Fast Realistic Rendering (FRR) (2023), 1–5

LAB ASSIGNMENT 2

Visibility

Student Name: Evripidis Avraam

1. Introduction

In this project, I will be exploring the significance of visibility techniques in OpenGL, which are vital for achieving efficient rendering in real-time computer graphics applications. The scene is limited by the amount of geometry processed in the GPU. The utilization of default rendering (no optimization), frustum culling, occlusion queries, and the evaluation of shading models such as Gouraud and Phong will shed light on their impact on rendering performance.

1.1 Compile and Run (Ubuntu Linux)

To compile and run the project in an Ubuntu Linux environment, use the following commands in the Terminal:

1. cd [downloaded project folder].../BaseCode/
2. mkdir build
3. cd build
4. cmake ..
5. make
6. ./BaseCode

Disclaimer: In case the project loads more meshes than the computer can handle, change the following lines of the Scene.h file:

- int modelCopies = [some-acceptable-number];
- The 4 float arrays right after "modelCopies" to contain [3 * some-acceptable-number] elements.

2. Visibility Techniques

2.1 Frustum Culling

Camera frustum represents the zone of vision of a camera. Without limits, we have an infinitely expanding pyramid in front of the camera. **Frustum culling** is a visibility optimization technique, which sorts visible and invisible elements, and renders only the visible ones. It can be useful to avoid the computational overhead of things that are not visible.

To apply frustum culling, the world-space camera frustum planes are computed. With these, we can check if an object is inside or outside the frustum by testing the frustum planes against the scene's models' **Axis-Aligned Bounding Box's (AABB)** corners. We typically use a bounding volume to test for frustum culling, since it is a time-efficient rough approximation of our model's mesh.

2.2 Occlusion Culling

Occlusion Culling is a feature that disables rendering of objects, when they are not currently seen by the camera, because they are obscured (occluded) by other objects.

The term **occlusion culling** refers to a method that tries to reduce the rendering load on the graphics system by eliminating (*i.e.*, culling) objects from the rendering pipeline if they are hidden (*i.e.*, occluded) by other objects. There are several methods for doing this.

In this project, I apply a simple occlusion query method, following these steps:

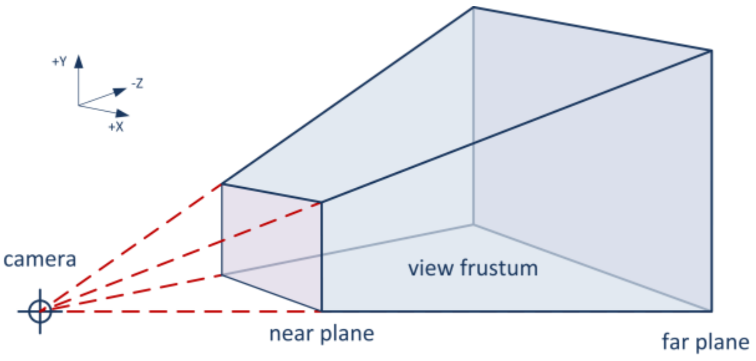


Figure 1. Camera frustum

1. Initiate an occlusion query.
2. Turn off writing to the frame and depth buffer.
3. Render a simple but conservative approximation of the complex object (AABB).
4. Terminate the occlusion query.
5. Ask for the result of the query (that is, the number of visible pixels of the approximate geometry).
6. If the number of pixels drawn is greater than some threshold (typically zero), render the complex object.

This method works well if the tested object is really complex, but step 5 involves waiting until the result of the query actually becomes available, resulting in potentially large delays.

2.3 Shading Models (Phong, Gouraud)

Phong and **Gouraud shading** are two different methods used to compute the shading of polygons. Gouraud shading, also known as smooth shading, computes the color intensity of a polygon at a vertex level (vertex shader), and interpolates those values across the polygon's surface. On the other hand, Phong shading is a more sophisticated technique, that computes the shading at a pixel level (fragment shader) on the surface of a polygon. Depending on the application and the result one wants to achieve, one could be more efficient than the other. In this project, we only compare their result in terms of rendering time (frames per second - fps).

3. Experiments

3.1 Setup

The timing setup is shown below in Table 1. The camera starts on an original position in front of the grid, on a higher level on the y-axis, looking at the whole scene. It starts going downwards on the y-axis, until it reaches the same y-level as the meshes. Then, it moves towards the mesh grid, reaching a point inside the irregular grid. The next 10 seconds, it moves along the (-x)-axis (left) for 5 seconds and the x-axis (right) for another 5 seconds. Afterwards, it moves "backwards" to reach the original position, on the mesh's y-level, and the last 3 seconds "upwards" on the y-axis to watch the whole grid again.

For better understanding of what is happening during execution, I have provided the possibility to render the AABBs of the meshes. The rendered meshes' AABBs are green and are on the size of the meshes, while the occluded meshes' AABBs are intentionally streched on the y-axis to be visible when the actual meshes are occluded.

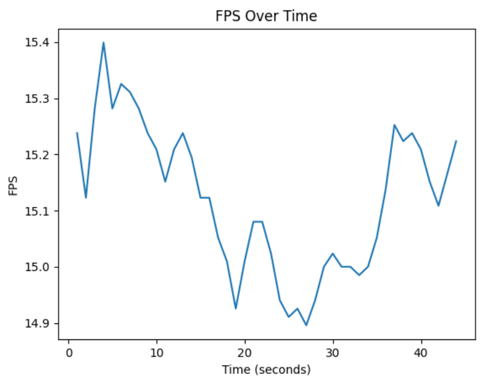
Disclaimer: The experiment was conducted on a computer with **144fps** on a none-mesh-rendering state, and during the renders, the total number of the meshes rendered (Stanford bunny) were 256. Also, due to technical issues (logical bugs in my code), I was unable to set a specific path for each experiment and export it. Instead, I used a time-counter and moved the camera based on pre-defined checkpoints.

Table 1. An Example of a Table

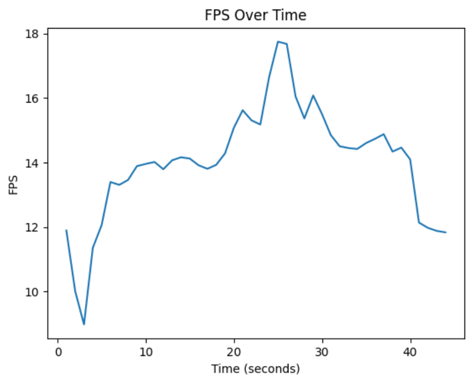
Time (s)	Viewing position	Frustum	Occlusion
0 - 5	Origin, High level	Whole scene	None
5 - 20	Origin, Same level	Front part of scene	Low
20 - 25	Among meshes, mid towards left	Variable	High
25 - 30	Among meshes, left towards mid	Variable	High
30 - 42	Among meshes, backwards to Origin, same level	Variable	High/Low
42 - 45	Origin, upwards to High level	Whole scene	Low/None

3.2 Results - Plots

The following plots illustrate the results measured in fps in the different scenarios. They appear in couples of rendering techniques ("Default" and "Occlusion Culling"), for Frustum Culling enabled/disabled using Phong and Gouraud shading.

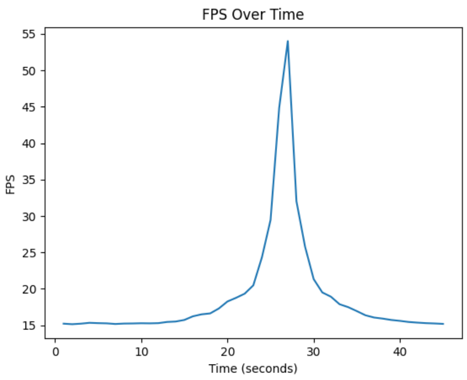


Default rendering

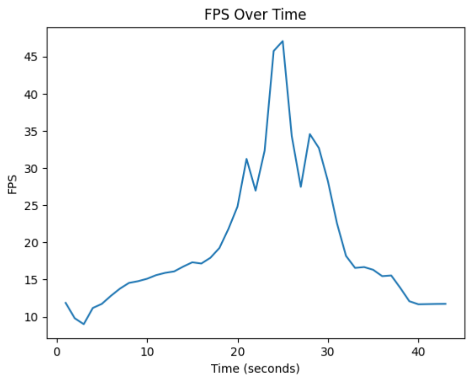


Occlusion Culling rendering

Figure 2. Phong Shading (**without** Frustum Culling)

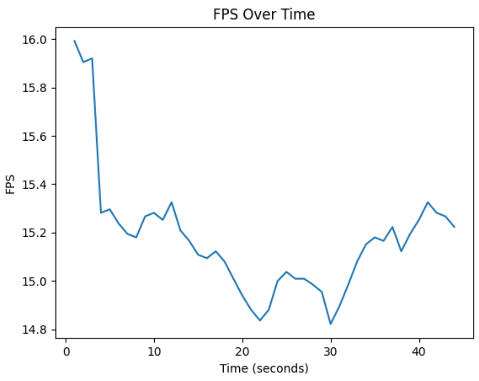


Default rendering

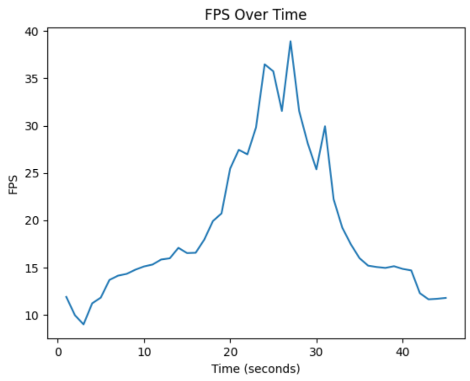


Occlusion Culling rendering

Figure 3. Phong Shading (**with** Frustum Culling)

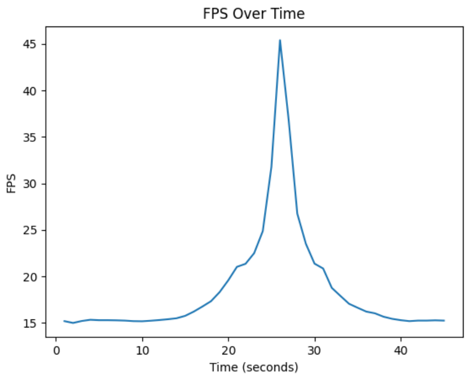


Default rendering

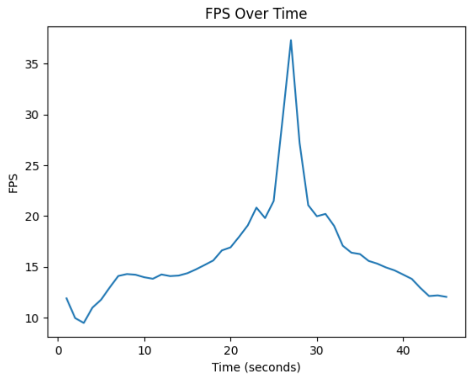


Occlusion Culling rendering

Figure 4. Gouraud Shading (**without** Frustum Culling)



Default rendering



Occlusion Culling rendering

Figure 5. Gouraud Shading (**with** Frustum Culling)

3.3 Results - Analysis

From the plots (Figures 2 - 5), we can notice the following:

- Frustum Culling improved the performance in every scenario. Its simplicity in applying it suggests that it should always be used in applications like this one.
- In the cases that Frustum Culling was OFF, Occlusion Queries were **outperforming** the Default rendering. We can see that when the whole scene was rendered, the Occlusion fps were **lower** than the Default rendering, due to the extra computational overhead in the CPU from the Occlusion Query method I applied. However, when the camera was among the meshes, the performance gain is noticeable.
- In Gouraud shading, the gain of Occlusion Queries is much higher than its Phong counterpart. In general, the shading method is not directly related to the Occlusion Culling performance. However, Phong shading tends to be more computationally expensive than Gouraud, because it requires more calculations per pixel. So this could be the reason for the results I got.

4. Conclusions

In this section I will summarize the main conclusions of this project:

- By applying culling techniques to a scene bounded by geometry processing in the GPU, we can achieve significant performance improvements. In my scene, which was densely populated by complex meshes, in most cases I noticed peaks over 30 fps, achieving "real-time" performance.
- More sophisticated Occlusion Query techniques (*e.g.*, Stop and Wait) are able to provide much better results than my simplified version of Occlusion Culling. However, as already noticed in the result analysis, Occlusion Queries need to be handled with caution, since they could in fact produce worse results in low-occlusion scenarios (*e.g.*, rendering a sparsely populated scene).
- Regarding Occlusion Culling, modern methods can also take advantage of more complex techniques, such as considering an object to be visible/invisible for multiple frames. As a result, we could get even better performance using less conservative algorithms.

5. Issues Encountered

In this section, I will summarize the issues I encountered during the development of the project and the experiment:

- First of all, the lack of a hard-coded path induces a marginal, yet existent, error in the rendering timing. However, given the minor fps changes I got per frame, this error could be slightly ignored, since the resulting plots provide sufficient data for comparison.
- Secondly, while I developed functions for camera path recording/exporting-as-file, and for better FPS recording, there was some implication on the ImGui part of these functions. As a result, I have left the ImGui UI and the code for the path recording, but it is currently not working.
- Finally, I tried to implement a Coherent Hierarchical Culling (CHC) optimization, by partitioning the world space in a Quadtree fashion, but in the end it wouldn't work properly and decided to leave it outside the rendering options.