Eurographics Symposium on Parallel Graphics and Visualization (2014)
M. Amor and M. Hadwiger (Editors)

Parallel Methodologies for a Micropolygon Renderer

Mark A. Bolstad

University of Maryland, Baltimore County, Baltimore, Maryland, USA

Abstract

This paper compares the performance of three different methodologies for a multi-threaded micropolygon-based renderer. We extend the REYES [AG99] algorithm for multi-threaded rendering, which we call CASCADE. CASCADE processes one bucket per thread, forwarding primitives and micropolygons to other buckets/threads through split and dice operations. ROUND_ROBIN runs N single-threaded versions of CASCADE and a compositor, where primitives are distributed to each thread in a semi-random manner. NO_FORWARD executes split and dice operations, but a primitive that spans multiple buckets is processed independently by different threads and the primitives generated through split and dice operations that project outside the current bucket are discarded. In addition, bucket scheduling is used in this case to ensure that no thread is starved for work. Extensive analysis demonstrates that none of these methodologies are clearly superior to the others under all combinations of primitive size, count, transparency, and parallelism, so, a hybrid algorithm is proposed whose performance characteristics make it the best choice under all but the most pathological cases.

1. Introduction

Rendering is the process of taking a three-dimensional description of geometry, its material properties, and the environment, and projecting them through a camera onto a two-dimensional plane. The focus of this paper is not on the shading system, but on the handling of models or scenes with large geometric complexity. Techniques for rendering large scenes typically fall into one of three categories: reduction in complexity, data reorganization, and brute force techniques. At their heart, all methods for handling large scenes are about reducing the number of primitives that are pushed through the renderer. Additionally, most of these methods are designed for improving the performance of interactive rendering, where the models are typically written once, but viewed/rendered multiple times. In these situations, allocating additional effort up front to reduce complexity or reorganize the data for more efficient access will be beneficial. For this paper we are targeting off-line renders with a typical usage pattern of write-once, render-once. For these scenarios, the extra cost to reorganize the data is typically not an efficient use of resources. Therefore, we will focus on the brute force methodology, of which parallel rendering is one technique.

Micropolygon rendering is the process of tessellating input geometric primitives into polygons typically of less than one pixel in size. Micropolygon rendering was first described by Cook et. al. [CCC87] and has been used to render complex scenes for motion pictures for 25 years. Over that period this algorithm has been shown to be stable and robust when rendering large, geometrically complex scenes. In this paper we investigate parallel extensions of a micropolygon renderer and show that none of the common work distribution methodologies are appropriate for all scene compositions. While the results in this paper are strictly applied to a micropolygon renderer, the results are applicable to any rendering algorithm that processes the scene in small, limited parts, such as a ray caster using a Hilbert curve for screen traversal.

2. Previous work

The original REYES algorithm by Cook et. al. [CCC87] processed each primitive in the geometric database one at a time independent of all other primitives. Each primitive that survived the culling phase is split into finer pieces and eventually diced into grids of sub-pixel sized quadrilaterals called micropolygons. The grids are then shaded, busted into individual micropolygons, and then sampled. Because primitives are processed one at a time, all of the samples from the

DOI: 10.2312/pgv.20141080

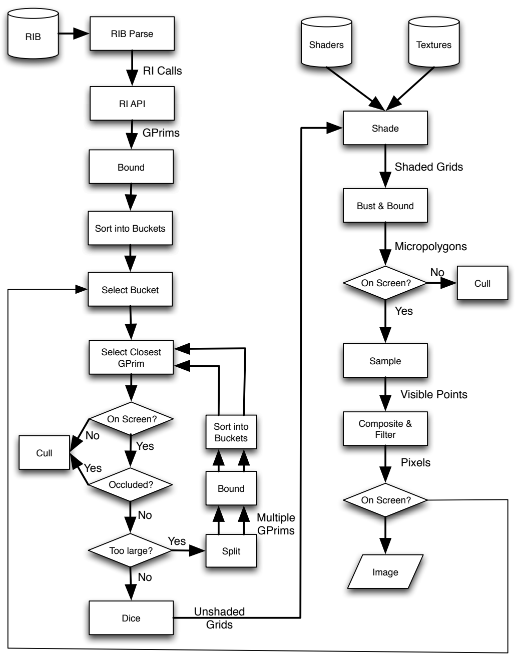


Figure 1: *The REYES Rendering pipeline*

micropolygons are retained until the final primitive is processed, otherwise, the visibility could not be properly determined. As a result, the memory consumption of the original REYES algorithm was linear in the number of micropolygons that survived to the final stage of the process, and the number of micropolygons generated were several orders of magnitude larger than the original geometric database. An improved REYES algorithm [AG99] was designed that removed the requirement for the retention of the visible point lists. The main change was that rather than process each primitive separately, the entire geometric database is read, bound, and sorted into buckets. Then, each of the buckets is processed as described in the original algorithm. When a primitive is split, it falls into one of three conditions: inside, outside, or straddles the current bucket. If the split primitive falls outside of the current bucket, the primitive is forwarded to the first bucket that overlaps its bounds and further processing of the primitive is delayed until its new bucket is processed. The primitives remaining in the current bucket then complete the entire dice-shade-bust-hide algorithm described above. When the entire bucket is processed, all of the memory used by the overlapping primitives, the visible point lists, and the micropolygons can be reclaimed, and utilized by the next bucket. The trade-off made to achieve the lower memory footprint is the assumption that the entire geometric database can be resident in memory. Several extensions to the RenderMan Interface Bytestream (RIB) alleviated some of that assumption by delaying the reading of a primitive until the renderer began the processing of a bucket that overlapped its bounds.

While the REYES algorithm has been parallelized before (NetRenderMan) [AG99], its primary function is to speed up rendering through the distributed processing of the individual buckets by using a replicated database.

In recent years work has focused on enabling the REYES algorithm for real-time applications using the GPU [FLB*09, FFB*09, PO08, WGER05, ZHR*09]. While exciting results are coming out of this line of research, in most part it is not applicable to us as we are investigating scenes sizes that far exceed the capabilities of modern GPUs except through streaming extensions. Relevant parts of the research have been investigated and where pertinent, implemented, e.g., DiagSplit by Fisher et. al [FFB*09].

2.1. Parallel Rendering

Two methods for rendering large models, parallel rendering and multi-resolution methods, attempt to improve rendering performance by reducing the number of primitives processed by an instance of the renderer. Multi-resolution methods try to reduce the number of primitives rendered through the use of a hierarchy of simplified representations that are visually indistinguishable from the full resolution model when viewed from a particular distance. The parallel rendering approach typically uses a brute force algorithm that tries to render the full resolution of the model by breaking the data into smaller chunks that can be processed efficiently on a large number of processors, thereby reducing the total workload of a single processor. More recent advances have seen the combination of parallel rendering methods with multi-resolution techniques.

Through the 1980's and 1990's, a large number of researchers began tackling the problem of parallel rendering. While many different topics were covered during this time, the topics can be sorted into three broad categories: hardware systems [Ull83, EAF*88, FPE*89, PH89, MEP92], rendering, [Wei81,DS84,FK90,Cox95] including parallelizing the ray-tracing algorithm [Lef93, Pit93, BBP94, Neu94, Fun96], and scheduling and load-balancing [Whi93, Whi94, RCJ99]. Dippé [DS84] described a prototype parallel ray-tracing system that redistributed primitives whenever the load imbalance between neighbors exceeded a threshold. Although not explicitly stated, their redistribution algorithm was performed on frame boundaries, they could still have interframe imbalances depending on the initial distribution of primitives.

To comprehend the differences between parallel rendering techniques, one has to understand that all parallel rendering algorithms require that the primitives are sorted somewhere in the rendering pipeline. Parallel algorithms can be classified into one of three categories, sort-first, sort-middle, or sort-last, depending on where in the rendering pipeline the sorting occurs [MCEF94].

Sort-first algorithms partition the screen-space into a set of non-overlapping tiles, each of which is rendered independently. In a sort-middle approach, the graphics primitives are sorted after vertex processing, but before fragment processing. Currently, this approach is not efficient for a software-based system due to the high bandwidth requirements necessary for transmission of the data between the stages, and the redistribution of the data between frames. And, sort-last algorithms function by distributing the primitives evenly amongst the processors. For each class of algorithms, there are many possible techniques to distribute the primitives to the rendering processes, including random allocation and round-robin. After each processes has completed its rendering, a list of tiles or samples are sent/gathered to/from another process for compositing.

Eldridge [Eld01] further refined this taxonomy by splitting the definition of sort-last into two components, *sort-last fragment* and *sort-last image composition.* The distinction occurs as to whether sorting occurs before or after fragment shading. For the rest of this paper, when we refer to sort-last we are using Eldridge's definition of sort-last image composition.

In 2000, Samanta et al. [SFLS00] proposed an algorithm that uses a combination of sort-first to decompose the tiles into N distinct groups, then uses sort-last to resolve the depth on the overlap at the edges of the tiles. Their objectives were to balance the load across the processors and minimize the screen space overlaps. The data had to be replicated across all the nodes of the cluster, thereby limiting the size of the model to the machine with the smallest memory.

In the following year, Samanta et al. [SFL01] described a method to achieve nearly the performance of full database replication with only partial replication. It used the hybrid sort-first, sort-last architecture from their previous paper with partial replication of the model data. Even with these improvements, the size of the model was still limited to the size of the smallest memory in the cluster.

A pure software renderer provides a finer grain of control in how the memory of a system is used for rendering. Green and Paddon [GP94] described a set of methodologies to increase the performance of a multi-processor ray-tracing system by mimicking a virtual memory model. They showed the effect on performance of varying memory allocation between the resident set and the cache, and between the voxel hierarchy (octree) and the objects. By rendering a low resolution image they could acquire a lower bound on the objects and the memory required for the resident set.

3. Algorithms

In this paper we describe four different methodologies for parallelizing the REYES algorithm. In all of these parallel variants we are only looking at algorithms that are suitable for running on a shared memory architecture. This allows us to look at the efficiency of the algorithms independent of any communication costs that would be associated with a message passing approach. We started by implementing the CASCADE algorithm. During testing and analysis, the performance was measured with several tools looking for bottlenecks in the code. The results from the analysis led to each of the following algorithms as a method to overcome one/several bottlenecks. The fourth algorithm was conceived by using the best aspects of the others. In the following sections we describe each of the algorithms in detail.

3.1. CASCADE

One of the most straightforward approaches to parallelizing the REYES algorithm is for each bucket to be rendered by one thread. Since we will nearly always have less threads than buckets, threads are reused on multiple buckets. In Molnar's parallel taxonomy this approach is a modified version of the sort-first algorithm. The modifications are two-fold: first, threads process multiple tiles, and second, there is no replication of data amongst the tiles. The split/dice loop proceeds as in the improved REYES algorithm [AG99] with the exception that the fragments output from the loop are forwarded as in the original method, albeit to a potentially different thread. To reduce the amount of synchronization involved with the forwarding of primitives, a per-thread queue is added to each bucket. Each thread writes to only a single queue per bucket thereby eliminating any write conflicts between threads. When the split/dice loop is ready to process the next primitive, the thread queues are scanned for any forwarded primitives which are then sorted into the main queue. The primary point of synchronization is due to the unpredictable arrival of forwarded primitives. Since these primitives can arrive at any point during the rendering of a bucket, no bucket can complete until all buckets previous to it have completed. Buckets are statically assigned to threads modulo the thread count. A queue of buckets was originally used,

3.2. ROUND_ROBIN

Using a sort-last methodology, we implemented the initial distribution of primitives through round robin allocation. Once the primitives have been allocated, each thread proceeds with a single-threaded variant of the CASCADE algorithm with the exception that the visible point lists for a completed bucket are placed in a thread specific queue of the master process for compositing. The master is responsible for freeing the visible point lists once it has received the lists from all active threads and compositing has completed.

3.3. NO_FORWARDING

The third algorithm is also based on a sort-first methodology. In the REYES algorithm, as primitives are bound they are placed into the first bucket that overlaps the lower left corner. NO_FORWARD follows a traditional sort-first distribution by placing a reference to each primitive into all buckets overlapped by the primitive projected bounds. During splitting and dicing, all primitives and micropolygons that project outside the current bucket are discarded. Otherwise, the algorithm proceeds similar to CASCADE. The other difference from CASCADE is that rather than having threads stride through the buckets based on the number of threads, buckets are scheduled to the first available thread.

3.4. MODIFIED_NO_FORWARDING

After analyzing the previous methods, this algorithm was derived by combining CASCADE and NO_FORWARDING such that we minimize the bottlenecks from either algorithm (discussed in the next section). From NO_FORWARDING we retain the bucket/thread allocation and the primitive distribution to the buckets. One issue in NO_FORWARDING is that primitives can be split and diced multiple times, one for each bucket the primitive overlaps. Since the primitive is split and diced in the first bucket it overlaps, CASCADE does not suffer from this issue. We modify NO_FORWARDING such that the primitive is split by the first thread to access the primitive. Once split or diced, the new items are added back as children of the original node. If another thread is unable to lock the primitive, it moves that primitive to a deferred list and then it moves to the next primitive. On each iteration through the primitive loop, it checks if the deferred primitive is available and closer than the next one. If the deferred primitive has been split we add the children to the primitive queue. If it was diced, we shade and bust as normal. Figure 2 shows the changes made to the original REYES algorithm.

.

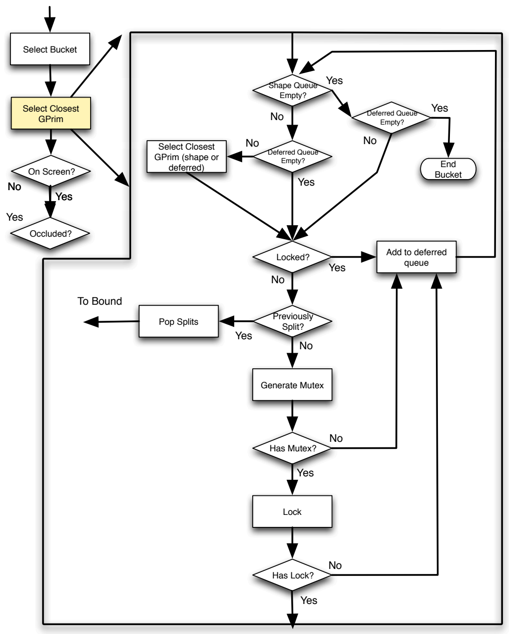


Figure 2: *Modified REYES algorithm*

4. Results

The results in this section were obtained on a single Linux system with two 2.67 GHz Intel "Nehalem" quad-core processors and 24 GB of RAM with hyper-threading disabled. The renderer is run with N threads for rendering and a master that handles the parsing of the scene description, converting the visible point lists into final pixel color, and for ROUND_ROBIN, compositing the point lists.

We used synthetic and procedurally generated scenes consisting of polygons (triangles and quadrilaterals) to enable easy comparison of results across the different algorithms. One of our procedural datasets is cityscapes generated using the Houdini modeling and animation software. We vary a large number of parameters for each building including width, depth, the number of floors, windows, and window and floor inset and thickness. Buildings are randomly placed on a grid, one per cell, with the grid ranging in size from 10x10 cells to 160x160 cells, resulting in polygon counts between 147,000 and 50 million. Two different camera views (see Figure 3) were used for testing the effect of minimal and heavy occlusion culling. The synthetic scenes consist of the Stanford bunny randomly replicated 10 to 1600 times in two sets, one opaque, the other with an alpha value of 0.02 (see Figure 4). In addition, the number of rendering threads used was varied between one and eight (two to nine actual threads including the master). For all tests, parsing the input data was a single threaded process and the timing for parsing was consistent across primitive counts and algorithms, and so was excluded from the timing results.

Figure 5 shows the results for different thread counts versus problem size for the city scene for each of the two views. In the heavily occluded street canyon view, MODIFIED_NO_FORWARDING outperforms all the other algorithms for any combination of threads and scene size. Depending on the problem size, MODIFIED_NO_FORWARDING is between 16-40% faster for 1 thread, and between 8-90% faster for 8 threads. In the second test scene, the view is modified to be looking at the en-

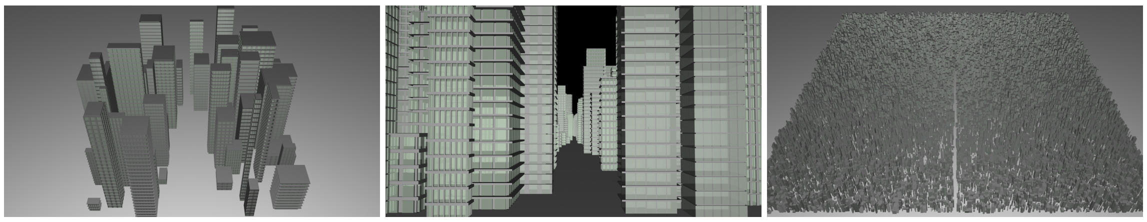


Figure 3: *Aerial and street canyon views of cityscapes with 0.1, 6, and 46 million polygons*

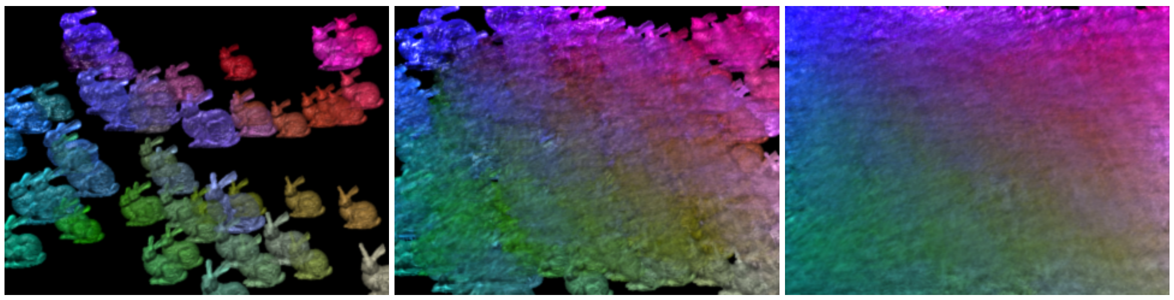


Figure 4: *Image result for 50, 400, and 1600 transparent Stanford Bunny tests*

tire city. Occlusion plays a role in this scene, but the viewpoint was chosen to minimize its impact. As a consequence, MODIFIED_NO_FORWARDING is the fastest algorithm in nearly all combinations of thread count and problem size, except for the largest scene with the two highest thread counts. Two items to note in these results: First, the algorithm that is second fastest changes depending on the scene size and thread count, and second, the differences between these algorithms (excluding ROUND_ROBIN) is typically small, on the order of 3-10% for the larger scenes.

In both scenes, ROUND_ROBIN consistently performs the worst except for the single threaded cases. In the three other algorithms, the master thread's only function post-parsing is to write the final image pixels to file. In ROUND_ROBIN, compositing of the pixels from the rendering threads is performed on the master thread, which gives it a slight advantage over some of the other algorithms. However, in all of the other cases, the performance gain due to occlusion culling is lost as the primitives are assigned to the buckets in random order.

For the second set of test cases involving the Standford bunny, the results are significantly more varied (see Figure 6). In most of the cases, the original REYES algorithm, CASCADE, is the fastest with MODIFIED_NO_FORWARDING surpassing it only in the large scenes with low thread counts. But, like the low occlusion city scene, the differences between the algorithms is minimal (excluding ROUND_ROBIN in most of the cases). For the transparent cases, again the results are significantly different in that ROUND_ROBIN is the fastest in nearly all cases. The reason is that with the very low alpha value, occlusion never comes into play except in the largest scene. The offloading of the compositing of the pixels to the main thread allows the worker threads to return rendering faster than the other algorithms. Varying the opacity value produces results in between these two extremes, depending on how quickly occlusion culling comes back into play (e.g., for an opacity value or 0.1, it takes approximately 45 samples before we reach fullly opaque).

5. Discussion

In this paper we have shown that there is no one best method for parallelizing a micropolygon renderer under a variety of conditions. Future work will explore two extensions to these algorithms. First, we propose a scene heuristic that would analyze the scene during parsing and estimate which algorithm would be the most efficient for the given conditions e.g., high transparency and thread count, then MODIFIED_NO_FORWARD. We would like to investigate the use of a two-pass heuristic that first stochastically samples the input scene and then chooses the algorithm for the full data pass. Second, we propose to investigate the scaling of the algorithms under a distributed memory model with message passing. This will allow for testing of data sizes much larger than can be processed on a single node.

While not reported in the results, a bottleneck in the rendering process is in the parsing of the input files. There was only a small variance between the different methods with regards to scene processing and setup, but for the large data sets could be as much as 40% of the total render time. Future work will look into efficient methods for parallelizing scene setup.

6. Acknowlegements

We would like to thank the Stanford 3D Scanning Repository for the bunny model. Also, we would like to thank Side Effects Software for supporting this research.

References

[AG99] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., 1999. 1, 2, 3

[BBP94] BADOUEL D., BOUATOUCH K., PRIOL T.: Distributing Data and Control for Ray Tracing in Parallel. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 69–77. 2

[CCC87] COOK R., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniq* (1987), ACM Press, pp. 95–102. 1

[Cox95] COX M.: *Algorithms for parallel rendering*. PhD thesis, 1995. 2

[DS84] DIPPE M., SWENSEN J.: An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 149–158. 2

[EAF*88] EYLES J., AUSTIN J., FUCHS H., GREER T., PAULTON J.: Pixel-planes 4: a summary. (1988), 183–207. 2

[Eld01] ELDRIDGE M.: *Designing Graphics Architectures around Scalability and Communication*. PhD thesis, Stanford University, June 2001. 3

[FFB*09] FISHER M., FATAHALIAN K., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: DiagSplit: Parallel, Crack-free, Adaptive Tesselation for Micropolygon Rendering. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 1. 2

[FK90] FRANKLIN W., KANKANHALLI M.: Parallel object-space hidden surface removal. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniq* (1990), ACM Press, pp. 87–94. 2

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-parallel rasterization of micropolygons with defocus and motion blur. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09* (2009), 59. 2

[FPE*89] FUCHS H., POULTON J., EYLES J., GREER T., GOLDFEATHER J., ELLSWORTH D., MOLNAR S., TURK G., TEBBS B., ISRAEL L.: Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniq* (1989), ACM, pp. 79–88. 2

[Fun96] FUNKHOUSER T.: Coarse-grained parallelism for hierarchical radiosity using group iterative methods. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniq* (1996), ACM Press, pp. 343–352. 2

[GP94] GREEN S., PADDON D.: Exploiting coherence for multiprocessor ray tracing. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 12–26. 3

[Lef93] LEFER W.: An efficient parallel ray tracing scheme for distributed memory parallel computers. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering* (1993), ACM Press, pp. 77–80. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32. 3

[MEP92] MOLNAR S., EYLES J., POULTON J.: PixelFlow: high-speed rendering using image composition. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniq* (1992), ACM Press, pp. 231–240. 2

[Neu94] NEUMANN U.: Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 49–58. 2

[PH89] POTMESIL M., HOFFERT E.: The pixel machine: a parallel image computer. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniq* (1989), ACM, pp. 69–78. 2

[Pit93] PITOT P.: The Voxar Project (Parallel Ray-Tracing). *IEEE Comput. Graph. Appl. 13*, 6 (1993), 27–33. 2

[PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5 (Dec. 2008), 1. 2

[RCJ99] REINHARD E., CHALMERS A., JANSEN F.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics* (1999), ACM Press, pp. 21–28. 2

[SFL01] SAMANTA R., FUNKHOUSER T., LI K.: Parallel rendering with k-way replication. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphic* (2001), IEEE Press, pp. 75–84. 3

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J.: Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2000), ACM Press, pp. 97–108. 3

[Ull83] ULLNER M.: *Parallel machines for computer graphics*. PhD thesis, 1983. 2

[Wei81] WEINBERG R.: Parallel processing image synthesis and anti-aliasing. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniqu* (1981), ACM, pp. 55–62. 2

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM Press, pp. 7–14. 2

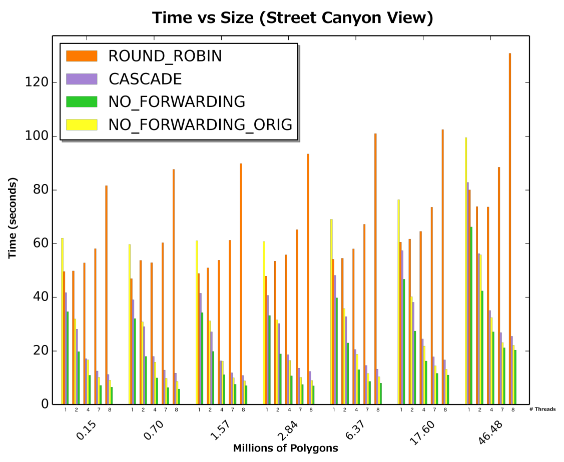
[Whi93] WHITMAN S.: A task adaptive parallel graphics renderer. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering* (1993), ACM Press, pp. 27–34. 2

[Whi94] WHITMAN S.: Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 41–48. 2

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: RenderAnts: interactive Reyes rendering on GPUs. *ACM Transactions on Graphics 28*, 5 (2009), 155. 2

# Polygons	# Rendering Threads	Round Robin	Cascade	No Forwarding	No Forwarding Original
147428	1	49.65	41.73	34.67	62.08
	2	49.81	28.13	19.73	31.96
	4	52.82	17.10	10.98	16.65
	7	58.15	12.51	7.15	10.20
	8	81.65	11.26	6.50	9.07
701374	1	46.951	39.10	32.053	59.71
	2	53.79	29.10	17.96	30.84
	4	52.91	17.99	9.93	15.86
	7	60.33	12.91	6.38	9.78
	8	87.69	11.76	5.80	8.70
1568796	1	48.89	41.50	34.28	61.10
	2	51.00	27.14	19.81	31.20
	4	53.88	16.35	11.17	16.25
	7	61.28	11.92	7.57	9.99
	8	89.87	10.87	7.06	8.89
2836538	1	47.83	40.77	33.23	60.79
	2	53.48	30.25	18.89	31.57
	4	55.85	18.62	10.72	16.44
	7	62.22	13.57	7.50	10.14
	8	93.48	12.38	7.02	9.05
6374458	1	54.20	48.16	39.78	69.12
	2	54.54	32.81	22.99	35.77
	4	58.05	20.52	13.00	18.75
	7	67.27	14.58	8.63	11.61
	8	101.03	13.27	8.02	10.38
17600502	1	60.60	57.42	46.78	76.43
	2	61.72	38.19	27.46	40.23
	4	64.59	24.46	16.26	21.87
	7	73.63	17.81	11.68	14.44
	8	102.52	16.71	11.00	13.21
46481006	1	80.08	82.83	66.22	99.57
	2	73.78	56.25	42.43	55.87
	4	73.69	35.06	27.13	32.47
	7	88.50	26.84	21.22	23.13
	8	103.94	25.51	20.36	22.10

Table 1: *Timing values for the Building scene. This table show the time in seconds for the four techniques versus the number of polygons for the street canyon scene Figure3. Note, that in most of the cases, there is very little improvement between 7 and 8 threads. The 8 thread case results in oversubscription of the system as the master thread increase the thread count to a total of 9 (on an 8-core system).*



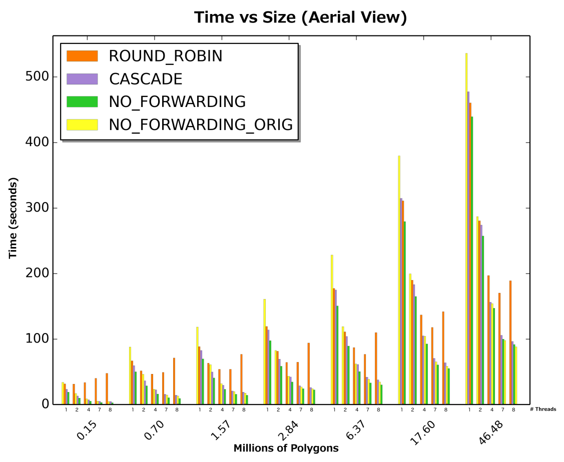
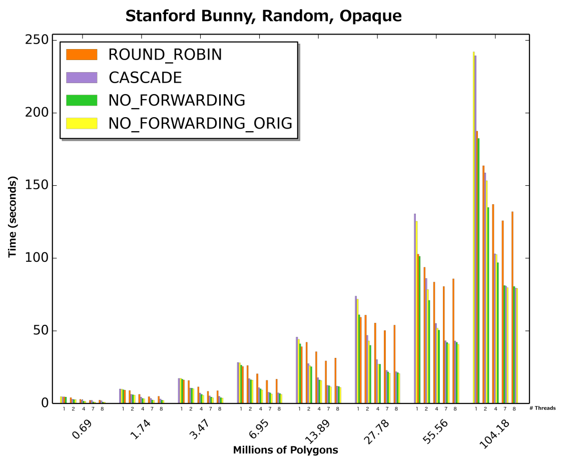


Figure 5: *Timing results for the two different city views. Note that the two plots have different maximums for the Y-axis.*



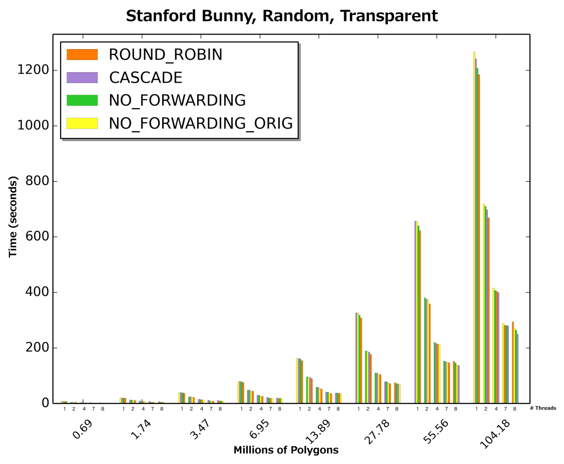


Figure 6: *Timing results for the two different Stanford bunny test scenes*