# 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.

# Introduction

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 is imperative in order to manage that kind graphics complexity.

Frustum Culling is commonly used techniques to avoid rendering those meshes that are outside the viewing volume. These unnecessary 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 been block by others in front of them (occluders).

Unfortunately it doesn’t support to avoid rendering hidden objects been covered by others models in front of them. To solve this issue, Occlusion Culling techniques are required. 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 by means of a two steps procedure. Fist it draw the entire scenes 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 rejected the fragment with an occluded depth value. 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 and CPU reads back. The algorithm presents an earlier discard strategy than z pre-pass and it also does not restrict the pixel shader to write depth information. 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.

# Related work

Many techniques exist to solve the Occlusion Culling problem. A complete survey is presented in detail in [SILVA OH SENIOR].

Geometry-space techniques, like the one presented in [2 PAPER MATIAS], detect non visible objects by constructing a shadow frusta from a sets of Occluders, but they commonly does not support the property of Occluder fusion.

Pre-computed solutions like [4], creates in offline time a conservative visibility solution for the scene. Normally only indoors environments are supported, where discrete cells and connectors can easily be placed, and the pre-compute data cannot be easily updated in real time.

Occlusion Queries is a commonly feature in commodity GPU, which allows to detect if a mesh would finally result visible on screen. This feature brings a new set of tools to attack Occlusion culling, but there are still many difficulties to solve. Especially when trying to read a query result without suffering from latency, CPU stall and GPU starvation. Different techniques are proposed [5][6][7 DE PAPER MATIAS] to solve these issues.

The original Hierarchical Occlusion Culling presented in [CITAR HOM] has recently been adapted to take advantage of the present increase in multi core CPU architectures. The techniques presented in [PAPER\_LEA Y PAPER\_MATIAS] developed a strategy for Software Occlusion Culling

FALTA hablar de Patch Based Occlusion Culling y del otro paper similar.

Quizás también del blog de Nick darnell.