FRR Visibility Project

Guillem Pérez Delgado

June 2021

1 Introduction

In this project, I will be exploring the increase in performance that can be obtained by applying culling strategies to a scene that is limited by the amount of geometry processed in the GPU. In particular, the two strategies that I am going to explore are frustum culling and occlusion culling via GPU occlusion queries.

1.1 Frustum Culling

At each frame, the world-space viewing frustum planes are computed. With these, we can check if an object is outside the frustum by looking for the existence of a frustum plane that leaves all the vertices of the bounding box of that object on the outside. If there is no such plane, we assume that the object might be potentially visible.

Note that this is just a conservative algorithm [2] since it is possible for an object to be outside the frustum and no such plane exist, for example if the object covers one of the corners of the frustum. Nevertheless, I didn't observe this happening in the implementation because it is an issue that appears specially when objects are big compared to the viewing frustum, which is not the case.

1.2 Occlusion Culling

For performing the occlusion culling via GPU occlusion queries, I have considered four different approaches that will later be compared. All of them have

been designed to be conservative. Sorted in increasing order of complexity, these four approaches are the following:

- None (N): Occlusion culling is not performed.
- Stop and Wait (SW): Simple usage of occlusion queries following the stop and wait approach.
- "Advanced" (A): A more advanced usage of occlusion queries that takes into account the visibility of the previous frame. Objects are rendered in a front to back order.

  If an object was visible the previous frame, the object is directly drawn and an occlusion query using the exact geometry is issued, this occlusion query will only need to be resolved at the beginning of the following frame.

  If the object was not visible the previous frame, the object is not drawn and an occlusion query using its bounding box is issued. However, we do not wait for the result of it and we keep rendering objects. In order for the final image to be correct, these occlusion queries have to be resolved before the end of this frame, and in case it is determined to be visible, the object has to be rendered.

  Given that we temporarily do not draw this objects that might still be potential blockers, instead of checking for the results of the queries after drawing all the scene, we interleave the drawing of objects with checking the availability of the queries result. This way we draw them as soon as we know they are visible and thus provide a better occlusion for the rest of scene.
- Coherent Hierarchical Culling (CHC): Simple implementation of the CHC approach as described in [1]. Since for the scene I am considering the quadtree is full, I have implemented it as a simple array of nodes where the children of a node `i` are nodes `4*i + 1` to `4*i + 4`.

Any of the four strategies might be combined with frustum culling to further improve performance.

2 Experiments

2.1 Setup

For the experimental part, I have designed a 45 seconds path through the scene that captures a wide variety of situations regarding visibility. Using this path, I

have measured the frame rate along it at intervals of 250ms and using the eight possible rendering algorithms that result from the combination of one occlusion query strategy and frustum culling enable/disabled.

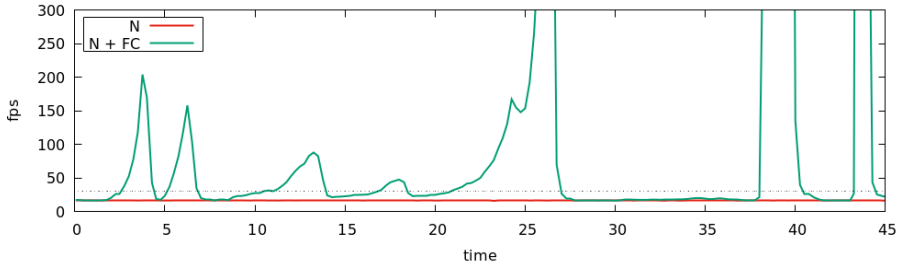
The following table, describes the sections in which the path can be roughly decomposed.

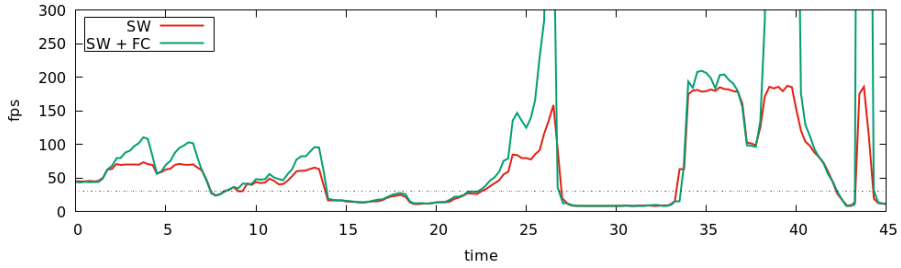
Time	View from	Frustum	Occlusion
0-8	Corner	Whole scene/Few objects	High/Low
8-27	Through Scene	Variable	High
27-33	Above	Whole scene	None
33-38	Below	Whole scene	Full
38-40	Turn around	Empty	-
40-42	Transition from above to below	Whole scene	Variable
42-44	Turn around	Empty	-
44-45	Above	Almost all scene	None

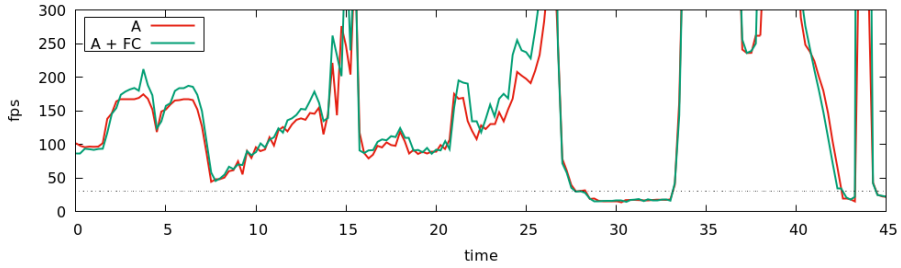
2.2 Results

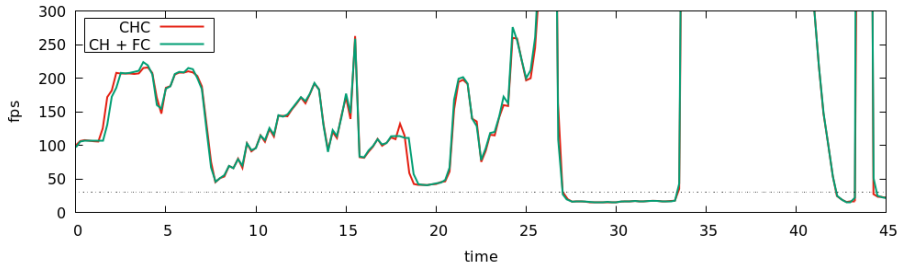
I have summarized the data gathered from the previous setup in the following plots, in all of them the 30 fps threshold is marked with a dotted line as baseline for a realtime experience.

First, let's compare each occlusion culling strategy with and without frustum culling enabled.



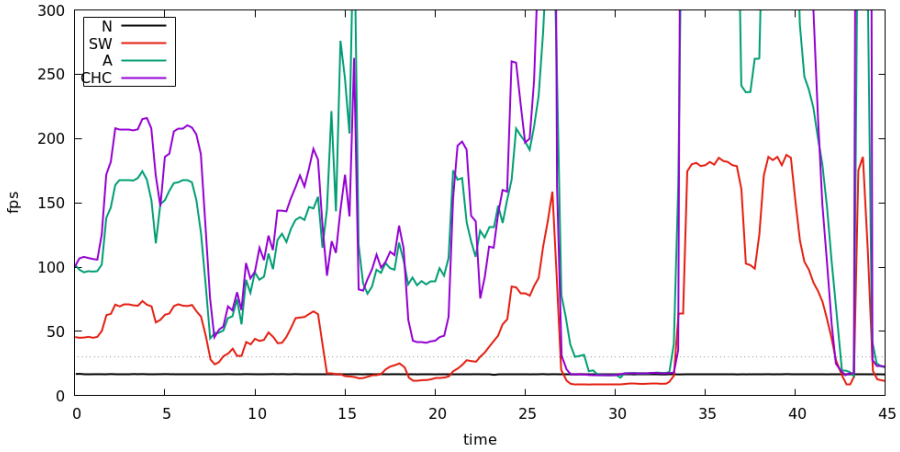


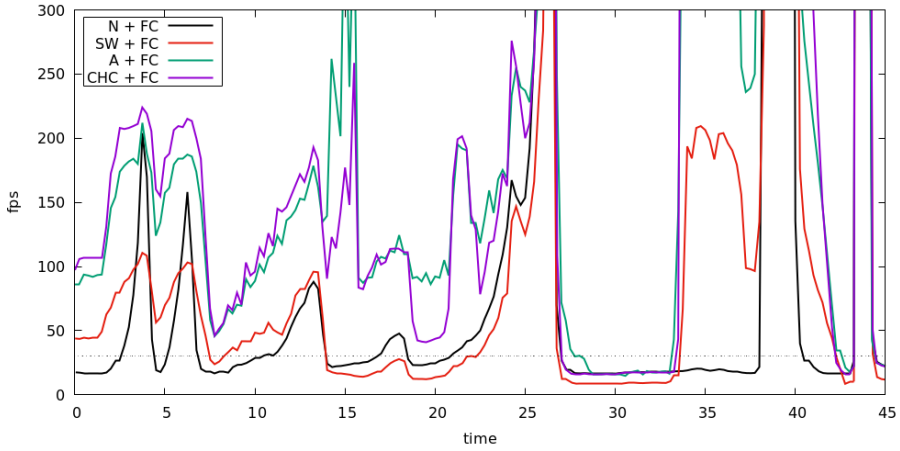




As expected, the variants that use frustum culling outperform the ones that don't. Interesting to note, however, that it makes almost no difference for the CHC algorithm.

Now, for a more interesting comparison, let's compare the different occlusion culling strategies, the comparison is done both with and without frustum culling enabled.





In this plots, we can observe how both A and CHC are always able to deliver at least as good frame rates as the other alternatives and in some favorable situations the difference is measured in hundreds of fps.

And not only that, with the exception of the sections of the path where the whole scene has to be rendered, they are both able to maintain frame rates higher than 30. Whereas both N and SW fail at doing so in other situations.

An important take away of these plots is how badly SW performs when there is little occlusion in the scene, performing worse than N in several situations.

2.3 Conclusions

In this section I will summarize the conclusions I obtained from working in this project:

- By applying culling strategies to a scene bounded by geometry processing in the GPU, it was possible to optimize an application that was delivering constant 15 fps, to one that, with the exception of some extreme cases, is able to deliver a frame rate above 30 fps, thus providing a realtime experience.
- Frustum culling has proved to be useful in all the situations, introducing low overhead and great improvements in the frame rate. And not only that: it is also easy to implement, so it should always be applied if possible.
- Occlusion queries are able to provide performance improvements greater than those that frustum culling is able to deliver. However, performing a simple SW implementation is not enough and can worsen performance in low occlusion situations.

  Improving the usage of occlusion queries with some simple optimizations like using the information of the previous frame can make occlusion queries much more efficient, without being very hard to implement.

  Other more aggressive optimizations that have not been explored in this work (such as considering and object to be visible/invisible for several frames) might further improve the performance at the cost of losing the conservativeness of the algorithm.

References

[1] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," *Computer Graphics Forum*, 2004, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2004.00793.x.

[2] I. Quilez. (2013). "Fixing frustum culling," [Online]. Available: https://iquilezles.org/www/articles/frustumcorrect/frustumcorrect.htm. (accessed: 12.06.2021).