Term project number 17
Hierarchical View-Frustum Culling for Z-buffer Rendering

Aleš Koblížek[1]
Department of Computer Graphics and Interaction,
Faculty of Electrical Engineering, CTU in Prague

Abstract

Implement hierarchical view frustum culling for large scale scenes consisting of triangles. First, construct a bounding volume hierarchy (BVH) using top-down method, middle point subdivision. Avoid rendering such BVH nodes that cannot be visible (out of viewing frustum) usually known as view frustum culling.

Keywords: view frustum culling, bounding volume hierarchy, middle point subdivision

1. Introduction

View frustum culling is the process of identifying objects (or triangles) that lie outside the view frustum so they can be removed from the rendering pipeline. This reduces the load of the pipeline stages up to and including clipping. The goal of this project is to create an efficient implementation of a frustum culling algorithm for large static scenes, utilizing a bounding volume hierarchy, where the bounding volumes are axis-aligned bounding boxes.

2. Algorithm Description

2.1. Building the BVH

First, the BVH must be built. The top-down method is used, which means that all triangles are put into a single node, the root, and the nodes are recursively split in two by dividing the triangles inside the node into two groups, according to chosen scheme. Each group is then passed to the newly created child node. This process is repeated until each node contains fewer triangles than specified threshold.

The chosen subdivision scheme is midpoint subdivision [3], which first calculates a midpoint and extents from triangle centroids along each axis and chooses the splitting axis as the one with the largest extent, thus forming a splitting plane, which is perpendicular to this axis and contains the midpoint. Then it divides the primitives into two groups based on the position of their centroids relative to the splitting plane.

The shape of the tree is not known in advance, so it is easier to use the dynamic representation where each node is allocated separately and the children are linked by pointers. This is however not suitable for traversal because the nodes are not stored on subsequent memory locations, so the they can not be loaded into cache in advance. Also the pointers waste space

[1]B4M39DPG – Aleš Koblížek, summer 2018/19

in cache. To avoid this, the tree is transformed into an array representation, where left child is always the following node in the array and the index of the right child is stored inside the node. Therefore the order of nodes in the array corresponds to the DFS pre-order traversal.

2.2. Hierarchy traversal

By using a hierarchical data structure it is possible to lower the computational cost of culling when high accuracy is not needed or when the objects (meaning their bounding volume) are well aligned with the frustum. The hierarchy is traversed using DFS pre-order. Whenever a node is fully inside the frustum, there is no need to test its children and triangles of all leaves in this subtree can be sent for rendering.[2] When a node is fully outside the frustum, all of its children will be outside as well so this subtree can be skipped. If the node intersects the frustum and it is not a leaf, it might still be possible to eliminate some triangles by recursing down the tree and doing more tests. In case the intersecting node is a leaf, all its triangles have to be rendered since it does not pay off to test every single triangle. Therefore the number of triangles in leaves should be suitably chosen.

2.3. Frustum - box intersection test

One approach to this problem is to transform the frustum into perspective coordinate system, therefore it becomes a box. Each bounding volume also has to be transformed, in case of AABB it means to transform all eight vertices and construct their AABB. Thus the problem becomes an intersection test of two AABBs, which can be solved by six comparisons. This approach pays most of its price on the vertex transformations, leaving no room for optimizations.

Second approach, the one used in this project, is to test the box against each of the frustum planes, either in world space or in model space.[3] It offers more room for optimizations, which will be described in further sections. More detailed description of both methods can be found in [1]. The planes in world-space can be extracted from view-projection matrix by an algorithm described in [2]. Analogically, the planes in model-space can be extracted using the exactly same algorithm from model-view-projection matrix.

2.4. Efficient plane - box intersection test

Exact test cosists of testing position of eight points against a plane, meaning eight dot products and comparisons. There is a more efficient conservative algorithm, which can be used instead.[4] The idea is that it is sufficient to test the two vertices that lie on the diagonal which is best aligned with the normal of the plane. They can be found by using the binary encoded signs of components of the plane normal vector as index into a lookup table. The vertex farther along the normal is called *p-vertex*, the other one *n-vertex*. If the *n-vertex* is inside the plane, then the box is inside. Else if the *p-vertex* is inside the plane, then the box is intersecting the plane. Otherwise it is outside the plane. The indices of the *p-* and *n- vertices* are the same for a given plane and all AABBs.

[2]Therefore it is beneficial to sort the triangles in order corresponding to the order of leaves in DFS traversal – triangle ranges of siblings will be adjacent and their union will form a continuous range.

[3]The tests must be done in model space if the object transformation changes, so the BVH does not have to be rebuilt. Also if there are multiple objects with different transforms, the same BVH can be used for all of them.

[4]A conservative algorithm can be used, because when the result of the test is *intersecting* instead of *outside*, the only consequence is that the children will be tested as well.

2.5. Culling optimization techniques

There are several possible optimizations, as described in [1]. *Plane masking* exploits the hierarchical structure – when a node is inside a plane, all the child nodes must also be inside that plane, therefore the tests against those planes can be skipped. *Plane coherency* tries to end the testing of box against set of plains earlier by starting with the plane the box was outside of last time. *Octant test* is usable only for symetric frustum (meaning horizontal and vertical field of view is the same) and only if the radius of the bounding sphere of a node is not greater than the smallest distance from frustum center to any of its planes. It works by splitting the frustum into eight symetric octants, identifying the octant that contains the centroid of the node bounding sphere (or bounding sphere of the AABB, which may give a worse fit) and testing only against the three outer planes of that octant.

3. Implementation details

The most annoying part of the project was getting it to compile on Windows. It took me around three hours to setup the project with all the dependencies and resolve name collisions with Windows constants and macros. Also on Windows some destructors were not called after closing the application window, which was solved by passing an option to GLUT to not terminate after the window is closed but rather only exit the main loop. The most tricky to find programming error was switched function arguments. Worse than that was a false assumption about the *octant test*, that the planes it eliminates from testing can also be masked out from the child nodes. This is not true, because the bounding volume can also intersect those, meaning the children can intersect them as well.

The total time spent on the project is likely around 100 hours, including this report, but not counting the preparation of presentations. I had some code to start with from a previous project, but it had to be refactored and quite extensively so starting from scratch would likely not add much time. The evaluation (measurements, processing of the data, creating graphs) took significant portion of the time spent on the project – my estimate is 15 to 20 hours. I had to repeat it several times, first because I started measuring in debug mode, then because of errors in command line arguments passed to the program and then again because the scenes used were found to be too small.

4. Results

All the measurements are done on the scenes listed in table 1. For each scene a single prerecorded camera route is used, defining the position and orientation of the camera. The routes start such that the entire scene is visible, continuing through the scene, lowering the amount of visible geometry down to zero, where they end. During measurement, this route is steped through with equally sized steps and the statistics are recorded. Each twenty consequent records are then averaged.

First, a suitable maximum number of triangles per leaf (*MTPL*) has to be chosen. If it is too high, fewer nodes will be created and the culling will be faster, but less accurate and therefore more excess triangles will be sent for rendering, increasing the draw time. Figure 1 demonstrates that. Based on it, the *MTPL* value was chosen to be 10^4. Different values might be more suitable if the CPU-GPU power balance is different. For instance if the GPU is much weaker than in this case, it might be beneficial to use lower *MTPL* value, therefore

Scene num.	Scene name	# tris.	Draw time $[ms]$	# nodes
1	part_of_pompeii	28.0 m	5.5	8803
2	PowerPlantM	12.7 m	3.0	4623
3	asianDragon	7.2 m	1.6	2221
4	ten_blocks_in_pompeii	5.6 m	1.1	1783
5	City4M	4.7 m	0.9	1135
6	block_in_pompeii.high_lod	3.1 m	1.2	1137
7	vienna_cropped	0.9 m	0.4	283

Table 1: List of scenes used for testing. The given number of BVH nodes is for $MTPL = 10^4$.

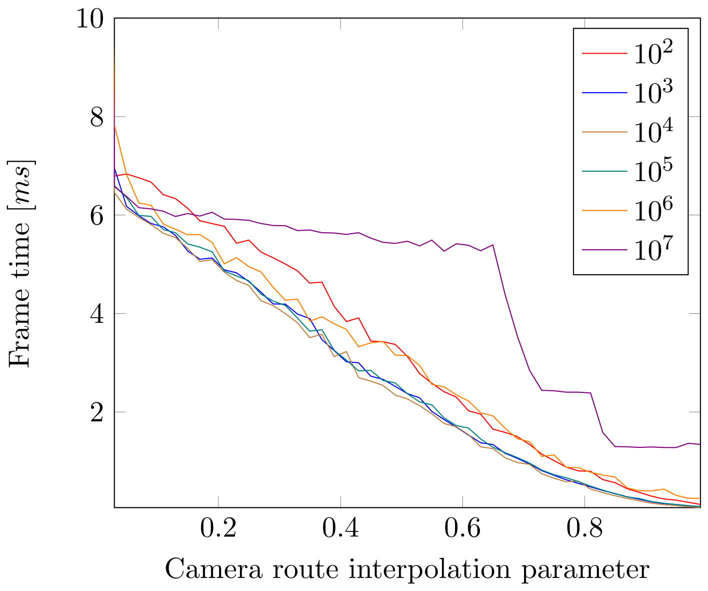

Figure 1: Frame time over the same camera route for various *MTPL* values, measured in scene 1

the result of the culling will be finer – fewer triangles will be sent for rendering, but with higher CPU and memory cost.

Next, the culling optimization techniques will be compared. Figure 2 shows comparison of culling time for different versions of the algorithm on the same camera route in scene 1. The *MTPL* value was set to 100 to increase the number of generated nodes and therefore improve the accuracy of measurements of the traversal efficiency. According to the obtained results, the optimal combination of optimization techniques is *plane masking* and *octant test.* It might be worth to further examine the slowdown caused by *plane coherency.*

Now the frustum culling algorithm with optimal parameters will be evaluated on all the scenes listed in table 1. The *MTPL* is set to 10^4 and only *octant test* and *plane masking* optimizations are enabled. Figure 3 shows the frame time speedup depending on the percentage of visible geometry. Note that the speedup is quite consistent over all of the scenes, only on scene number 5 it is slightly lower.

Table 2 contains the software and hardware configuration that was used for the mea-

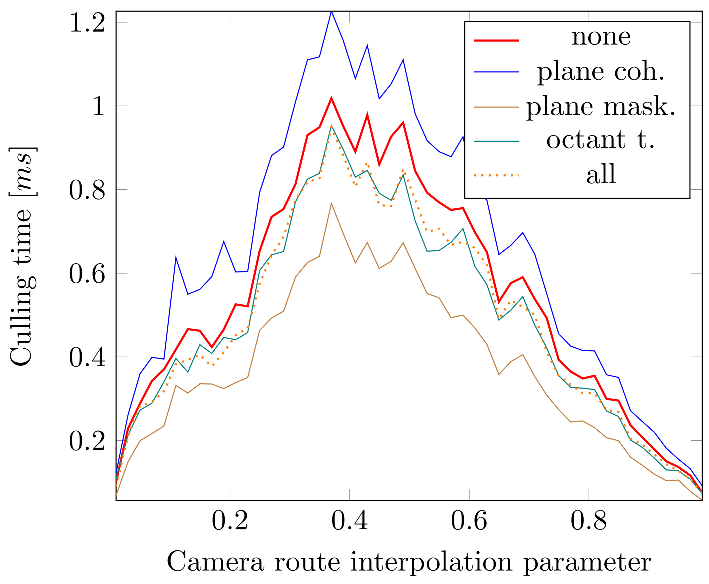

Figure 2: Comparison of culling optimization techniques – culling time along camera route in scene 1. Each of the techniques is measured individually and then all of them combined. Culling time without optimizations is also included.

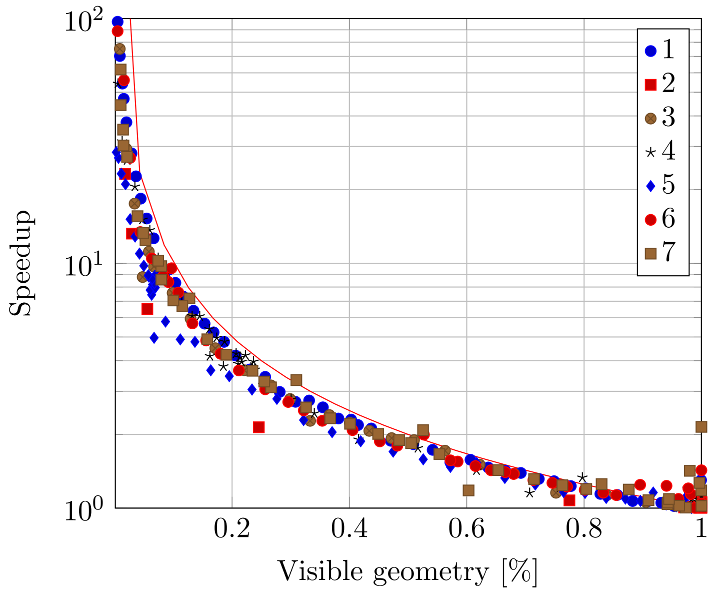

Figure 3: Frame time speedup obtained by frustum culling ($MTPL = 10^4$, *octant test*, *plane masking*) depending on the percentage of visible geometry. Measured on all scenes, numbered as in table 1 (according to the amount of geometry). The red line shows the optimal speedup: $\frac{1}{percentage\_of\_visible\_geometry}$.

CPU	Intel Xeon E3-1231 v3 @ 3.4 GHz, 8 MB cache, 4 cores, 8 threads
RAM	16 GB
GPU	Nvidia GeForce GTX 1070 Ti
Operating system	MS Windows 10
Compiler	MSVC 2017
Architecture	amd64
Compiler options	O2 (maximize speed), inline function expansion, enable intrinsic func.

Table 2: Hardware and software configuration used for measurements.

surements. The implementation uses a single thread. Vertical synchronization was disabled during testing. Back-face culling was disabled as well, which might be important to know for comparing the results with another implementation.

5. Conclusion

A frustum culling algorithm has been implemented, including a simple method for BVH construction. A consistent speedup over all of the tested scenes was achieved, which is almost inversely proportionate to the percentage of culled geometry. On scene number 5 the speedup is slightly lower, which may be worth examining. As part of the implementatin, three optimization techniques for frustum culling were described and evaluated: *plane masking*, *plane coherency*, *octant test*. From the obtained results it was found that *plane coherency* results in a slowdown, perhaps due to suboptimal implementation – the reason should be investigated.

References

[1] Assarsson, Ulf, and Tomas Moller. ”Optimized view frustum culling algorithms for bounding boxes.” Journal of graphics tools 5.1 (2000): 9-22.

[2] Gribb, Gil, and Klaus Hartmann. ”Fast extraction of viewing frustum planes from the world-view-projection matrix, 2001.” URL http://www. cs. otago. ac. nz/postgrads/alexis/planeExtraction. pdf (2004).

[3] Pharr, Matt, Wenzel Jakob, and Greg Humphreys ”Physically Based Rendering: From Theory To Implementation” [online]