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

# Primary Point

# Abstract

No CPU readback

Z-Prepass

No hierarchy

Available in GPU shaders

Utilzie compute shaders

# Introduction

## Importancia de Occlusion Culling. (reciclar del paper del cacic)

## Intro a la técnica

### Present techique:

Complex scenes with thousands of meshes and expensive shading computations are common place in current Real-Time graphics applications. Although commodity hardware continues to increase its computational power every day, most scenes of this kind cannot be directly supported at real time frame rates. Application of optimization techniques are imperative in order to manage that kind graphics complexity.

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

As frustum culling doesn’t determine if hidden objects are being covered by other models in front of them, several Occlusion Culling techniques can be applied. Applications with expensive pixel shaders may greatly improve their performance by reducing fragments overdraw.

The Z pre-pass technique [CITAR] 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 again, but this time the GPU can early reject the occluded fragments based already present depth values in the Z buffer. These way non visible fragments are not computed.

This technique is used by many applications to reduce its pixel overdraw but its main limitation is that GPU cannot take advantage of this optimization when the pixel shader uses a depth writing operation.

In this work we present a technique for solving Occlusion Culling in GPU, without the need of special hardware extensions or CPU readbacks.

The method consists of including a visibility test in the vertex shader of the application in order to discard those vertices that belongs 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 fragment computations.

A previous step computes in GPU the visibility state of each mesh and stores its result in the output texture called Occlusion Map. This state is acquired with an overlap test and depth comparison procedure. This procedure uses a depth map computed from proxy meshes, called which are low-poly conservative versions of the scene meshes, simplified in offline time.

As our method discards occluded objects before they get rasterized, they never get to alter the Z buffer, so one of the advantages is that the hardware Early-Z or Hyper Z culling technologies can still be used effectively when the pixel shaders override the original depth values. Altering the depth value in pixel shaders is a common technique applied to achieve more realistic effects [[1](#Eng04)], but after it the GPU hardware no longer can apply the early z culling optimizations.

# 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

There is a great amount of research conducted in Occlusion Culling. A classification and overview of those methods is presented by Cohen-Or et al. [[1](#cohen2003survey)]. Among those techniques the ones that work in point-space are Hierarchical Z Buffer [[2](#Gre93)] and Hierarchical Occlusion Culling [[3](#Zhang97visibilityculling)].

On modern GPUs hardware occlusion queries [[2](#NVI02)] provide a built in way to determine if a draw call contributes to the current frame, but suffer from stalling effects due to the CPU readback. To address this issue temporal coherence techniques are applied [[3](#CGF:CGF793)] [[4](#Mat08)] [[5](#Gut06)], but they require spatial hierarchies to limit the number of issues queries.

Some newer hardware capabilities allow conditional rendering without CPU intervention, like OpenGL conditional rendering [[4](#NVi13)] which is implemented as GL\_NV\_conditional\_render extension and DirectX 11 predicated rendering implemented as the ID3D11Predicate interface. These methods determine whether geometry should be processed depending on the results of a previous draw call. Current hardware Conditional Rendering doesn’t allow the GPU shaders to access the occlusion results, but Engelhard et al. [[7](#Eng09)] implement a method that allows this. Other authors [[10](#Rák13)] [[11](#Dar13)] also implement HZB on GPU using available compute shaders.

More recently Nießner [[13](#Nie12)] proposes a patch primitive approach to perform occlusion culling applying Hierarchical Z buffer and temporal coherence, but requires the use of newer hardware capable of executing compute shaders.

In recent years, as the CPUs increased the number of cores and the set of SIMD instructions were extended, there were some approaches to perform Point-Based Occlusion Culling such as HOM using highly optimized software rasterizers [[5](#DBLP:conf/siggraph/Vale11)] [[6](#andersson2009parallel)] [[7](#Int13)] [[8](#Bar12)] [[13](#Bar12)].

Z-Prepass?

# Vertex Discard Occlusion Culling

## Algorithm Overview

### In our proposed method we perform an image precision and point based [[1](#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 [[9](#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 [[10](#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 [[1](#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 [[9](#Zhang97visibilityculling)], our map does not contain opacity information, therefore the buffer is more similar to the Hierarchical Z buffer [[11](#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 resolve if 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 furthest from camera depth value of those extreme points. The resulting occludee 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 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 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 completely culled, all the pixels must be examined exhaustively and proved to be farther than the values stored in the Occlusion Map.

Some methods perform this Overlap and Depth Test in CPU side [[6](#andersson2009parallel)] [[7](#Int13)] [[12](#Hey:2001:ROC:647653.732306)] [[13](#Leo12)], and others use special GPU hardware capabilities such as hardware occlusion queries [[2](#NVI02)], [[3](#CGF:CGF793)] or the more modern predicate/conditional rendering [[4](#NVi13)]. Our method manually computes the visibility result pixel by pixel utilizing HLSL pixel shaders.

However as explained before, to actually conclude that a occludee is culled, we have to exuastively test all the pixels inside the occludee bounding rectangle, resulting in a NxM texture fechtes 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 high numbers.

To accelerate this some methods build a pyramid of downsampled versions of the occlusion map where each increasing level is half of the size of the previous one. There are two approaches to utilize the pyramid, one is like the method used in HOM [[9](#Zhang97visibilityculling)] and HZB [[11](#Gre93)] which they begin at some level of the pyramid depending on the occlude bounding rectangle size, but in the worst case scenario these methods have to go to the finest level of the pyramid to assure that the occludee is completely culled by the occluders. Moreover some methods [[14](#Dar13)] [[15](#Rák13)] only stick to a selected level of the pyramid, limiting the possible number of texture samples to avoid the worst case scenario to have to move to greater detail levels. 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 samples and level of conservativeness. In the next section we discuss the methods used to leverage the GPU hardware to perform this visibility test.

### Block subdivision

Despite having a downsized version of the Occlusion Map, performing all the NxM texture samples in a single Pixel Shader execution does not perform as expected, because of the serial nature of the algorithm presented in figure XXXX. In the best cases this inner loop could take only a few cycles when in other worse cases the same execution could take hundreds of thousands of cycles before it finished.

For this reason, in our method the visibility test is parallelized taking advantage of the parallel execution of the pixel 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 split up in blocks that concurrently perform the visibility test 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 each visibility test block goes to a rendering target texture called Unreduced Visibility Map (UVM) where holds the occludees block visibility results one next to the other as seen in fig XXXXX. Thus every occludee in the scene has an assigned region inside this map to hold the results of each tested block.

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 case we determined that every occludee would have a preset number of 32 blocks in width by 32 in height assigned resulting in a total of 1024 blocks. This gives us a maximum occludee screen size of 256x256 pixels and if the dimensions are larger than that, the occludee is simply not considered as a good candidate for occlusion culling.

To implement this algorithm using shader model 3 (without compute shaders), we carefully position a 32x32 pixel quad and render it using a pixel shader that executes the visibility test code. Each pixel of this quad represents 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 8x8 pixels overlap and depth test.

Using this blocks 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, then the whole occludee is considered culled, conversely when at least one of the blocks results visible the whole occludee is potentially visible.

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

### Visibility Map Reduction

In order reduce the UVM and consolidate each 32x32 region into a single value, we need to determine there is at least a non-zero value inside that matrix. For this we find the maximum of the matrix to see if there is any value other than zero.

The straightforward implementation for finding the maximum value inside the 32x32 matrix would require 1024 loop iterations; however we utilize separable reduction approach using two render 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 by finding the maximum value in each row, reducing the total number of iterations to 64.

After this reduction now have the Visibility Map containing the results of the occlusion culling process for each occludee tested in the current frame, and will be heavily utilized in the next by the next step of the method.

## 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 it sends the results back. To address this issue, we propose an asynchronous mechanism where the CPU doesn´t 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 passed 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 modified the vertex shader that performs the World-View-Projection transformation. 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 to let the GPU know that it has to be culled. This process is performed for every vertex that constitutes the occludee geometry.

# 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 Computer Shader capabilities) so we could test in the current commodity hardware. The implementation of our occlusion culling module was designed so that it 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, 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 [[13](#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 that were not sent to the GPU in each frame due to occlusion culling.

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.

# Future work

# 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 especially if they alter the default depth value of the fragments. Conversely we found that for scenes with lightweight pixel shaders and no z overrides, our method did not outperform the GPU built in early z culling.

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 commodity hardware GPUs. Some older hardware, particularly those without Unified Shader architecture, the vertex texture lookup downgraded the performance significantly.

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 enhancements to be made to our method, the is to overcome the limitaton of the 256x256 pixel occludees mentioned before.

Also we would like 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, similar to the work presented by Nießner et al [[13](#Nie12)] and Rákos [[11](#Rák13)]. 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.

## 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.

### Aknowledgements

Thanks UTN Andres Bursztyn, Profesora ingles.

References