Vertex Discard Occlusion Culling

Sibgrapi paper ID: 114496

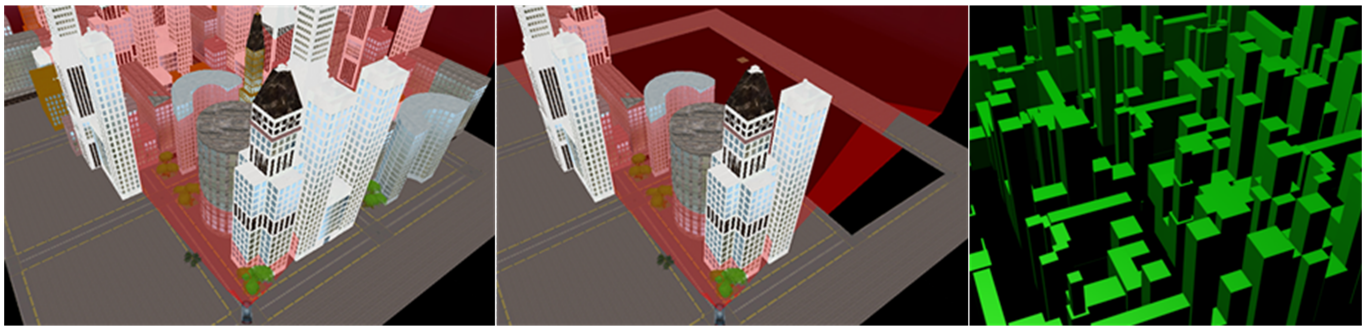

Fig. 1. Left: The densely occluded scene as viewed from the camera. Middle: The Occlusion Culling algorithm avoids rendering completely occluded objects. Right: The simplified occluder set used for occlusion.

***Abstract*—Performing visibility determination in densely occluded environments is essential to avoid rendering unnecessary objects and achieve high frame rates. In this work we present an implementation of the image space Occlusion Culling algorithm done completely in GPU, avoiding the latency introduced by returning the visibility results to the CPU. Our algorithm utilizes the GPU rendering power to construct the Occlusion Map and then performs the image space visibility test by splitting the region of the screen space occludees into parallelizable blocks. Our implementation does not need special hardware extensions and the visibility results are accessible by GPU shaders. It can be applied with excellent results in scenes where pixel shaders alter the depth values of the pixels, without interfering with hardware Early-Z culling methods. We demonstrate the benefits and show the results of this method in real-time densely occluded scenes.**

***Keywords*-Occlusion Culling; GPU; Visibility Determination**

I. Introduction

Complex scenes with thousands of meshes and expensive shading computations are increasingly frequent in current real-time graphics applications. Although commodity hardware continues to increase its computational power every day, scenes like this cannot be directly supported at real-time frame rates. Optimization techniques are crucial in order to manage that kind of graphics complexity.

Frustum culling is a commonly used technique to avoid rendering meshes that are outside the viewing volume. These invisible models can be discarded at an early stage in the pipeline obviating expensive computation that will not contribute to the final image. Unfortunately it does not consider objects (occludees) that do not contribute to the final image because they are being blocked by others in front of them (occluders).

Because of this, several Occlusion Culling techniques were developed to overcome this limitation. Applications with expensive pixel shaders may greatly improve their performance by reducing fragments overdraw.

The Z-PrePass [1] technique avoids computing unnecessary pixel shaders following a two step procedure. First it draws the entire scene in order to store in the Z-Buffer all the depth values of the scene visible points. Second the scene is drawn once more, but this time the GPU can early reject the occluded fragments based on already present depth values in the Z-Buffer. This way non visible pixel shaders are not executed.

This technique is used by many applications to reduce its pixel overdraw but its main limitation is that GPU cannot take advantage of the Early-Z [2] or [3] optimization when the pixel shader uses a depth writing operation [4], [5]. Since our method discards occluded objects before they get rasterized, no restrictions related to depth writing are imposed to pixel shaders.

Contributions: In this work we present a technique for solving Occlusion Culling in GPU, without the need for special hardware extensions or CPU read back. It includes a visibility test in the vertex shader of the application in order to discard those vertices that belong to occluded meshes. If the mesh is occluded then all its vertices can be discarded in the vertex shader, avoiding the rasterization step and the pixel computations. A previous step computes the visibility state of each mesh in the GPU and stores its result in an output texture called *Occlusion Map*. This result is acquired after performing a highly parallelized overlap and depth test comparison.

II. Related work

There is a great amount of research conducted on Occlusion Culling. A classification and overview of all these methods is presented by Cohen-Or et al. [6]. Among those techniques the ones that work in point-space are Hierarchical Z-Buffer (HZB) [7] and Hierarchical Occlusion Culling (HOM) [8].

On modern GPUs hardware occlusion queries [9] provide a built-in way to determine if a draw call contributes to the current frame, but suffer from latency and stalling effects

due to the CPU read back. To address this issue temporal coherence techniques are applied [10], [11], but they require spatial hierarchies of objects to limit the number of issues queries.

Some newer hardware capabilities allow conditional rendering without CPU intervention like OpenGL conditional rendering which is implemented as GL_NV_conditional_render [12] extension and DirectX 11 predicated rendering implemented as the ID3D11Predicate interface [13]. These methods determine whether geometry should be processed or culled depending on the results of a previous draw call. Current hardware conditional rendering does not allow the GPU shaders to access the occlusion results, but Engelhard et al. [14] implement a method that do allow this. Other authors [15], [16] also implement HZB on GPU using available in the newer compute shaders.

More recently Nießner [17] proposes a patch primitive based approach to perform occlusion culling applying HZB and temporal coherence. In recent years, since CPUs increased the number of cores and the set of SIMD instructions were extended, some approaches perform point based Occlusion Culling such as HOM using highly optimized software rasterizers [18], [19], [20], [21].

III. Vertex Discard Occlusion Culling

A. Algorithm overview

In our proposed method we perform a from-point, image-precision [6] occlusion culling process completely in GPU without the need for the CPU to read back the results. The method consists of a series of steps that must be followed by each frame to generate the *Occlusion Map*, perform the *Visibility Test* and obtain the Potentially Visible Set. Finally the method uses those results, already present in GPU, to discard all the vertices of the occluded objects before they reach further stages of the pipeline.

The steps are:

1 *Occludee Generation*: Select occluders and generate simplified volumes.
2 *Occlusion Map Generation*: Render occluder simplified volumes into the *Occlusion Map* Texture.
3 *Visibility Testing*: Determine which occludees are occluded and stored them in the *Visibility Map*.
4 *Vertex Discard*: Cull all the vertices that belong to invisible occludees.

B. Occlusion Map Generation

The method begins Offline by creating a database of selected occluders that meet a predefined criteria [22], and storing the proxy meshes which are simplified, low-poly and conservative versions of the original occluders. These simplified occluders will be rendered faster than the original meshes at the expense of more conservativeness Fig. 2.

In each frame, object-precision culling techniques such as Frustum Culling, PVS and Portal Culling [6] are applied to discard as much occluders as possible. With this obtained reduced subset of occluders we perform the first step of the method which is to render the proxy meshes into the *Occlusion Map*. This buffer stores the closest to camera depth values of every rasterized occluder and is implemented as a 32-bit floating point render target texture which is preferably a one fourth downscaled version of the screen framebuffer.

Unlike the HOM's Occlusion Map [8], our map does not contain opacity information, therefore the buffer is more similar to the HZB [7] which only stores the depth values of the occluders in each point, leaving the highest depth value to indicate no occluder presence.

The generation of the Occlusion Map is relatively inexpensive as the GPU massively parallel power is utilized to render the low-poly convex volumes of the proxy meshes and also because the pixel shader applied is extremely straightforward because it only outputs the depth value of each point.

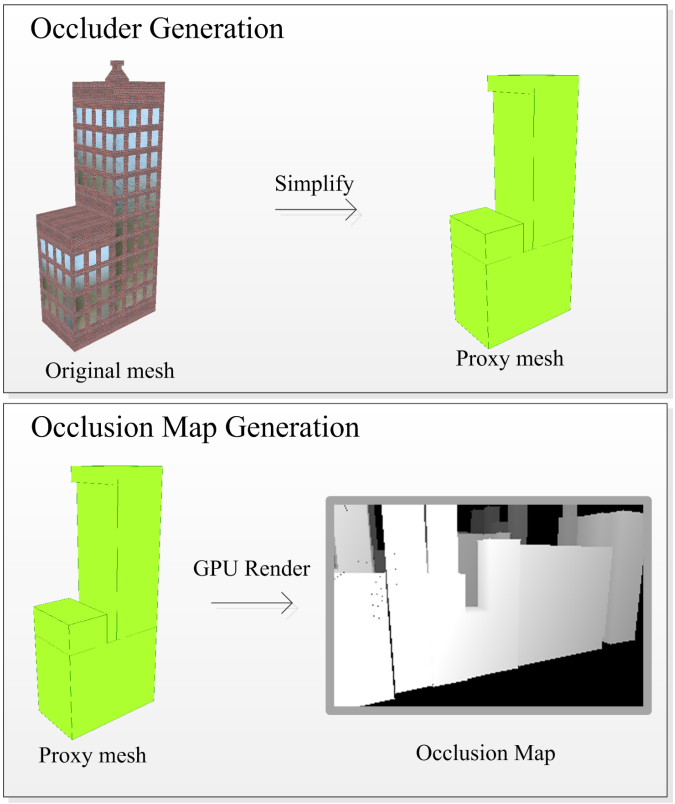

Fig. 2. First step is to obtain the simplified occluders as proxy meshes. Second step is to render all proxy meshes to the *Occlusion Map* texture.

C. Visibility Test

The core of this image based Occlusion Culling algorithm is to perform the Visibility Test for each selected occludee against the fusion of all the occluders represented by the *Occlusion Map*. Then it is used to determine whether the occludee geometry will continue along the pipeline or if it will be culled immediately. Visibility testing is performed by contrasting the points inside the occludee screen space bounding rectangle against the *Occlusion Map* depth values that contain the aggregated information of the occluders. In

each frame, for every occludee in the viewing frustum, the algorithm performs a screen space projection of the occludee bounding box vertices. With those eight screen projected points, it determines the clipped 2D screen space bounding rectangle and finds the nearest from camera depth value of those extreme points. The resulting occludee bounding rectangle becomes a conservative superset of the actual pixels covered by the occludee (see Fig. 3). Afterwards, the visibility test determines if the occludee would actually contribute to the final image and starts by comparing all the depth values inside the occludee bounding rectangle against the ones in the *Occlusion Map*; when at least one point of the occludee is closer to the camera than the one stored in the same position in the *Occlusion Map*, the algorithm can now assume that such point is visible and therefore the whole occludee is considered potentially visible.

On the other hand, to determine that an occludee is completely culled, all the pixels must be examined exhaustively and proved to be farther than the values stored in the *Occlusion Map*.

Some methods implement this overlap and depth test in CPU [19], [20], [23], [21], and others use special GPU hardware capabilities such as hardware occlusion queries [9] or the more modern predicate/conditional rendering [13], [12]. Our method manually computes the visibility result pixel by pixel utilizing GPU pixel shaders.

However as explained before, to actually conclude that a occludee is culled, we have to exhaustively test all the pixels inside the occludee bounding rectangle, resulting in $N \times M$ texture fetches to the *Occlusion Map*. As the screen space regions covered by the occludees get larger, the number of texels to fetch and test can reach very large numbers.

To accelerate this, some methods build a pyramid of downsized versions of the *Occlusion Map* where each increasing level is half the size of the previous one. There are two approaches to utilize the pyramid, one is like the method used in HOM [8] and HZB [7] which they begin at some level of the pyramid depending on the occludee bounding rectangle size and have to go to the finest level to assure that the occludee is completely culled by the occluders. The other approach [15], [16] only sticks to a selected level of the pyramid, limiting the possible number of texture fetches to a given constant to avoid the worst case scenario where they have to move to levels with greater detail. After implementing this last variation we found that the level of conservativeness was higher than expected for medium to large screen space occludees.

In this work we found that using a single level *Occlusion Map* of a fourth of the original screen buffer was a good tradeoff between number of texture fetches and level of conservativeness. In the next section we discuss the methods used to leverage the GPU hardware to perform this visibility test.

D. Block subdivision

Despite having a downsized version of the *Occlusion Map*, performing all the $N \times M$ texture fetches in a single pixel shader execution does not perform as expected, because of the serial nature of the algorithm. In the best cases this inner loop could take only a few cycles whereas in other cases the same execution could take hundreds of thousands of cycles before it is finished.

For this reason, in our method the visibility test is parallelized taking advantage of the parallel execution nature of the pixel shaders, splitting the total region covered by each occludee into a series of fixed size blocks where each one only performs a maximum of 8×8 texture lookups to the *Occlusion Map* (see Fig. 4). This way each occludee bounding rectangle is split up in blocks that concurrently perform the visibility test by executing pixel shaders that return only two possible output colors: 0 meaning the block itself is completely occluded or 1 if the block is potentially visible. The output of each pixel shader goes to a rendering target texture called *Unreduced Visibility Map (UVM)* that holds the block visibility results one next to the other as seen in Fig. 5.

In order to simplify the way each region is assigned, every occludee is assumed to have a fixed number of blocks, no matter its screen space size. In our study we determined that every occludee would have a preset number of 32×32 blocks assigned, resulting in a total of 1024 blocks. This gives us a maximum occludee screen size of 256×256 pixels and if the dimensions are larger than those, the occludee is simply considered potentially visible.

To implement this algorithm using shader model 3 (without compute shaders), we carefully position a 32×32 pixel quad (GPGPU quad) and render it using a pixel shader that executes the visibility test code. Each pixel of this quad represents

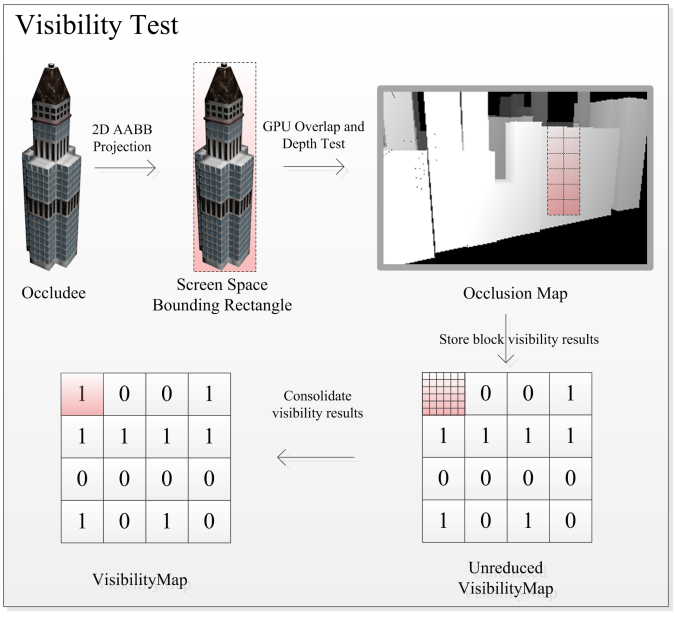

Fig. 3. The occludees in the scene are projected in 2D and the Bounding Rectangle is calculated. For each rectangle the algorithm performs the visibility test in GPU accessing the *Occlusion Map*, storing the visibility result in the *Visibility Map*.

a block visibility test of the occluder. The shader gets the occludee bounding rectangle coordinates, depth value and the block number as parameters, and then executes the 8×8 pixels overlap and depth test.

Require: $occludeeSize$
Require: $occludeePos$ {occludee AABB position}
Require: $occludeeDepth$
Require: $occlusionMap$
Require: pos {quad texture coordinates}
Require: $quadSize$
1: $base \leftarrow occludeePos \times pos + quadSize \times 8$
2: $result \leftarrow 0$ {not visible}
3: **for** $i = 0$ to 8 **do**
4: **for** $j = 0$ to 8 **do**
5: $p \leftarrow base + (i, j)$
6: $depth \leftarrow$ read p from occlusionMap
7: **if** $occludeeDepth \geq depth$ **then**
8: $result \leftarrow 1$ {visible}
9: break
10: **end if**
11: **end for**
12: **end for**

Fig. 4. Visibility test algorithm performed in a pixel shader

Using this block subdivision strategy, the visibility test is split into smaller task units and performed in parallel making use of the available GPU shader execution cores. If all the blocks comprising the occludee rectangle output 0 values, then the whole occludee is considered culled, conversely when at least one of the blocks results visible the whole occludee is considered potentially visible.

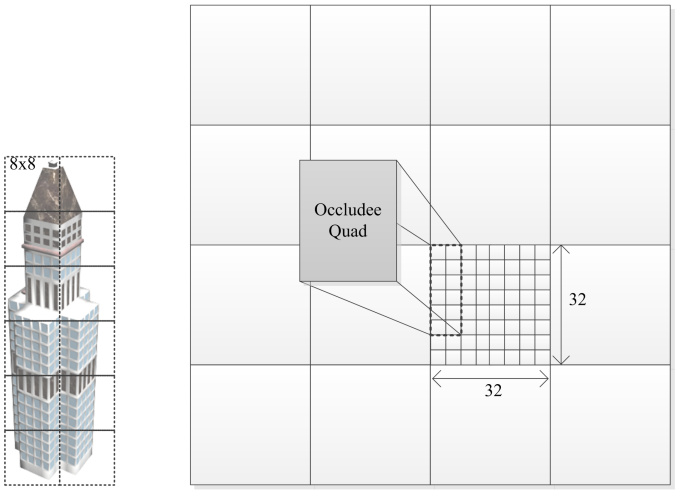

Fig. 5. The occludee is split into 8x8 blocks, then each block performs the visibility test and stores the result into the *Unreduced Visibility Map*. Each occludee has an pre-assigned region of 32x32 blocks inside this Map. *Occlusion Map* texture.

Nevertheless the visibility result of each occludee is not consolidated into a single value, but spread into a series of 32×32 matrices inside some region of the *UVM*. The next step of our method reduces each 32×32 occludee visibility result matrix into a consolidated *Visibility Map* that will hold the results of each visibility test one next to the other.

E. Visibility Map Reduction

In order to reduce the *UVM* and consolidate each 32×32 region into a single value, we need to determine if there is at least a non-zero value inside that matrix. To achieve this, we search for the maximum value of the matrix to see if there is any value other than zero. The search is done utilizing a parallel reduction approach with two rendering passes to limit the total number of operations. In the first pass we search the maximum value in each matrix column of 32 pixels and store it in an intermediate texture. In the second pass, we obtain the final *Visibility Map* looking for the maximum value in each row. Finally we end up with the *Visibility Map* containing the results of the occlusion culling process for each occludee tested in the current frame, which will be heavily accessed in the next step of our method.

F. Vertex Discard

This *Visibility Map* texture could be sent back to the CPU and processed there to avoid having to execute the draw calls to occluded objects; however this would produce a stalling effect on the GPU while sending the results back. To address this issue, we propose an asynchronous mechanism where the CPU does not need the results of the visibility test.

In our method the CPU always performs the draw calls for all the geometry that is potentially visible (the subset that passes frustum culling, portal culling, PVS, etc.), and the GPU is responsible for discarding the occluded geometry based on the *Visibility Map* content.

In our implementation we slightly modify the vertex shader that performs the *World-View-Projection* transformation as seen in Fig. 6. Before drawing an occludee, we send a parameter to the pixel shader indicating the *ID* of occludee that is about to be rendered. Based on that value, the vertex shader will perform a texture lookup in the *Visibility Map* to find the occlusion status for that particular occludee. If it is potentially visible, then the vertex shader does its usual computation letting the vertex continue throughout the pipeline. On the other hand, if the occludee is invisible we assign a negative z value to the output vertex so it can be culled by the GPU. This process is performed for every vertex that constitutes the occludee geometry.

IV. Implementation and Results

Our method was implemented using C# 4.0 with DirectX 9 and Shader Model 3. We decided not to use newer shader models (with Compute Shader capabilities) so we could test in the current commodity hardware. The implementation of our occlusion culling module was designed in a way that can

be easily adapted to other graphics frameworks, where only certain parts have to be added or modified.

We tested our method in a densely occluded 3D city scene Fig. 7, composed of 210 meshes, adding up a total of 379.664 triangles. For this scene 258 occluder proxies were generated in Offline time based on the ideas presented by [22].

In order to analyze the algorithm performance, 15 representative scene View Points were taken, where in each position we compute the following occlusion metric:

$$Value = (\frac{t - v}{t}) \times 100$$

Where t is the total scene meshes and v is the total visible meshes. With this metric we can determine the percentage of meshes that were discarded by the GPU in each frame due to occlusion culling (see results in Fig. 8).

These values are computed with Occlusion Culling deactivated and then with it activated. We also include the frames per second that resulted from rendering the scene with and without Occlusion Culling (see results in Fig. 9). The values were computed using a PC with Intel Core i3 2.40GHz processor with 2GB RAM and Intel HD Graphics 3000 GPU.

Require: vp {Vertex 3D Position}
Require: $vMap$ {Visibility Map}
Require: i {Occludee index}
1: $vis \leftarrow$ read visibility info from $vMap$ using i
2: **if** vis = 0 **then**
3: {Continue with normal vertex shader calculations}
4: **else**
5: $vp.z = -1$ {Discard vertex}
6: **end if**

Fig. 6. Vertex cull algorithm performed in a Vertex Shader

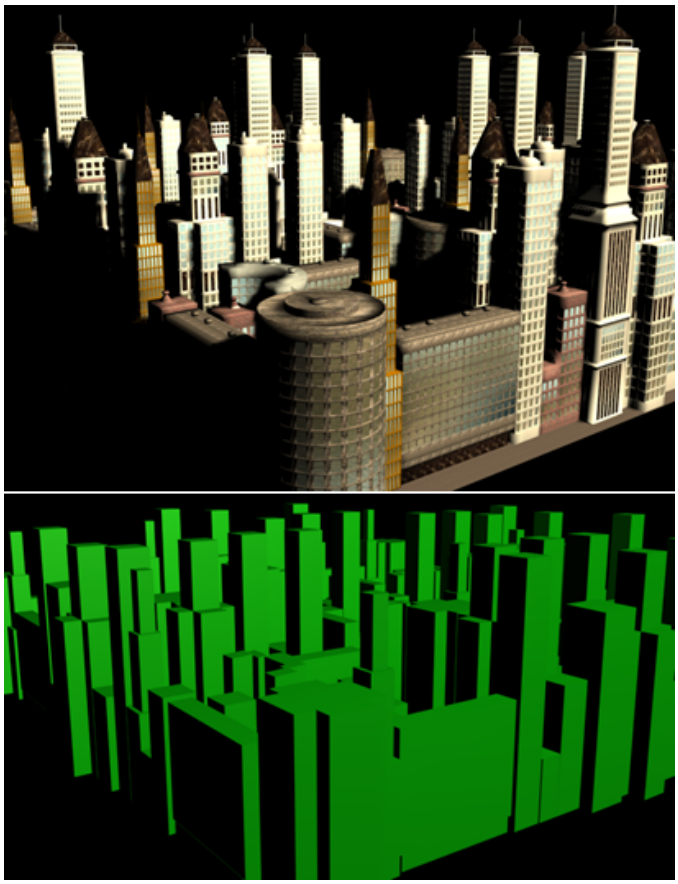

Fig. 7. Top: The 3D city scene used to test the algorithm. Bottom: The simplified occluder set used for Occlusion Culling

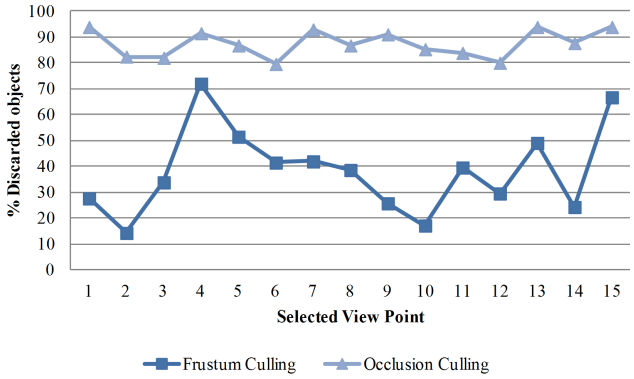

Fig. 8. Discarded mesh percent, first with only Frustum Culling and then activating Occlusion Culling, at the fifteen different selected view points.

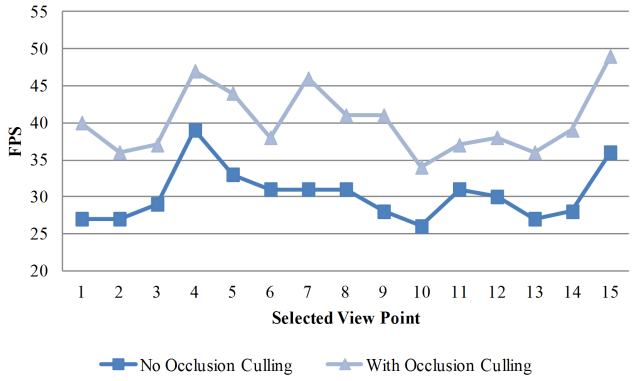

Fig. 9. FPS rendering performance only with Frustum Culling and then with Occlusion Culling activated, at the 15 different selected view points.

V. Conclusions and Future Work

We have implemented a method that performs image space occlusion culling completely in GPU, taking advantage of its rendering power to build the *Occlusion Map* and leveraging its parallel architecture to perform the visibility test.

According to our results, this occlusion culling method is applicable in densely occluded scenes where pixel shaders are computationally expensive and specifically if they alter the default depth value of the fragments, like in [4] and [5]. Conversely we found that for scenes with lightweight pixel shaders and no depth overrides, our method performs similar

to the GPU built-in Early-Z culling, making it suitable for mixed case scenarios.

As our implementation is based on Shader Model 3, it does not require special hardware requirements, beyond the vertex shader texture lookup capabilities present in most GPUs. However we found that in some older hardware, particularly those without Unified Shader architecture, the vertex texture lookup may downgrade the performance significantly [24].

It is also important to have some considerations before applying this technique. As all the occludees are sent to the GPU, no matter if they are occluded or not, there is a CPU-GPU bus bandwidth required to transfer the primitives to the graphic adapter. Moreover, as many other similar occlusion culling algorithms, the occluders have to be preprocessed in order to simplify the geometry into simpler conservative volumes.

Among the numerous enhancements to be made to our method, we would like to modify it to overcome the limitaton of the 256×256 pixel size occludees and to explore built in hardware options to reduce the *UVM*, avoiding the current two rendering pass method.

Finally as newer versions of DirectX and OpenGL become available we could explore the option of implementing this method using compute shaders, orienting it to the work presented by Nießner[17] and Rákos[15]. We could also count the number of visible blocks in each occludee and utilize the results to determine some level of detail in geometry and pixel shaders.

REFERENCES

[1] W. Engel, "Shaderx7," *Charles River Media*, 2009.

[2] E. Haines and S. Worley, "Fast, low memory z-buffering when performing medium-quality rendering," *J. Graph. Tools*, vol. 1, no. 3, pp. 1–6, Feb. 1996. [Online]. Available: http://dx.doi.org/10.1080/10867651.1996.10487459

[3] G. Riguer, "Performance optimization techniques for ati graphics hardware with directx 9.0," *ATI Technologies Inc*, 2002.

[4] V. Krishnamurthy and M. Levoy, "Fitting smooth surfaces to dense polygon meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/237170.237270

[5] A. Lee, H. Moreton, and H. Hoppe, "Displaced subdivision surfaces," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 85–94. [Online]. Available: http://dx.doi.org/10.1145/344779.344829

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

[7] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," Anaheim, CA, 1993, pp. 231–238.

[8] H. Zhang, D. Manocha, T. Hudson, and K. Hoff, "Visibility culling using hierarchical occlusion maps." Los Angeles, CA: In Computer Graphics (Proceedings of SIGGRAPH 97), 1997 1997, pp. 77–88.

[9] NVIDIA Corporation, "Nv_occlusion_query," http://www.opengl.org/registry/specs/NV/occlusion_query.txt, Mar. 2013.

[10] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha, "Fast and simple occlusion culling using hardware-based depth queries," *Techinical Report TR02-039, Dept. Comp. Sci., University of North Carolina*, 2002.

[11] D. Staneker, D. Bartz, and M. Meissner, "Improving occlusion query efficiency with occupancy maps," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, ser. PVG '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 15–. [Online]. Available: http://dx.doi.org/10.1109/PVGS.2003.1249049

[12] NVIDIA Corporation, "Nv_conditional_render," http://www.opengl.org/registry/specs/NV/conditional_render.txt, Mar. 2013.

[13] Microsoft, "Id3d11predicate interface," http://msdn.microsoft.com/en-us/library/windows/desktop/ff476577%28v=vs.85%29.aspx, Mar. 2013.

[14] T. Engelhardt and C. Dachsbacher, "Granular visibility queries on the gpu," Boston, 2009, pp. 161–167.

[15] D. Rákos, "Hierarchical-z map based occlusion culling," http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/, Mar. 2013.

[16] N. Darnell, "Hierarchical z-buffer occlusion culling," http://www.nickdarnell.com/2010/06/hierarchical-z-buffer-occlusion-culling/, Mar. 2013.

[17] M. Nießner and C. Loop, "Patch-based occlusion culling for hardware tessellation," in *Computer Graphics International*, 2012.

[18] W. Vale, "Practical occlusion culling in killzone 3," 2011, p. 49.

[19] J. Andersson, "Parallel graphics in frostbite-current & future," *SIGGRAPH Course: Beyond Programmable Shading*, 2009.

[20] Intel Corporation, "Software occlusion culling," http://software.intel.com/en-us/articles/software-occlusion-culling/, Jan. 2013.

[21] L. R. Barbagallo, M. N. Leone, M. M. Banquiero, D. Agromayor, and A. Bursztyn, "Techniques for an image based occlusion culling engine," in *XVIII Argentine Congress on Computer Sciences*, ser. CACIC 2012, Bahia Blanca, Oct 2012, pp. 405–415.

[22] M. N. Leone, L. R. Barbagallo, M. Banquiero, D. Agromayor, and A. Bursztyn, "Implementing software occlusion culling for real-time applications," in *XVIII Argentine Congress on Computer Sciences*, ser. CACIC 2012, Bahia Blanca, Oct 2012, pp. 416–426.

[23] H. Hey, R. F. Tobler, and W. Purgathofer, "Real-time occlusion culling with a lazy occlusion grid." London, UK, UK: Springer-Verlag, 2001, pp. 217–222.

[24] NVIDIA Corporation, "Geforce 8 and 9 series gpu programming guide," http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf, Mar. 2008.

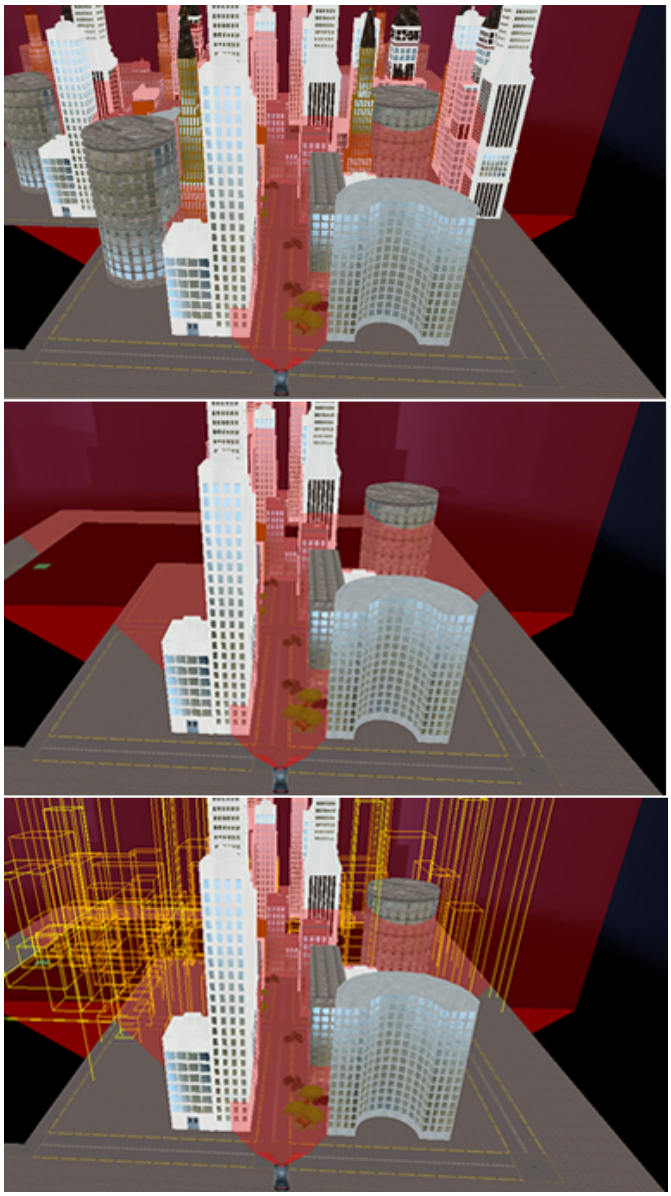

Fig. 10. Top: The original city scene without Occlusion Culling. Middle: The scene with Occlusion Culling. Bottom: The yellow boxes show the occluded meshes of the scene.