Parallel Rasterization, Z-Buffering, and Occlusion Culling

Umair Shahzad and Zachary Susskind

Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, TX 78712
{ushahzad, zsusskind}@utexas.edu

Abstract. In computer graphics, rasterization is an algorithm which converts objects from vector format to a set of pixels. In this project, we implement several variants of rasterization in a multicore CPU environment. We also investigate a technique for culling hidden polygons as an optimization.

1 Introduction

In graphics processing, it is usually more efficient to store objects as sets of geometric coordinates (in *vector* form) rather than as sets of pixels (in *raster* form). For instance, a straight line segment can equivalently be defined by every pixel composing it, or just by its two end points. However, eventually vector objects must be converted into raster form for display. The process is known as *rasterization*[Wor95].

Rasterization consists of two major operations:

1. Determining which pixels are interior to each object in a scene
2. When objects overlap, resolving which one should be drawn. This is known as *z-culling*.

In this project, we investigated the implementation of a parallel algorithm for rasterization of triangles. While GPUs provide hardware acceleration for

rasterization, we chose to create a multi-CPU, pure software implementation. We also implemented an algorithm for occlusion culling, which detects and skips the rasterization of polygons which are fully hidden behind other objects.

2 Proposal

2.1 Basic Rasterization

Our rasterization algorithm is based on the technique described in [Pin88]. This algorithm is built around the *edge function*:

$$edgeFunction(\vec{a}, \vec{b}, \vec{p}) = (p[1] - a[1])(b[2] - a[2]) - (p[2] - a[2])(b[1] - a[1])$$

Note that this equation is equivalent to the z-component of $(\vec{p} - \vec{a}) \times (\vec{b} - \vec{a})$. Let P be the projection matrix which maps a point in $\mathcal{R}^3$ onto the XY plane, and suppose $\vec{p}$ lies on the XY plane. Then $edgeFunction(\vec{a}, \vec{b}, \vec{p})$ is negative if $\vec{p}$ is to the left of $P(\vec{b} - \vec{a})$, positive if $\vec{p}$ is to the right, and 0 if $\vec{p}$ lies directly on the projected vector.

For a triangle ΔABC with clockwise-ordered vertices and point $\vec{p}$ on the XY plane, $\vec{p}$ is in $P\Delta ABC$ if and only if $edgeFunction(A, B, \vec{p})$, $edgeFunction(B, C, \vec{p})$, and $edgeFunction(C, A, \vec{p})$ are all non-negative. This gives us a way to test if a pixel lies inside the projection of the triangle on the screen.

This algorithm can clearly be run in parallel for multiple pixels. However, this is a very fine granularity of parallelism; while it is easily done in dedicated hardware, overhead on CPU is likely to dominate any benefits except perhaps for very large triangles. We chose to instead parallelize across different triangles.

Testing whether every pixel is interior to a triangle is inefficient. A better approach only tests pixels in a two-dimensional bounding box around the triangle, defined by the min and max x and y coordinates of its vertices. This

may still test many pixels outside the triangle if it has a long diagonal leg. A third approach, proposed in [Pin88], interpolates a near-vertical line through the triangle and explores horizontally away from this line. This approach tests the fewest pixels, but introduces significant complexity in traversing the triangle.

2.2 Z-Culling

Z-culling tracks an associated *depth* for each pixel in a scene. The z-coordinate of each pixel in a triangle is computed using its barycentric coordinates, which are the normalized results of its edge functions. Multiplying these with the vertices, summing, and taking the z-component gives the depth of the pixel [Scr15]. If the depth of the pixel is less than its z-buffer entry, the triangle is part of the foreground at that pixel.

Since we rasterize triangles in parallel, there is a potential race condition if two threads test the same pixel at the same time. We expect this contention to be uncommon, so we use a lock-free update model based on atomic read-modify-write operations.

2.3 Occlusion Culling

We implemented the Hierarchical Occlusion Map (HOM) technique described in [Zha98], which seeks to cull fully occluded objects before rasterization. An object is occluded if it is fully overlapped by other objects, and if those other objects are strictly in front of it. These conditions can be checked with an overlap test and a depth test respectively.

Overlap Test We use downsampled occlusion maps to check for overlap. An occlusion map is a transparency matrix, where 1 represents a completely opaque region and 0 represents a completely transparent region. The HOM algorithm

constructs progressively lower-resolution maps by recursively averaging 2x2 pixel blocks. To determine if an object is occluded, we check whether its bounding box is fully opaque in the lowest-resolution map. We progressively check higher-resolution maps until either the box is fully occluded in a map or all maps have been checked. Overlap estimates are conservative and do not produce false positives.

Depth Test We use a low-resolution (64x64) depth buffer to check if overlapped objects are occluded. We say an object is occluded if all pixels in the low-resolution depth buffer have a depth less than its minimum depth. This is also a conservative estimate.

2.4 Methodology

We generated scenes consisting of random triangles with specified number and average area. Our rasterizer was written in C++ using OpenMP for multithreading and the std::atomic library for lock-free updates. We used a simple shader to produce bitmap outputs; see Figure 1 for an example.

We then swept across 2520 configurations, as shown in Table 1, recording the rasterization time for each configuration.

Parameter	Values
Number of Triangles	2^8, 2^{11}, 2^{14}, 2^{17}, 2^{20}
Average Triangle Size	10^{-6}, 10^{-4}, 10^{-2}
Output Image Resolution	256, 512, 1024, 2048
CPUs	1, 2, 4, 8, 16, 32, 64
Raster Modes	All Pixels, Bounding Box, Interpolated
Hierarchical Occlusion Mapping	Disabled, Enabled

Table 1: Run parameters, the permutations of which were swept across

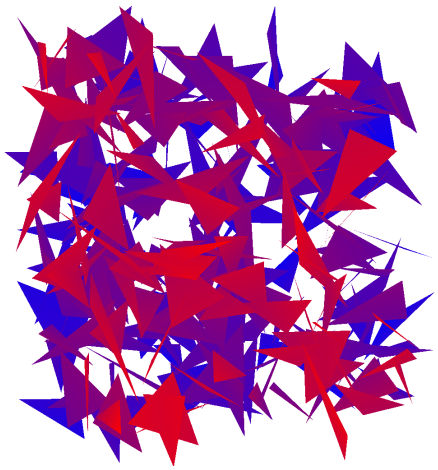

Fig. 1: Sample image with 256 input background triangles and average triangle area of 0.01 (where the area of the screen is 1.0). Blue pixels are further from the viewpoint; red pixels are closer

2.5 Alternatives

Ray Tracing Ray tracing is a method of rendering scenes which involves modeling the behavior of simulated light rays [App68]. Ray tracing produces results with high visual fidelity, but was historically too computationally intensive for real-time applications; however, dedicated hardware accelerators have emerged in the last several years [NVI18].

Hierarchical Z-buffer This algorithm traverses octrees and hierarchical pyramids to efficiently cull objects that are not in view [GKM93]. The disadvantage of this approach is that the octree is computationally expensive to construct, so it is not well-suited to dynamic scenes.

3 Results

Figure 2 shows our results for scenes with 16384 triangles; for additional results, refer to Appendix B. Increasing the number of cores did not always improve

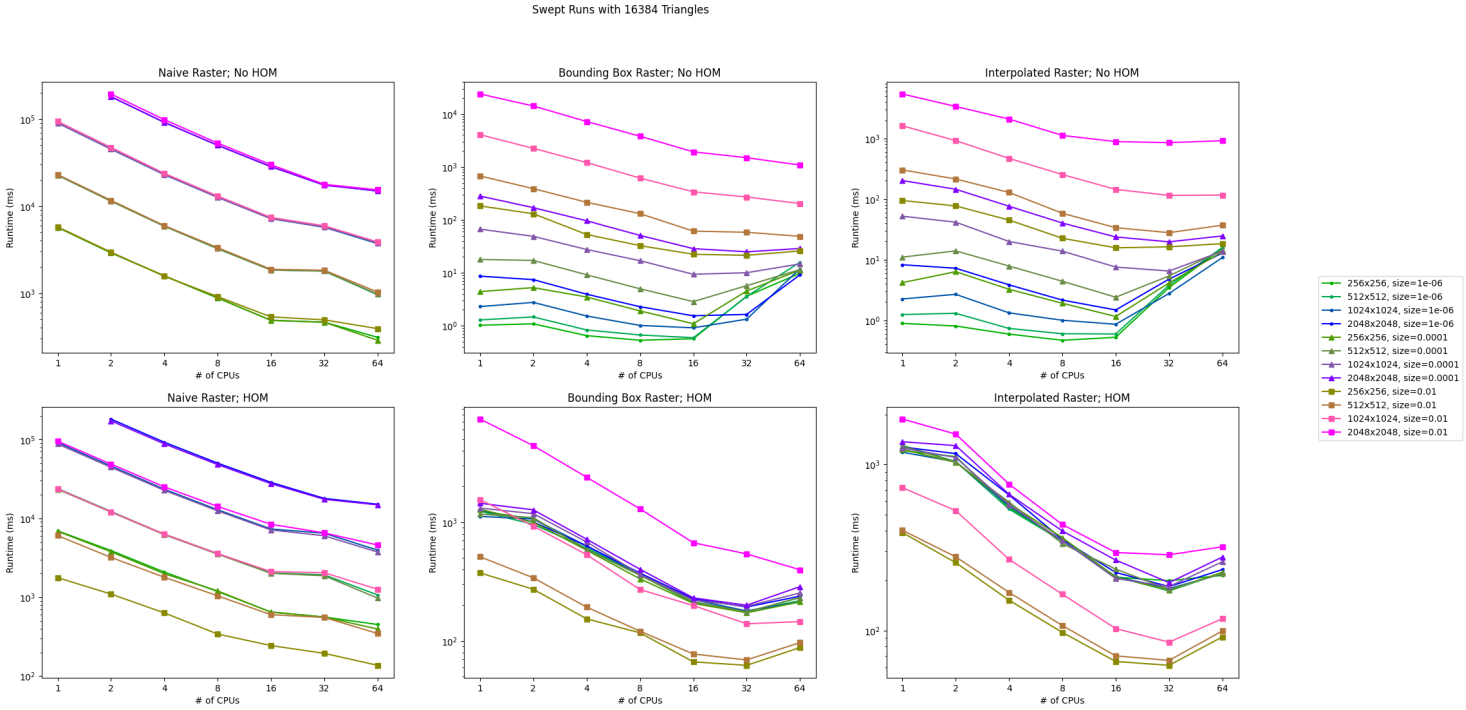

Fig. 2: Sweeping results with 16384 triangles, using the parameters defined in Table 1

the performance of our algorithm. Analysis with the *perf* utility did not reveal significant differences in L1D or LLC miss rates when the number of cores was increased, so contention and false sharing seem like unlikely root causes. The machine used for these experiments only had 32 physical CPUs (but 64 logical CPUs; see Appendix C), which may explain some degradation with 64 CPU runs; otherwise, it seems likely that parallelization overhead was a contributing factor.

While naive rasterization was generally the slowest technique, interpolation was not always significantly faster than the bounding box method. The difference seemed to be more noticeable for larger triangles and fewer CPUs.

Occlusion culling did not always improving the overall performance. The overhead to create and update the HOM becomes increasingly significant when there are fewer triangles in the scene. Additionally, when triangles are smaller, occlusions become less common, reducing the benefits of HOM.

4 Conclusion

In this project, we examined the performance of rasterization and occlusion culling for a variety of scene configurations. We found that optimizations which provide performance improvements in one configuration may be ineffective or even harmful in another.

References

[App68] Appel, A. "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 37–45. ISBN: 9781450378970. DOI: 10.1145/1468075.1468082. URL: https://doi.org/10.1145/1468075.1468082.

[GKM93] Greene, N., Kass, M., and Miller, G. "Hierarchical Z-buffer visibility". en. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*. Not Known: ACM Press, 1993, pp. 231–238. ISBN: 9780897916011. DOI: 10.1145/166117.166147. URL: http://portal.acm.org/citation.cfm?doid=166117.166147 (visited on 11/13/2021).

[NVI18] NVIDIA. *NVIDIA Turing GPU Architecture*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[Pin88] Pineda, J. "A Parallel Algorithm for Polygon Rasterization". In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, pp. 17–20. ISBN: 0897912756. DOI: 10.1145/54852.378457. URL: https://doi.org/10.1145/54852.378457.

[Scr15] Scratchapixel. *Rasterization: A practical implementation*. 2015. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation.

[Wor95] Worboys, M. *GIS: A computer science perspective*. Taylor & Francis, 1995.

[Zha98] Zhang, H. "Effective Occlusion Culling for the Interactive Display of Arbitrary Models". UML Order No. GAX99-14933. PhD thesis. USA, 1998.

A Link to Code

Code to accompany this paper is available at https://github.com/ZSusskind/parallel-project-f21.

B Additional Figures

Refer to Figures 3, 4, 5, and 6 for results with different numbers of triangles in the input. The impact of parallelization overhead becomes generally less pronounced as the number of triangles in the scene increases.

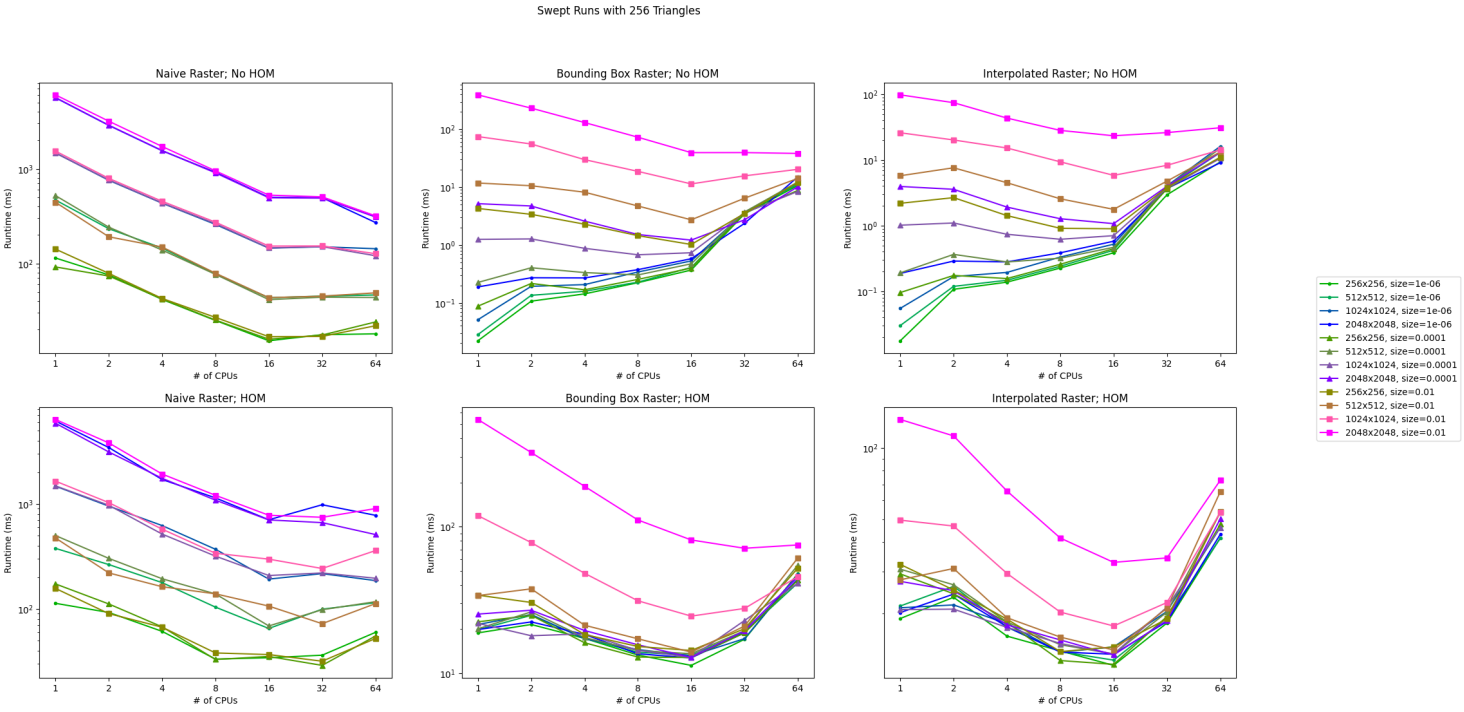
Swept Runs with 256 Triangles
Naive Raster; No HOM
Bounding Box Raster; No HOM
Interpolated Raster; No HOM
Naive Raster; HOM
Bounding Box Raster; HOM
Interpolated Raster; HOM
Runtime (ms)
# of CPUs
256x256, size=1e-06
512x512, size=1e-06
1024x1024, size=1e-06
2048x2048, size=1e-06
256x256, size=0.0001
512x512, size=0.0001
1024x1024, size=0.0001
2048x2048, size=0.0001
256x256, size=0.01
512x512, size=0.01
1024x1024, size=0.01
2048x2048, size=0.01

Fig. 3

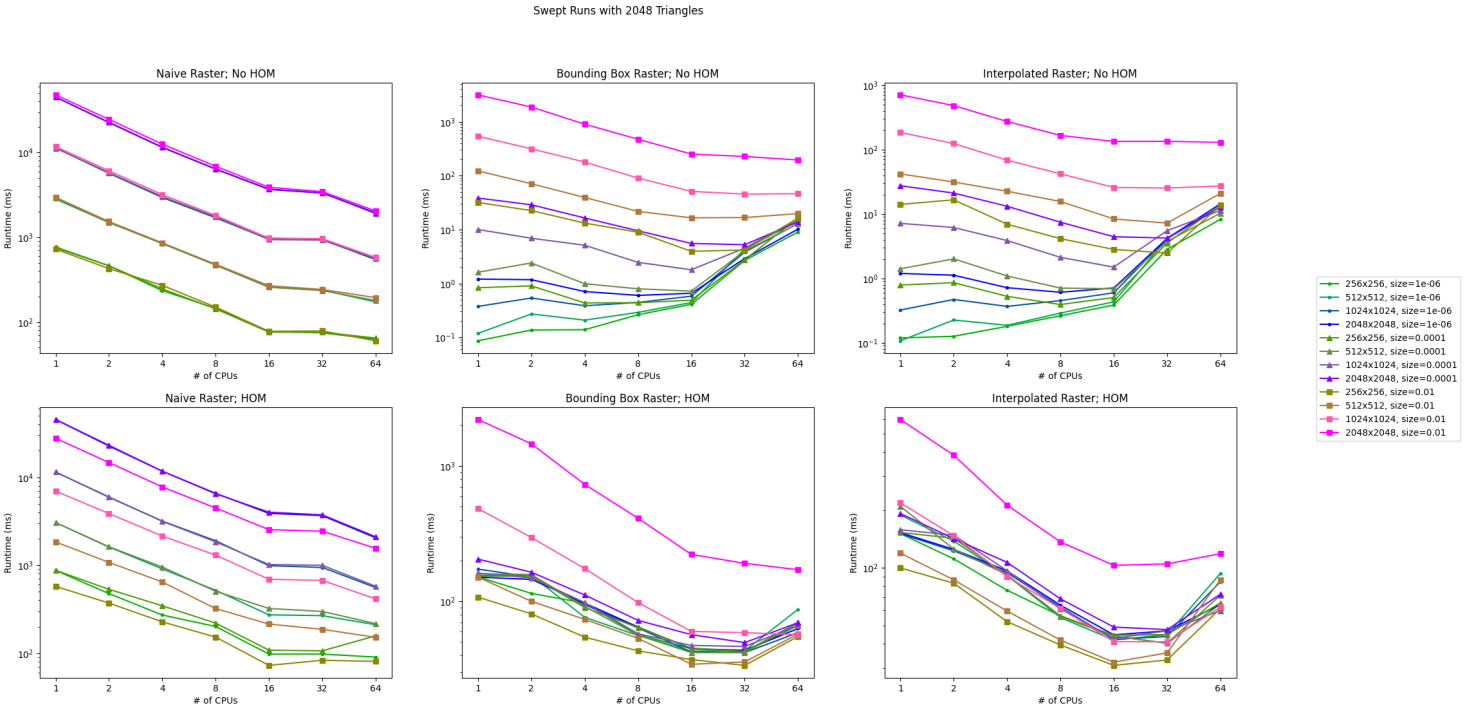
Swept Runs with 2048 Triangles
Naive Raster; No HOM
Bounding Box Raster; No HOM
Interpolated Raster; No HOM
Naive Raster; HOM
Bounding Box Raster; HOM
Interpolated Raster; HOM
Runtime (ms)
# of CPUs
256x256, size=1e-06
512x512, size=1e-06
1024x1024, size=1e-06
2048x2048, size=1e-06
256x256, size=0.0001
512x512, size=0.0001
1024x1024, size=0.0001
2048x2048, size=0.0001
256x256, size=0.01
512x512, size=0.01
1024x1024, size=0.01
2048x2048, size=0.01

Fig. 4

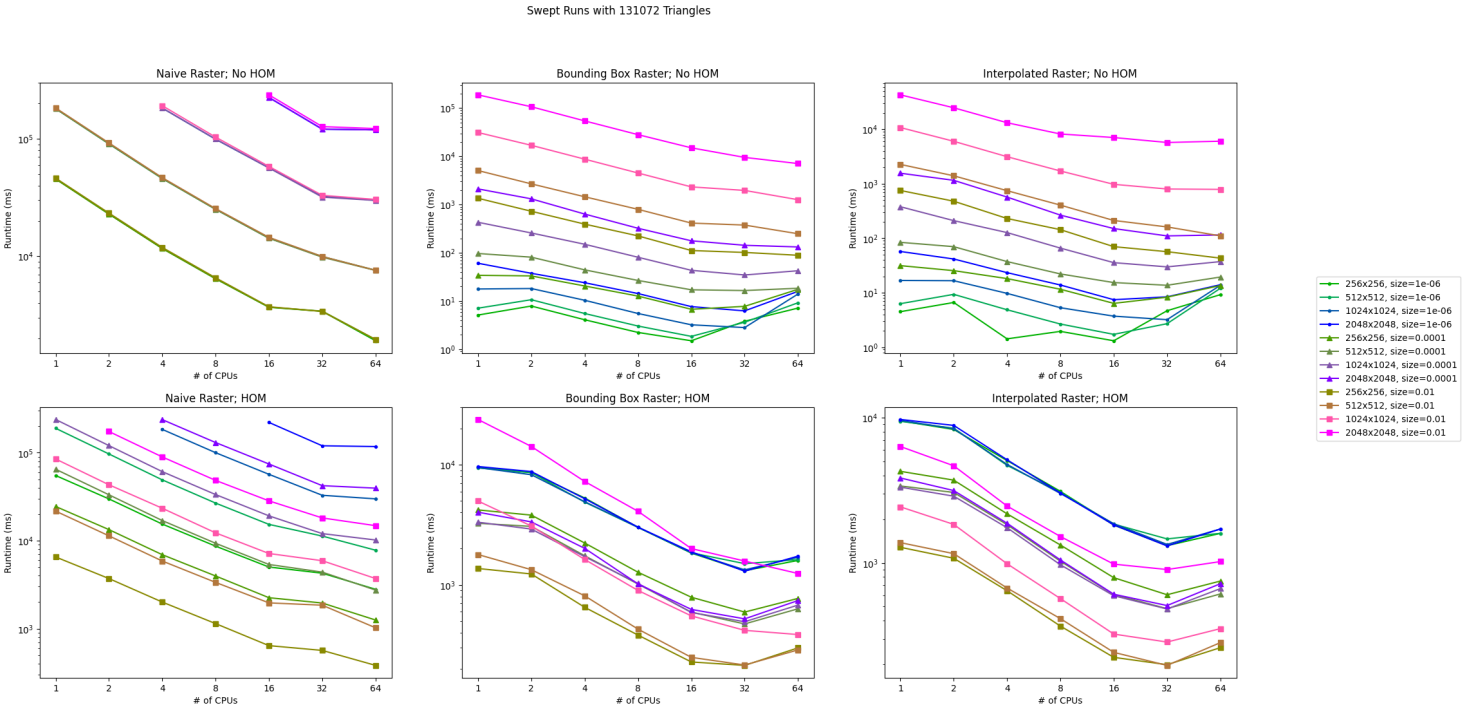
Swept Runs with 131072 Triangles
Naive Raster; No HOM
Bounding Box Raster; No HOM
Interpolated Raster; No HOM
Naive Raster; HOM
Bounding Box Raster; HOM
Interpolated Raster; HOM
Runtime (ms)
# of CPUs
256x256, size=1e-06
512x512, size=1e-06
1024x1024, size=1e-06
2048x2048, size=1e-06
256x256, size=0.0001
512x512, size=0.0001
1024x1024, size=0.0001
2048x2048, size=0.0001
256x256, size=0.01
512x512, size=0.01
1024x1024, size=0.01
2048x2048, size=0.01

Fig. 5

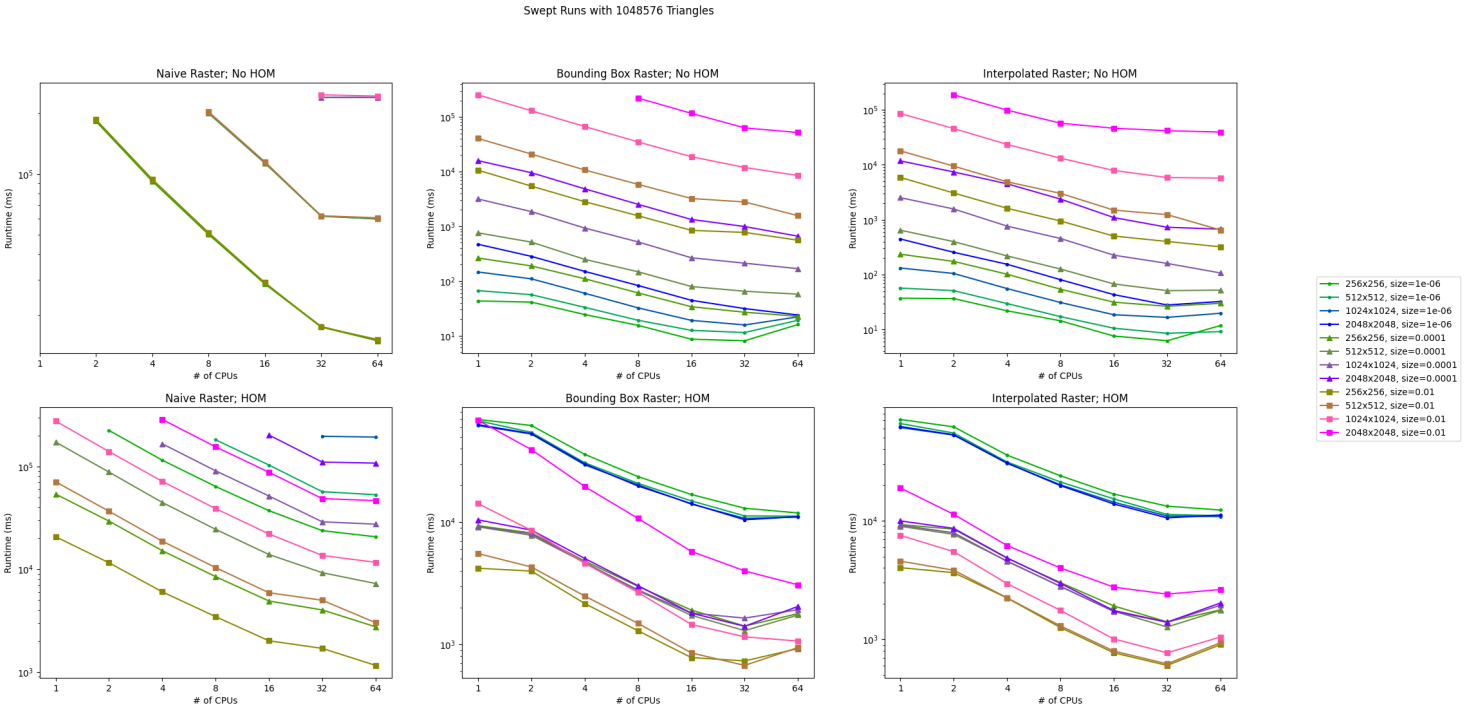

Fig. 6

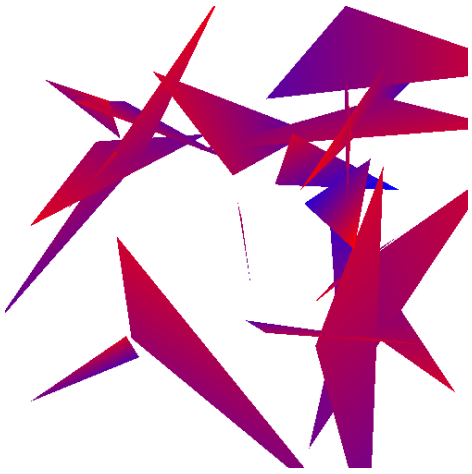

Fig. 7: Original Scene

(a) 128x128 (b) 64x64 (c) 32x32

Fig. 8: A set of progressively downsampled occlusion maps for a sample input

Figure 8 shows the set of occlusion maps for a sample input. Note that successive occlusion maps are progressively lower resolution.

C Profiling Details

Profiling was performed on a machine with 2x Intel Xeon E5-2698 v3 CPUs (16 physical / 32 logical cores each; 32 physical / 64 logical cores total) and 128GB DDR4-2133MHz ECC memory, running Ubuntu 20.04 LTS. Profiling runs were performed sequentially, with runs timing out after five minutes.