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 [[5](#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 [[4](#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.

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 Result Texture.

## 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

#### 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

## Walktrhough de Silva

## Hierarchical Occlusion Map

## 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. This method is very similar to HOM proposed by Zhang [[1](#Zhang97visibilityculling)] but without a pyramid.

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 PVS and finally use the results stored in GPU side to discard the geometry vertices of the occluded objects before they reach further stages of the pipelinr.

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

## Occlusion Map Generation

### Occluders are chosen. Citar paper de Mati. Prepare a set of proxy meshes.

Convenient to use not original geometry.

### Occludes are lightweight and conservative boxes, easy to render.

### After occludee list is prepared, frustum culling applied, leaving a subset of the ocludees.

### Occlusion map is generated using GPU rendering power. Simplified VS and PS is used to simple output Z value.

### .25 of original frame buffer, using 32F floating point to store depth.

### The render target is Occlusion Map (depth Map)

The method begins offline by creating a database of good occluders that meet a predefined criteria [[2](#Germs01geometricsimplification)], in this database 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.

In 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 the most number of occluders 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 of those occluders into the Occlusion Map. This Occlusion Map 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, consequently the buffer only stores the depth values of the occluders in each point, leaving the higest depth value to indicate no occluder presence.

The create this Map is relatively inexpensive as the GPU massively parallel power is utilized to render the low-poly convex volumes of the proxy meshes and because the pixel shader to accomplish this is extremely simple 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 occluder fusion represented by the Occlusion Map to determine if

if the receiving occludee geometry will continue thru the pipeline or 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. With those eight screen projected points, it determines the 2D screen space bounding rectangle

The occludees only have assigned a constant furthest depth value.

by checking the screen space projected occludee conservative region against the occluder fusion represented by the Occlusion Map

For each object in the viewing frustum, the algorithm conservatively

approximates its projection with a screen-spacebounding

rectangle of its bounding box. This rectangle covers a superset

of the pixels covered by the actual object. The extremal values

of the bounding rectangle are computed by projecting the corners

### Need to perform Overlap and depth test.

### To get the occludee bounding rectangle project in 2D screen space.

### Paralellizing Occlusion Culling

#### Naive

##### Overlap and depth test naïve implementation would require NxM texture access in a single shader execution.

#### Occludee subdivision into Blocks

##### To take advantage of the parallel architecture of the GPU, the texture lookups are split into Block 8x8.

##### Max 32 blocks per size. Otehrwise considered visible.

##### Screen Quad is generated to execute compute shader and calculate the visibility of each block.

##### The position of the quad is important, the result will be rasterized in a carefully chosen region of the unreduced Visibility Map.

##### Poner Pseudocodigo.

##### What happens if occludee is less than 32x32 blocks.

##### Diagrama de Unreduced Visilibity Map.

#### Reducing visibility Blocks in each block

##### To reduce the result of each block of coccluder into a single visibility result we need to reduce.

##### Using separable filters. Max Two passes of 32x32.

## Vertex Discard

### 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

The solution was built using C# 4.0, DirectX 9, shader model 3.0, without compute shader). 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.

A 3D city model was built, composed of 210 meshes, adding up a total of 379.664 triangles. For this scene 258 Occluders were generated in Offline time by the process detailed in [PAPER\_MATIAS]. In order to analyze the algorithm performance, fifteen representative scene View Points were taken. For each position we compute the following metric: Value = (t - v) / t \* 100, where t is the total scene meshes and v is the total visible meshes.

This metric allows us to see the percent of discarded meshes that Occlusion Culling prevented from sending to the GPU in each frame. The metric is 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