EFFICIENT OCCLUSION CULLING AND NON-REFRACTIVE TRANSPARENCY RENDERING FOR INTERACTIVE COMPUTER VISUALIZATION

POON Chun-Ho
M. Phil. THESIS

A dissertation submitted to
the University of Hong Kong
in partial fulfillment of the requirements for
the degree of Master of Philosophy

The University of Hong Kong

August 2000

Acknowledgement

I would like to thank my supervisor, Dr. Wenping Wang for his advises and support. Throughout my graduate studies, Dr. Wang has always directed and enlightened my research.

In particular, I would also thank Mr. Dominic Cheng for his contributions and many discussions during the formative phase of this work.

Moreover, I would thank my family and Mandy Cheuk for their support and encourage throughout the years.

The last but not the less, thanks go to my groupmates of Computer Graphics laboratory in the Department of Computer Science and Information Systems, K. B. Cheung, Y. K Choi, Y. L. Leung and K. H. Yeung, for their support and joy.

Declaration

I hereby declare that the dissertation, submitted in partial fulfillment of the requirements for the degree of Master of Philosophy and entitled "Efficient Occlusion Culling and Non-Refractive Transparency Rendering for Interactive Computer Visualization", represents my own work and has not been previously submitted to this or any other institution for any degree, diploma and other qualification.

POON Chun-Ho

Abstract of thesis entitled

Efficient Occlusion Culling and Non-Refractive Transparency Rendering for Interactive Computer Visualization

submitted by

POON Chun-Ho

for the degree of Master of Philosophy
at The University of Hong Kong
in August 2000

Hidden surface removal and non-refractive transparency rendering are two fundamental problems for interactive visualization. Hidden surface removal helps to identify the hidden primitives, bypass their rendering processes, so as to reduce the workload of graphics engine and provide an interactive frame rate. Without the distortion from refraction, non-refractive transparency rendering gives more insight of the interior structure of geometric models. This research is aimed at studying these critical problems.

Exact hidden surface removal is usually implemented at pixel level, such as z-buffering, in order to achieve the image precision. It is computationally power demanding. A conservative algorithm that finds a minimal superset of visible primitives in a short time is considered as the most suitable approach for interactive visualization. One significant technique that makes use of occlusion relation among the primitives of the model, and culls a significant amount of invisible primitives at different viewpoints quickly, is studied and devised to an occlusion culling algorithm.

In this thesis, we present two algorithms to improve occlusion culling for a highly occluded virtual environment. The first is a novel method to select occluders with multiple criteria at pre-processing stage for static environment, using the idea of the *minimum occluder set* (MOS). The MOS of an occludee is the minimal set of primitives that occludes the occludee. The second is 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.

A common method for non-refraction transparency rendering is interpolated transparency. Interpolated transparency calculates the resultant intensity by linearly interpolating the corresponding intensities of two pixel fragments. The main disadvantage of interpolated transparency is that, when there are more than two pixel fragments, the graphics pipeline has to apply pair-wise interpolations in a sorted visibility order. However, finding visibility order at object space is a complex task, especially for a self-penetrating model.

Another approach is screen door transparency. Screen door transparency generates a set of two dimensional masks that represents the different levels of opacity. The pixels of the masks have only two states, which determine whether they are visible or not. At higher opacity level, the corresponding mask contains less filled pixels. Though this technique is order invariant, the correctness of resultant opacities depends heavily on the size of mask and the distribution of pixels inside each mask. An improved screen door transparency rendering is presented, that applies a simple and fast mask generation algorithm and is feasible to be implemented at sub-pixel level.

The algorithms mentioned above have been implemented and applied to different experimental tests and a learning toolkit for medical student called "Virtual Brain". A significant speedup and a high accurate order invariant transparency rendering have been observed.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Interactive visualization is one of the major applications of computer graphics.

Hidden surface removal and non-refractive transparency rendering are two well known fundamental problems. They are still valuable research topics because of fast growing data size and complexity. These demands mainly come from the interactive visualization of architectural model, scientific and medical dataset and walkthrough of outdoor scene. Hidden surface removal helps to identify the hidden primitives, bypass their rendering processes so as to reduce the workload of graphics engine, and provide an interactive frame rate. Non-refractive transparency rendering outputs a correct amount of luminance that transfers through several transparent objects without consideration of optical refraction. Without the distortion from refraction, non-refractive transparency rendering gives more insight of the interior structure of geometric model, such as medical dataset, which is important for scientific visualization and pathological analysis. This research is aimed at studying these two problems and applying two novel techniques for interactive visualization.

Exact hidden surface removal is usually implemented at pixel level, such as z- buffering [2, 6], in order to achieve the image precision. It compares the depth value of the incoming fragment and the current one in the frame buffer at each pixel, and only keeps the closest one. It is computationally power demanding, and requires hardware implementation for interactive display. Another approach applies a visibility determination at primitive (or grouped primitives) level. At this level, an exact solution is also feasible, that makes use of spatial subdivision, but the overhead of visibility determination is high as well [13, 27, 28, 32]. Moreover, further subdivision of primitives is often necessary, that increases the overhead at the same time. A conservative algorithm that finds a minimum superset of visible primitives in a short time is considered as the most suitable approach for interactive visualization [7, 8, 17, 21, 35, 40]. One significant approach that makes use of occlusion relation among the primitives of the model, and culls a significant amount of invisible primitives at different viewpoints quickly, is studied and devised to an occlusion culling algorithm.

An occlusion culling algorithm consists of two processes, selection of occluders and occlusion culling. An occluder is a primitive among the whole model, which is lain before other primitives at certain view point. If the data or model is assumed not to be changed, it is a static model environment. The selection of occluders will be run at the pre-processing stage, therefore a computational consuming but highly effective algorithm is allowed. On the other hand, a simple method for selection of occluders will be applied for dynamic environment. In this thesis, we present the following algorithms for the occlusion culling:

1. a novel method to select occluders with multiple criteria at pre-processing stage for static environment, 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.

A common method for non-refraction transparency is interpolated transparency [11]. Interpolated transparency calculates the resultant intensity by linearly interpolating the two corresponding intensities of pixel fragments. For example, let I_1 and I_2 be the intensities of two pixel fragments P_1 and P_2, where P_1 is in front of P_2 at certain view point. The coefficient k_1, range from 0.0 to 1.0, represents the opacity value of P_1. The combined intensity I is calculated as below.

$$I = k_1 I_1 + (1 - k_1) I_2$$

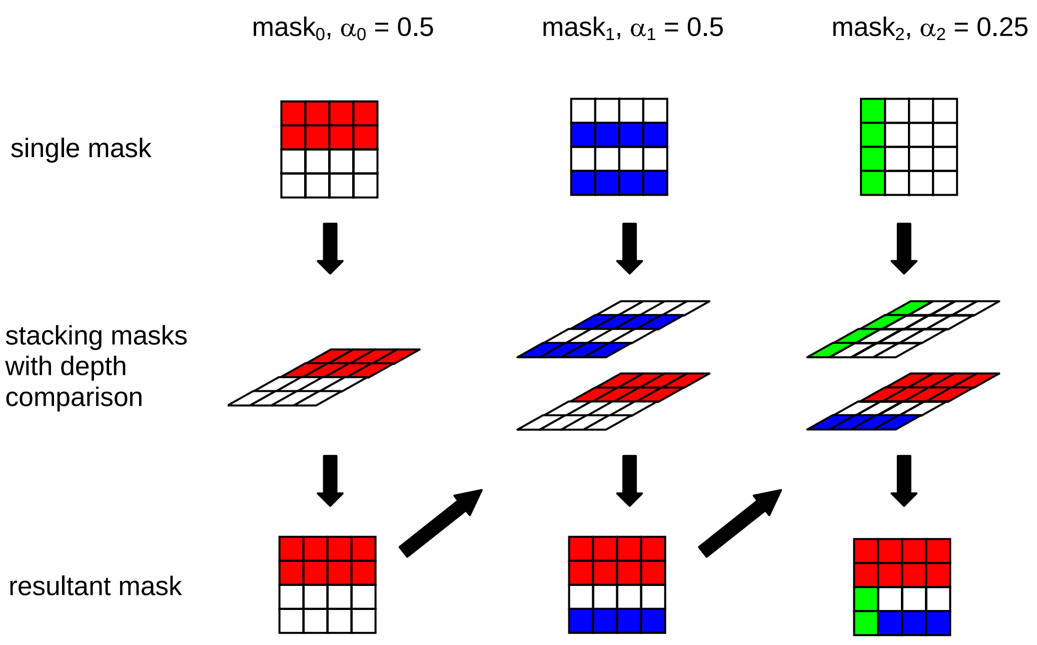
The main disadvantage of interpolated transparency is that the correctness only holds for two pixel fragments. When there are more than two pixel fragments, the graphics pipeline has to apply pair-wise interpolations in a sorted visibility order. However, the visibility order at object space is a complex computation task, and even worse for self-penetrating model.

Interpolated transparency is further accelerated by alpha channel and commonly regarded as alpha blending [30]. Since alpha blending provides a precise visual effect, there are several literatures [5, 20, 22, 38] to provide order invariant algorithm, by using A-buffer or multi-pass rendering.

Another approach is screen door transparency. Screen door transparency [12, 26] generates a set of two dimensional masks, that represent the different levels of opacity. The pixels of the mask have only two states, and determine whether they are visible or not. At highly transparent object, the corresponding mask contains fewer filled pixels. Using the masks, we will render the transparent objects with masked rasterization. If there are two or more transparent objects, their masks are stacked together one by one with depth comparison. Therefore, some pixels will be covered, but some are still visible finally. If the mask size is infinite, the portions of visible pixels from different objects should be the same as alpha blending.

Though this technique does not require a visibility order, the correctness of opacities depends heavily on the size of the masks and the distribution of pixels inside each mask. Moreover, a distracting pattern will occur if screen door transparency is applied at pixel level.

We give an example to show the principal of screen door transparency in Figure 1. We have three masks, with the alpha values of *0.5*, *0.5* and *0.25* correspondingly. We also assume that their visibility order is $mask_0$, $mask_2$ and $mask_1$. The first row shows their individual masking pixels. We then stack them together according their depth order in the second row. The third row shows the resultant masks. We observe that the portion of visible pixels has the same effect as the resultant alpha blending.



mask$_0$, α_0 = 0.5
mask$_1$, α_1 = 0.5
mask$_2$, α_2 = 0.25
single mask
stacking masks with depth comparison
resultant mask

	mask$_0$	mask$_1$	mask$_2$
alpha blending	0.5	$(1-\alpha_0)\times(1-\alpha_2)\times\alpha_1$ = 0.1875	$(1-\alpha_0)\times\alpha_2$ = 0.125
No. of visible pixels / mask size	8/16 = 0.5	3/16 = 0.1875	2/16 = 0.125

Figure 1: The principal of screen door transparency.

An improved screen door transparency rendering, that applies a simply and fast mask generation algorithm and is feasible to be implemented at sub-pixel level, is presented as another contribution of this research.

A learning tool kit for medical student called "Virtual Brain", that applies both the occlusion culling and non-refractive transparency rendering mentioned above will be introduced in this thesis as well.

1.2 Difficulties

1.2.1 Occlusion Culling

In general, the performance of hidden surface removal has a combined factor of culling percentage and computation cost. An exact hidden surface removal has the highest culling percentage but large computational time.

For the purpose of interactive visualization, we apply a conservative occlusion culling, that means we take a trade off between the computational time and an acceptable culling percentage.

The culling percentage depends on the quantity and quality of the occluders selected. As a single occluder rarely covers other primitives wholly, we usually choose a small portion of primitives grouped as an occluder set. For simplicity, the primitives are not in the occluder set are called occludees, though their roles are potential occludees indeed, since they will be tested against occlusion at run time. Even in a static environment, we need different occluder sets at different view points.

To select more occluders, we may reach a higher culling percentage. However, this also increases the computation time of culling. On the other hand, if we only apply a small amount of occluders, the culling percentage may be too low, and has no significant reduction of rendering time. Therefore, we define the optimal set of occluders is the set of primitives giving the maximum ratio of its culling percentage to its computation cost. The general occluder selecting criteria consider four properties of a primitive. They are the size or projected size, first hit, redundancy and computation cost of the primitive. The former two criteria are typically used to determine good occluders. However 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, they do not take into account the combined gain with neighbor occluders. These criteria only give an approximation of optimal occluder set, and there is a challenge to find out the minimal set of primitives that gives the highest culling percentage.

The culling algorithm usually involves a lot of floating point calculations for an atomic operation. If an environment consists of thousands of primitives, the computation time is not acceptable. More considerations about hierarchical data grouping and simplification of atomic operation are required to fulfill the narrow time slot of real time display.

1.2.2 Non-Refractive Transparency

The visual effect of screen door transparency depends on the size of mask, and the distribution of filled pixels. The size of mask limits the number of opacity level, and also affects the error in the case of stacking pixel masks for several transparent objects. The output of stacked pixel masks is also regarded as higher order opacity. In our approach, we apply screen door transparency at sub-pixel level, and we render the scene with general frame buffer. The size of frame buffer bounds the size of mask and output window. If the mask size is too large, the output window will be small, and vice verse. Moreover, in order to provide a direct implementation on the cutting edge graphics engine [25] with sixteen sub-pixel sampling, we have to keep the mask be reasonably small, such as sixty four pixels.

Since the mask is small, the number of erroneous pixels becomes significant for higher opacity order. A systematic mask generation is important to compute the coverage mask with minimum error. An algorithm is presented in [26], which gives precise masks with minimum error. However, its complexity order is 2^m, where m is the number of transparent objects. In practice, the computation time is too long for interactive display if we have tens of transparent objects.

1.3 Contributions

1. A novel method to select occluders with multiple criteria for static geometric data, 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 spatial hierarchy of the whole model at run-time.
3. A fast visibility order independent transparency rendering algorithm, which applies the screen-door transparency at subpixel level.
4. An application called "Virtual Brain", that supports interactive visualization of the surface-based data model of human organs on affordable PC platform and is used for computer-based anatomy and pathology teaching tool.

1.4 Overview of This Thesis

In this thesis, we shall briefly discuss related works of hidden surface removal and non-refractive transparency rendering in section 2. The details of occlusion culling including the minimum occluder set algorithm, the occlusion culling algorithm using opacity map and sparse depth map will be presented in sections 3. In section 4, mask generation of screen door transparency and its relative consideration are given. The experiments and applications of the algorithm mentioned above will be described and analyzed in section 5. The thesis concludes in section 6.

2 Related Work

2.1 Hidden Surface Removal

Hidden surface removal is a fundamental problem in computer graphics. The conventional z-buffer algorithm is implemented in hardware or software [2, 6] that yields exact visibility information by pixel-wise comparisons of depth values of every primitive.

The binary space partitioning (BSP) tree algorithm [13, 28], which refines the work in [32], 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 [27]. Further research in computational geometry on randomized algorithms for maintaining a BSP tree for a dynamic model has been conducted [1, 36], which, however, does not lead to practical results.

The potentially visible set (PVS) [21, 35] 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 [14] 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 [15] 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 [7, 8] 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 needs to be computed only once for static models. However, it is relatively computationally consuming, especially with a floating point implementation. Another occlusion culling algorithm [17] 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) [40]. Our approach is closely related to this work. The main innovations of HOM are occluder fusion and efficient usage of conventional hardware acceleration.

The problem of exact visibility sorting of geometric objects without the help of hardware z-buffer is addressed in [34]. 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.

2.2 Non-Refractive Transparency Rendering

There are two major approaches for non-refractive transparency rendering, they are alpha blending and screen door transparency. Alpha blending [30] is the widely accepted method and has well supported from hardware alpha channel. However, it requires to blend the pixel fragments pair-wise in a far-to-near order. In practice, it is difficult to find the visibility order in real time. Therefore, several researches extend the usage by providing order invariant methods.

The A-buffer [5] is one major direction for order invariant alpha blending. It stores all fragments of every pixel in a depth-sorted order, and the resultant intensity of each pixel is simply weighted interpolation according to the order. Though the storage space can be saved by merging the fragments that come from the same primitive and overlapped in depth, it still has an unbounded storage requirement. An extended hardware architecture is proposed in [18], which requires a small fixed amount of storage. To limit the usage of storage, it merges the fragments that are very close in their depth-values. There are some more improvement issues in this literature, but the details will not be given here.

Another approach is multi-pass based transparency rendering [22, 38]. The multi-pass based techniques only keep an opaque pixel map and a sort depth pixel map. The algorithm first renders all the opaque objects and stores their intensities and depth values into the opaque pixel map. Then, it renders all the transparent objects into the sort depth pixel map, and only keeps the fragment, that is the closest to the one in the opaque pixel map. The opaque pixel map now blends with the sort depth pixel map. In this pass, one transparent object is resolved. The operation repeats for the remaining transparent objects until all of them are resolved. Obviously, the maximum number of passes is the maximum number of transparent layers among any pixels. Although it provides an order invariant alpha blending, the rendering time of multi-pass is not suitable for interactive visualization.

A hybrid method [20] keeps a buffer of constant number of image layers, such as four layers. It follows the approach of multi-pass based technique for the first four closest layers. If there are more fragments coming, the buffer overflows. We composite these four layers into one, and free the remaining three layers to find the next three closest fragments repeatedly.

The reason for multi layered buffer is based on the observation of the overflow happening at a low chance, and most of the common cases can be resolved within the number of image layers.

Screen door transparency is not a new method, but it seems to be a supplementary approach in the past. Since it is closely related to the supersampling implementation of hardware, some issues [2, 16, 25, 31] are given to address it as architectural features. There is only a few pin-pointing literatures [11, 12]. A detailed discussion of screen door transparency has been raised in [26]. It explores the higher order opacity of using different types of pixel masks adapted from digital halftoning [3, 10, 19, 23, 24, 37], and presents a new systemic algorithm. The algorithm gives a set of less erroneous pixel mask, but its complexity order is 2^m, where m is the number of masks or transparent layers. Our approach is aimed to reduce its complexity order, and make it be applicable to an interactive application.

3 Occlusion Culling

In this chapter we describe the whole process of occlusion culling, including MOS algorithm that helps to select an effective and efficient occluders, and the occlusion culling algorithm which uses opacity map and sparse depth at run time. We first show the overview of the whole process of occlusion culling, and then we further explain the details of MOS and occlusion culling algorithms.

3.1 Overview

The aim of occlusion culling is to cull a significant amount of invisible primitives at different viewpoints in real time. To reduce the overhead, 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 a large portion of hidden nodes of the hierarchical bounding volume tree at run-time. Figure 2 shows the process flow of the rendering pipeline integrating this approach.

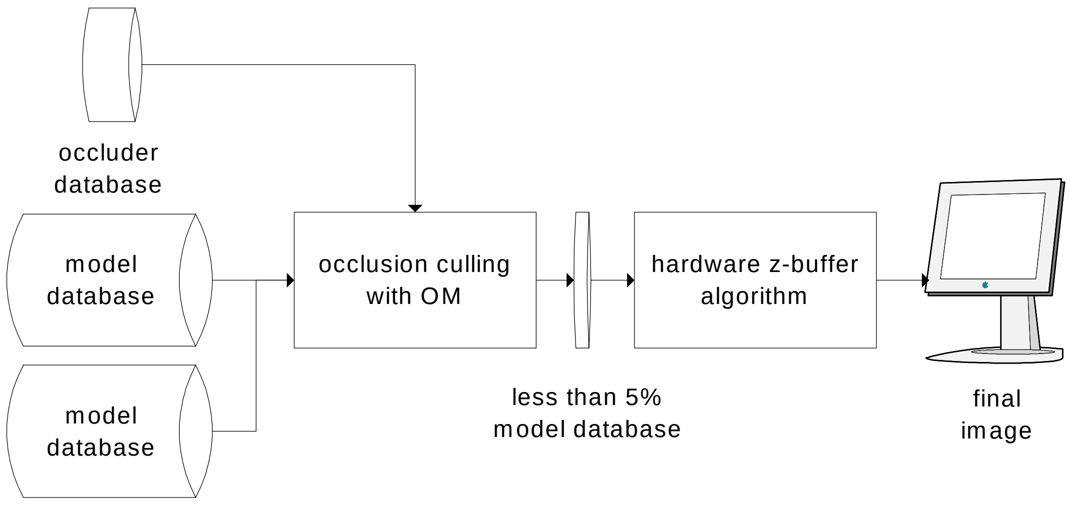

occluder database
model database
model database
occlusion culling with OM
hardware z-buffer algorithm
less than 5% model database
final image

Figure 2: The occlusion culling algorithm using opacity map acts as a fast filter to cull a large portion of hidden primitives in the model database.

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 many 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.

At 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 optimized 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 for occlusion recursively with 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.

3.2 Minimum Occluder Set Algorithm

As mentioned before, a single occluder selection criterion, such as the size or projected size, first-hit, redundancy or computation cost, has its own weakness and does not consider the combined gain of culling percentage with neighbor 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. These criteria fail to give the combined culling percentage of the whole set of occluders, and only provide a rough approximation. In contrast, our scheme tries to find the minimum set of primitives that occludes an occludee, as shown in Figure 3. It chooses a set of primitives at one time, instead of picking only one primitive. Therefore it leads to more efficient elimination than those occluders only yield incomplete coverage. With a suitable scoring scheme, we can find 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.

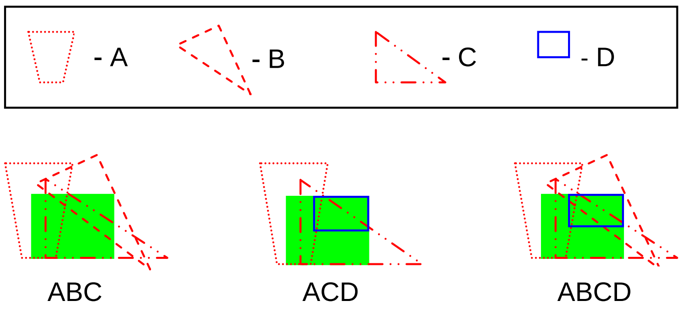
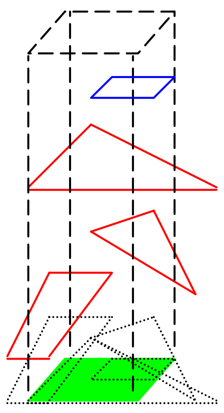


Figure 3: The idea of MOS. The primitives and their labels are shown in the box above. The shaded rectangle is the image of an occludee. The left and middle figures show two MOS (ABC and ACD) of the same occludee, while the one on right shows the wrong selection for MOS, as either B or D is redundant.

3.2.1 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 4. 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 objects containing only few primitives.



no. of primitive	1	2	3	4
	A	AC	ABC	ABC
	C	BC	ABD	
		BD	ACD	
		CD		

Figure 4: Occluder stack and combination table.

3.2.2 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 4. 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 covering certain pixels within the projected image of occludee. 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 other 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. In the other words, after the primitive *A* is picked, certain pixels within the projected image of occludee is supposed to be covered, and we can skip the consideration of which other primitives can cover these pixels. 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. Moreover, there may be many different combinations of primitives that covered all the slots as well. Therefore, 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 the 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 at a time, until we find all the MOSs. According to Figure 4, we first collect *A* and *C* as intermediate MOS. Then, only the third slot (*BD*) of the 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 executed 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.

3.2.3 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 projected image sizes affect the rasterization time. Combining the gain, cost and user preferences, 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 colors. 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.

3.3 Occlusion Culling

The occlusion culling consists of three parts. They are view frustum culling, overlap test with opacity map, and depth comparison with sparse depth map. The view frustum culling is the typical algorithm to be applied on the hierarchical bounding volume tree at first. It culls the nodes falling outside the view frustum, but not those hidden by occluders. In our occlusion culling algorithm an occludee is occluded if (*a*) the 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 testes, 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 are described.

3.3.1 Opacity Map

The opacity map is a two dimensional array of the down 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 lying inside the rectangular area from lower left corner up to the pixel. In Figure 5, 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 Figure 5c is equal to the number of 1's in the black bordered region in Figure 5b. The algorithm uses scan-line conversion to calculate the opacity values at each pixel. A row and a 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.

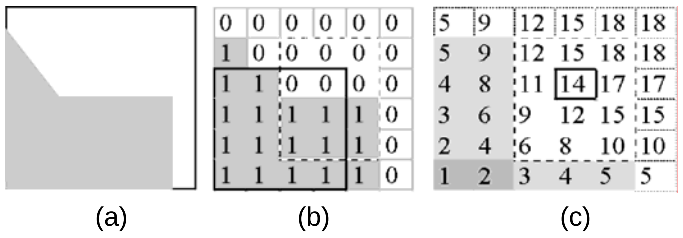


Figure 5: (a) The back buffer for rendering the occluders. The grey area is covered by occluders. (b) The bitmap of the occluders. (c) The opacity map, and the shading showing the usage of opacity function.

3.3.2 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 fully filled 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 5c 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,

$$\begin{aligned} & OPF(2, 5, 1, 5) \\ &= Op(2, 1) - Op(2, 5) - Op(5, 1) + Op(5, 5) \\ &= 2 - 9 - 5 + 18 \\ &= 6 \end{aligned}$$

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.

The projected image of the occludee can be obtained by either using the three-dimensional bounding box or the convex hull of the occludee, as a simplified representative. The computation cost of projected image of three-dimensional bounding box is much cheaper. However, in the case of rounded-shape model, the void space of bounding box is too large that an overlap test often fails. Therefore, we adapt the Quick Hull algorithm [4] to find the convex hull of the occludee at the pre-processing stage, and use its projected image for overlap test. This trade off can be adjusted according to the model.

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 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 extra occlusion culling in the case where the rendering capacity is much larger than the number of primitives falling in the view frustum.

3.3.3 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 6. 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 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.

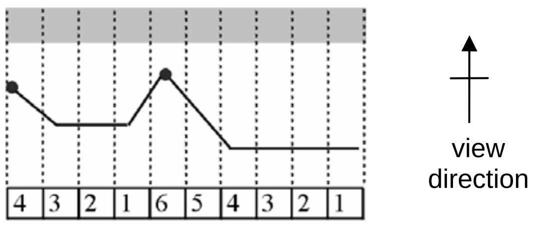


Figure 6: One row segment of the sparse depth map. It is the top view of an occludee (the grey rectangular box), and some occluders (the black lines). The two black dots mark the local farthest pixels (with the locally largest depth values) of this segment, called the peaks.

3.3.4 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 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 larger 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.

We have gone through the part of Occlusion Culling, and now proceed to the part of Non-Refractive Transparency Rendering. Later, the experimental results and application performances will be given for both parts.

4 Non-Refractive Transparency Rendering

In this chapter, we describe the overall procedure for applying screen door transparency at sub-pixel level. Afterwards, we study a precise mask generation, called pixel tree mask method. Since the pixel mask affects the accuracy of resultant opacity, and it is regarded as a core part of screen door transparency. We also present a new mask generation method, called tabular pixel mask, that considers the pixel distribution with other depth neighbors. We would show their computational complexity and accuracy in the later part.

4.1 Overview

To recall the basic of screen door transparency, it generates a set of two dimensional masks that represent the different levels of opacity. The pixels of the mask have only two states, which determine whether they are visible or not. A highly transparent object, the corresponding mask contains fewer filled pixels. Using the masks, we will render the transparent objects with masked rasterization. If there are two or more transparent objects, we stack their masks together one by one with depth comparison. Therefore, some pixels will be covered, but some are still visible finally. If the mask size is infinite, the portions of visible pixels from different objects should be the same as alpha blending. In fact, the pixels within the mask are sub-pixels of the whole frame buffer; however, because we focus on the mask generation, we simply call them "pixels", which are used in previous literatures.

To apply screen door transparency at the sub-pixel level, basically, we use a larger conventional frame buffer with magnification equals to the size of pixel mask. We first render all opaque objects. Afterwards, We generate a pixel mask for a transparent object, which is filled with the number of pixel as the same portion as its alpha value. Using a stencil buffer, we render the transparent object with the masked rasterization. We repeat to render all the remaining transparent objects. At last, we re-sample the image back to its original size. Figure 7 shows the process flow of screen door transparency rendering.

If the hardware capability is allowed, we can simply apply the pixel mask into sub-pixel buffer. In our case, the platform is aimed for PC compatible; we have to use the alternative as mentioned above, with some loss of performance. However, it does not affect our later analysis. In our implementation, we use 8×8 mask size for a practical application, and 16×16 and 32×32 for experimental tests. In the practical application, we provide a 96×96 region for screen door transparency rendering, as it is limited by the size of hardware frame buffer.

The performance of screen door transparency depends on the accuracy of higher order opacity obtained from the stacked masks, which requires a careful computation of pixel distribution among the masks. Therefore, our research is focused on the precise mask generation for a small mask and tens of transparent objects.

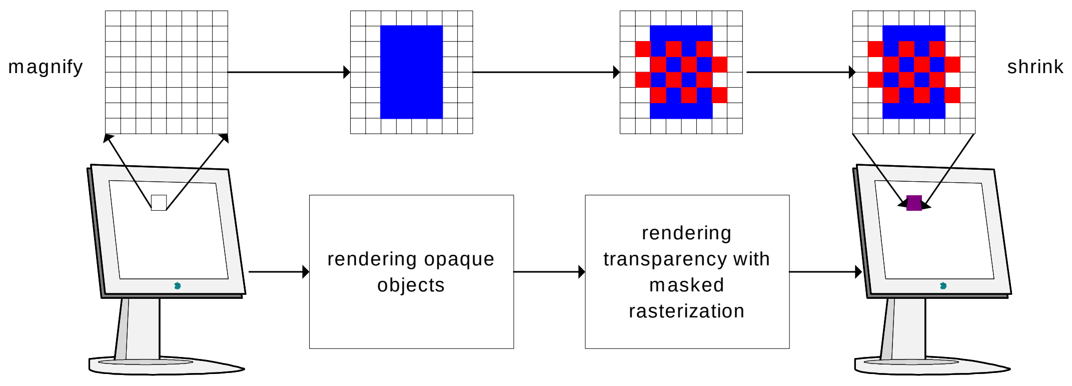


Figure 7: The process flow of screen door transparency rendering. An enlarged frame buffer is used, and all opaque objects are rendered first. Then, transparent objects are rendered with masked rasterization. Finally, the frame buffer is shrunk back to its original size for visual output.

4.2 Precise Mask Generation

The major advantage of screen door transparency is order invariant. The accuracy depends on the sizes of mask and the filled pixel distribution. Usually, white noise mask is suggested for mask generation, but it does not guarantee the accuracy, and acts as a rough approximation instead. To find a set of precise masks of different opacities, besides an exhaustive search of all possible masks, there is a binary tree approach, called pixel tree mask [26]. We base on the idea of binary tree partitioning of mask, further furnish for the situation of 8×8 mask size and tens of transparent objects, with a comparable computational complexity.

4.2.1 Binary Tree Approach – Pixel Tree Mask

In [26], a precise mask generation, called pixel tree mask method, is presented. The algorithm makes use of a binary tree structure to partition the mask into sub-regions hierarchically, so that the overlapped portion of any two masks should be the product of their alpha values. This is an important behavior for the correctness of higher opacities. To illustrate the idea, we give an example in Figure 8. Suppose there are three transparent primitives p_0, p_1 and p_2, with alpha values *0.5, 0.25, and 0.5* respectively. The rendering order is p_0, p_1 and p_2, while their depth order is also p_0, p_1 and p_2, and p_0 is the nearest. For the pixel tree mask, since we will binary partition the masks into sub-regions hierarchically, similar to building a binary tree structure, we define s_j^i be the mask region where it is *i*th node at *j*th level of the tree. The root of the tree is the whole mask s_0^0. We now generate the first mask by binary partition the whole mask s_0^0 into two sub-regions s_1^0 and s_1^1. s_1^0 is the region of *zero* (empty) and s_1^1 is the region of *one* (filled). The size of region of *one* is equal to the product of its alpha value and the number of pixels of the parent node. Then, we generate the second mask, by further partitioning s_1^0 and s_1^1 into s_2^0, s_2^1, s_2^2 and s_2^3

respectively. The process continues similarly for the remaining primitives, and is shown in Figure 8.

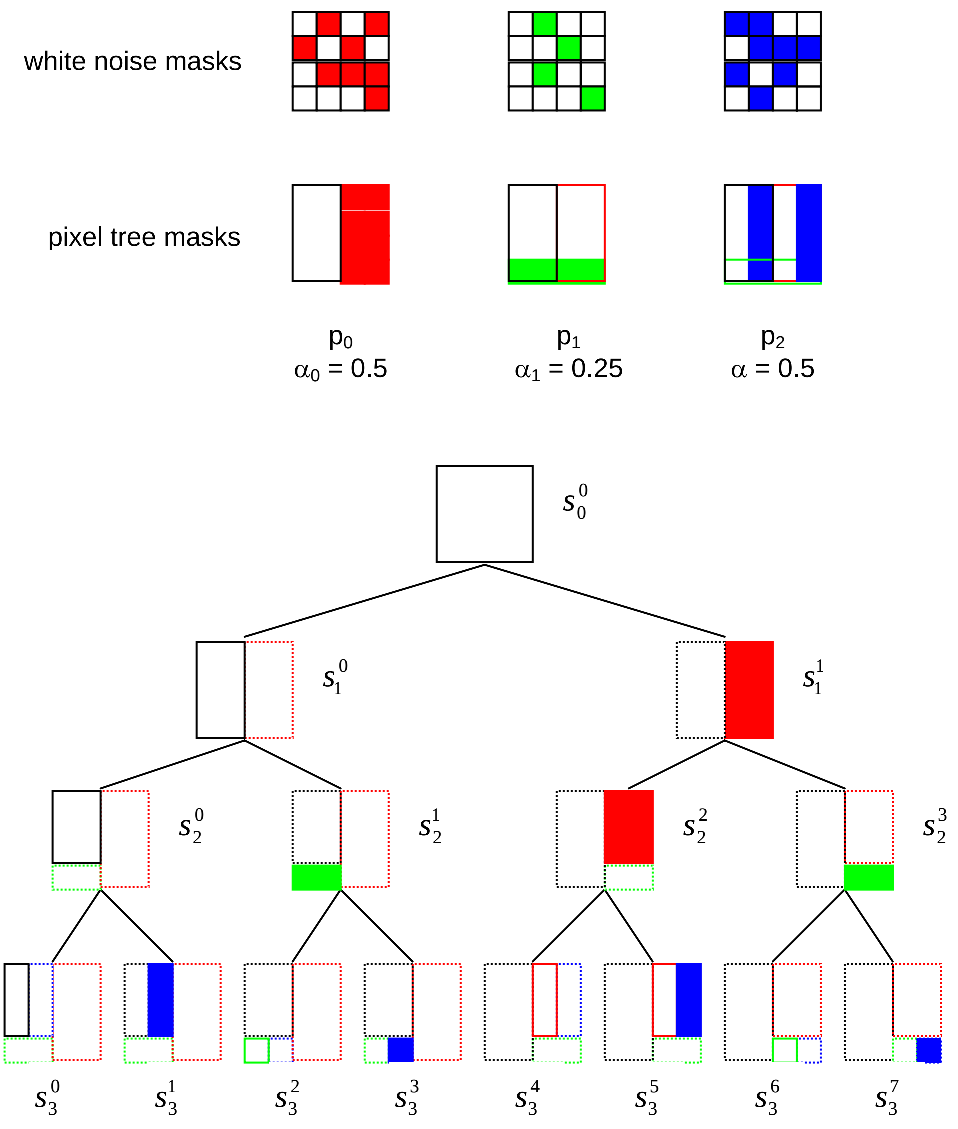
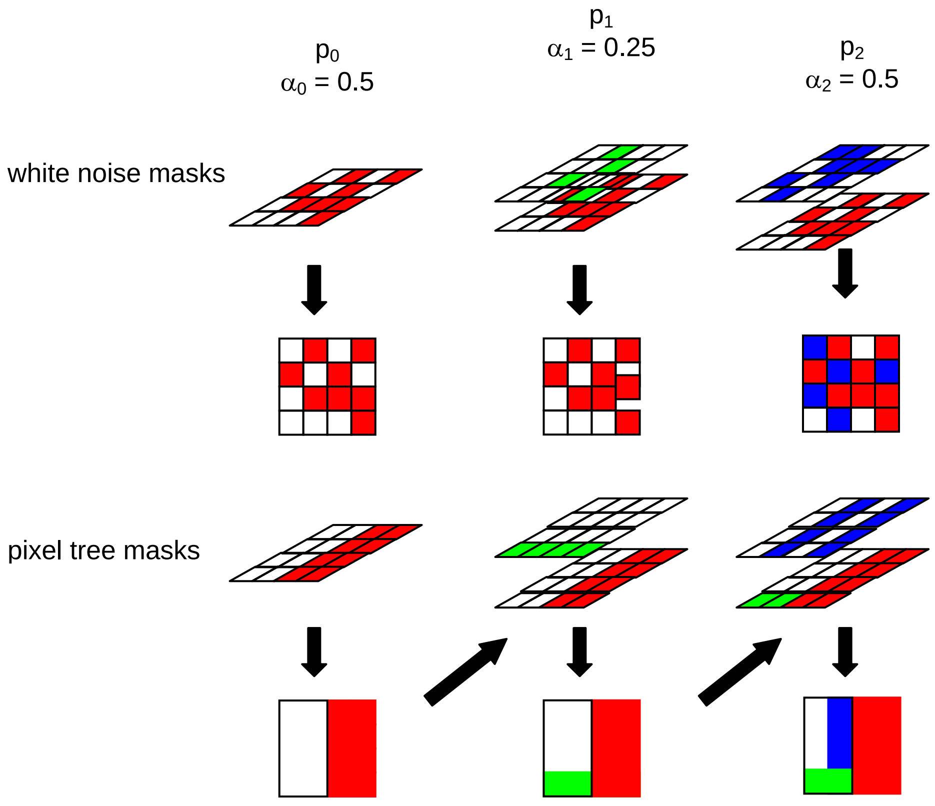


Figure 8: Examples of white noise masks and pixel tree masks. The binary tree structure of pixel tree masks are also shown.

4.2.2 Error Comparison

After the generation, we compare the pixel tree masks with white noise masks. We count the faulty pixels for both sets of masks, by stacking them together. The stacking order is the rendering order, without any assumption on the depth order. To stack one mask onto another, we also carry out the depth comparison each pixel, like conventional z-buffering. The number of faulty pixel is the difference between the actual number of visible pixels after stacking and the number of pixels by multiplying their alpha values and mask size together. In Figure 9, we can see that the white noise masks give one or more faulty pixels at the second and higher opacities, while the pixel tree masks do not.

Obviously, it is because the pixel tree mask method distributes the filled pixels with careful consideration. As the mask is small, the error of the white noise masks is significant. On the other hand, if the mask size is infinite, we can achieve the exact solution as well. Of course, it is not practical for infinite mask size, or even 32×32 mask size.



	p_0, $\alpha_0 = 0.5$	p_1, $\alpha_1 = 0.25$	p_2, $\alpha_2 = 0.5$
exact no. of filled pixels	8	2	3
no. of filled pixels of white noise masks	8	0	5
no. of filled pixels of pixel tree masks	8	2	3

Figure 9: Comparison of white noise masks and pixel tree masks.

The mask size is discrete, so it introduces a quantization error from mapping a continuous alpha value to a discrete number of filled pixels. Moreover, the pixel tree mask has another source of error. When the tree grows, the sub-regions at the leave nodes become smaller; they may have a few of pixels, such as one or two pixels only. These sub-regions cannot afford further partitioning. We have to "round off" the number of pixels within these sub-regions, and compensate the "round off" from their peer sub-regions. This gives an intra-quantization error. In order to solve the intra-quantization error, we try to sequence the splitting order of leave nodes in a randomized manner, so that hopefully, the intra-quantization error will not be accumulated into a single branch of tree vertically. Mostly, the intra-quantization error should be settled with a proper randomized splitting sequence. However, if we share the intra-quantization error from the peer nodes randomly, we have chance for putting some pixels from the region of *one* to the region of *zero*, not at the leave level, but at the ancestor level. It still violates the binary partition with its ancestors. In Figure 10, we have three masks, s_1, s_2 and s_3. They have alpha values of *0.5*, *0.25* and *0.25* respectively. For simplicity, we split the nodes in the order as the same as their indexes. There is no intra-quantization error at the first two masks. At the third mask, a 0.5 pixel error is borrowed from s_3^3 and s_3^7 to s_3^1 and s_3^5 correspondingly. It violates the binary partition of s_2, as it takes one pixel form the region of *one* to the region of *zero*. Sometimes, if the tree has sixteen levels, the new mask may violate its ancestors up to ten higher levels.

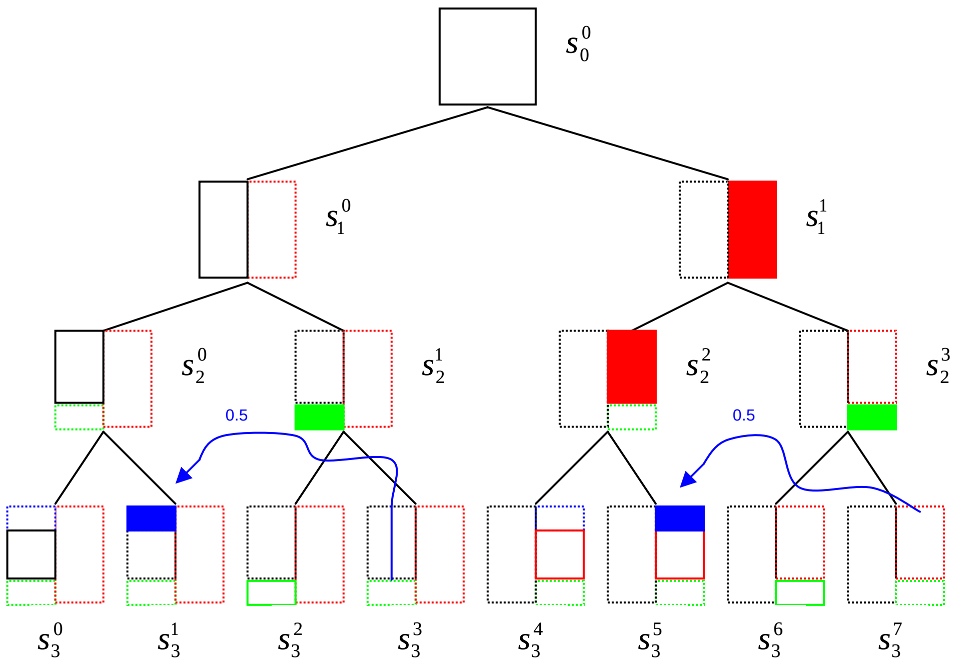


Figure 10: A violation of binary partition to ancestors by an intra-quantization error.

If an 8×8 mask is used, and the tree grows up to 16 levels, the violation of binary partition is significant. Though the top six levels in the tree are able to skip the violation, but they may not be the first six objects in depth order. Therefore, all the lower levels that suffer from violation introduce faulty pixels to the final output. It leads to a limitation of the number of transparent objects or we simply call it mask capacity, that equals to $log_2 n$, where n is the mask size.

4.2.3 Computational Complexity

A disadvantage of pixel tree mask or binary partition approach is the computation complexity, $O(2^m)$, where m is the number of transparent objects. For an 8×8 mask, if there are 8 transparent objects, we will have 256 leave nodes. It is a redundant overhead, as we have 64 pixels only and thus more than half of them are null nodes. Another disadvantage is pixel tree mask method requires knowing the alpha values of all transparent objects first. Otherwise, if we want to generate a new mask with a built tree, we have to transverse the whole tree from the root again.

4.2.4 Feasibility

The pixel tree mask method gives a precise stacked mask output, however, the computational complexity and the limited number of transparent objects make it improper for an interactive application. In order to give a practical usage of screen door transparency, the algorithm should use a small mask, have a low computational complexity, allow an unlimited number of transparent objects and give a precise stacked mask output.

4.3 Tabular Pixel Mask Generation

To achieve the requirement of practical screen door transparency, we propose a tabular pixel mask generation, that is based on the pixel tree mask generation. The tabular pixel mask generation is pin pointing to the case of small mask (8×8) and many transparent objects (≥ 6). In this situation, the first target is to reduce the storage requirement and computational complexity. As if the binary tree of pixel tree mask (8×8) has more than six levels, some null nodes appear. Moreover, when the tree grows to sixteen levels, there are at least 65472 ($2^{16} - 64$) null nodes. It implies an excess storage need and redundant computational overhead. The second aim is to minimize the occurrence of violation when there are more than six transparent objects. This requires a novel method for sequencing the order of splitting sub-regions. In this section, we describe the details of tabular pixel mask generation, including: structure of mask table, the process of mask generation and the method of sequencing the splitting order.

4.3.1 Structure of Mask Table

The structure of mask table is simple; it has a number of row entries, two row indexes and an accumulated error. A row entry is similar to a node of binary tree structure of the pixel tree mask, and represents a sub-region of the whole mask. It includes a list of pixels that are inside this sub-region, a count of the pixels and a boolean value that indicates whether this sub-region is filled or not. We refer the row entry with a filled region as *filled* *row entry*, and another is *empty row entry* in contract. Every row entry has at least a pixel, which is the main difference comparing with the content of a node of pixel tree mask. The two row indexes, named *start* and *end*, point to the first and the last previous spawned row entries. The accumulated error is a floating point variable, that is used for mask generation.

The structure of a mask table is shown in Figure 11.

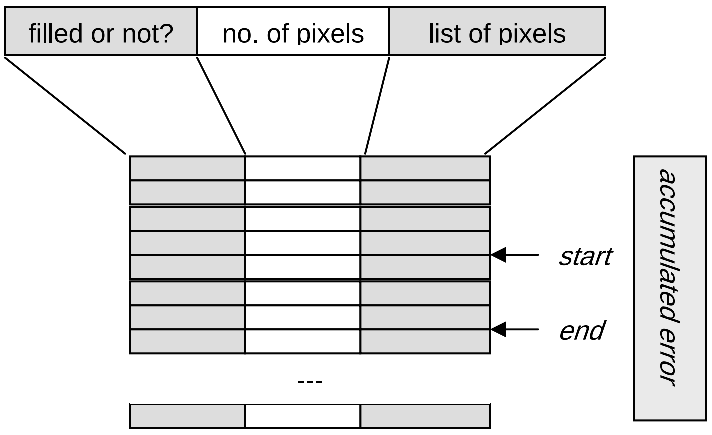


Figure 11: Structure of mask table.

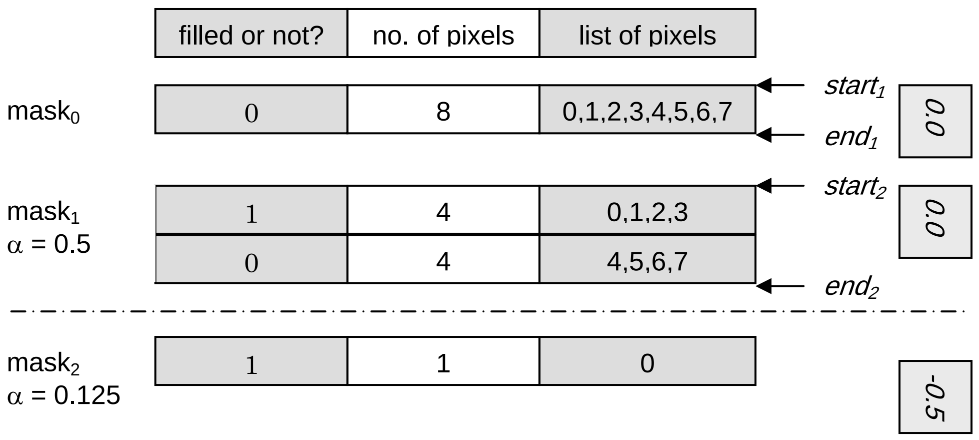
4.3.2 Mask Generation by Table Rolling

Initially, the mask table contains a row entry only at the index position *0*. This row entry represents the whole mask. It contains all pixels of the whole mask in its pixel list, and has a boolean value of false, indicating it is empty mask. The *start* row index point to position *0*, while the end row index point to position *1*.

The process of mask generation is by spawning those row entries within the *start* and *end* row indexes. Normally, each row entry will spawn two new row entries at the bottom of the mask table. One new row entry is for filled sub-region, taking an amount of pixels equals to the alpha value times the pixel count of its parent. Another child is for empty sub-regions, taking the remaining pixels from the parent. If any of these new row entries have zero pixel count, this row entry is abandoned, and the next new row entry is inserted here instead. After these row entries are processed, the *start* and end *row* indexes are updated to surround these new row entries.

As mentioned before, partitioning a sub-region introduces an intra-quantization error at the later stage. It is because the new row entry takes a rounded number of pixels from the parent. In order to compensate this error, we use a variable for accumulating the error. At each new mask is going to be generated, we set the accumulated error to zero. Then, every new row entries take the original rounded amount plus the accumulated error for splitting. Afterwards, the rounded-off amount is further set to the accumulated error for the next row entry.

An example is shown in Figure 12. The table has only one row entry as $mask_0$ initially. We assume the mask has eight pixels. The alpha value of $mask_1$ is *0.5*, so the list of pixel from $mask_0$ is simply divided into two rows without an accumulated error. The $start_1$ and end_1 row indexes are updated to $start_2$ and end_2 row indexes respectively. The alpha value of $mask_2$ is *0.125*. When splitting the first row entry from $mask_1$, we have an accumulated error -0.5 (*i.e.* $4 \times 0.125 - 1$). Then, we set the number of pixels of the second row entry to *3.5* (the original number of pixels plus the accumulated error), therefore, this new row entry has *0* pixel (*i.e.* 3.5×0.125), and will be abandoned. The remaining *4* pixels are added to the next empty row entry. The generation of $mask_2$ is completed.



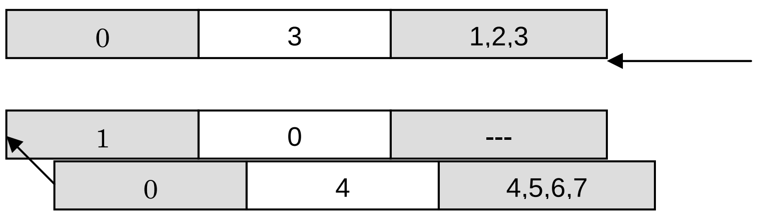


Figure 12: An example of tabular mask generation.

4.3.3 Neighborhood Error Compensation

The intra-quantization error compensation introduces the violation of binary partitioning. It gives a noticeable amount of faulty pixels when the number of transparent objects exceeds the capacity of the mask size (i.e. $log_2\ n$, where n is the mask size). The number of faulty pixels can be reduced by sequencing the splitting order with consideration of the depth neighborhood.

The faulty pixels are usually located at the masks of nearby objects. It is because the masks of far away objects are mostly covered, and have only a few of visible pixels at all. Moreover, if the nearby objects have already produced faulty pixels, the far away objects will inherent those errors as well. Therefore, if we prevent the violation for nearby objects' masks, we can improve the accuracy of overall opacity.

The idea of neighborhood compensation is that, we keep track of several nearby transparent objects. When we add a new mask, we sequence the splitting order of the new mask, in the way of sharing the intra-quantization error within the region of *one* and the region of *zero* separately, for the object with the highest priorities first. Now, we re-arrange the pixel list of splitting order of the region of *one* partially, so that it can share the intra-quantization error within the region of *one* of the object with the second highest priority. The operation is similar for the region of *zero*, and repeats for the lower priority. We will have fewer combinations of the splitting order for the lower priorities, as the pixel list of splitting orders becomes shorter. It means that the new mask may have violation to these low priority object masks inevitably at last.

Now, we would give an algorithmic description and an example. We first define RF^j & RE^j be the filled and empty row entries for the *j*th objects, where *j* is the index of the mask table; in this case, *j* is also the rendering order. Both RF^j and RE^j contain the list of pixels, which represent their mask coverage regions, but without the ordering among these pixels. Then, we define F^i and E^i be the partial pixel list of splitting order of the new mask, with considering for the object up to the *i*th priority, or simply depth value in this case. It means that F^0 is the pixel list of splitting order, sharing intra-quantization error within the region of *one* of the *0*th priority object. In addition, F^1 is the partial pixel list of splitting order, which shares intra-quantization error among the regions of *one* of both the *0*th and *1*th priority object. In fact, F^{m-1} is the intersection of *RF* from *0*th to *(m-1)*th priority.. The definition of E^i is similar. Therefore, when we add a new mask, we find out these F^m and E^m to the certain priorities, say 6. Afterwards, we collect the pixels from F^m to F^0, with discarding the duplicated ones, which is the final splitting order for this new mask. We apply the same procedure to E^m to E^0 as well.

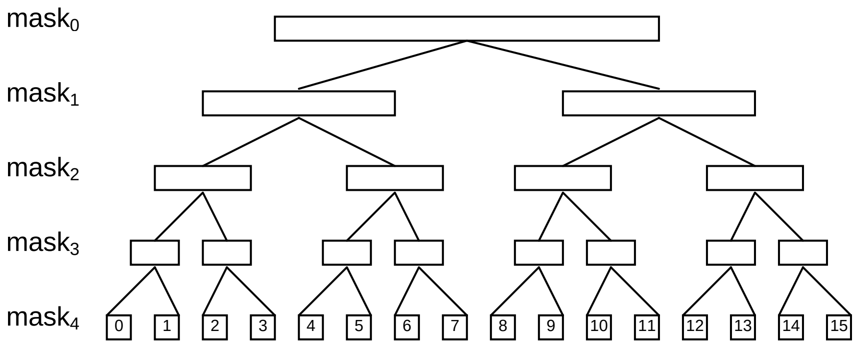


Figure 13: An example of neighborhood error compensation.

In Figure 13, we assume that there is no abandoned row entry while splitting the $mask_{0\sim4}$ for simplicity. The indexes of row entries are shown in $mask_4$. We keep track of three nearest objects, and in this case they are $mask_2$, $mask_1$ and $mask_4$ in the descending order of their depth. We find RF^2, RF^1, RF^4 and RE^2, RE^1, RE^4 shown in Table 1. Initially, F^0 is equal to RF^2, containing { *4, 5, 6, 7, 12, 13, 14, 15* }. To consider the second priority as well, we calculate the intersection of RF^2 and RF^1, that is { *12, 13, 14, 15* }, which is stored in F^1. For the third priority, F^2 is the intersection of RF^2, RF^1 and RF^4 or { *13, 15* }. The generation of $E^{0\sim2}$ are similar and the resultant $F^{0\sim2}$ ad $E^{0\sim2}$ are shown below. Finally, the splitting order of the new mask is { *13, 15, 12, 14, 4, 5, 6, 7, 0, 2, 1, 3, 8, 9, 10, 11* }.

filled row entry	list of pixels	empty row entry	list of pixels
RF^2	4,5,6,7,12,13,14,15	RE^2	0,1,2,3,8,9,10,11
RF^1	8,9,10,11,12,13,14,15	RE^1	0,1,2,3,4,5,6,7
RF^4	1,3,5,7,9,11,13,15	RE^4	0,2,4,6,8,10,12,14

splitting order for filled row entry	list of pixels	splitting order for empty row entry	list of pixels
F^0	4,5,6,7,12,13,14,15	E^0	0,1,2,3,8,9,10,11
F^1	12,13,14,15	E^1	0,1,2,3
F^2	13,15	E^2	0,2

Table 1: Lists of pixels for filled and empty row entries and their corresponding splitting orders.

4.3.4 Computational Complexity

Assume we use a mask of size n, and there are m transparent objects. For the mth mask generation, if $2^m \leq n$, the computational complexity is $O(2^m)$; otherwise, it is $O(n)$. For the computational complexity of generation of m masks, if $2^m \leq n$, it is $O(2^{m+1}-1)$, else it is $O(\ n*(m - \log_2 n + 2) -1\ )$ or simply $O(m*n)$.

To compare the computational complexities of pixel tree mask and tabular pixel mask method, we assume that there are m transparent objects, and each mask has size n again. Then we count for the total numbers of atomic operations for generating m masks of the two methods. We define the atomic operations of pixel tree and tabular pixel mask methods to be the splitting of nodes and row entries respectively. We calculate the theoretical computation complexity for four different cases and the result is shown in Table 2. We clarify that, since the actual computational times of atomic operations are different, therefore, the figures in Table 2 can only be used as supplementary reference.

In the case 2 and 4, we observe that the pixel tree and tabular pixel mask generation have the same computational complexity. It is because in these cases the number of transparent objects is still within the capacity of the mask size. In the test case 1 and 3, the number of transparent objects exceeds the capacity of mask size, there are many null nodes produced for the pixel tree mask generation, and thus causes a large redundant overhead, while the tabular pixel mask method does not.

	Pixel tree mask	Tabular pixel mask

Computational complexity	$2^{m+1} - 1$	If $n \leq 2^m$, => 2^{m+1} - 1;
		else => $n \times (m - \log_2 n + 2) - 1$

1) $n = 1024$, $m = 24$	33554431	16383
2) $n = 1024$, $m = 10$	2047	2047
3) $n = 64$, $m = 16$	131071	767
4) $n = 64$, $m = 6$	127	127

Table 2: Computational complexities of white noise, pixel tree and tabular pixel mask generations.

With the same assumption as above, and also let R be the size of frame buffer. The computational complexities of applying masks are, $O(m)$ for geometric calculation, $O(R*n)$ for rasterization, and $O(R*n)$ for depth comparison. However, if hardware sub-pixel buffer is big enough, we can ideally achieve $O(R)$ for time complexity of rasterization though we have $O(R*n)$ for space complexity.

The main bottleneck of the tabular pixel mask algorithm is the part of mask generation, which has computational complexity of $O(m*n)$, if we generate all the masks again and again at each pixel. However, we usually apply this method as a fast approximation of rendering a lot of transparent objects. Thus, we accept a trade off between the visual accuracy and time complexity. In order to reduce the time complexity, we only re-generate the masks for the divided regions of the whole frame buffer, instead of generating all the masks again at each pixel. It is because Neighborhood Error Compensation does not require a full depth sorting, only require to know which certain number of objects are the nearest, and they are not sorted at all even. This property is able to afford a few depth changes of objects within the divided region, without any noticeable error. The mask generation scheme may depend on the number of transparent objects and the mask size.

Now, we compare the overall complexity of tabular pixel mask generation with the conventional A-buffer method [5], which has time complexity of $O(R*m*\log_2 m)$ and space complexity of $O(R*m)$. If we carry out the pixel mask generation per pixel, we have a time complexity of $O(R*m*n)$ and space complexity of $O(R*n)$. Assume n is smaller than m, pixel mask generation will use less space than A-buffer method, but the time complexity of pixel mask generation will be far behind. For example, if we use an 8×8 mask, n is 64, the pixel mask generation will be faster only when $m > 2^{64}$. However, if we allow generating the pixel masks for the divided regions sized of DR, then, pixel mask generation will have a reduced time complexity of $O(R/DR*m*n)$. In this case, the pixel mask generation will be faster when $m > 2^{n/DR}$. So if we use an 8×8 mask again, and assume DR is a 4×4 region, then, the pixel mask generation will be faster when $m > 2^4$. This performance is more practical. The Table 3 shows the computational complexity comparison.

	A-buffer algorithm	Tabular pixel mask generation (pixel)	Tabular pixel mask generation (divided region)
Time complexity	$O(R*m*log_2 m)$	$O(R*m*n)$	$O(R/DR*m*n)$
Space complexity	$O(R*m)$	$O(R*n)$	$O(R*n)$

Table 3: Comparison of computational complexities of the A-buffer algorithm and tabular pixel mask generation for pixel and divided region. *R* stands for the frame buffer size, *DR* is the size of a divided region, *m* is the number of transparent objects and *n* is the mask size.

5 Experiments and Applications

We have implemented the occlusion culling and non-refractive transparency rendering for experimental testing and practical application. In order to analyze and illustrate the performance of these algorithms, we use a simple walkthrough system as a test platform for the MOS and the occlusion culling algorithms. Moreover, we apply the occlusion culling algorithm to the application "Virtual Brain". However, as Virtual Brain setup a dynamic environment, so we bypass the implementation of MOS algorithm. For the tabular pixel mask generation, we use a set of randomized test data for comparison first, and also give a practical result from Virtual Brain. Table 4 shows the summary of the experiments and applications for the three algorithms.

	Experiments	Applications
Occlusion culling		
MOS algorithm	Walkthrough system – outdoor city scene	--
Occlusion culling	Walkthrough system – outdoor city scene	Virtual Brain
Non-refractive transparency rendering		
Tabular Pixel Mask Generation	Random test data	Virtual Brain

Table 4: Summary of the experiments and applications.

5.1 Experimental Result

5.1.1 Occlusion Culling on A Walkthrough System

We have implemented the MOS and occlusion culling algorithms on a simple walkthrough system, which uses OpenGL and runs on an SGI Indigo2 Max IMPACT workstation with R10000 CPU (195MHz) 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 source and no texture. A birdeye view of the test model is shown in Figure 26.

5.1.1.1MOS 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 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.

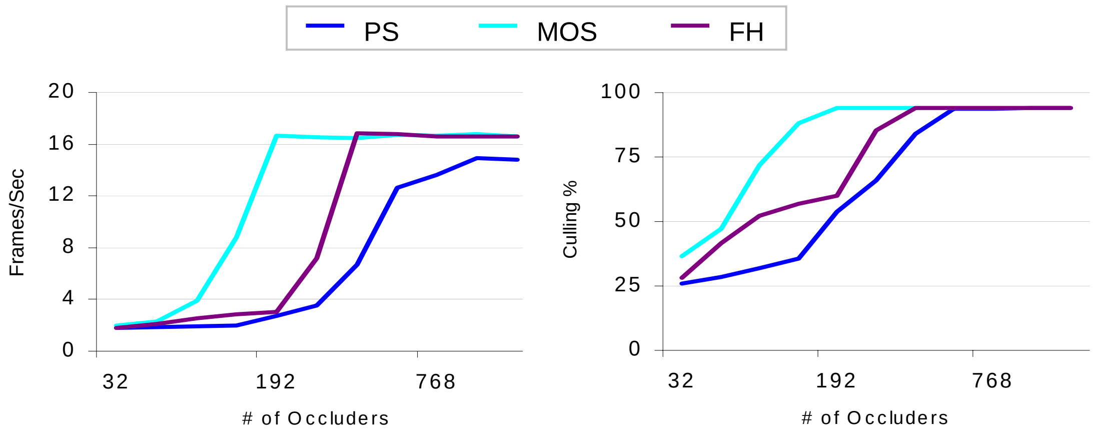
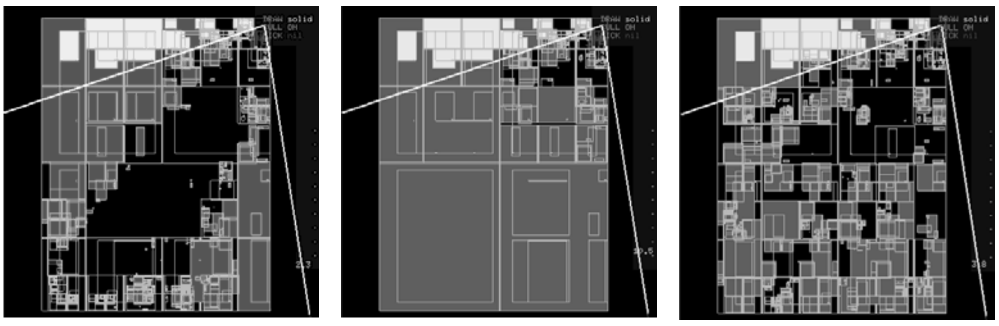


Figure 14: Frame rate and culling percentage of different occluder selection methods. *PS* stands for the criterion of projected size, *FH* stands for the criterion of first-hit.

Figure 14 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 15, 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 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.



(a) (b) (c)

Figure 15: 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.

5.1.1.2Occlusion 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.

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 16. It has the average frame rate of 17.7, 19.4 and 28.8 for the three routes, respectively.

According to Figure 16, 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 17 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 gradually 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 18. 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 the 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.

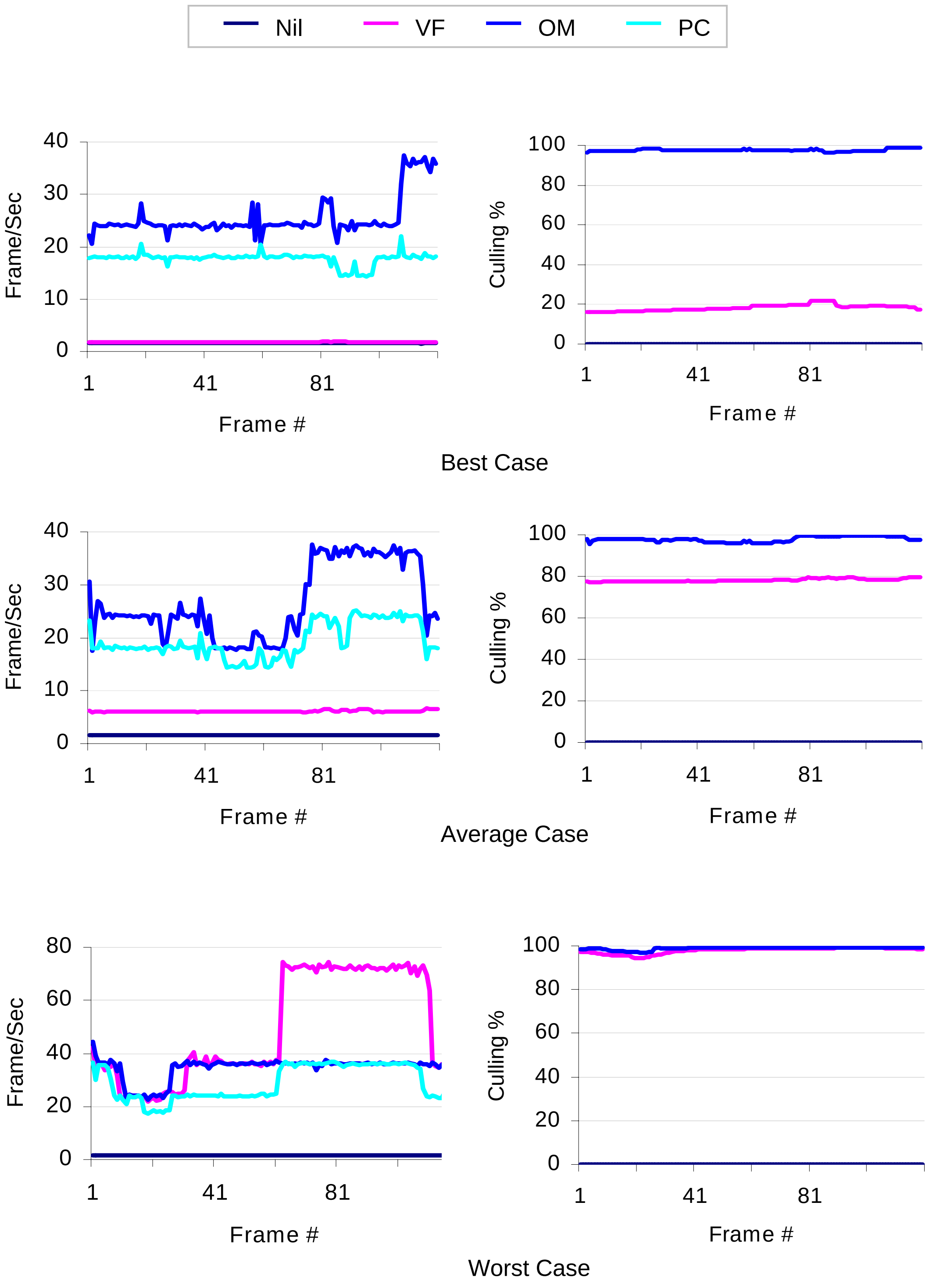


Figure 16: Performances of occlusion culling with different routes. *Nil* represents that no culling is applied. *VF* represents that view frustum culling is applied. *OM* represents that occlusion culling with opacity map and sparse depth map is applied. *PC* means occlusion culling with pixel-wise comparison.

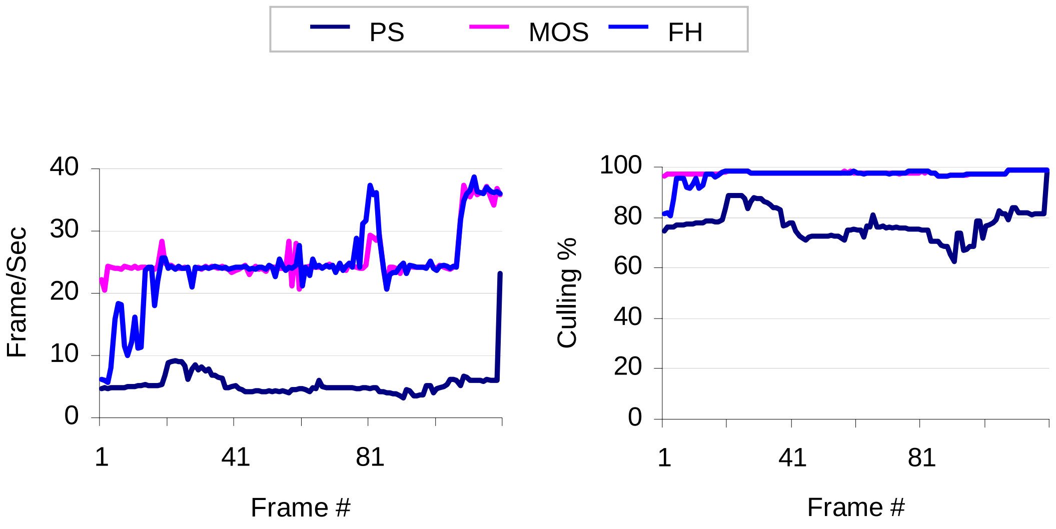


Figure 17: Performances of occlusion culling with different occluder selection criteria for the best case route. *PS*, *MOS* and *FH* represent the criteria of projected-size, minimum occluder set and first-hit, respectively.

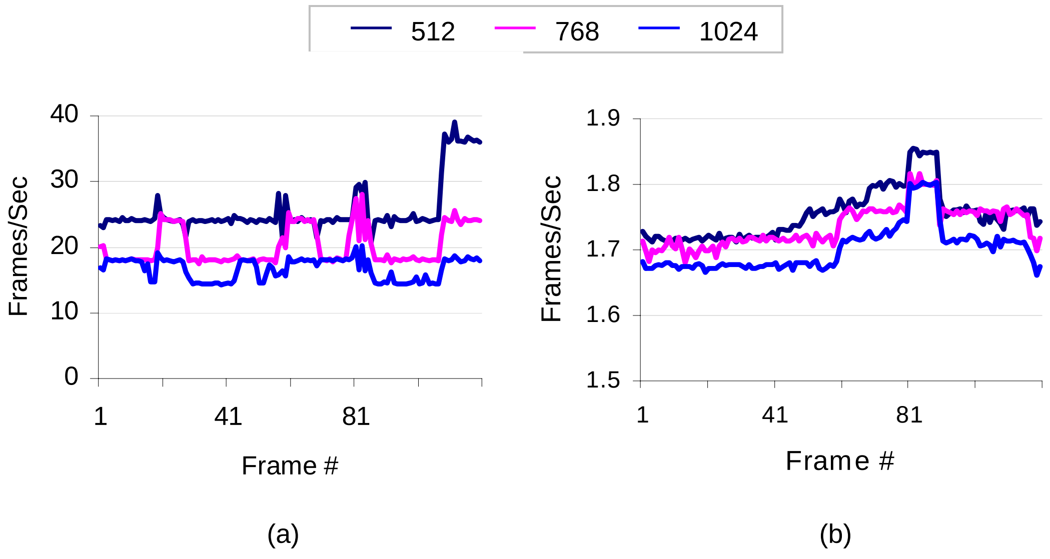
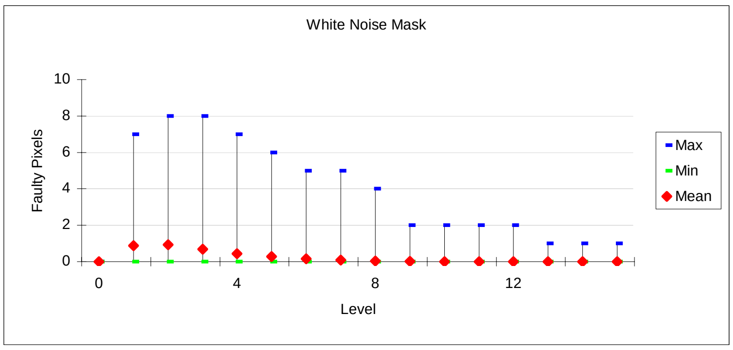
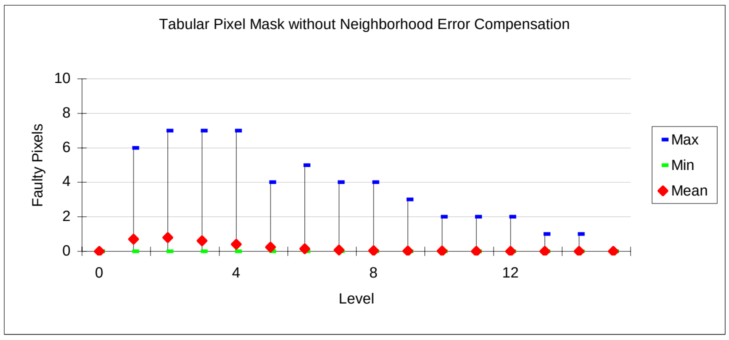


Figure 18: (a) Performances of occlusion culling using opacity map and MOS algorithm at different resolutions, 512×512, 768×768 and 1024×1024, (b) The result of view frustum.

5.1.2 Tabular Pixel Mask Generation with Random Test Data

Two test cases are conducted, for comparing the accuracies of white noise mask, tabular pixel mask with and without neighborhood error compensation. The first test case uses 16×16 sized masks for 16 transparent objects, and the second test case uses 32×32 sized mask for 32 transparent objects. In both cases, each object has a random opacity or alpha value, and a depth. They are generated in a randomized order, and we make no assumption on the depth or rendering order. We stack the masks together one by one, we records the number of faulty pixels at each level. The result of first test is shown in Figure 19 and the second one is shown in Figure 20.





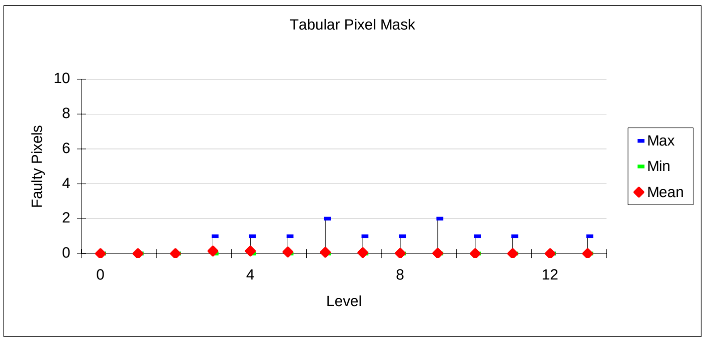


Figure 19: Results of white noise mask, tabular pixel mask with and without neighborhood error compensation for 16×16 sized mask.

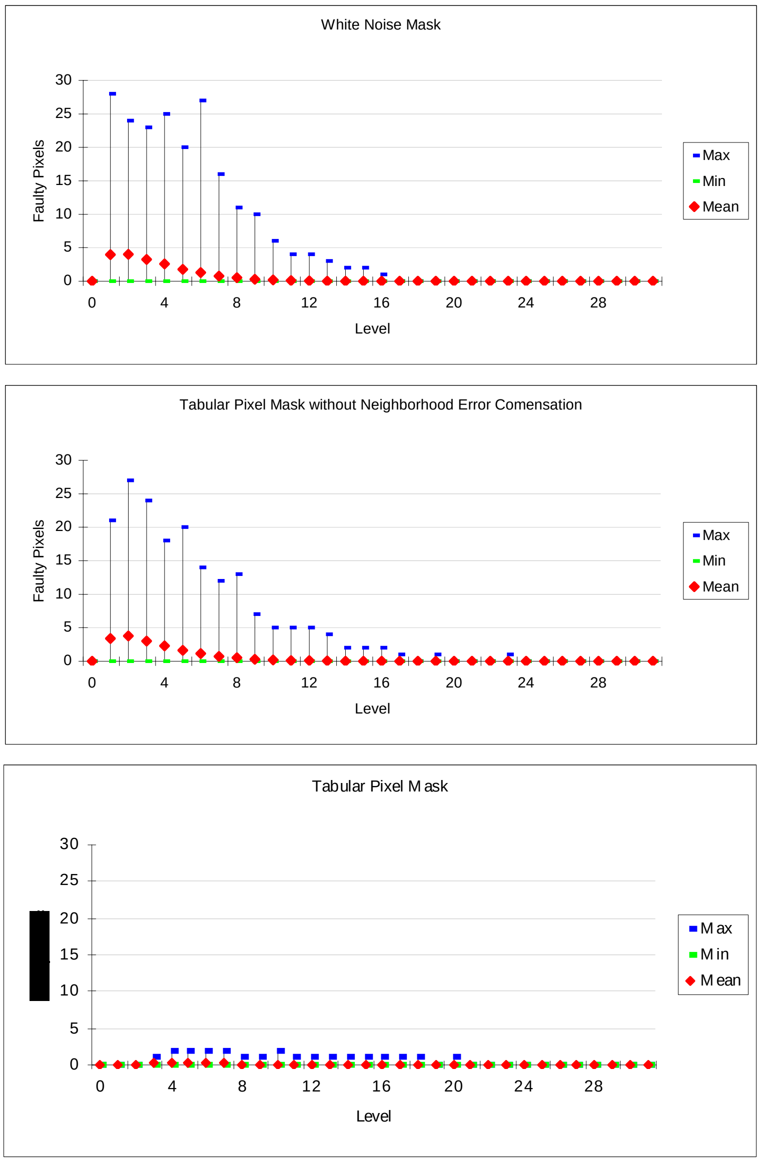


Figure 20: Results of white noise mask, tabular pixel mask with and without neighborhood error compensation for 32×32 sized mask.

The minimum numbers of faulty pixels are always zero at each level for the three methods. However, for the maximum and average number of faulty pixels, tabular pixel mask gives a better result, achieving only 1 to 2 faulty pixels. While the other two masks have similar behaviors, and give about 7 faulty pixels for 16×16 sized mask and 27 faulty pixels for 32×32 sized mask in the worst cases. It is because the mask size of white noise mask is not large enough to achieve the correct result by the ground of probability. About the tabular pixel mask without neighborhood error compensation, the number of transparent objects exceeds the capacity of mask, and causes serious violation of binary partitioning. On the other hand, the tabular pixel mask method with neighborhood error compensation has minimized the violation and provides a better result.

5.2 Application Performance – Virtual Brain

The occlusion culling algorithm and the tabular pixel mask generation is implemented in an application known as “Virtual Brain”. The platform of the application is a DELL Precision 410 PC with Intel Pentium II (400 MHz), 256 MB RAM, and Intergraph Intense 3D 3410GT display card. In this section, we demonstrate the performance of occlusion culling and visual output of non-refractive transparency by using tabular pixel mask generation.

5.2.1 Occlusion Culling

We have recorded the timing and culling percentage of 475 frames to show the performance of occlusion culling in the “Virtual Brain”. The camera path is simply rotating around the whole brain and skull, and several screen snapshots are captured as shown in Figure 21. As the organs of the brain and skull are expected to be movable, we use the projected size of primitive as the criteria of occluder selection, instead of the MOS algorithm. The view frustum always contains the whole model and view frustum culling does not help, so we only consider the cases of occlusion culling algorithm and brute force approach. In Figure 20, we draw the primitives that are culled in sharp colors, while the other ones are in transparent grey. It shows quite a lot of interior primitives are culled. According to the Figure 22, we achieve a maximum culling percentage of 78.6% and frame rate speedup of 3.1 while the total number of polygons is 207,372.

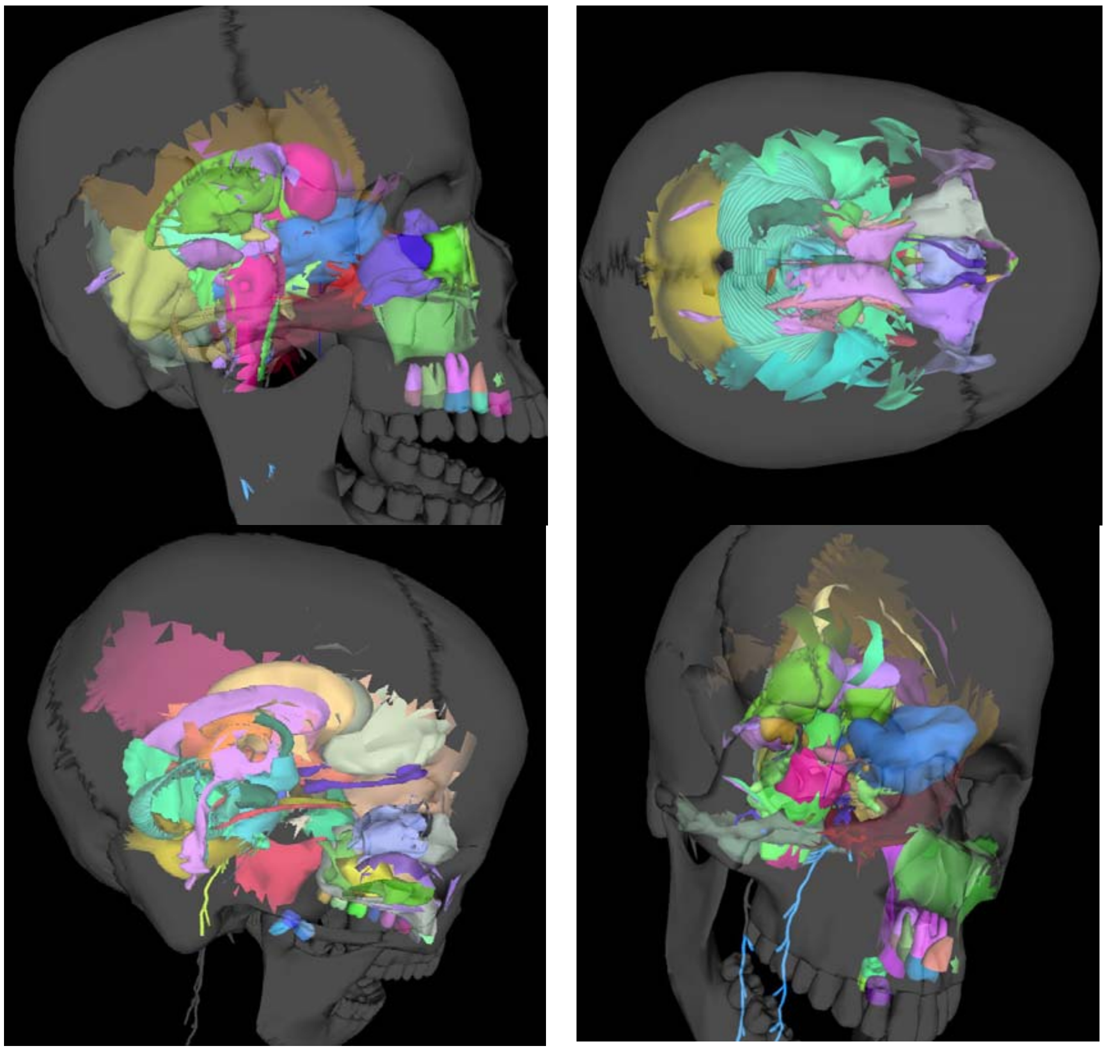


Figure 21: Screen snapshots of occlusion culling algorithm. The colorful interior primitives are detected to be invisible, and culled.

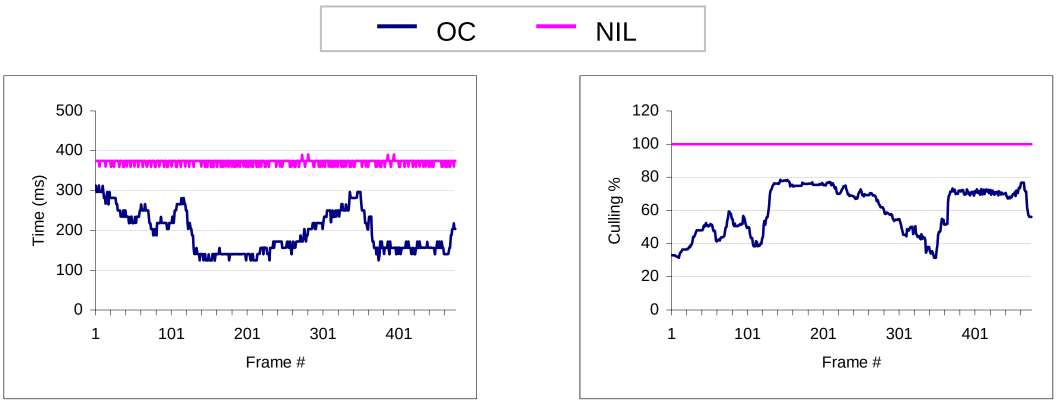


Figure 22: Performance of Occlusion Culling algorithm in "Virtual Brain". *OC* stands for Occlusion Culling while *NIL* stands for brute force approach.

5.2.2 Tabular Pixel Mask Generation

Firstly, we show two visual properties of screen door transparency rendering, they are order invariant and interior layer masking-off. Second, we demonstrate the visual outputs of transparency rendering using alpha blending, white noise mask, pixel tree mask and tabular pixel mask.

Figure 23 illustrates the order invariant property of screen door transparency. At the lower right part of both pictures, we have three organs; they are red, blue and green in color, following their depth order. The red and blue ones are transparent. However, they are rendered in the order of green, red and blue. Therefore, in the left picture, we apply the general alpha blending without visibility sorting, that cannot illustrate the blue organ through the red one. In the red rectangle of right picture, we can see that the blue organ is behind the red one and before the green one.

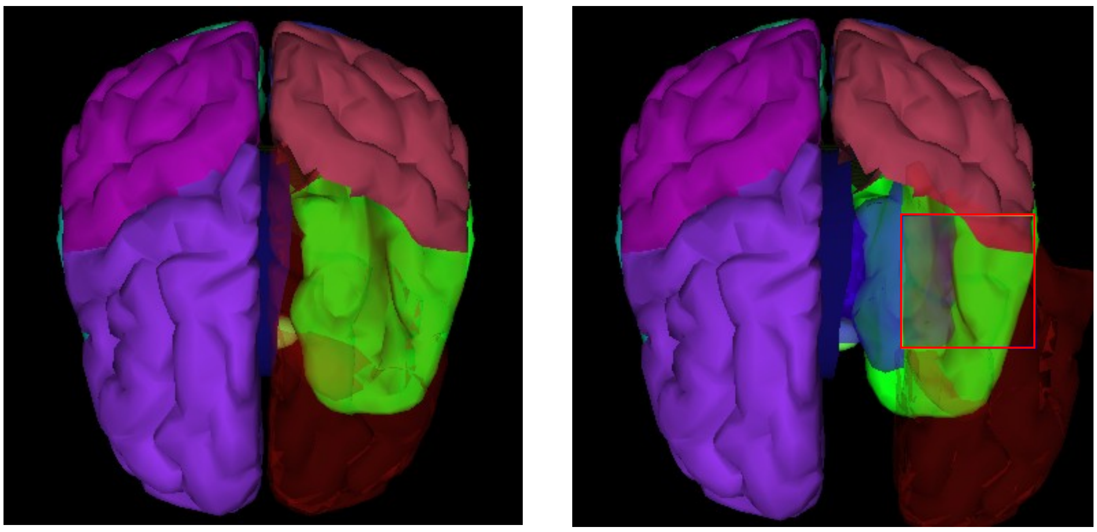


Figure 23: The left picture applies general alpha blending without visibility sorting, thus shows an incorrect opacity. The right picture applies screen door transparency and clearly shows that the blue organ is between the red and the green ones.

Figure 24 shows an additional advantage of screen door transparency. If a transparent object contains some interior primitives, which sometimes cause confusion of the overall shape and shading. For example, some interior primitives are shown in the lower right purple cerebrum of the left picture, but they are just confusing the visual appearance of the organ in clay yellow. Using pixel mask, we can have a visual output of the outermost layer of organs only, which is shown in the red rectangle of the right picture. It is because the same object is able to have the same mask, that will cover the interior primitives' mask pixels.

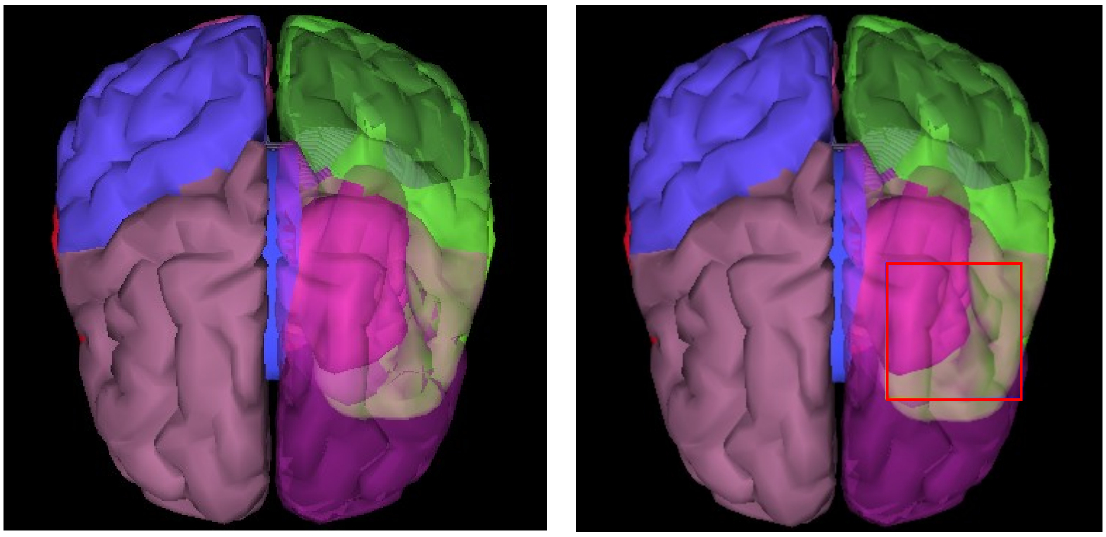
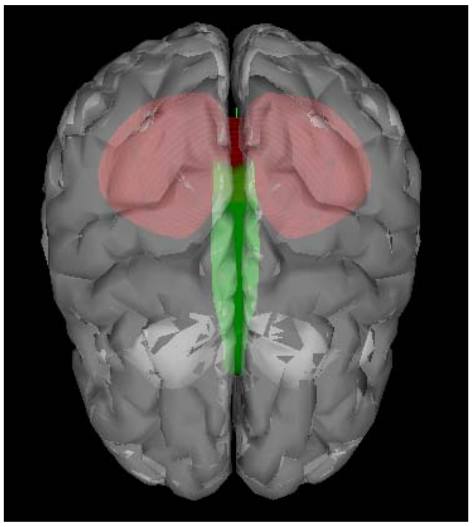
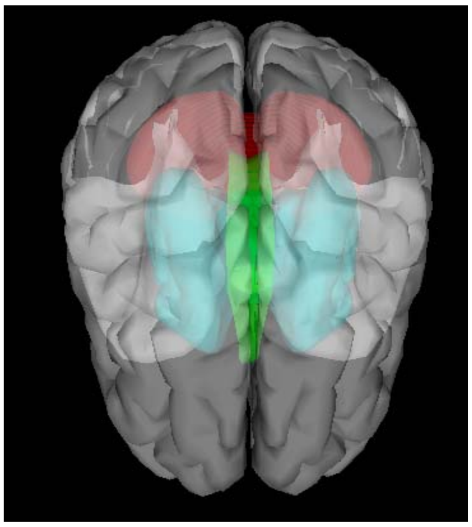
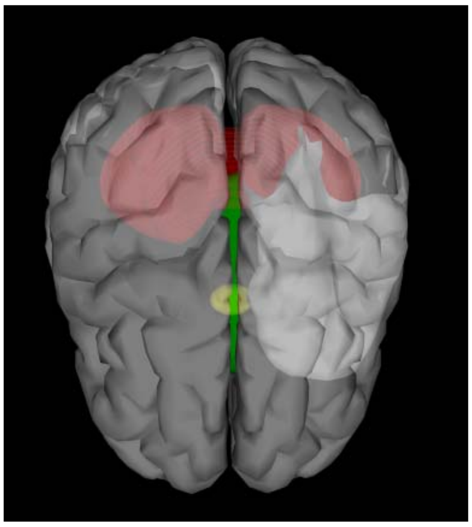


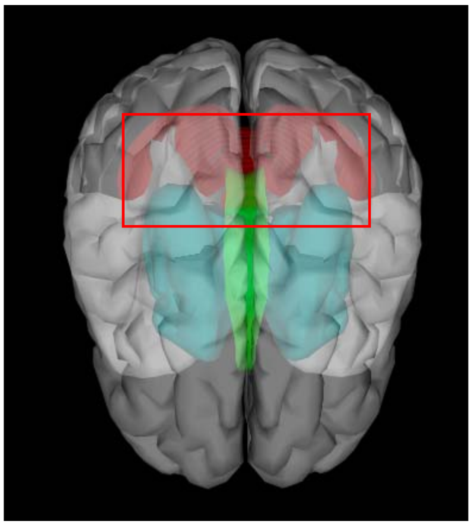
Figure 24: The left picture shows some interior primitives that give a confusing visual output, while the right one only shows the outermost layer of the organs, and provides a clear understanding.

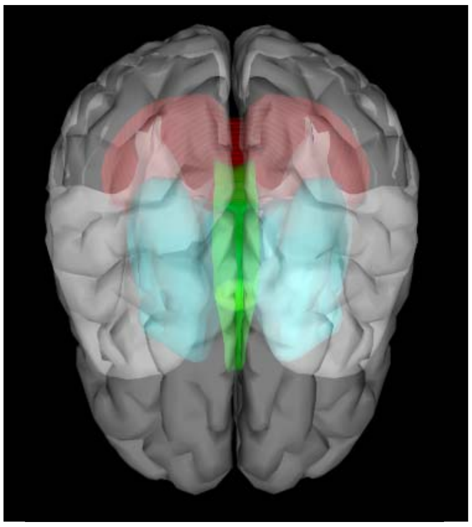
In Figure 25, we show the visual outputs of alpha blending with and without visibility sorting, white noise mask, pixel tree mask and tabular pixel mask. We render 12 transparent objects, with an 8×8 mask for those screen door transparency techniques. In Figure 25a, alpha blending without visibility sorting is applied, some organs are blended incorrectly, or even hidden wholly, since they are not rendered in a far-to-near order. In Figure 25b, we use alpha blending with visibility sorting, the picture shows the correct visual output. In Figure 25c and Figure 25d, we use white noise mask and pixel mask respectively. There are some parts are hidden, and we use the red line to highlight the incorrect portion. In Figure 25e, we use tabular pixel mask, and the visual output is accurate as alpha blending with visibility sorting.


(a) alpha blending without visibility sorting


(a) alpha blending with visibility sorting


(c) white noise mask


(d) pixel tree mask



(e) tabular pixel mask

Figure 25: Visual outputs of alpha blending with and without visibility sorting, white noise mask, pixel tree mask and tabular pixel mask.

6 Virtual Brain

Traditional neuroanatomy was taught by using cross-section slide of the brain, but in recent years many computer based applications have been developed. Through the use of computer graphics and virtual reality, the students of medicine can have an interactive visualization of three dimensional structure of human organs. With the shift in emphasis towards a medical curriculum which stresses more important on learning through the use of computers, the Department of Pathology recognizes the advantage of computerized learning toolkit for anatomy and pathology. Therefore, a project “Virtual Brain” co-operated by the Department of Pathology, and the Department of Computer Science and Information Systems is launched, in order to develop a learning product for medical students.

The developed system supports real-time visualization of the surface-based data model of human organs on affordable PC platform. The user can “walk through” the interior structure of human organs, and thus perceives a better understanding of the relationship of different organs. Moreover, the user can move, rotate, resize and zoom the organs, so as to recognize the geographical and anatomical location. The textual information of organ is further provided as a complete reference of medical knowledge.

As the floating-point computational power and 3D graphics API are limited on general PC, some advanced speedup techniques have been applied to achieve interactive frame rate and transparency effect. The development consists mainly of three parts: data modeling, speed-up technique and transparency rendering. The aim of data modeling is to unify the data file formats, correct the organ positions and orientations, and increase data efficiency. The source data comes in different file formats and coordinates system. We standardize their file formats, i.e. Virtual Brain Object, (*.vbo). This is a subset of the well known file format, WaveFront Object file (*.obj). Also, we modified the source data set so that all organs are in the right position and orientation. The overly tessellated surface data has been simplified to minimize the requirement of computational power. The number of polygons of resultant data set is about two hundred thousands. The application is speeded up by our occlusion culling algorithm. As an opaque skull usually occludes most of the interior tissues from a fixed view point, we can skip to render those culled primitives, in order to increase the frame rate. Transparency rendering is also important to visualization, as it can reveal the relationship between the outer shield and the inner tissues. The screen door transparency rendering using tabular pixel mask method is applied, since it does not require the visibility sorting., which is favorable to an interactive application. Though screen door transparency rendering is still under a pilot run and allows a small displaying region, its computational load is light enough for interactive visualization. If hardware supersampling is provided, its usage will be greatly increased.

In the coming phrase, the information cue, question module, and other usages of the existing infrastructure are the major areas for improvement. Our goal is to provide an interactive, user-friendly, and affordable visualization system for medical education. Some screen snapshots are shown in Figure 27.

7 Conclusion and Future Works

We have presented an occlusion culling algorithm using the minimum occluder set and opacity map, and a non-refractive transparency rendering method using the tabular pixel mask generation.

The occlusion culling 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.

The tabular pixel mask generation provides a fast, order invariant method for non-refractive transparency rendering. It is feasible to apply the screen door transparency at sub-pixel level, and gives a comparative accurate visual effect by avoiding the violation of binary partitioning among the depth neighborhood.

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 outdoor environment with a large number of visible primitives, we can apply impostors [33, 39] 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.

Also, about the screen door transparency rendering, we would break the limitation of frame buffer, by allowing a trade off from the mask size, shifting frame buffer requirement to main memory with frequent or incremental frame buffer reading, or using dedicated hardware with supersampling.

Furthermore, we would extend the functionality of the application "Virtual Brain", for example, adding the information cue, question module, and other usages of the existing infrastructure, so as to provide an interactive, user-friendly, and affordable visualization system for medical education.

9 Appendix

9.1 Chicago City Model

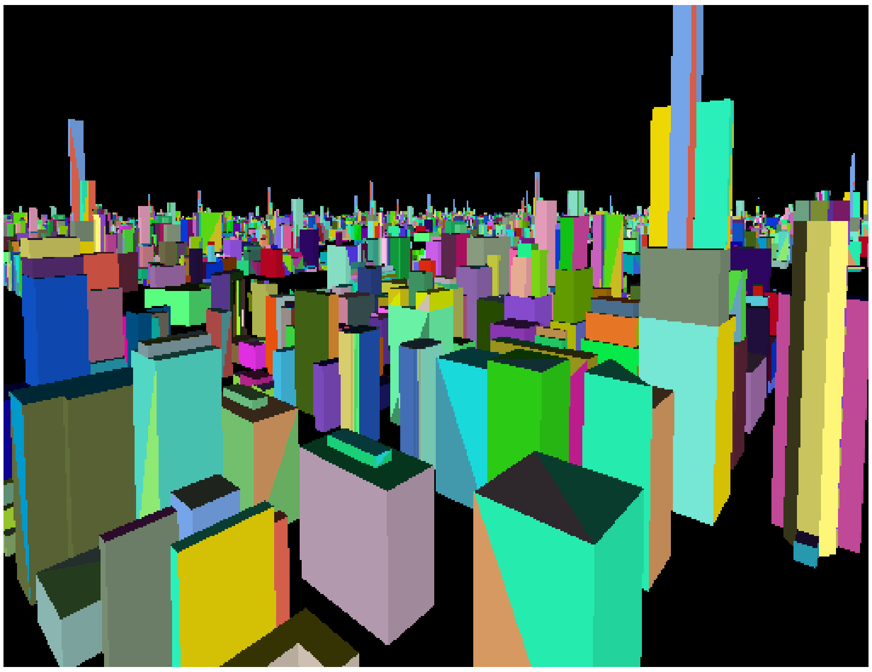


Figure 26: A birdeye view of the test model, which is composed of thirty copies of a Chicago city model and contains 300,540 polygons in total.

9.2 Virtual Brain Screen Snapshot

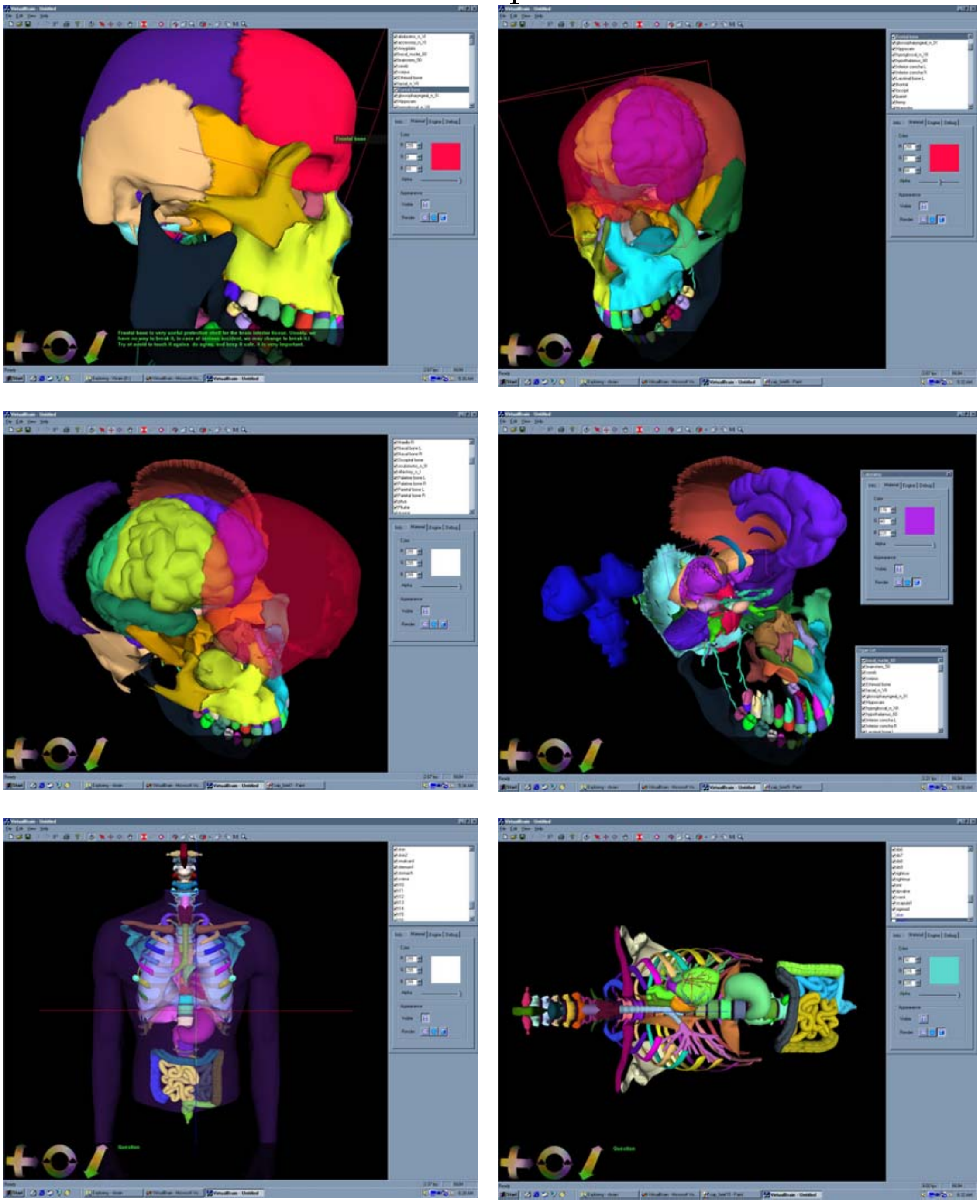


Figure 27: Screen snapshots of "Virtual Brain". The project is ongoing to import the whole body!

8 References

1. P. K. Agarwal, L. J. Guibas, T. M. Murali and J. S. Vitter. Cylindrical Static and Kinetic Binary Space Partitions. *Computational Geometry 1997*, pp. 39-48.

2. K. Akeley. RealityEngine Graphics. *SIGGRAPH 1993*, pp. 109-116.

3. J. P. Allebach. Selected Papers on Digital Halftoning. *SPIE The International Society for Optical Engineering*, 1999.

4. C. B. Barber, D. P. Dobkin and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Transaction on Mathematical Software, Vol. 22, No. 4, Dec. 1996*, pp. 469-483.

5. L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. *SIGGRAPH 1984*, pp. 103-108.

6. E. Catmull, A Subdivision Algorithm for Computer Display of Curved Surfaces. *PhD thesis, University of Utah, 1974.*

7. S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. *Symposium on Computationa Geometry 1996*, pp. 78-87.

8. S. Coorg and S. Teller. Real-time Occlusion Culling for Models with Large Occluders. *Symposium on Interactive 3D Graphics 1997*, pp. 83-90.

9. X. Decoret, G. Schaufler, F. Sillion and J. Dorsey. Multi-layered Imposters for Accelerated Rendering. *Eurographics 1999*, pp. C63-C72.

10. R. Floyd and L. Steinberg. An Adaptive Algorithm for Spatial Grey Scale. *Proceedings of the Society for Information Display, Vol. 17, 1976*, pp.75.

11. J. D. Foley, A. V. Dam, S. K. Feiner and J. F. Hughes. Computer Graphics – Principles and Practice. Addison-Wesley Publishing Company 1996.

12. H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, J. D. Austin, F. P. Brooks, Jr., J. G. Eyles and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *SIGGRAPH 1985*, pp. 111-120.

13. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH 1980,* pp. 124-133.

14. N. Greene, M. Kass and G. Miller. Hierarchical Z-Buffer Visibility. *SIGGRAPH 1993,* pp. 231-238.

15. N. Greene. Hierarchical Polygon Tiling with Coverage Masks. *SIGGRAPH 1996,* pp. 65-74.

16. P. Haeberli and K. Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. *SIGGRAPH 1990,* pp. 309-318.

17. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff and H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. *Computational Geometry 1997,* pp. 1-10.

18. N. P. Jouppi and C. F. Chang. Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparancy. *Eurographics 1999,* pp. 85-143.

19. H. R. Kang. Digital Color Halftoning. *IEEE Press,* 1999.

20. M. Kelly, K. Gould, B. Pease, S. Winner and A. Yen. Hardware Accelerated Rendering of CSG and Transparency. *SIGGRAPH 1994,* pp. 177-184.

21. D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potential Visible Sets. *Symposium on Interactive 3D Graphics, 1997,* pp. 105-106.

22. A. Mammem. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *Computer Graphics and Applications 1989,* pp. 43-55.

23. J. L. Mannos and D. J. Sakrison. The Effects of a Visual Fidelity Criterion on the Encoding of Images. *IEEE Trans. Inf. Theory, Vol. IT-20, 1974,* pp. 525-536.

24. T. Mitsa and K. J. Parker. Digital Halftoning Technique using a Blue Noise Mask. *Journal of the Optical Society of America A, Vol. 9(11), Nov. 1992,* pp. 1920-1929.

25. J. S. Montrym, D. R. Baum, D. L. Dignam and C. J. Migdal. InfiniteReality: A Real-Time Graphics System. *Computer Graphics and Interactive Techniques 1997,* pp. 293-302.

26. J. D. Mulder, F. C. A. Groen and J. J. van Wijk. Pixel Masks for Screen-Door Transparency. *IEEE Visualization 1998,* pp. 351-358.

27. K. Mulmuley. An Efficient Algorithm for Hidden Surface Removal. *SIGGRAPH 1989,* pp. 379-388.

28. B. Naylor. Partitioning Tree image Representation and Generation from 3D Geometric Models. *Graphics Interface 1992,* pp. 201-211.

29. C. H. Poon and Wenping Wang. Occlusion Culling Using Minimum Occluder Set and Opacity Map. *IEEE Information Visualization 1999,* pp. 292-300.

30. T. Porter and T. Duff. Compositing Digital Images. *SIGGRAPH 1984,* pp. 253-259.

31. A. Schilling and W. Straβer. EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer. *SIGGRAPH 1993,* pp. 85-91.

32. 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.*

33. F. Sillion, G. Drettakis, B. Bodelet. Efficient Impostor Manipulation for Real-Time visualization of Urban Scenery. *Eurographics 1997,* pp. 207-218.

34. J. Snyder and J. Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decomposition. *SIGGRAPH 1998,* pp. 219-230.

35. S. Teller and C.H. Sequin. Visibility Pre-processing for Interactive Walkthroughs. *SIGGRAPH 1991,* pp. 61-69.

36. E. Torres. Optimization of the Binary Space Partition Algorithm for the Visualization of Dynamic Scenes. *Eurographics 1990,* pp. 507-518.

37. R. Ulichney. Digital Halftoning. MIT Press, 1988.

38. S. Winner, M. Kelley, B. Pease, B. Rivard and A. Yen. Hardware Accelerated Rendering of Antialiasing Using A Modified A-buffer Algorithm. *Computer Graphics and Interactive Techniques 1997,* pp. 307-316.

39. P. Wonka and D. Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. *Eurographics 1999,* pp. C51-C60.

40. H. Zhang, D. Manocha, T. Hudson and K. E. Hoff III. Visibility Culling using Hierarchical Occlusion Maps. *SIGGRAPH 1997,* pp. 77-88.