Hand-Drawn Ray Tracing

Casey Conway and Jason Lee
Rensselaer Polytechnic Institute

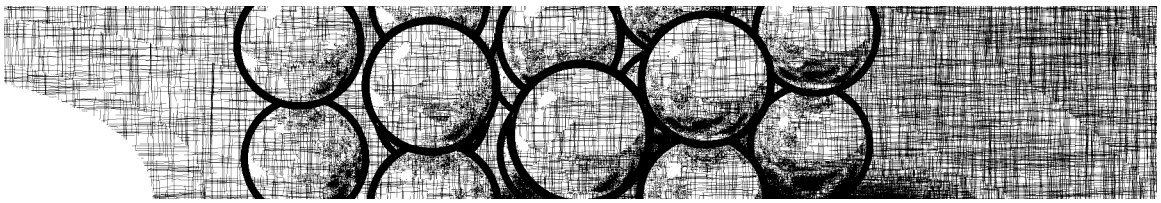


Figure 1: Hatched reflecting spheres

ABSTRACT

We present a strongly parallelized implementation of a two stage non-photorealistic ray-tracer designed to output rendered scenes in a hand-drawn aesthetic. Our program is designed to run on IBM's Blue Gene supercomputer and takes advantage of a novel, chunk-based bargaining system to maximize efficiency through strategic load balancing. We present a metric for estimating the cost of rendering each pixel, and we compare the performance of various parallel optimizations to analyze the efficiency gained by our final algorithm.

KEYWORDS

ray tracing, non-photorealism, hand-drawn, parallel

1 BACKGROUND

Ray tracing is a well studied technique for rendering high quality images with global illumination. First introduced in [5], it allows the detailed rendering of reflective and refractive surfaces as well as realistic shadows. [2] expands the method with stochastic techniques to include features such as anti-aliasing, soft shadows, and glossy reflections. We implement the stratified sampling techniques of [2] to try and reduce the noise and clumping artifacts introduced by adding randomness to the rays.

Ray tracing works by shooting rays from the camera position towards the scene through a grid of output pixels. The shading of each pixel is determined by the computations of each ray, which recursively construct an exponentially expensive ray tree throughout the environment. Each ray is cast through the scene and terminates when it hits an object. Depending on the material properties of the hit surface, it may produce additional rays to account for reflective bounces or shadows. Shadows are computed by shooting more rays from the hit surface towards each light source, and calculating the percentage of unobstructed rays to approximate the amount of light received at that particular point.

We extend this ray tracing implementation to include silhouette edge detection as described in [1] to determine the locations of mesh boundaries. This step is succinctly summarized by casting an extra stencil of rays for each pixel to determine if an edge should exist based on the percentage of hits from different objects. We require harder edges on our objects to help identify the borders between foreground and background objects when our hand-drawn step occurs.

Our final post-processing step makes the rendered images look hand-drawn by applying a similar technique as introduced in [3]. Although the authors' contribution is made towards real-time applications running on the traditional graphics pipeline, we are able to successfully adapt their technique to be used in ray-tracing. We overlay a paper shader based on Perlin noise to complete the simulation of a real hand drawn image.

2 OVERVIEW

Our algorithm accepts input files that are in the `.obj` format. It first exports a render of the given scene into a `.ppm` image file using our parallel ray-tracer implementation. This is a highly configurable and robust system that allows for the specification of various parameters, such as the camera position, the direction the camera should face, toggleable soft-shadows, the quality of the shadows, the number of reflected ray bounces, the number of anti-alias samples, the thickness of silhouette edges, the ambient lighting values, and the resolution of the output image, among many others.

On a whole, ray tracing is embarrassingly parallel as each pixel can be computed independently without needing information from surrounding pixels. This fact allows us to easily divide work between ranks (processes) and allow for parallel speedup to occur. However, a naïve partitioning of the pixels results in very poor load balancing for most images. This can result in processes finishing earlier than others and then idling for the remainder of the computation instead of actually working. We discuss in detail below a more complicated algorithm with complex inter-rank communication and threading to achieve full system usage.

Our raytracer, which uses a message passing library known as MPI to achieve parallelization, will produce a rendered, photorealistic output image. For many use cases, this is a fine stopping point. However, our particular project is more interested in producing a non-photorealistic pencil shaded effect to make 3D scenes seem as if they were drawn by hand.

To achieve this, we have a post-processing pipeline to generate bitmaps of pencil strokes corresponding to various light levels. We then read in the ray-traced image, threshold it based on the brightness of each pixel, and compute the final output by pulling from the corresponding bitmap texture instead. In all, this is a much

cheaper operation than the ray tracing step and does not require a supercomputer for reasonable performance times.

3 PARALLELIZATION APPROACH

We begin by describing the various levels of parallelization we experimented with and examine the relative improvement gained with each step. As our algorithm evolved in complexity, we were able to more effectively utilize our runs on the supercomputer as we improved the overall runtime.

3.1 Sequential

Our starting code base had no parallelization of any kind, so we were able to run a few test cases on just our laptops to see exactly how long a render would take. In one of our simplest examples, shown in (Figure 12), a 4k render with 1024 shadow samples and no anti-aliasing takes nearly 5 hours to compute. This scene only has two primitives and a quad for the floor, so intersection tests are very cheap to compute.

In examples such as the watchtower (Figure 9), we have many more potential collisions and thus each ray is much more expensive to compute. There are many very good solutions to this problem, such as using octree or kd-tree spatial data structures, but we currently do not have those implemented due to time constraints. There are numerous papers out there which implement these common data structures, and we just note that our version is not as efficient as we would like it to be. As such, the much higher cost of each ray due to collision checks leads to more complicated test cases being infeasible to render at higher resolutions on a single core. Anything higher than about 2000 faces is out of our render time restriction.

3.2 Naïve Parallel

The most obvious approach towards parallelizing raytracing is to evenly split the problem space between each rank. We divide each desired image into an $n \times m$ grid of pixels and assign each of the R ranks nm/R pixels worth of work.

As simple as this approach is, it works quite well and is effective in lowering runtime significantly. Ranks are able to compute sections of each image in parallel, and the more ranks allocated, the quicker the overall computation completes. Results confirming this speedup are available in section (8).

We envisioned some major issues to this approach before starting work on the parallelism and they were confirmed once we began running some test cases. The condition that each rank gets the same amount of pixels to work with, regardless of the actual scene, is non-optimal, because each pixel requires a different amount of time to compute depending on the behavior and materials within the target region. In consequence, some ranks can finish much more quickly than others and lay idle until completion, increasing the overall run times significantly. This problem and our first solution is described in the next section.

3.3 Cost Metric

In the Naïve parallel algorithm described in section 3.2, we point out that some ranks may finish sooner than others. This leads to having multiple ranks lay idle for potentially a very long time until the render completes. This issue is caused by the different costs to compute each pixel. In areas where rays only intersect the background, processing for that pixel ends relatively early; there is no need to compute expensive shadow rays, reflection rays, or any of the instances of additional recursion. In areas of the scene where there are reflective surfaces, the time required to calculate the shading at each pixel is much more involved as it can result in shooting thousands of additional rays.

This is further impacted by the principle of spatial locality. As ranks are assigned contiguous regions of the output image to compute, they often get lots of the same surface. As a result, a rank that is assigned some background pixels (such as those assigned the top portion of many of our test cases) will often get many background pixels, while a rank containing pixels for a reflective surface will usually need to compute a large majority of that expensive surface just on its own. This compounding effect is more noticeable on runs with fewer ranks, but is present on larger tests as well.

To combat this issue, we introduce a cost metric to more evenly distribute pixels to each rank based on estimated cost, as opposed to a simple pixel count. We experimented with a variety of parameters and found a formula which estimates the cost of each pixel. To achieve this, we first undergo a preprocessing step: we cast a single ray through the middle of each pixel to see what type of object it hits. If it doesn't hit anything (i.e. only hits the background), we consider the pixel cost to be the lowest possible. Otherwise we scale the cost of hitting diffuse and reflective surfaces based on various factors such as the number of potential bounces or the expected number of shadow rays we might produce.

We then apply this cost estimation to the division of labor between ranks. More specifically, an equal number of "cost units", not pixels, will be distributed across the ranks, with the goal of having each rank finish at roughly the same time. In casting rays during the preprocessing step, an intersection checking expense is incurred, which can lengthen the amount of time spent calculating a metric. However, this step, like the rest of the raytracer, is parallelized and its data results can be stored and reused after creation for new renders with the same resolution and camera position.

3.4 Chunks

While a metric based approach is appealing, it is not particularly effective when the approximations are poor. In larger scenes this could result in much slower runs if the metric fails to accurately predict the cost of pixels.

We instead shift to a more dynamic algorithm for managing this load balancing problem. Here we begin to stray from the idea of an embarrassingly parallel implementation into something with more complicated inter rank communication. For our premier technique, we need to introduce the idea of chunks.

Chunks can be considered smaller slices of groups of pixels than the divisions described above. In our code, we generate chunks of roughly fifty to one hundred pixels in size, which is a very small percentage of the overall image. For reference, a 4K image is roughly 8.3 million pixels, so there would be about 83,000 chunks in our average case.

Chunk sizes can be configurable based on the cost metric or given a fixed size. In our code, we use the metric of section (3.3) to have variable sized chunks to try and balance work even further.

With a good metric implementation, this could result in further performance improvements.

3.5 Bargaining

Our major contribution in this project is our load balancing scheme which we refer to as bargaining. In this approach, each rank is given a queue of chunks to process while communicating with other ranks to keep the contents of each queue roughly equal in cost.

We achieve this by adding some additional threading to handle communication between ranks. Each rank splits into two threads: one is responsible for computing the shading, while the other handles inter-rank communications to try and keep its queue balanced. Every second, we send out a heartbeat message to neighboring ranks on the left and right, as well as broadcast a message to all ranks to try and maintain an idea of how much more computation exists out there. If our upcoming workload is more than the global average, we peel off a chunk from our queue and offload it to a neighbor instead.

Over time, this technique results in a balancing out of the work over all ranks, sort of like spreading butter. We start out with the basic naïve distribution and over time this iterative technique converges to a healthier balance. As we're constantly updating workloads, we ensure no rank ends much earlier than the rest as they all finish roughly at the same time. This improves the results even with a poor metric that has bad approximations of the actual costs.

4 BITMAP GENERATION

To create the hand-drawn aesthetic for rendered three-dimensional scenes, we implemented bitmap generation similar to the method described by [3]. To generate the many levels of density needed, multiple bitmaps are created, each with a slightly different density parameter. Markov chains are used to dictate the generation of individual black pencil strokes on a white bitmap.

This behavior is defined by two random variables: the deviation chance, or the chance at any one iteration for a pencil stroke to move one pixel in a direction perpendicular to its prescribed direction, and the terminate chance, or the chance that a pencil stroke ends, or stops propagating, at any one iteration. In addition, there are two integer variables - one dictates the minimum length of any pencil stroke, which prevents pencil stroke termination until a certain number of recursions has completed, and one dictates the cell stroke density, or how often a new stroke should be generated in the bitmap. This is done in a stratified fashion; therefore, the bitmap is not too "clumpy".

Having created multiple bitmaps with multiple densities, our next goal is to calculate the approximate brightness level of each pixel, which comes from our ray-tracing step. This will allow us to decide which bitmap should be projected on that particular pixel. We do this by calculating thresholds for various brightnesses and mapping the corresponding bitmaps to those regions of the image. We end up with an image such as the one in Figure (4). As bitmap generation is not as computationally expensive as raytracing, it is permissible to run this routine as a post processing step without the aid of a supercomputer. Thus, we exclude the bitmap generation from our parallel performance study.

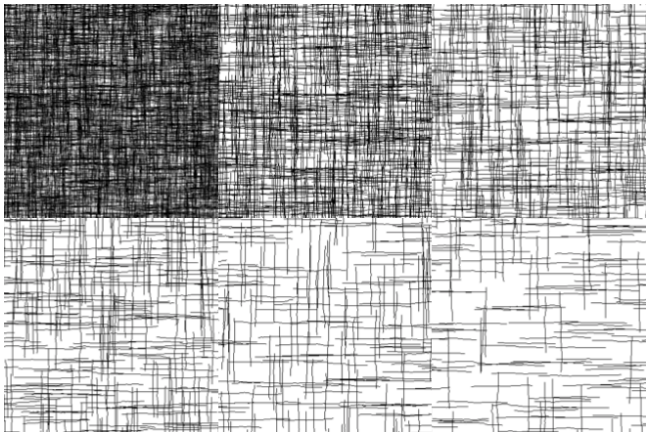


Figure 2: Bitmap levels

4.1 Bitmap Issues

We encountered a few issues in our implementation of bitmap generation. One early bug involved diagonal strokes being generated in the bitmap. This was because the random seeder was set up in a way such that the random number generator would always return the same value. This resulted in constant deviation from the stroke's original direction, as well as the bunching of multiple strokes. This buggy implementation is shown in Figure 3.One of the challenges with implementing bitmaps and thresholding is of ensuring a smooth gradation between the lightest areas of the image, as well as the darkest areas of the image. However, we decided that it would be more important that the brightest areas in the image remain at that level. Therefore, while we ensured that regions of maximum brightness in the original image would not have any bitmap applied, resulting in white areas in the processed image, the upper bound for a region of the processed image to have a bitmap applied remained independent of the brightness of the image. For example, in the processed reflective spheres example (Figure 13), the floor contains a distinct region with no bitmap, but the neighboring regions have a relatively dark bitmap applied. Ultimately, we find that this enhances the contrast of the image that we are trying to create.

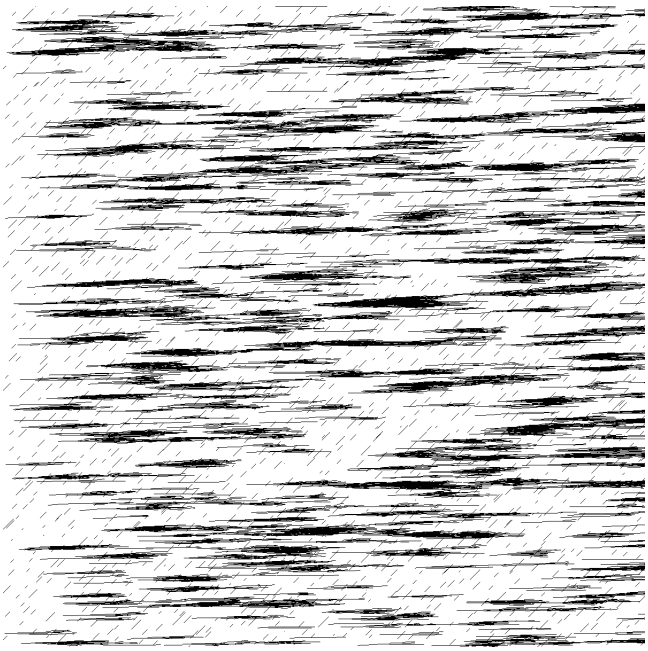


Figure 3: Bugged bitmap generation

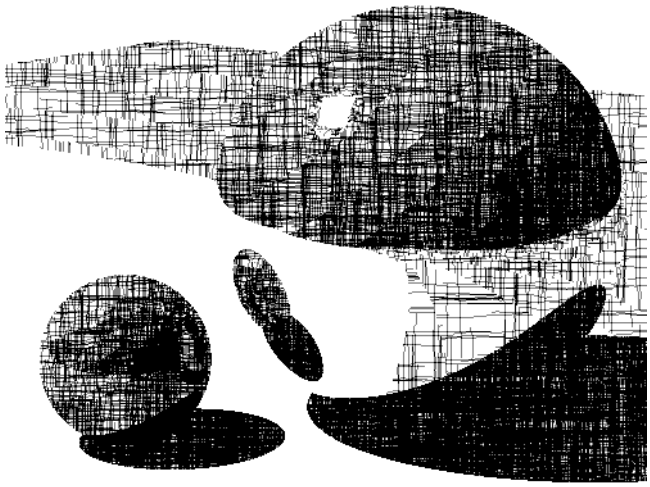


Figure 4: Spheres rendered without edges: notice the blending of the larger sphere into the background

5 EDGE DETECTION

The technique described in [3] includes an algorithm for detecting silhouette edges to maintain clarity after bitmap application. Their process works with the real time graphics pipeline, so they have access to the actual edges of the mesh. In our ray-traced approach we do not have that luxury, so we have to resort to a slightly more complicated calculation.

We were lucky to find a paper [1] that achieves exactly what we need. Their algorithm can detect all sorts of critical lines including silhouettes (borders of objects over a background or other objects), self occluding silhouettes, and crease edges (large variations in surrounding normals). Due to time constraints, we only considered major silhouettes.

To showcase the importance of edges, note figure (4). Here, we render a scene with our technique without edges. Notice that the larger sphere blends into the background surface, and it can be difficult to differentiate where one object begins and ends. Figure (5) re-renders the same image with an edge thickness of 0.01. This updated version works really well with our pencil shader as it includes the types of lines real life artists would use to sketch an image.

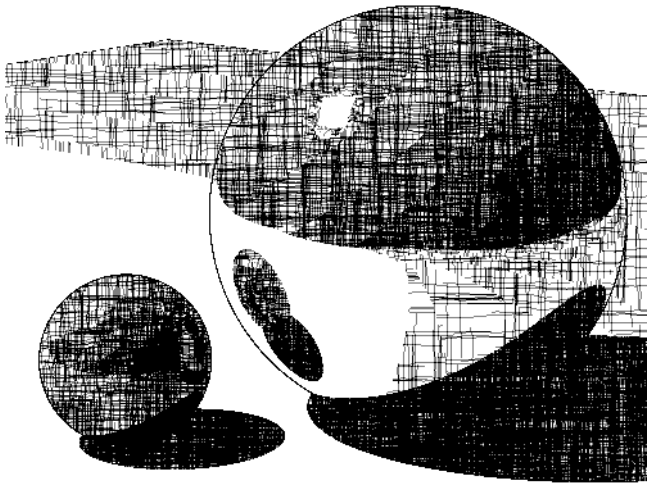


Figure 5: Spheres rendered with edges

5.1 Stencils

To achieve these silhouette edges, we construct nine additional rays for each pixel (none of which can generate a recursive bounce). We shoot one ray through the center of the pixel and determine the id

of the object it hits. We then shoot eight arrays in a circle of radius h around the center ray, with their positions calculated in image space. We tally up the number of different object ids that they hit, and note that a silhouette edge occurs when roughly 50% of the rays hit different objects.

Figure (6) is a simple visualization of our process. The edge width parameter h is simply the distance between the red center ray and any black stencil ray. A larger value of h will result in more pixels passing our edge test, which directly correlates to thicker borders on objects. This parameter should be experimentally determined for each particular scene. Our double sphere and watchtower examples were run with $h = 0.01$, while our multiple spheres (figure 11) were run with a larger value of $h = 0.1$.

If we have stencil rays hitting multiple different objects (in this case, the existence of green and black colored stencil rays), we know we should draw an edge. We only care about silhouette edges and use only one ring of samples. For higher quality edges with more potential features (e.g. self-occluding silhouettes), we direct you to [1] instead.

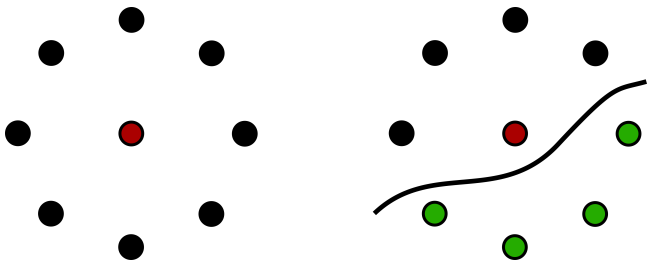


Figure 6: Stencil rays: (a) relative positions of each additional ray compared to the center ray (b) an example where the pixel color would return black

5.2 Antialiased Edges

One of the major feature of [1]'s algorithm is their ability to have smooth anti-aliased edges based on the percentage of the stencil rays that hit different objects. Closer to 50-50 results in a harder edge than an 80-20 split, as it corresponds to a darker shade of black. As the stencil ray hits get less evenly balanced, we linearly decrease the level of black into a lighter gray; this results in smooth curves in the final output.

We originally tried to implement this feature, but ran into some issues causing the doubling of edges (Figure 7). Time constraints caused us to scrap this bugfix entirely, so instead we stuck with non-antialiased edges. In our final results, we simply return a black color if anywhere between 10% and 90% of the stencil rays hit two different objects. We are reasonably happy with how the edges turned out regardless of these issues, and the pixelation is not particularly noticeable once we add the bitmaps on top.

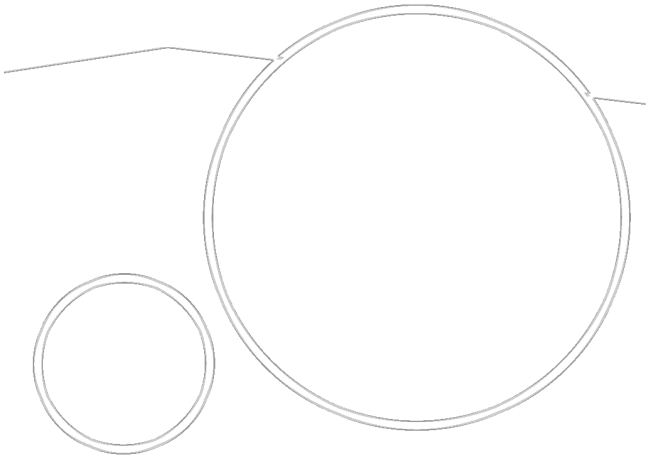


Figure 7: A buggy attempt at anti-aliased edges

6 PAPER SHADER

To increase the realism of the hand-drawn aesthetic, we have an additional post process step which adds a bit of modified Perlin noise to simulate the image being drawn on a piece of paper (figure 8). A noise map is generated and further discretized to increase contrast of the paper texture, and then applied on top of the combined bitmap image to yield the final result. This final image contains the hard edges from the ray tracing, the hatching from bitmaps, and the paper shader, the three core components that we set out to deliver.

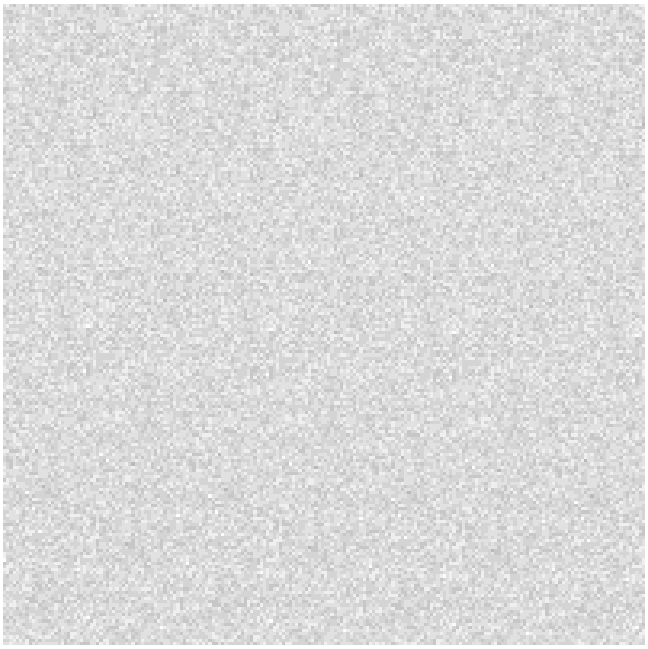


Figure 8: A small sample of the paper shader

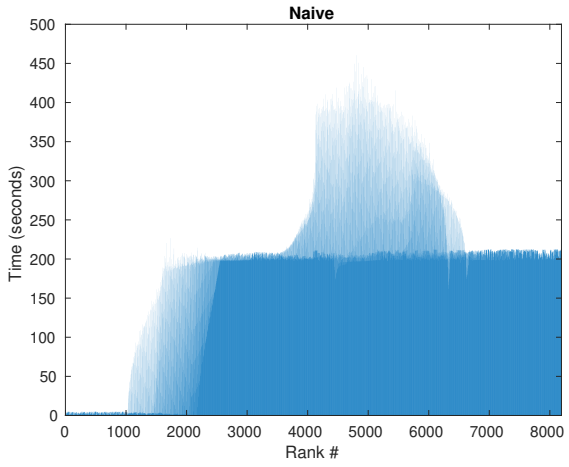
7 PARALLEL PERFORMANCE RESULTS

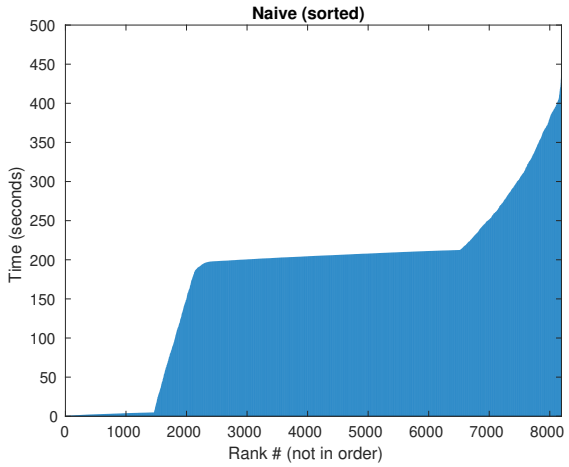
We can examine the completion time of each rank based on our various parallel algorithms by looking at the numbers obtained from our sphere test case. We first show the unsorted completion time of each rank, which is useful since we can identify which area of the image was being rendered. On the left, the lowest ranks typically handled the top of the image, which is mostly background. The spikes in the middle of some graphs roughly correspond to ranks which mostly calculated pixels for the two spheres, and the far right of each graph corresponds to ranks which were assigned pixels near the bottom of the image (most of which was a flat surface).

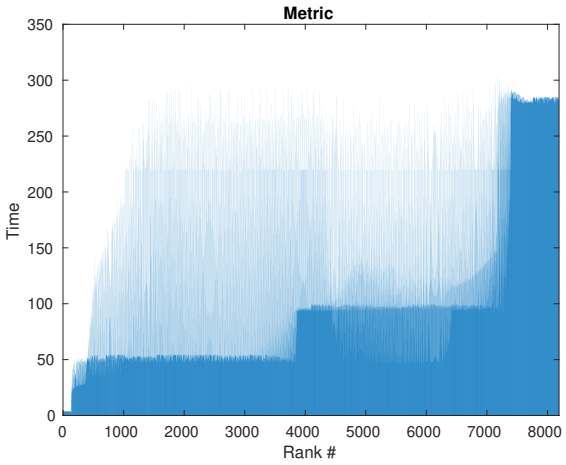
The second graph in each pair sorts the results by completion time. Here we can analyze how long each rank stayed alive to compute pixels relative to the lifetimes of the rest of the ranks. Ideally, we want to achieve a balanced result with all ranks finishing at the same time to avoid idling. Our earlier approaches fail to achieve this standard, and we can see the balancing in action with the last two graphs corresponding to our bargaining algorithm.

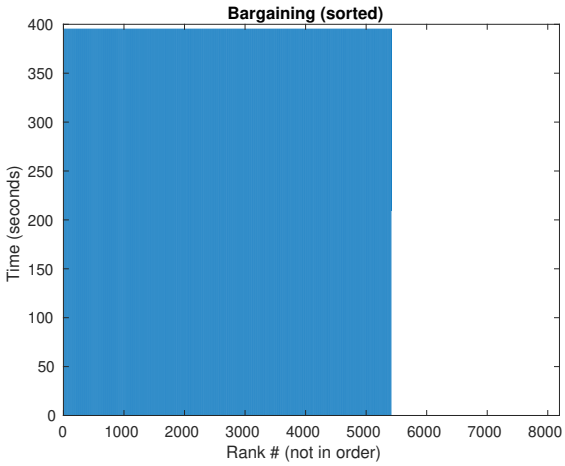
In [4], the authors investigate why their supercomputer was taking much longer than expected to complete basic calculation operations. Eventually, they discovered that OS noise, which matched the frequency of their operations, was severely degrading performance. In a previous implementation of the bargaining algorithm, our messaging thread had an "eager" approach towards message transmission and reception - after immediately processing messages to or from neighbors, it would resume listening for and sending information about the status of the ray-tracer thread on the same rank. However, this eagerness was degrading the performance of the ray-tracer thread, possibly due to OS scheduling - to the point where runs were expiring as they exceeded wall clock limits. As such, a one second delay introduced between the reception and transmission of interprocess messages substantially improved the performance of the overall ray tracing. A comparison between naive, metric, and bargaining yields an easy observation - the naive algorithm is certainly the worst - certain ranks finish much faster than others. For example, certain pixels in an image may require more processing as ray tracing may encounter materials that are reflective, or need to be checked for shadows, and so on.

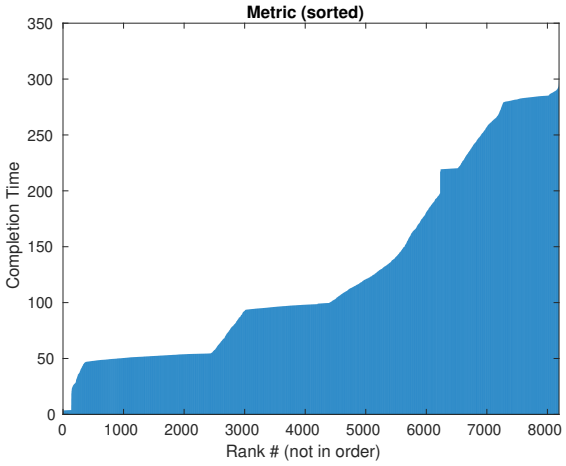
Of particular note is that our bargaining algorithm comes with a minor performance hit compared to the other cases. We still need to work on our threading and balancing code to ensure less impact on the main rendering thread.

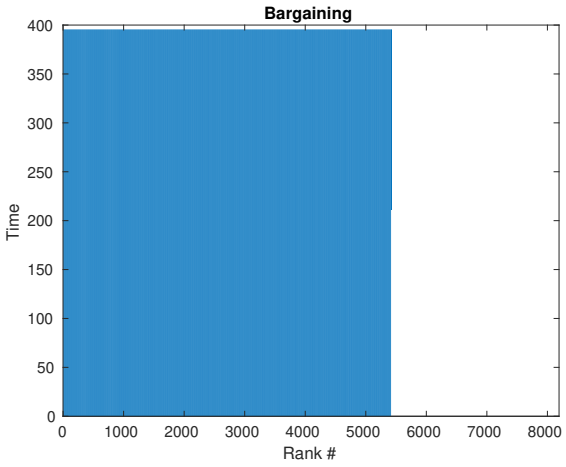






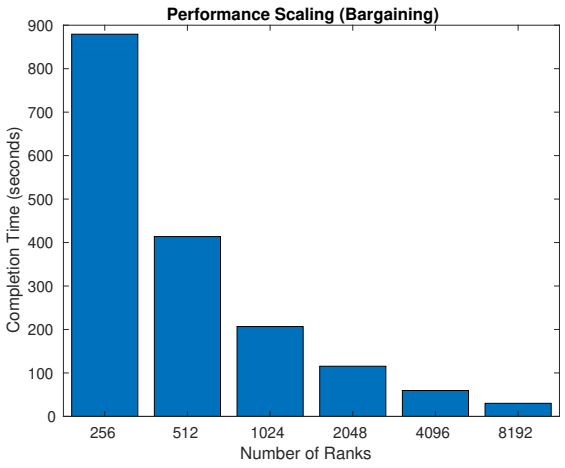






8 PERFORMANCE SCALING

We have a single graph to demonstrate how our major bargaining algorithm behaves with the addition of more physical resouces. As expected, we are able to achieve great speedup by adding more ranks. This is due to the ease in which ray tracing can be made parallel and is a good confirmation that our algorithm actually takes advantage of the existing resources.



9 TECHNICAL LIMITATIONS

We ran into a number of issues during this project that each needed a substantial amount of time and effort to solve. For one, our starting codebase (provided by Professor Cutler) depended greatly on library calls to OpenGL, a massive library which was not compatible with our goals of compiling for the Blue Gene. After stripping out all OpenGL code and reimplementing the necessary camera transformations, we also added a system to directly render results into a `ppm` file.

Furthermore, we needed to backport much of the modern C++ code of the original codebase into C++98, which took substantial effort. To complete the parallel portion of the project, we iterated

upon more and more efficient parallelization strategies as described in section (3).

Originally, we had real time progress updates occuring at each pixel with parallel file I/O; as each rank finished a pixel, it wrote the output instantly. This gives us the opportunity to see the progress at any particular point in time without issue, as well as stopping early due to timeouts or crashes giving us (useable) partial results.

This real time updates originally were intended to serve as a starting point for implementing very intuitive checkpointing. However, the Blue Gene did not allow us to have as many file I/O calls as required for larger renders. We instead chose to write batches of pixels at a time; in our final implementation we are able to leverage the chunk system described in section (3.4) as a natural division for combining the write calls into more efficient groupings. We still maintain the visual "progress bar", but the final results update much slower due to the batching.

Our access to the supercomputer here at RPI is limited to runs of 30 minutes in length. Rather than develop a specific system for getting around this limitation (which can be as trivial as indicating to our RayTrace routines which pixels to start and end at, rather than rendering the whole scene), we decided to just keep smaller scale test cases. Thus we could focus on making the few examples we have look as good as possible.

Our skill with LaTeX is still developing, so this writeup portion is not as visually appealing as it could be. We chose to have images relatively close (using forced "H" positioning) by the corresponding text. This resulted in some unnecessary white space that the typical "h" position hint would not generate.

10 FUTURE WORK

Our metric can be effective on some small examples, but has too much variance in its efficacy when the number of shadow samples goes up. Our collision detection algorithm is very primitive, as that was not a priority focus of our project. The expense of checking ray-object collisions adds up greatly with the sheer number of rays we end up shooting.

We can dive further into the ray-traced edge detection paper to obtain the full spectrum of edges required for many scenes. Our mesh processing techniques currently only works on quad meshes, so adding the logic for triangles is desirable but not done due to lack of time.

We can try and apply our technique to produce color pencil drawings, which we imagine should not be too difficult to achieve. Further improvements on the bitmap generation can be in producing smoother lines, more cohesive strokes, and a less pixelated output. Lastly, our paper shader has some noticeable repetitions in its generation. We think this may be an issue with the random number generator we used, and we think that we could improve it with some extra debugging efforts.

11 TEAM WORKLOAD

We decided to use the same codebase for final projects in two RPI computer science courses: Parallel Programming, taught by Professor Christopher Carothers, and Advanced Computer Graphics, taught by Professor Barbara Cutler. Casey worked on the initial porting of the ACG homework codebase to support the Blue Gene architecture, as well as the naive and cost-metric parallel implementations. He also implemented the edge detection algorithm for ray-traced silhouette edges. Jason worked on the bitmap generation, as well as the chunk interface and bargaining parallel implementation and the paper shader. Overall, both members spent a combined 75 hours of work on this project, split roughly halfway between the graphics component and the parallel component.

12 CONCLUSIONS

In this paper we present a strongly parallelized ray tracer which uses intelligent load balancing to maintain equal work between ranks. Further, we introduce a method for post processing ray traced renders to redraw them in a non-photorealistic hand-drawn result. Our tests examine the efficiency of our algorithm, and our results produce high quality wallpaper-ready images of arbitrary 3D scenes.

13 ACKNOWLEDGEMENTS

We would like to thank Professor Cutler for the initial raytracing code, Professor Carothers for his help with the supercomputer, Dennis Haupt of Free3D.com for the wooden watchtower model, and the invaluable assistance of Stack Overflow, without whom we would be nothing.

REFERENCES

[1] A. N. M. Imroz Choudhury and Steven G. Parker, *Ray tracing npr-style feature lines*, Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering (New York, NY, USA), NPAR '09, ACM, 2009, pp. 5–14.

[2] Robert L. Cook, Thomas Porter, and Loren Carpenter, *Seminal graphics*, ACM, New York, NY, USA, 1998, pp. 77–85.

[3] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein, *Stylized rendering techniques for scalable real-time 3d animation*, Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering (New York, NY, USA), NPAR '00, ACM, 2000, pp. 13–20.

[4] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin, *The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q*, Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (New York, NY, USA), SC '03, ACM, 2003, pp. 55–.

[5] Turner Whitted, *An improved illumination model for shaded display*, Commun. ACM **23** (1980), no. 6, 343–349.

14 IMAGES

The remaining pages are dedicated to housing various renders of our project.

Figure 9: Watchtower

Figure 10: Watchtower (dark version)

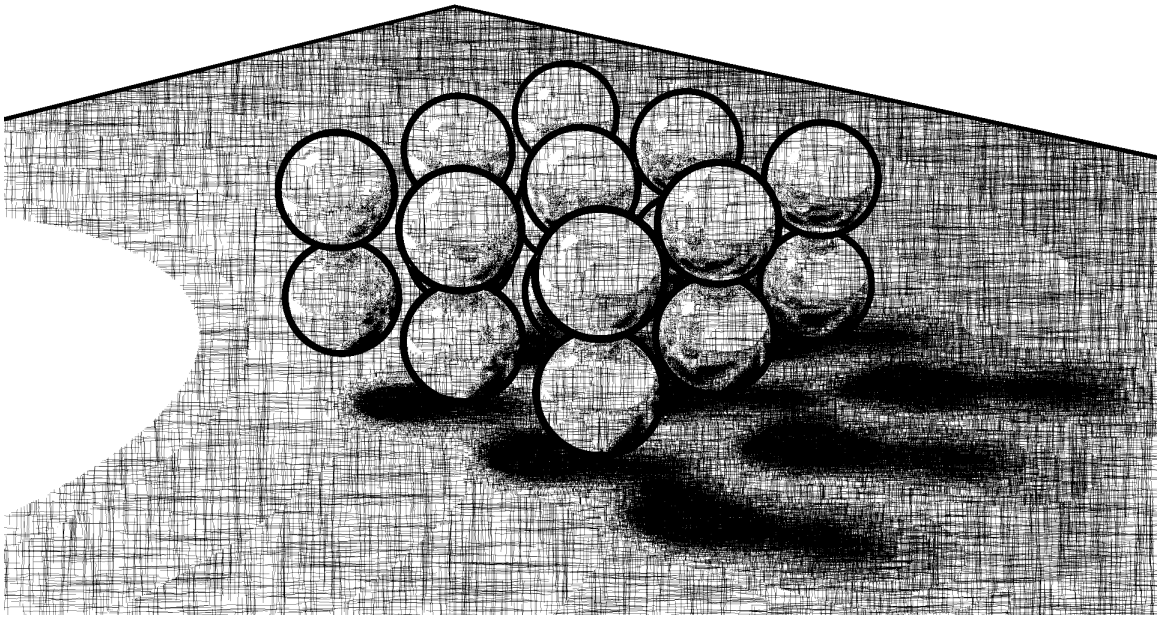


Figure 11: Multiple spheres (low shadow samples)

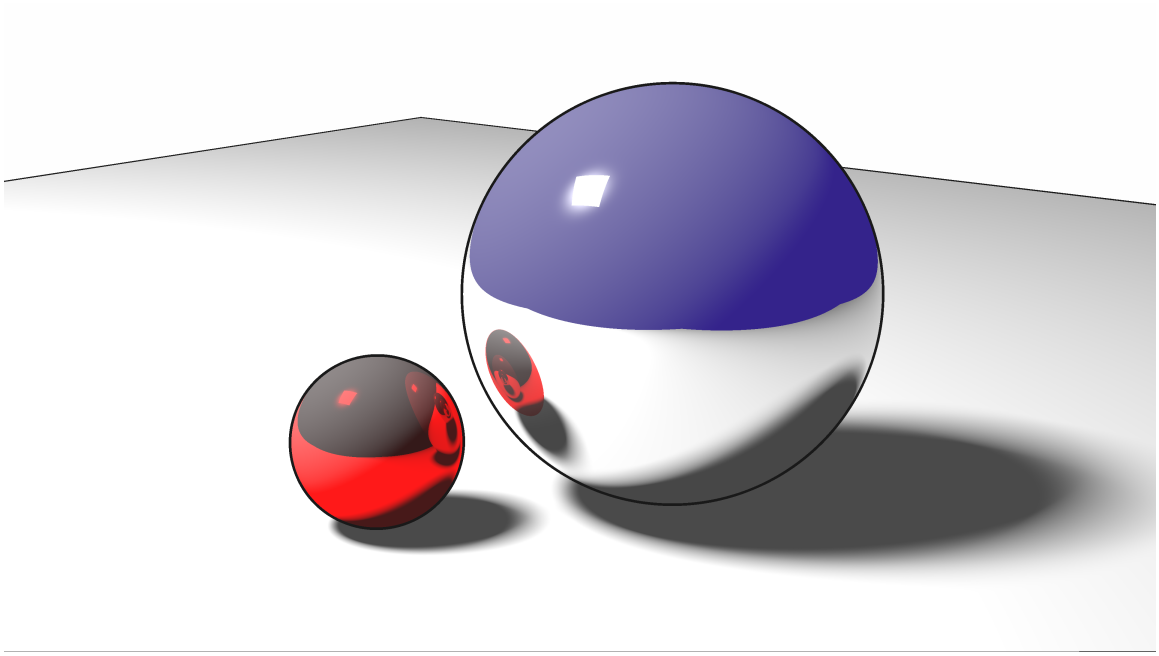


Figure 12: Reflective spheres (no post-process)

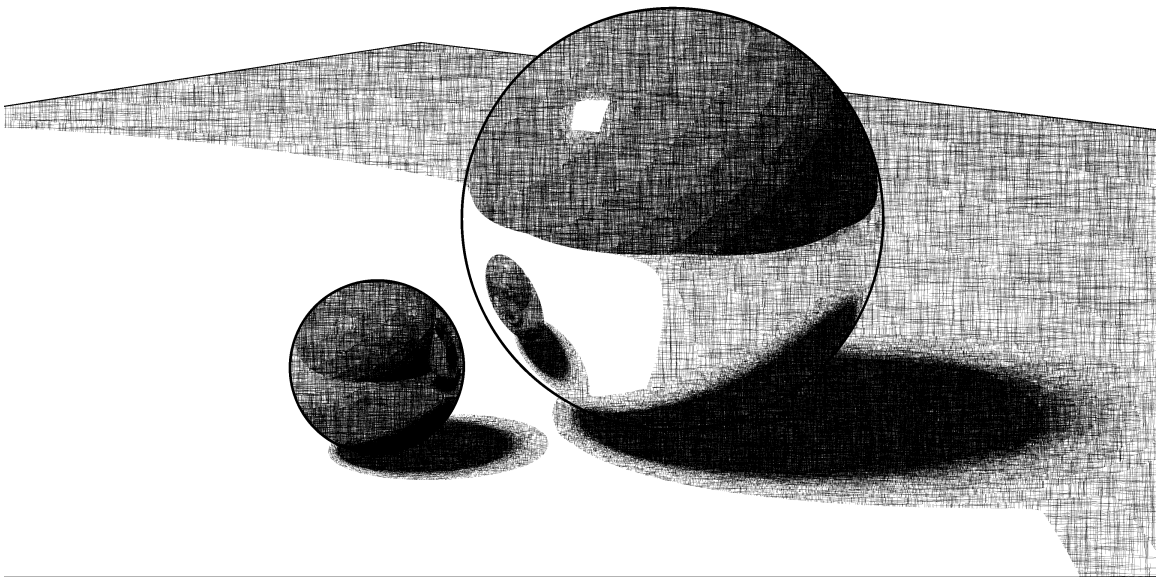


Figure 13: Reflective spheres (after post-process)

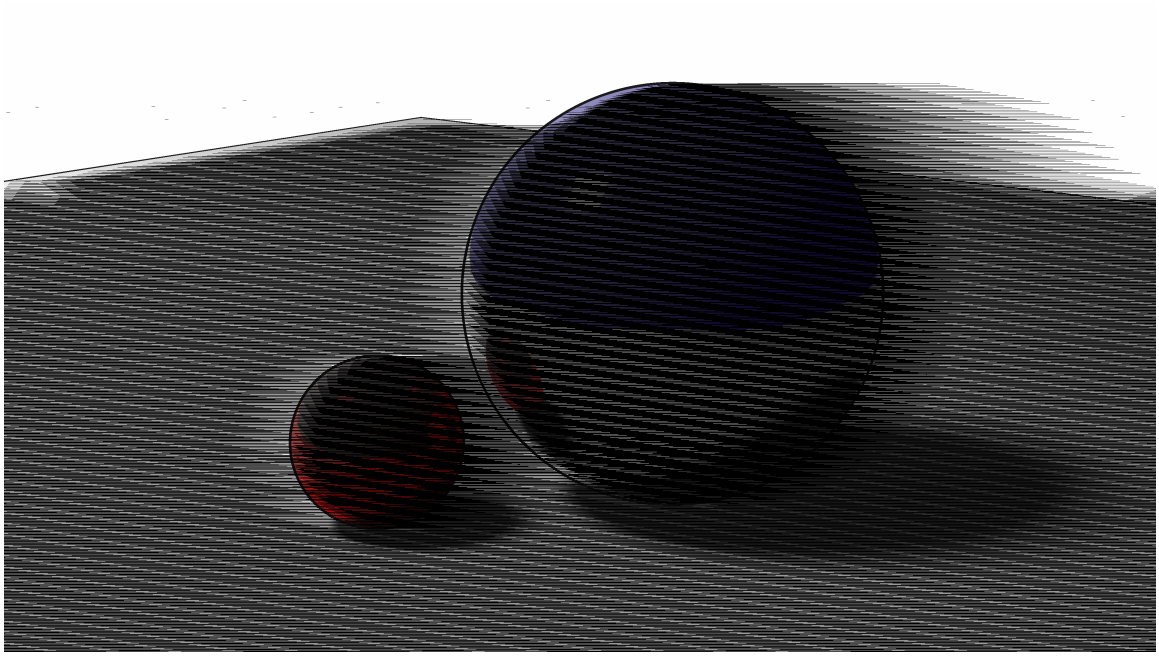


Figure 14: Mid-render progress of our bargaining algorithm; notice the diagonalization produced by sending chunks to neighboring processes

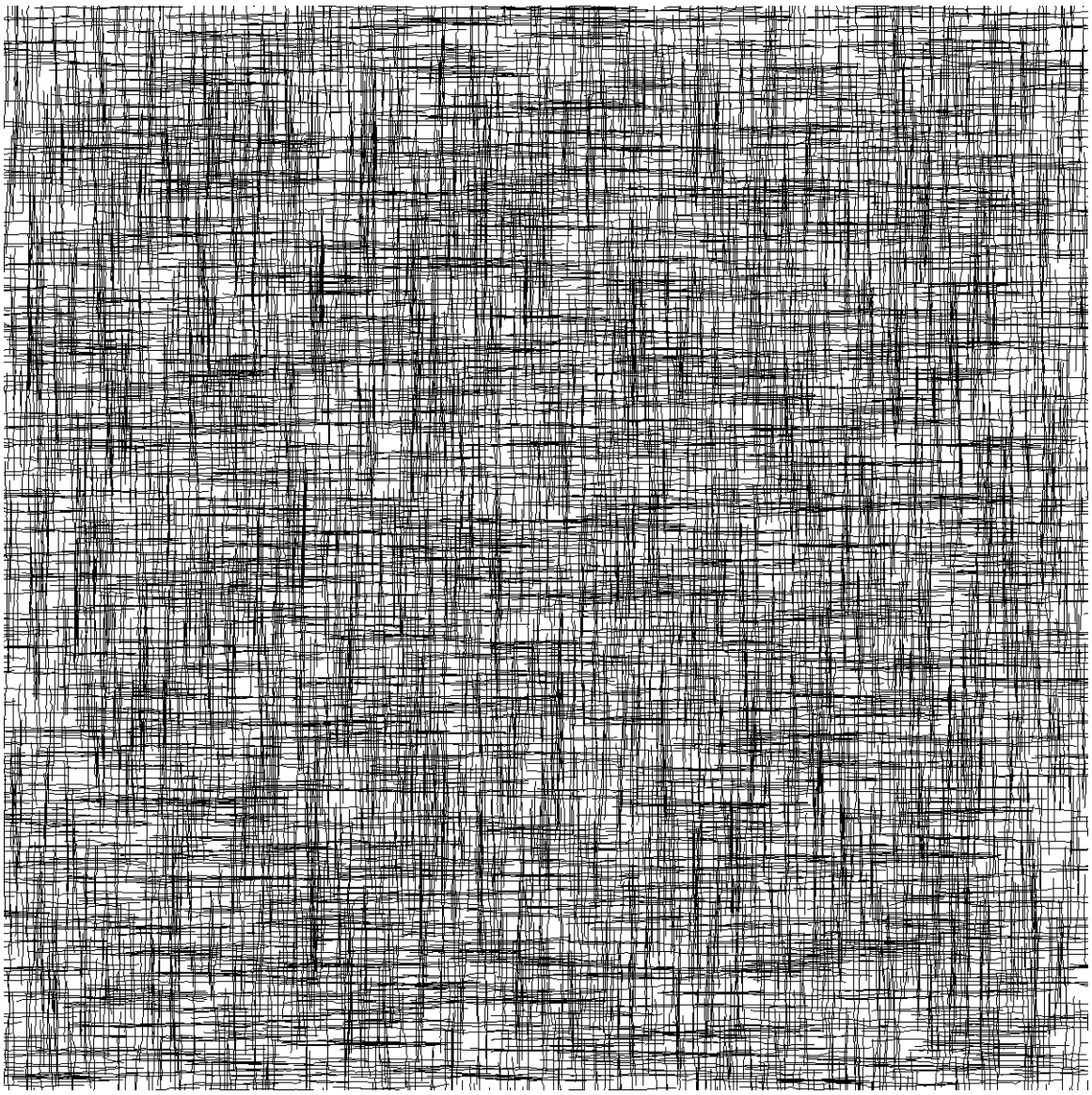


Figure 15: Close up of a bitmap with pencil strokes

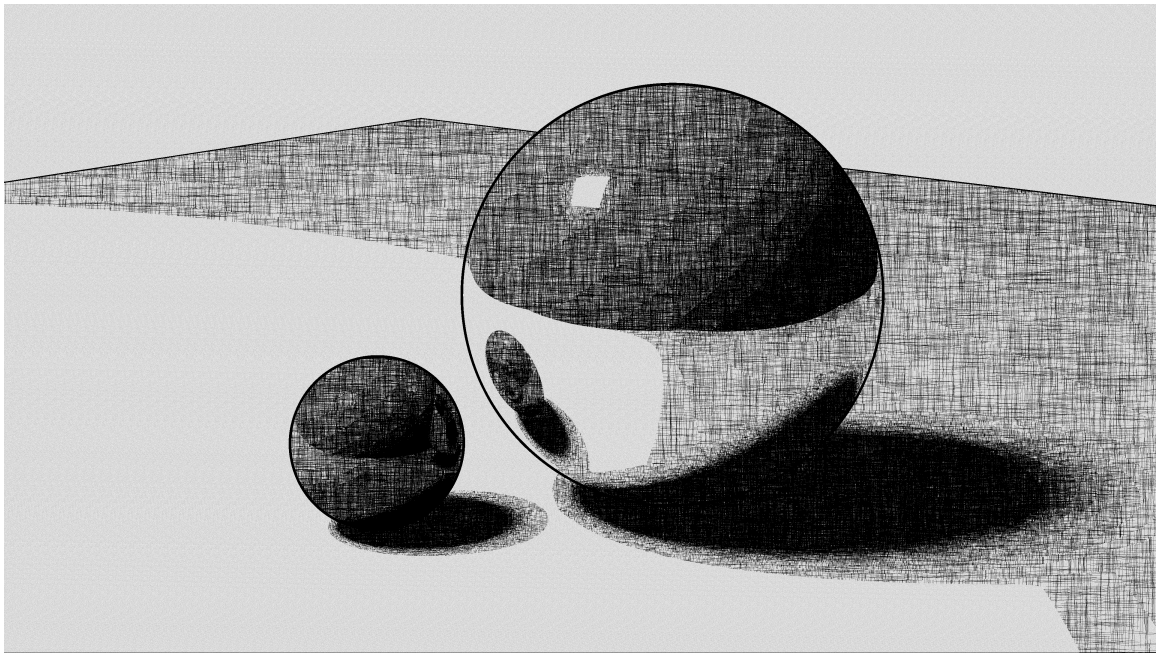


Figure 16: Reflective spheres with paper shader applied

Figure 17: Watchtower with paper shader

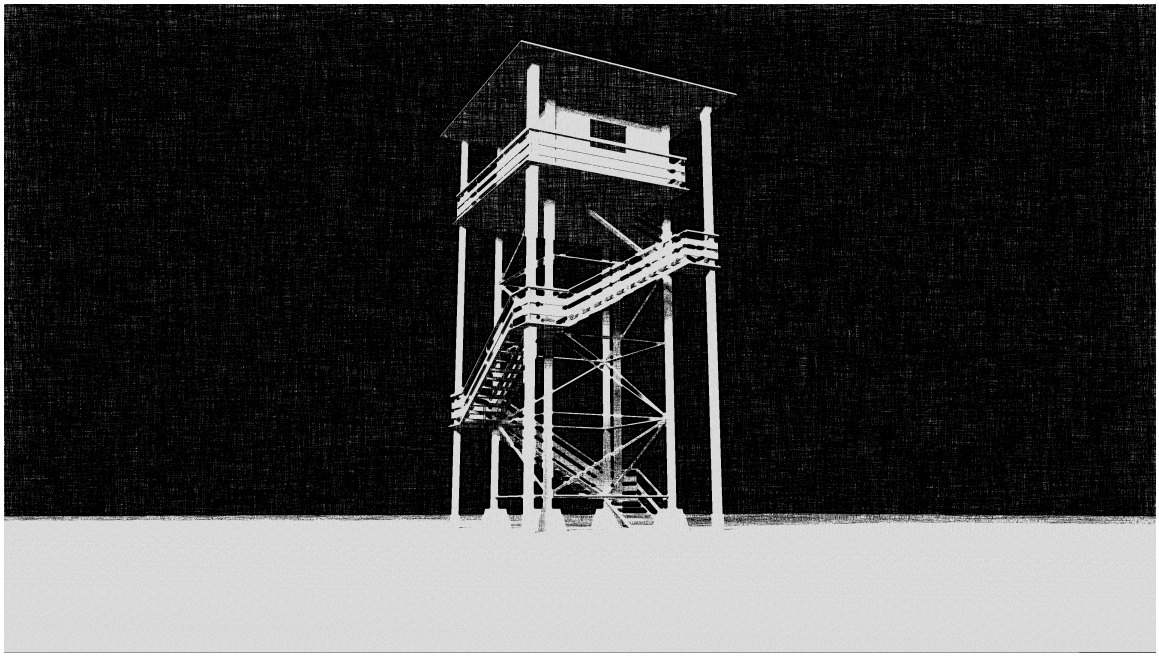


Figure 18: Watchtower (dark) with paper shader