Primary points

Present a simple but powerful solution to GPU Stalls

Avoid latency when obtaining result of occlusion.

Discard is done at geometry stage discarding vertices.

Avantages

No hardware extensions needed

Occlusion Result available in shaders

Objective

Show that in certain scenarios, a image space occlusion culling can be done completely at GPU without any hardware extensions, preventing GPU stalls.

#### Abstract

#### Per object, draw call granularity, non stalling.

#### Accessible by shaders

Culled Geometry is discarded at geometry culling stage

#### No GPU-CPU sync needed

#### Introduction

#### Nombrar walktrhough de silva

#### Hay hardware occlusion culling, pero que require hardware especial. Queries agregan latencia. Y Predicate rendering ( o conditional) no permite acceder a los resultados por los shaders.

* Granular: Uses Summed Area Tables to identify the ids of ítems that are not culled completely.
* We propose a implementation of gpu occlusion culling based on HIerarchial Z-Buffer
* When z values are modified in ps the technique is useful. Z prepass is not useful this technique works.

#### Related work

#### Hierarchical Occlusion Culling Zhang

* Generacion de occluders (proxy mesh)

#### Nombrar Predicate Rendering

#### Hardware occlusion culling

#### Granular visibility (ver buen resumen de esto en patch based)

#### Patch based, goes at primitive level

### Asyncronous GPU Occlusion Culling

#### Overview

* + - Nombrar las dos faces:
    - Decir que no se necesita conocer resultado del occlusion test
    - No vuelve a cpu

We divide the async occlusion culling method in three phases. The first phase is the Hierarchical Z Buffer (or Occlusion Map) construction in which the most representative occluders in the scene are rendered to a texture which will contain the coverage and depth information. In order to speed up the visibility test that takes place later in the process, this Hierarchical Z Map is downsampled in a pyramid style similar to HOM by [[1](#Zhang97visibilityculling)].

In the second phase of the process, the Visilibty test takes place where the actual potentially visible set of occludees is obtained. Using the HiZMap calculated before and based on the screen space bounded rectangle of the occludees, we perform the overlap and depth test to determine if the occludee is potentially visible or if it is completely occluded. The list of occludees and their Screen Space Bounding Rectangles are already stored in the GPU, so the calculation is performed in a Pixel Shader.

The result of each visibility test can be either potentially visible or completely occluded, and this binary result is stored in Occlusion Result texture, which will be used in the final phase which is the geometry discard. Unlike the Occlusion Query methods, where the CPU waits stalled to get the results from the GPU, in this method the all the geometry is sent to the GPU where it is in charge of discarding the geometry (vertices) at the earliest stage of the pipeline. For every object that is sent to the GPU an ID is also sent with it, so that way the Vertex Shader can perform a lookup in the Occlusion Result Texture and either project and propagate it into the rendering pipeline or cull it and prevent it from being rendered later.

In the next sections we will discuss these phases in more detail.

#### Hierarchical Occlusion/Z Map Construction

The Async Occlusion Culling method begins with the construction of the Hi Z map based on a carefully chosen subset of occluder objects, where a simplified, low poly and conservative version is preferred in most cases [[2](#Germs01geometricsimplification)] [[3](#Leo12)]. To obtain the Hi Z Map, we first create a buffer of the same size or half of the size of the framebuffer, that will hold the depth and coverage information of the occludeers at a given time of the scene. The application rasterizes the occluders into the HiZ Map texture using a pixel shader that only outputs the depth value of the occluder geometry, taking advantage of the rendering power of the GPU, in contrast with other alternatives that rasterize in CPU [[4](#Int13)].

The reason for the creation of this Hi-Z Map is to be used later to perform the visibility test pixel by pixel that lies inside the occludee projected bounding rectangle. However as large occludees will have a larger area, that could potentially have hundreds of thousands of pixels, the pixel level depth test is prohibitive. The solution for it is to build a hierarchy of Maps where the level 0 contains the original depth buffer, and the subsequent level, with half of its size, is constructed by getting the farthest depth value of the four neighboring texels and combining into one single texel. This buffer is very similar as the Hierarchical Z-Buffer described by [[5](#Gre93)] and is the reason for the name Hi Z Map.

The chain of downsampled Maps is built by performing several render passes until the 1x1 pixel level of it is reached or when any threshold level is reached.

This Hi-Z Map pyramid with all its levels will be used to reduce the number of depth value comparisons in the next phase in order to determine the visibility of the occludees.

* + - Render HiZ map based on occlude simplification.
    - Fast rendering, render states.
    - Texture format.
    - Build mipmap chain
    - Determine limits of chain

### Visibility Test

* + - Perform overlap test and Depth test
    - Hierarchical problem
    - Mip map determination
    - Results go to a texture

The visibility test is one of the most important phases and it is responsible for finding which occludees are fully occluded and which are potentially visible by contrasting every pixel covered by the occludee bounding rectangle and the information provided by the Hi-Z map.

The occludee information is coded into two different textures; one with 128bit containing the Bounding Rectangle and the other 32bit with the Depth information. This data is calculated on CPU and sent to the GPU every frame, at the same time with the occluders and all the geometry of the occludees.

When the Hi-Z map with the pyramid is already built and ready to use, the Visibility test is executed as a series of Compute Shaders (implemented as pixel shaders in this case) which will perform the overlap and depth to determine the occluded objects. For every pixel contained inside the screen space bounding rectangle of the Occludee the shader performs a depth comparison, where as soon as it detects that a point of the occludee is closer to the camera than the pixel stored in the Hi Z Buffer, it concludes that the Occludee is potentially visible. On the other hand if all the rectangle points are found to be farther than the points stored in the Hi-Z Map, then the algorithm concludes that is it fully occluded.

TO perform this visibility test for occludees that have an area of several pixels, the texture lookups in the original level of th HiZ map would be really high, we need to determine which Level or mip map Level we need to use.

Larger occluders will use levels that are smaller and smaller occluedees can use the larger levels of HI Z map.

To perform the depth test we must first get the level correct level of mipmap to use

[poner psuedocodigo de doble for]

### Geometry Discard

* + - Once we have occlusion results
    - Send all potentially visible objects to GPU. It will discard the pipeline by doing geometry culling.

## Implementation

Directx 9, no compute shaders. Shader model 3

C#

Geometry culling done by setting z to -1

## Results

What gpu was used.

Time to build mipmap chain

Time to discard vertexes

Occlusion effectiveness

## Conclusions

## Future work

#### Webgl

Compute Shader

Level of detail obtained in shaders

Speheres instead of AABB

#### Aknowledgements

## References

Daniel Es journal

Daniel Rákos - Rastergrid

Nick Darnell

Granular visibility engelhardt

# Pasos

## Occluder generation (Proxy mesh)

## Oclusion Map Generation

## Visiblity test. Occludee.

### Occludee Bounding rectangle

### Occludee Compute Quad

### Unreduced Visibility Map

### Visiblity Map

## Occludees Vertex discard

# Primary Point

# Abstract

# Introduction

## Importancia de Occlusion Culling.

## Intro a la técnica

### Present techique:

#### No Hardware Extensions

From-point visibility determnation

#### Occlusion Result Available in Shaders

#### Avoids Latency CPU Stall and GPU Starvation.

#### Enables Z Modify in PS (even if Z Prepas disabled).

#### Culls Vertex Shaders early in the pipeline.

#### Early Avoid Culled Geometry Shader and heavy Pixel shaders.

# Related Work

Occlusion Culling

## Walktrhough de Silva

## Hierarchical Occlusion Map

Hardware Occlusion Culling

## Occlusion Queries en HW (el de Nvidia y Predicate Rendering)

## Granular Queries

## Patch Based Occlusion Culling.

## Z-Prepass

# Vertex Discard Occlusion Culling

## Algorithm Overview

### In our proposed method we perform an image precision and point based [[6](#cohen2003survey)] occlusion culling process completely in GPU without the need for the CPU to read back the results. This method is based on Hierarchical Occlusion Maps HOM proposed by Zhang [[1](#Zhang97visibilityculling)], but instead of utilizing a pyramid map, it keeps a single level buffer.

The method consists of a series of steps that must be followed each frame to generate the Occlusion Map, perform the Visibility Test to get the potential visible set and finally use those results already present in GPU to discard the all the vertices of the occluded objects before they reach further stages of the pipeline.

### Steps: Dibujo del pizarron. Occludee: bounding rectangle, quad, Visiblity block Map, etc.

## Occlusion Map Generation

The method begins Offline by creating a database of selected occluders that meet a predefined criteria [[2](#Germs01geometricsimplification)] , where we also store the proxy meshes which are simplified, low-poly and conservative versions of the original occluders that will be rendered faster than the original meshes at the expense of more conservativeness.

For every frame, object-precision culling techniques such as PVS or Portal Culling [[6](#cohen2003survey)] are applied of the occluders present in the scene to discard as much occludders as possible. With this obtained reduced subset of occluders that fit inside the current viewing frustum, 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 32bit floating point render target texture which is preferably a 1/4th version of the screen framebuffer.

Unlike the HOM’s Occlusion Map [[1](#Zhang97visibilityculling)], our map does not contain opacity information, therefore the buffer is more similar to the Hierarchical Z buffer [[5](#Gre93)] which only stores the depth values of the occluders in each point, leaving the highest depth value to indicate no occluder presence.

The generation 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 as it only outputs the depth value of each point.

## Visibility Test

The heart 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 and then to verify if the receiving occludee geometry will continue along the pipeline or if it will be culled immediately.

The process of visibility test 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 it also stores the furthest depth value of those points. This occlude bounding rectangle is a conservative superset of the actual pixels covered by the occludee.

Then actual visibility test tries to determine if the occludee would actually contribute to the final image and consists of 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 position in the Occlusion Map, the algorithm can assume that that point is visible, and that the occludee as whole can be considered potentially visible. On the other hand, to determine that an occludee is culled, all the pixels must be examined and proved to be farther than the values stored in the Occlusion Map.

Some methods perform this Overlap and Depth Test in CPU [[7](#andersson2009parallel)] [[4](#Int13)] [[8](#Hey:2001:ROC:647653.732306)] [[3](#Leo12)], and others use special GPU hardware capabilities such as hardware occlusion queries [[9](#NVI02)], [[10](#CGF:CGF793)] or the more modern predicate/Conditional rendering. Our method computes the visibility result pixel by pixel manually utilizing GPU shaders [[11](#Dar13)]

Get NxM texture accesses. Some propose building a pyramid `referencias+` but resulted too conservative for our purposes. Large occluders were almost never occluded and conservatiness was too high.

To get an acceptable level of conservatiness at expense of lower samples, we decided to make 1/4 of the original size of framebuffer.

Despite this downsampling, perfoming NxM naïve texture lookups in a nested for loop resulted in poor performance.

Had to take advantage of parallel architecture and texture cache was not being leveraged.

### Block subdivision

As a single pixel shader (or compute shader) performing the total number of samples to the Occlusion Map, in a single nested loop, our method takes advantage of the parallel …

In our method the visibility test is parallelized taking advantage of the parallel execution of the pixel/compute shaders, by splitting the total region covered by each occludee bounding rectangle into a series of fixed size blocks, where each one only performs a maximum of 8x8 texture lookups to the Occlusion Map.

This way each occludee bounding rectangle is made up of these blocks that perform the visibility test concurrently by executing pixel shaders that output a 0 color value meaning the block itself is completely occluded or 1 if the block is potentially visible. The output of these visbility test blocks goes to a rendering target texture called Unreduced Visibility Map (UVM) where it stores every occludee block visibility result one next to the other as seen in fig XXXXX

To simplify the texture arrangement each occludee is assumed to have 32x32 blocks giving a maximum occludee screen size of 256x256 pixels, and if the dimensions are larger the occludee is not a candidate for occlusion culling.

To implement this in shader model 3 without the use of compute shaders, we carefully position a 32x32 pixel Quad with a pixel shader that executes the visbility test and sets the occludee bound rect coordinates and depth value as parameters. The position of each quad is arranged one next to the other so, after the pixel shader execution.

If all the blocks comprising the occludee rectangle output 0, then the whole occludee is considered culled, on the other hand when at least one of the blocks results visible the whole occludee is potentially visible.

Using this subdivision into blocks, the visibility test is performed in parallel making use of the available shader execution cores.

However the visibility result of each occludee is not consolidated into a single value, but spread into a series of 32x32 matrices inside some part of the Unreduced Visibility Map. The next step of our method reduces each 32x32 occludee visibility result matrix into a single Visibility Map that will hold the results of each visibility test one next to the other.

### Visibility Map Reduction

To reduce the UVM, we need to determine if any of the values inside the visibility map has a non zero value.

We use a Max separable filter to accomplish this by performing two rendering passes. In the first pass we search the max value of each sub row of 32 pixels inside the UVM and output a 0 or 1 value into an intermediate texture. In the second pass we obtain the final Visibility Map performing the same process but calculating the columns instead.

After this reduction we obtain the final Visibility Map which contain the results of the occlusion culling process for each occludee tested in the current frame, and will be utilized in the next by the next step of the method.

## Vertex Discard

After the visibility test has been performed, the result texture now contains for every occludee a value that indicates if it has been identified as occluded or potentially visible.

This result texture could be requested by the CPU and then treated there, however this will produce a stalling effect on the GPU while it sends the results back. To address this issue, we propose the asynchronous mechanism where the CPU doesn´t need the results of the visibility test.

In this case the CPU always performs the draw calls for all the geometry that is potentially visible (i.e. passed frustum culling, portal culling, PVS, etc), and the GPU is responsible of discarding the occluded geometry based on the Visibility Map.

To implement the occlusion culling in GPU

The vertices are in fact culled by following the vertex shader pseudocode in fig XXXX.

### Using result of Visility Map.

### Poner Pseudocodigo del cull y el transform comun.

### Vertex is set outside view frustum. GPU culls it automatically, releasing from pipeline.

### Advantages: Culling before rasterization, and even before Geometry Shader

# Implementation and Results

Our method was implemented using C# 4.0 with DirectX 9 and Shader Model 3, where we decided not to use newer shader models that count with Computer Shader so we could test it current commodity hardware.

The implementation of our occlusion culling module was designed so that it can be easily adapted to other frameworks or graphic packages.

It was design to be easily been added to an existing application. The only considerations are fist to execute the occlusion optimization routines, and second to add the visibility check in the main vertex shader of the application.

We tested the method in densely occluded 3D city scene, composed of 210 meshes, adding up a total of 379.664 triangles. For this scene 258 occluder proxies were generated Offline time based on the ideas presented by [[3](#Leo12)].

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 = (t - v) / t \* 100, where t is the total scene meshes and v is the total visible meshes. With this metric we can determine the percentage of discarded meshes by our algorithm that were not sent to the GPU in each frame.

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. The results were computed using a PC with Intel Core i3 2.40GHz processor with 2GB RAM and Intel HD Graphics 3000 GPU.

Fig. XXXXXXXXX. Top: FPS rendering performance only with Frustum Culling and then with Occlusion Culling activated, at the fifteen different selected View Points. Bottom: Discarded mesh percent, first with only Frustum Culling and then activating Occlusion Culling, at the fifteen different selected View Points.

## Results

### Cuanto tarda en hacer el discard y vertex texture lookups.

### Escenarios, FPS, Tiempo en hacer build del Occlusion Map.

### Como integrar con existing rendering frameworks.

# Future work

## Compressing Z Buffer to reduce visibility texture lookups.

## Overcome 32x32 block limitation for large screen space occludeers.( 256 screen space pixels).

## Use GPU built in hardware to perform Unreduced Visiloibity Map Reduce using mip map chain generation.

## Implement using CUda or Compute Shaders DX.

# Conclusions

## Advantages

### Can be applied effectively. Highly dense Occludeed environments, Heavy load pixel shaders, PS that modify Z values.

### Uses Raw power of GPU to render Occlusion Map.

### Takes advantage of the parallel architecture of the GPU to perform Visiblity Test.

## Disadvantages

### Geometry is sent to GPU even if it is not ended visible, can make a bottleneck in bus.

### Occluder generation

### Some older architectures may suffer performance penalties when performing texture lookup to the visilibity texture. Doesn´t happen in Unified Shader adapters.

### Not suitable For scenes with lightweight PS and when Early Z features es aprovechalo.

# References

# Trabajos citados

x

|  |  |
| --- | --- |
| [1] | Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth Hoff, "Visibility Culling Using Hierarchical Occlusion Maps," in *In Computer Graphics (Proceedings of SIGGRAPH ´97)*, Los Angeles, CA, August 1997, pp. 77-88. |
| [2] | Rick Germs and Frederik W. Jansen, "Geometric Simplification For Efficient Occlusion Culling In Urban Scenes," in *Proc. of WSCG 2001*, 2001, pp. 291-298. |
| [3] | Matias N Leone, Leandro R Barbagallo, Banquiero Mariano, Diego Agromayor, and Bursztyn Andres, "Implementing Software Occlusion Culling for Real-Time applications," in *CACIC 2012*, Bahia Blanca, 2012. |
| [4] | Intel Corporation. (2013, Jan.) Software Occlusion Culling. [Online]. <http://software.intel.com/en-us/articles/software-occlusion-culling> |
| [5] | Ned Greene, Michael Kass, and Gavin Miller, "Hierarchical Z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, Anaheim, CA, 1993, pp. 231-238. |
| [6] | D. Cohen-Or, Y.L. Chrysanthou, C.T. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *Visualization and Computer Graphics, IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412-431, 2003. |
| [7] | J. Andersson, "Parallel Graphics in Frostbite-Current & Future," *SIGGRAPH Course: Beyond Programmable Shading*, 2009. |
| [8] | Heinrich Hey, Robert F. Tobler, and Werner Purgathofer, "Real-Time Occlusion Culling with a Lazy Occlusion Grid," in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, London, UK, UK, 2001, pp. 217-222. [Online]. <http://dl.acm.org/citation.cfm?id=647653.732306> |
| [9] | NVIDIA Corporation. (2002, Feb.) NV\_occlusion\_query. [Online]. <http://www.opengl.org/registry/specs/NV/occlusion_query.txt> |
| [10] | Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," *Computer Graphics Forum*, vol. 23, no. 3, pp. 615-624, 2004. [Online]. <http://dx.doi.org/10.1111/j.1467-8659.2004.00793.x> |
| [11] | Nick Darnell. Hierarchical Z-Buffer Occlusion Culling. [Online]. <http://www.nickdarnell.com/2010/06/hierarchical-z-buffer-occlusion-culling/> |
| [12] | Will Vale, "Practical occlusion culling in KILLZONE 3," in *SIGGRAPH Talks*, 2011, p. 49. |
| [13] | Marc Olano and Trey Greer, "Triangle scan conversion using 2D homogeneous coordinates," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, New York, NY, USA, 1997, pp. 89-95. [Online]. <http://doi.acm.org/10.1145/258694.258723> |
| [14] | Matthias Nießner and Charles Loop, "Patch-Based Occlusion Culling for Hardware Tessellation," in *Computer Graphics International*, Bournemouth, 2012. |
| [15] | Krzysztof Narkowicz. (2012, April) Software occlusion culling. [Online]. <http://kriscg.blogspot.com/2010/09/software-occlusion-culling.html> |
| [16] | Joel McCormack and Robert McNamara, "Tiled polygon traversal using half-plane edge functions," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, New York, NY, USA, 2000, pp. 15-21. [Online]. <http://doi.acm.org/10.1145/346876.346882> |
| [17] | Intel. (2012, April) Intel 64 and IA-32 Architectures Optimization Reference Manual. [Online]. <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html> |
| [18] | Ned Greene, "Hierarchical polygon tiling with coverage masks," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1996, pp. 65-74. [Online]. <http://doi.acm.org/10.1145/237170.237207> |
| [19] | Thomas Engelhardt and Carsten Dachsbacher, "Granular visibility queries on the GPU," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics*, Boston, 2009, pp. 161-167. |
| [20] | Xavier Decoret, "N-Buffers for efficient depth map query," *Computer Graphics Forum*, vol. 24, no. 3, pp. 393-400, 2005. |
| [21] | Nicolas Capens. (2004, Nov.) Advanced Rasterization. [Online]. <http://devmaster.net/forums/topic/1145-advanced-rasterization/> |
| [22] | Z. Bethel. (2011) A Modern Approach to Software Rasterization. [Online]. <http://cse.taylor.edu/~zbethel/MSR/ModernApproachToSR.pdf> |
| [23] | Tomas Akenine-Moller, Eric Haines, and Natty Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008. |
| [24] | Michael Abrash. (2012, June) Rasterization on Larrabee. [Online]. <http://www.drdobbs.com/parallel/217200602> |
| [25] | (2012, April) The OpenMP® API specification for parallel programming. [Online]. <http://openmp.org/> |
| [26] | Daniel Rákos. Hierarchical-Z map based occlusion culling. [Online]. <http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/> |

x