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

#### Related work

#### Hierarchical Occlusion Culling Zhang

#### 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.

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

**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

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