%. The criterion of first-hit uses about 384 occluders to reach the same culling percentage. The projected size criterion has about 93% culling with 512 occluders. The culling percentage of the projected size criterion has the slowest growth rate. Also, more occluders are used, more computation overhead is introduced for occlusion culling, thus decreasing the frame rate shown in the tail part of the curve. The MOS algorithm uses a half of occluders as by the first-hit criterion to yield the optimal culling percentage, as it considers the combined gain and redundancy of primitives. These points are illustrated in Figure 7, which shows the top view of the whole model. The light grey boxes are nodes outside the view frustum, and the dark grey boxes are culled away by the occluders. These are the results when 192 occluders are used. Except in the MOS algorithm, the incomplete coverage caused by other two methods reduces the culling percentage, while the redundancy of occluders leads to increased overhead without improving culling ratio.

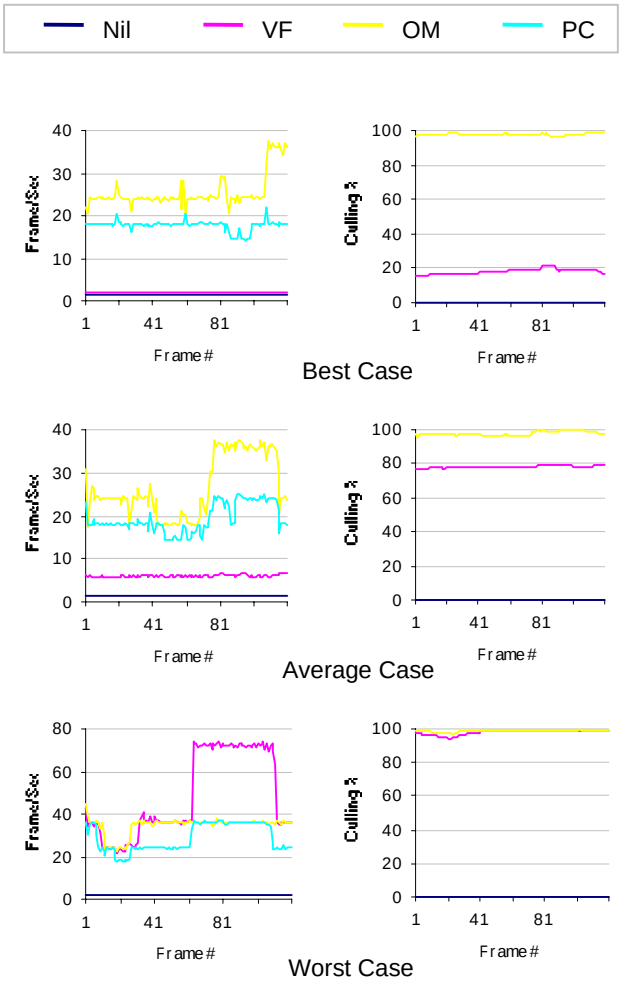


Figure 8: Performances of occlusion culling with different routes. *Nil* represents that no culling is applied. *VF* represents that view frustum culling is applied. *OM* represents that occlusion culling with opacity map and sparse depth map is applied. *PC* means occlusion culling with pixel-wise comparison.

6.2 Occlusion Culling

We have conducted two groups of tests for the occlusion culling. The first group is aimed to illustrate the speedup of occlusion culling with different depth complexities; and the second group shows performances and bottleneck at different resolutions.

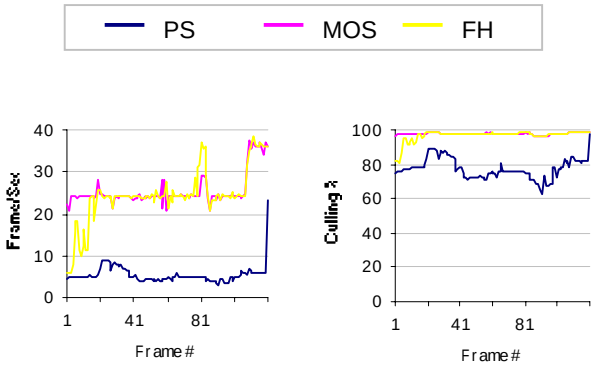


Figure 9: Performances of occlusion culling with different occluder selection criteria for the best case route. *PS*, *MOS* and *FH* represent the criteria of projected-size, minimum occluder set and first-hit, respectively.

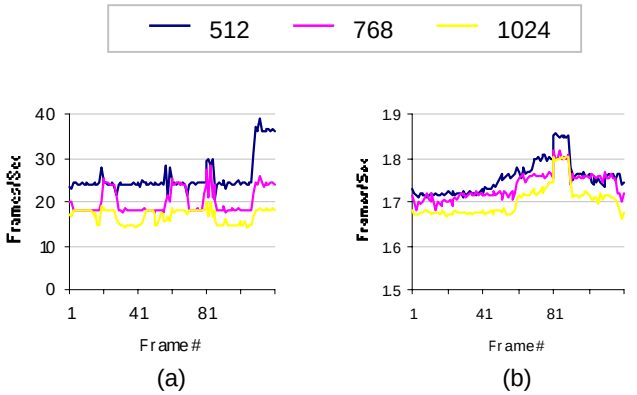
Tests at Different Routes: The following three tests are carried out with the same Chicago model, but alone different routes. The three routes are located with different depth complexities, and classified as *best*, *average* and *worst* cases for the speedup. The tests use 64 occluders and have 512×512 resolution. The three routes have 120 frames each.

For the best case, the route starts at the lower left corner of the environment, and heads towards the center part. It has the highest depth complexity. The speedup of occlusion culling to view frustum is 14.6 and the average frame rate is 25.5. For the average case, the route is located at the center of the environment, the depth complexity is medium. It has the speedup of 4.4 and average frame rate of 26.7. For the worst case, the route is set at the upper right corner of the environment, with the viewer looking outwards. It has lowest depth complexity, and the speedup and average frame rate are 0.7 and 34.6, respectively. For reference, the frame rate of occlusion culling with pixel-wise comparison is also shown in Figure 8. It has the average frame rate of 17.7, 19.4 and 28.8 for the three routes, respectively.

According to Figure 8, the occlusion culling has adverse effect on the frame rate in the worst case. That is because the computation cost of view frustum culling is lower than occlusion culling. If the environment has low depth complexity, occlusion culling causes overhead instead of profit to culling percentage.

Figure 9 shows the performance of occlusion culling using different occluder selection criteria for the best case route. The average frame rates for projected size and first-hit criteria are 5.4 and 24.5, relatively. The difference between MOS and first-hit criteria decreases progressively in the first twenty frames, and their performances are similar in the remaining frames. That is because the routes do not have too much visible primitives, so the superset of occluders (first-hit ones) converges to the optimal set after the first twenty frames.

Tests at Different Resolutions: The performance of occlusion culling using the opacity map is shown in Figure 10. The test is based on the best case route, using MOS. The two figures show the results of view frustum culling and occlusion culling at resolutions of 512×512, 768×768 and 1024×1024. The average frame rates are 25.6, 20.1 and 16.7 of the three ascending resolutions. As the sizes of opacity map and sparse depth map applied for three resolutions are the same, their culling percentages are constant. It is regarded as no change for the geometric computation. The drop in frame rate is caused by the rasterization of hardware rendering process, which is also the bottleneck of walkthrough system now. Although the frame rates of 768×768 and 1024×1024 resolutions are lower, we still have a speedup of 9.8 and 11.6 respectively.



(a) (b)

Figure 10: (a) Performances of occlusion culling using opacity map and MOS algorithm at different resolutions, 512×512, 768×768 and 1024×1024, (b) The result of view frustum.

7 Conclusions and Future Work

We have presented an occlusion culling algorithm (MOS algorithm) using the minimum occluder set and opacity map. Our algorithm results in significant speedup of the frame rate and a reduced number of occluders required. The speedup by occlusion culling is due to the use of the opacity map and sparse depth map. The opacity map needs only two integer additions and one subtraction to do the overlap test. The sparse depth map further simplifies depth comparison, by not using pixel-wise comparison. Moreover, the high culling percentage is achieved by the MOS algorithm, which takes into account the combined gain and redundancy of occluders. The occlusion culling algorithm makes no special assumption on occluders and models and is suitable for implementation on current graphics systems.

Further research includes the extension of the MOS algorithm to dynamic environments and integration with impostors for scalability. The MOS algorithm can be adapted to a dynamic model if the probability of dynamic occlusion is considered in the process of scoring. For an environment with a large number of visible primitives, we can apply impostors [18] for distant objects. Integration with impostors would make a walkthrough system into a semi-image-based VR system. Thus we would still have geometric data for nearby objects, which allows collision detection and interaction for the users, and the total number of primitives handled by graphics hardware is greatly reduced since distant primitives are represented as impostors.

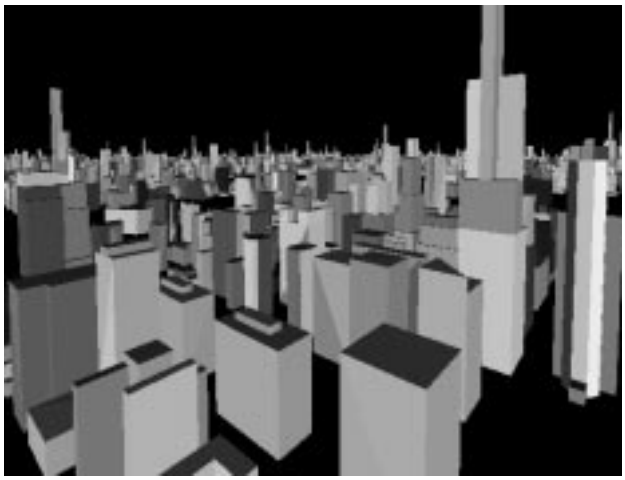


Figure 11: A birdeye view of the test model, which is composed of thirty copies of a Chicago city model and contains 300,540 polygons in total.

8 Acknowledgements

We thank Mr. Cheng Kin Shing for his contributions and many discussions during the formative phase of this work.

9 References

1. P. K. Agarwal, L. J. Guibas, T. M. Murali and J. S. Vitter. Cylindrical Static and Kinetic Binary Space Partitions. *Computational Geometry '97*, pp. 39-48.
2. K. Akeley. RealityEngine Graphics. *SIGGRAPH '93*, pp. 109-116.
3. E. Catmull, A Subdivision Algorithm for Computer Display of Curved Surfaces. PhD thesis, University of Utah, 1974.
4. S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. *Symposium on Computational Geometry 1996*, pp. 78-87.
5. S. Coorg and S. Teller. Real-time Occlusion Culling for Models with Large Occluders. *Symposium on Interactive 3D Graphics 1997*, pp. 83-90.
6. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH '80*, pp. 124-133.
7. N. Greene, M. Kass and G. Miller. Hierarchical Z-Buffer Visibility. *SIGGRAPH '93*, pp. 231-238.
8. N. Greene. Hierarchical Polygon Tiling with Coverage Masks. *SIGGRAPH '96*, pp. 65-74.
9. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff and H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. *Computational Geometry '97*, pp. 1-10.
10. D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potential Visible Sets. *Symposium on Interactive 3D Graphics, 199*, pp. 105-106.
11. K. Mulmuley. An Efficient Algorithm for Hidden Surface Removal. *SIGGRAPH '89*, pp. 379-388.
12. B. Naylor. Partitioning Tree image Representation and Generation from 3D Geometric Models. *Graphics Interface '92*, pp. 201-211.
13. R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for Applying Computer-Generated Images to Visual Simulation. *Technical Report AFHRL-TR-69-14. 1969*.
14. J. Snyder and J. Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decomposition. *SIGGRAPH '98*, pp. 219-230.
15. S. Teller and C.H. Sequin. Visibility Pre-processing for Interactive Walkthroughs. *SIGGRAPH '91*, pp. 61-69.
16. E. Torres. Optimization of the Binary Space Partition Algorithm for the Visualization of Dynamic Scenes. *Eurographics '90*, pp. 507-518.
17. H. Zhang, D. Manocha, T. Hudson and K. E. Hoff III. Visibility Culling using Hierarchical Occlusion Maps. *SIGGRAPH '97*, pp. 77-88.
18. F. Sillion, G. Drettakis, B. Bodelet. Efficient Impostor Manipulation for Real-Time visualization of Urban Scenery. *Eurographics '97*, pp. 207-218.