Fast Realistic Rendering: Occlusion Queries

Albert Gil Saavedra

April 2019

1 Occlusion Queries Plugin

1.1 Contents:

The Occlusion Queries Plugion contains:

- OcclusionQueries.h : Main header.
- OcclusionQueries.cpp: Main program you need to execute.
- OcclusionQueries.pro: Plugin .pro file.
- Shaders Folder: Contain the vertex and fragment shader used.
- Frustum.h: Frustum class header.
- Frustum.cpp: Frustrum class which includes functions used in the main program.

In order to guarantee the correct behavior of the Occlusion Queries plugin all theses files must be in the same folder, except for the shaders that must be inside the shader folder in the same directory.

1.2 User Instructions:

The Occlusion Queries plugin is implemented in LINUX using the Viewer. In order to compile the main program you need to:

1. Open the viewer: "./GLarenaPL"
2. Press *Space Bar* and select "Plugins" and "Open Plugin".
3. Select the "OcclusionQueries.cpp" or ".pro" file and open it.

The main program will start with the default mode. In order to change modes, or move the camera, use the following instructions:

- Right click + Mouse Movement: Translate the camera.
- Left click + Mouse Movement: Rotate the camera.

- Press o : Change to Occlusion Queries Mode.
- Press x : Change to Frustum Culling Mode.
- Press o or x depending on the mode to return no the Default Mode.

The mode is indicated in the lower left corner. In the upper left corner are indicated the fps, the total number of objects and the objects occluded(Occlusion Queries Mode) or the number of objects inside the frustum(Frustum Culling Mode).

It is possible to load other models by going into the "Scene" menu of the plugin (press *Space Bar* to show the menu). Also the number of objects(N) can be also changed by changing the integer variables *NumberOfCopies_x*, and *NumberOfCopies_z* that are located at the start of the .cpp file

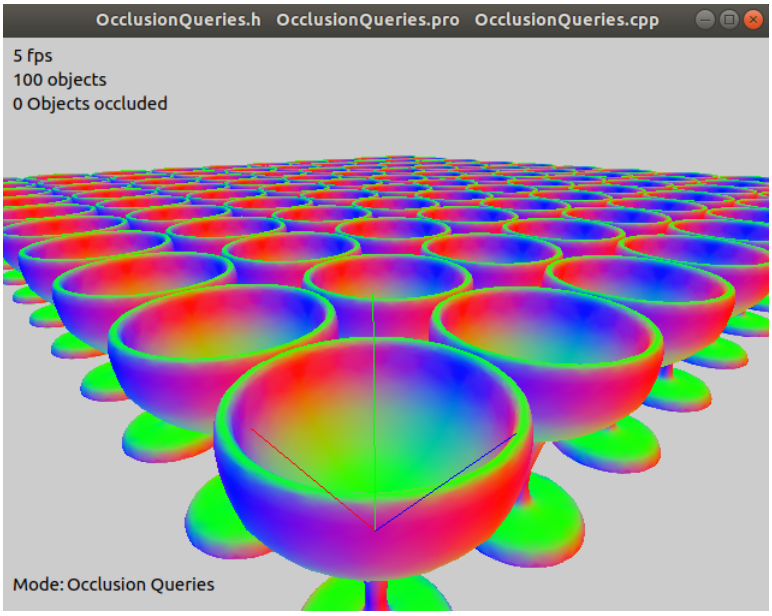

Figure 1: Visualization of the display of the plugin.

2 Implementation

2.1 General Features:

The objective of the program is to render N objects using occlusion queries, frustum culling or without applying any technique in order to compare the performance between them.

The main render loop and the implementation of these techniques are located in the drawScene() method, which uses the other functions[1] in order to render the objects. As we only render one type of object there is only one vertex buffer object and one vertex array object for the current object and another VBO and VAO for the bounding box of the object. In order to translate the vertices of the object and the bounding box we use the MVP matrix, which is translated every iteration of the render loop.

As it can be seen the bounding box implementation creates a VBO using a vertices of cube of length 1. This is because the transformation of the cube to the correct bounding box for each object is done in the vertex shader. These shaders are the same used for the object, but we use a boolean uniform to indicate if we are drawing the object or the bounding box. The fragment shader does not have a boolean option because we are going to render the bounding box with depth, draw and lighting disabled.

2.2 Occlusion Queries:

The occlusion queries implementation follows the next scheme:

1. Variable creation.
2. Disable depth, draw and lighting.
3. For each query/object:
   (a) Translate MVP matrix.
   (b) Begin Query.
   (c) Render the bounding box.
   (d) End Query.
4. Wait for the 3/4 of the queries to have the result.
5. Enable depth, draw and lighting.
6. For each query/object:
   (a) Translate MVP matrix.
   (b) Get query result.
   (c) Render the object if the result is different than 0.

As it can be observed the render loop is in fact two loops, in the first one we translate the object in the x direction and in the other we translate it in the z direction. For each object we render, we need to pass the MVP matrix to the shader to translate the object, and the boolean which indicates if we are rendering the object or the bounding box. As mentioned before the bounding

[1]The functions implemented to create the VBOs, VAOs and shaders are based in other plugins already created.

box shader part receive the bounding box maximum and minimum vertices just only once, and this translation of the MVP matrix is the one changing the position of it. Also there is a counter inside the rendering object part that allow us to know how many objects are rendering at the time.

2.3 Frustum Culling:

For the *Frustum Culling* implementation we used the *Frustum* class to calculate the six planes, which define the frustum, and to calculate if the Bounding Box of one object is inside the frustum or not.[2]

The implementation of the frustum culling follows the next scheme:

1. Compute frustum planes.
2. For each object
   (a) Translate MVP matrix.
   (b) Compute if the bounding box is inside the frustum.
   (c) If it is inside the frustum: draw the object.

The calculation to determine if the bounding box is inside the frustum is done by the *IntersecctionBBCenter* function defined in the *Frustum* class, and it returns a boolean variable which indicates if the bounding box is in the frustum or not. The methodology for this calculation is the following:

Using the six planes calculated previously and the center of the bounding box we calculate the distance between the center and the planes, if the distance is positive for all the planes, the center of the box will be inside the frustum, so the box will be in the frustum as well. In addition we have to take account of the boxes that are inside the frustum and the center is not inside, this is done by calculating the distance with the plane and compare it with the maximum distance of the bounding box, if it is less than this distance the bounding box will be also inside the frustum. Using the maximum distance from the center to the vertex we will draw some objects that are not inside because of the direction and position of his bounding box, but although we draw more objects that we need we will get an improvement in our performance.

As it is seen in the code we need to modify every time the vector defining the maximum and minimum vertices that we pass to the function to calculate if the bounding box is inside the frustum, because as it is defined in our program we only have one bounding box and it is translated through the MVP matrix as said before.

[2]The implementation of the calculation of the six planes is based on the article: Gil Gribb, Klauss Hartmann *Fast Extraction of Viewing Frustum Planes from the WorldView-Projection Matrix*

2.4 Comparison:

We have been able to prove that the *Occlusion Queries* and *Frustum Culling* techniques improve the performance of our program. But this improvement is not equal all the time, it depends on the camera position and direction, and the objects positions. We have to keep in mind that if there is no objects occluded or outside the frustum we will not be able to see any improvement in performance.

Here we will show some examples of how the performance improve. Note that all of them have been executed in a virtual machine using 100 objects, so the fps we have get are lower than the expected with a non-virtual machine.

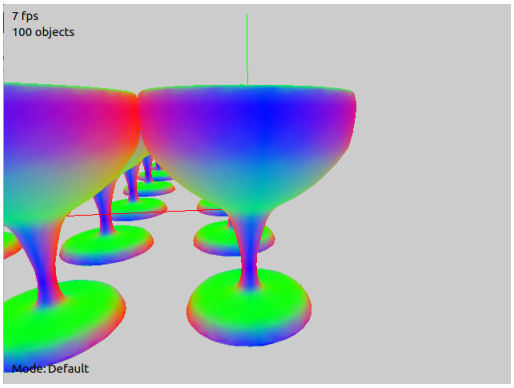

Figure 2: Default Mode, 7fps.

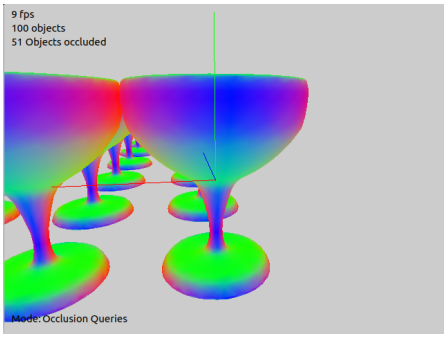

Figure 3: OQ Mode, 9fps, 51 objects occluded.

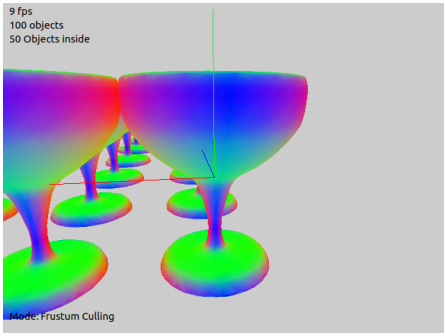

Figure 4: FC Mode, 9fps, 50 objects inside the frustum.

In these three picture we can see how half of the objects are occluded or outside the frustum in each case an the performance goes from 7 fps to 9 fps, which is almost an improvement of 30%.

Here what we have is that almost all the objects in the scene are occluded/not

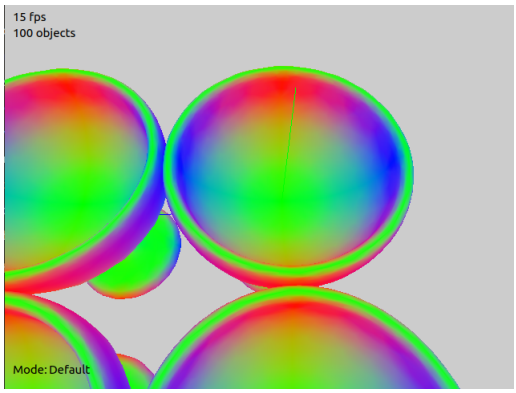

Figure 5: Default Mode, 15fps.

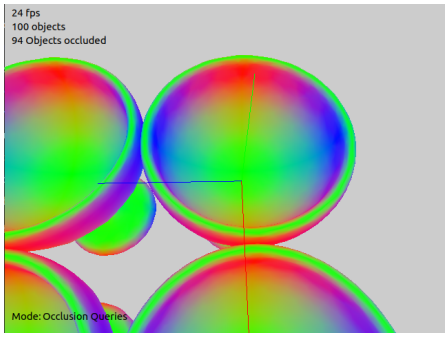

Figure 6: OQ Mode, 24fps, 94 objects occluded.

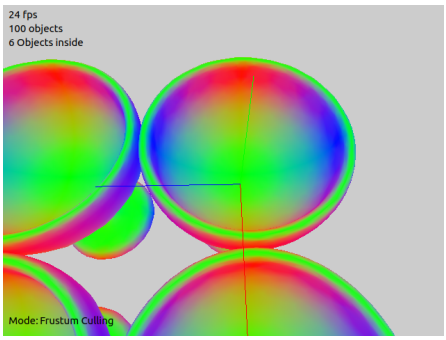

Figure 7: FC Mode, 24fps, 6 objects inside the frustum.

int the frustum, in each case we only draw 6 objects, and the performance goes from 15 fps to 24 fps, which is an improvement of 60%.

As we can see as we increase the objects occluded or outside the frustum in each case the improvement gets bigger. Moving the camera along the scene it is easy to check how the fps increase as we get more objects occluded or decrease as we get less objects occluded.

3 Conclusions

As we have seen the *Occlusion Queries* and *Frustum Culling* techniques improve the performance when there are objects in our scene that we do not need to render, as this number gets bigger the performance improvement is also bigger.For the implementations that we have used for both techniques, we have get similar results in the same position of the camera.

One step further in order to improve the performance may be to include

both of them in a single render loop and compare if the combination of both will get us even better results than just applying one of them.