Speeding Up a Simple Standard Ray-Tracer using Surface-Level Interpolation and Computational Geometry

Jamila Pegues[1], Cecilia Garraffo (Primary Adviser)[1], and Pavlos Protopapas (Secondary Adviser)[1]

[1]AC299R, Institute for Applied Computational Science (IACS), Harvard University

Ray-Tracing | Computer Graphics | Interpolation : Radial Basis Functions | Computational Geometry | Performance

1. Motivation

Ray-tracing is a fascinating technique for simulating how objects and light interact in a three-dimensional (3D) space. It is used for a variety of applications, from digital renderings of shadows to simulations of irradiated molecular gases in astrophysics. The technique is particularly well-lauded in computer graphics, where ray-tracing is used in animated films, digital art, and much more (Suffern, 2007).

Despite its theoretical simplicity, however, standard ray-tracing is inherently computationally expensive. Standard ray-tracing involves iterating through each pixel in the image, sending a ray from that pixel, and calculating that pixel's irradiance and color from the ray's interaction with light and objects in the 3D space. Therefore the computational intensity (i.e., the runtime) of a standard ray-tracer increases as the number of pixels, number of lights, and the number of objects in the space increases. This is certainly an undesirable outcome for rendering more complex (and thus most likely more interesting) scenes at higher resolution. It is the reason that applications with static simulated scenes, such as animated/CGI films and digital art, take such a painstakingly long time to produce. It is also the reason that dynamic systems, such as video games, which rely on real-time updates and computation, cannot afford to wait so long for scenes to be rendered.

A typical approach to decreasing the runtime of standard ray-tracers has been to throw more resources at the problem - e.g., by tracing rays in parallel, employing GPUs, and/or by increasing the number of cores used for the calculations. Another approach is to use clever tricks and approximations, such as rasterization, so that the viewer *thinks* the scene has been ray-traced (as long as they do not look too closely).

Here we take a different approach. We modify a simple standard ray-tracer using surface-level interpolation and computational geometry. We then evaluate how our modified ray-tracer performs relative to a simple standard ray-tracer in terms of both runtime and accuracy. Finally, we discuss ways that our modified ray-tracer can be improved and expanded through future work.

2. The Modified Ray-Tracer

A. Terminology. We briefly review the terminology we use for this report. For more discussion, see Suffern (2007).

We use the window-based framework, which treats ray-tracing like viewing the world through a window. We use ***world*** to refer to a collection of objects and light sources within three-dimensional

(3D) space. We use ***camera*** to represent the viewer's perspective (in other words, the viewer's 'eye'). The camera views the world through the ***window***, which refers to a two-dimensional (2D) grid of pixels. Finally, a ***scene*** refers to a set containing a camera, a window, and a world altogether.

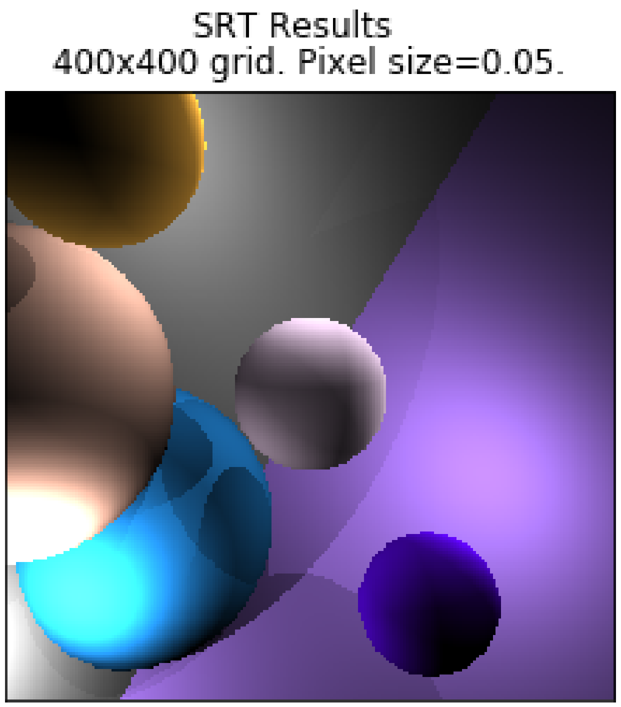


Fig. 1. Rendering (using a Simple Standard Ray-Tracer) of a scene containing 5 spheres, 2 planes, and 4 point light sources. Technical details of the scene (e.g., pixel size) are given in the title.

B. The Algorithm.

We now present the make-up of our modified ray-tracer (MRT).

The overarching goal of the MRT is to approximate a fully ray-traced image. The MRT aims to reduce the computational intensity (i.e., runtime) of rendering an image, without sacrificing too much accuracy of the final image relative to the exact ray-traced result. In practice, our MRT algorithm does the following: it fully ray-traces a subset of pixels in the window, and then uses the measured irradiance (i.e., light reflected back onto that pixel) for that known subset to "guess" the irradiance and color of the remaining unknown pixels.

Our MRT's approach can be broken down into three major sequential components: (1) the underlying ray-tracer, (2) the edge finder, and (3) the interpolator. We discuss each of these components in detail in the subsequent subsections. We will use the scene presented in Figure 1 as a working example.

B.1. The Underlying Ray-Tracer.

We built our MRT on top of a *very* simple standard ray-tracer. Our underlying ray-tracer (Algorithm 1) is nothing special, is not parallelized, and is not efficient (e.g., it is written in Python, rather than, say, Julia or C++). There are many other standard ray-tracers out there that are faster, sleeker, and far more fantastic then the simple one we built here. We emphasize that the *modifications* are what make our MRT special, not the underlying ray-tracer.

Algorithm 1 Part 1 of the MRT: The Underlying Ray-Tracer.

```
1: Define a world: light sources, objects
2: Define a camera: shape, orientation
3: Define a window: pixel size, length, height, orientation
4: Extract a grid-based subset $p_k$ of pixels in the window
5: for each pixel in the subset $p_k$, do
6:     Send ray from camera through current pixel
7:     Determine if ray intersects any objects in the world beyond the window
8:     if intersection exists, then
9:         Determine the intersecting object that is closest to the pixel
10:        Store a unique id=(positive integer) that corresponds to that object and maps to this pixel
11:        Compute the irradiance and color of the pixel, based on object's material and world's light sources
12:        Store the irradiance and color of the pixel
13:        if the intersection point is blocked from all light, then
14:            Uniquely alter the id of this pixel (e.g., add 0.5)
15:    else
16:        Store id=0 for this pixel, representing empty space
17:        Store the color of this pixel as black (representing the absence of color)
```

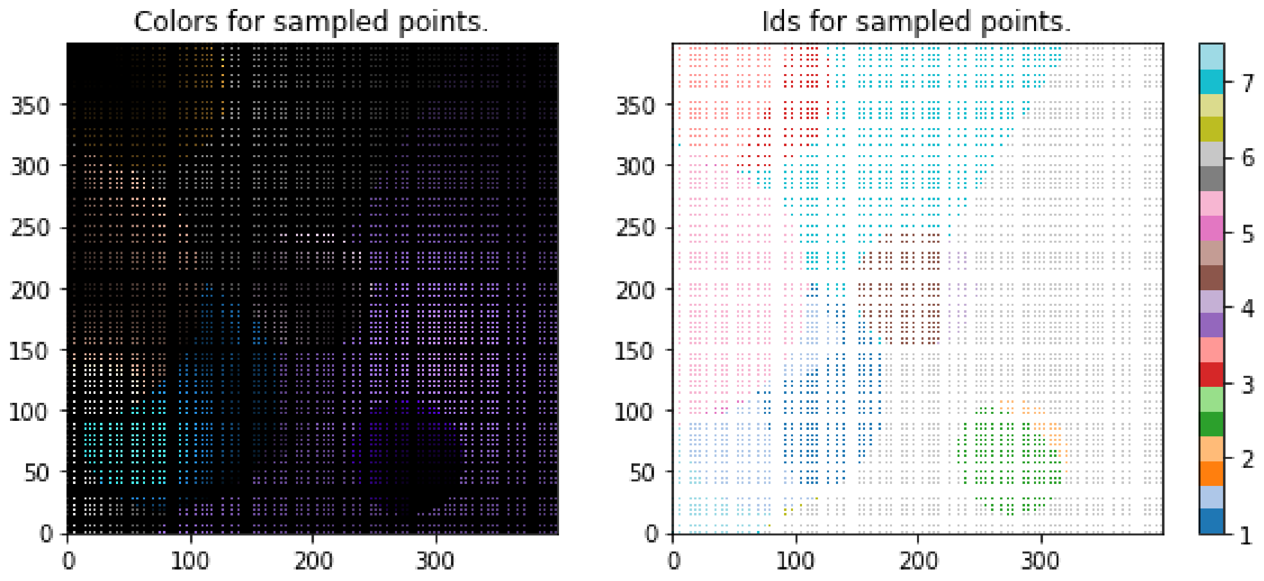


Fig. 2. Left: The grid-sampled subset of pixels for the scene in Figure 1 that have been fully ray-traced. The sample size is ∼10% of the original number of pixels. **Right:** The exact ids for the sampled subset of known pixels.

We only use the underlying ray-tracer as a vehicle to test our modifications. We highly encourage readers to try these modifications out on their own (presumably fancier and more efficient) standard ray-tracers, and to let us know of their results.

With that said, from here on we will assume the reader has familiarity with how standard ray-tracers work. We will focus on presenting the modifications of our MRT rather than describing the underlying ray-tracer in any detail. We only note that our ray-tracer supports a few world components (planes, spheres, and point light sources), and that all materials are assumed to be "dull" (i.e., not mirror-reflective). For a detailed description of the standard ray-tracing approach, refer to any of the extensive references and documentation that exist elsewhere (e.g., the fantastic ray-tracing textbook by Suffern, 2007).

The MRT relies on a subset of fully ray-traced pixels in the window, which the MRT uses to "guess" the irradiance, and therefore color, of the unknown pixels. The MRT selects this subset on a fixed grid (i.e., all sampled pixels in the window are equally-spaced), because even sampling of the window provides an even sampling of all objects seen within that window. Figure 2 displays the grid-sampled subset of pixels, which have been fully ray-traced, for the scene depicted in Figure 1. This sampling step consumes most of the MRT's runtime, but we have found from our tests (Section 3) that the fraction of the total pixels sampled does not have to be large (e.g., 10%).

An important step for the underlying ray-tracer is to take note of the specific objects that each sampled pixel's ray intersects. In the case of our dull, non-reflective object materials, the first intersected object is the most important. Essentially, we "tag" the pixel with the object's unique id (i.e., a positive integer). If that point of intersection is in shadow (i.e., blocked from all light sources), we take note of that too (i.e., by adding 0.5 to the stored unique id). If the pixel's ray intersects nothing at all (i.e., sees only empty space), then we tag the pixel accordingly (i.e., with an id of 0). These ids are crucial for the next step of our MRT algorithm.

Algorithm 2 Part 2 of the MRT: The Edge Finder.

```
 1: Divide the image into four rectangles
 2: for each rectangle $R$, do
 3:     if all known pixels $p_k \in R$ have the same id, then
 4:         Assign all unknown pixels $p_u \in R$ to have that same id
 5:     else
 6:         Divide this rectangle into four more rectangles, as able
 7:         Recurse from Step 2
 8: for each id $i$, do
 9:     Temporarily represent all pixels with id=$i$ with value 1
10:     Temporarily represent all pixels with id$\neq i$ with value 0
11:     Apply a Gaussian filter (which has a sigma parameter) to this binary representation
12:     Normalize this binary representation to 1.0
13:     Set all pixels with a binary representation value above a cutoff (e.g., 0.05) to have id=$i$
```

B.2. The Edge Finder. We now have a subset of sampled pixels for which we know exactly what the irradiance and final color are. Based on these known values, we want to "guess" (i.e., interpolate) what the unknown values of the unknown pixels are. This is quite difficult to do flexibly for a large, complex image. Each of the objects in a scene can be positioned anywhere in space, and so their surfaces will display different patterns of irradiance. These objects can also block each other from different light sources, leading to regions of both soft and hard shadows in our final image.

We could try to quickly interpolate the entire image, all at once, given those known pixels. However, this form of interpolation assumes that the surfaces behave according to some function

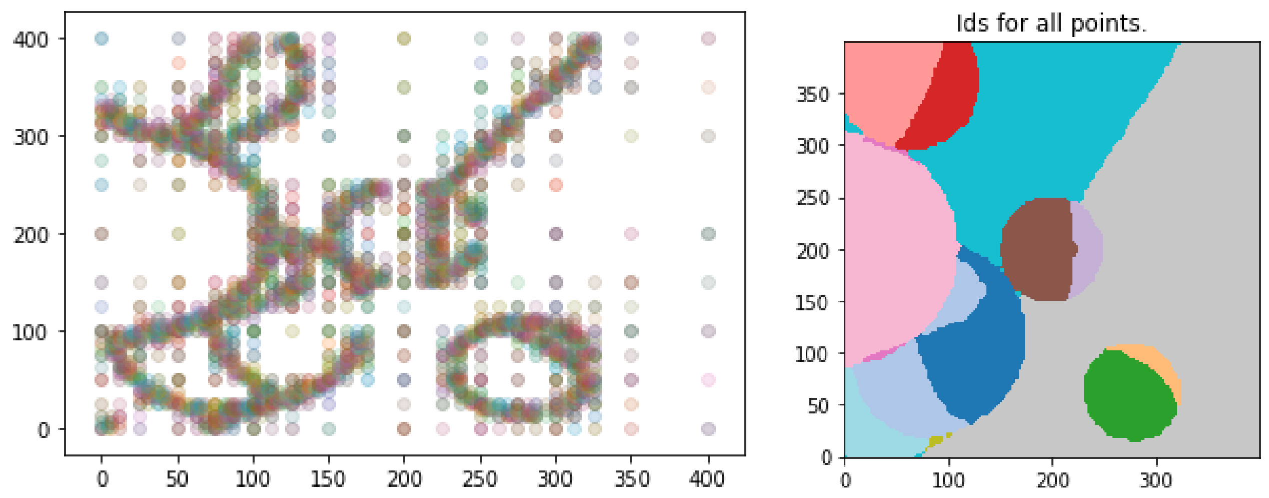


Fig. 3. Left: A trace of the edge-finding algorithm. Each dot corresponds to the corner of a rectangle explored by the algorithm. Note how the smallest rectangles were formed around the edges of different objects and regions of shadow. **Right:** The ids assigned to the unknown pixels. Note the jagged edges.

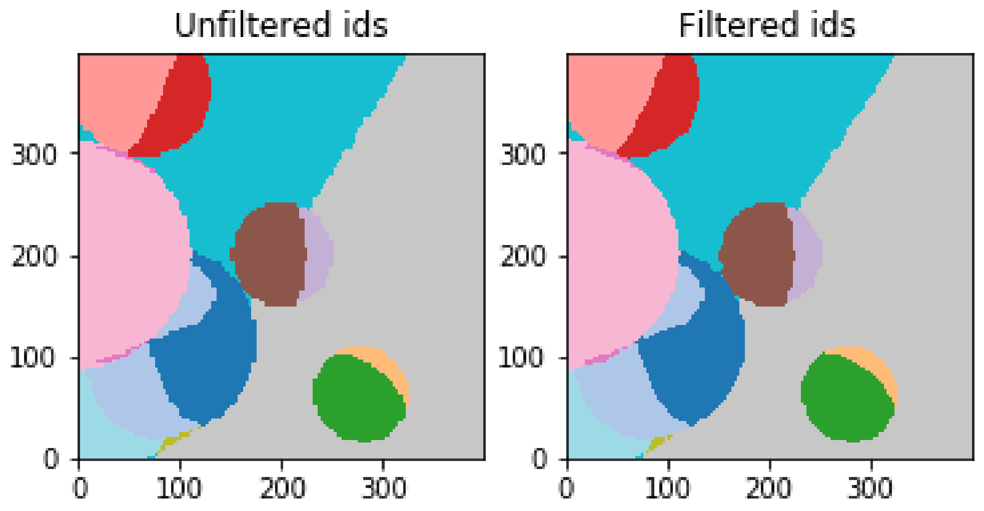


Fig. 4. The assigned ids before (left) and after (right) smoothing them with a Gaussian filter.

or model (e.g., a high-degree polynomial or Gaussian). Unfortunately, it is quite difficult to come up with a single function/model that accurately describes a complex image, especially when that image has multiple objects and regions of shadows. We could instead try performing more localized interpolation (i.e., an spline) of the unknown pixels using the known pixels. Unfortunately, while this could certainly be an accurate approach, local interpolation is extremely slow and expensive for large numbers of pixels, as it requires heavy calculations at each individual pixel.

So before we proceed to the interpolation step, we first build a sense of the "edges" in our images, namely the edges of different objects and shadows, using an edge-finding algorithm (Algorithm 2). This algorithm uses a recursive space partitioning approach that is based on the k-d tree procedure. For this algorithm, we first split the entire image into four rectangles. For each rectangle, we check to see if all ids contained within that rectangle are the same (i.e., correspond to the same object or region). If the ids in the rectangle are the same, then we assign all unknown pixels within that rectangle to have the same id. However, if any two ids in the rectangle are different, then we further divide that rectangle into four smaller rectangles. We then repeat the entire process for each of those new rectangles. Figure 2 displays the exact ids for the sampled pixels of our example. Figure 3 then illustrates the recursive space partitioning process and the ids assigned to the unknown pixels.

The downside of this approach is that it leads to jagged edges, which get worse as the number of sampled points decreases. We mitigated the "jaggedness" by applying a Gaussian filter to each collection of pixels with the same id relative to all other pixels with a different id (this step is included in Algorithm 2). Figure 4 shows that this filtering is not perfect. We discuss possible improvements to this smoothing approach in Section 4.

Algorithm 3 Part 3 of the MRT: The Interpolator.

```
1: for each id $i$, do
2:     Define $p_{\mathrm{k,i}}$=(all known pixels with id=$i$)
3:     Define $p_{\mathrm{u,i}}$=(all unknown pixels with id=$i$)
4:     Extract all pixels $c_{\mathrm{C}} \in p_{\mathrm{k,i}}$ that make up the convex hull of $p_{\mathrm{k,i}}$
5:     Extract subset (e.g., 10) of pixels $c_+ \in p_{\mathrm{k,i}}$ that have the highest irradiance values of all pixels in $p_{\mathrm{k,i}}$
6:     Extract subset (e.g., 10) of pixels $c_- \in p_{\mathrm{k,i}}$ that have the lowest irradiance values of all pixels in $p_{\mathrm{k,i}}$
7:     Extract limited subset (e.g., up to 1000) of pixels $c_{\mathrm{in}} \in p_{\mathrm{k,i}}$ that are $\notin c_{\mathrm{C}}$, $\notin c_+$, and $\notin c_-$
8:     Build a linear radial basis function (RBF) interpolator using $c_{\mathrm{in}}$, $c_{\mathrm{C}}$, $c_+$, and $c_-$
9:     Use the linear RBF interpolator to interpolate all pixels in $p_{\mathrm{u,i}}$
10:    Calculate the colors for $p_{\mathrm{u,i}}$ from the interpolated irradiance values
```

B.3. The Interpolator. Once we have split all pixels by their ids, we are ready to interpolate the subset of pixels with each id separately, without fear of the complications that arise from interpolating a complex image all at once. We use a radial basis function (RBF) to carry out our interpolation. This was chosen only after much trial and error (including lengthy experimentation with high-degree polynomials and tapered Gaussians). We specifically use SCIPY.INTERPOLATOR's linear RBF interpolator, as it is much faster than any RBF we coded up ourselves.

Radial basis function (RBF) interpolators are quite accurate. Unfortunately, they are also quite computationally expensive. Their runtimes increase significantly as the number of pixels increase, which is of course *never* desirable for a ray-tracer. To avoid incurring these costs while still maintaining high levels of accuracy, we first force the RBF interpolator to interpolate from known "capstone" pixels. The "capstone" pixels include (1) all known pixels with the given id that form the convex hull of those pixels, (2) the top ten brightest known pixels with the given id, and (3) the top ten darkest known pixels with the given id. By forcing the RBF interpolator to include

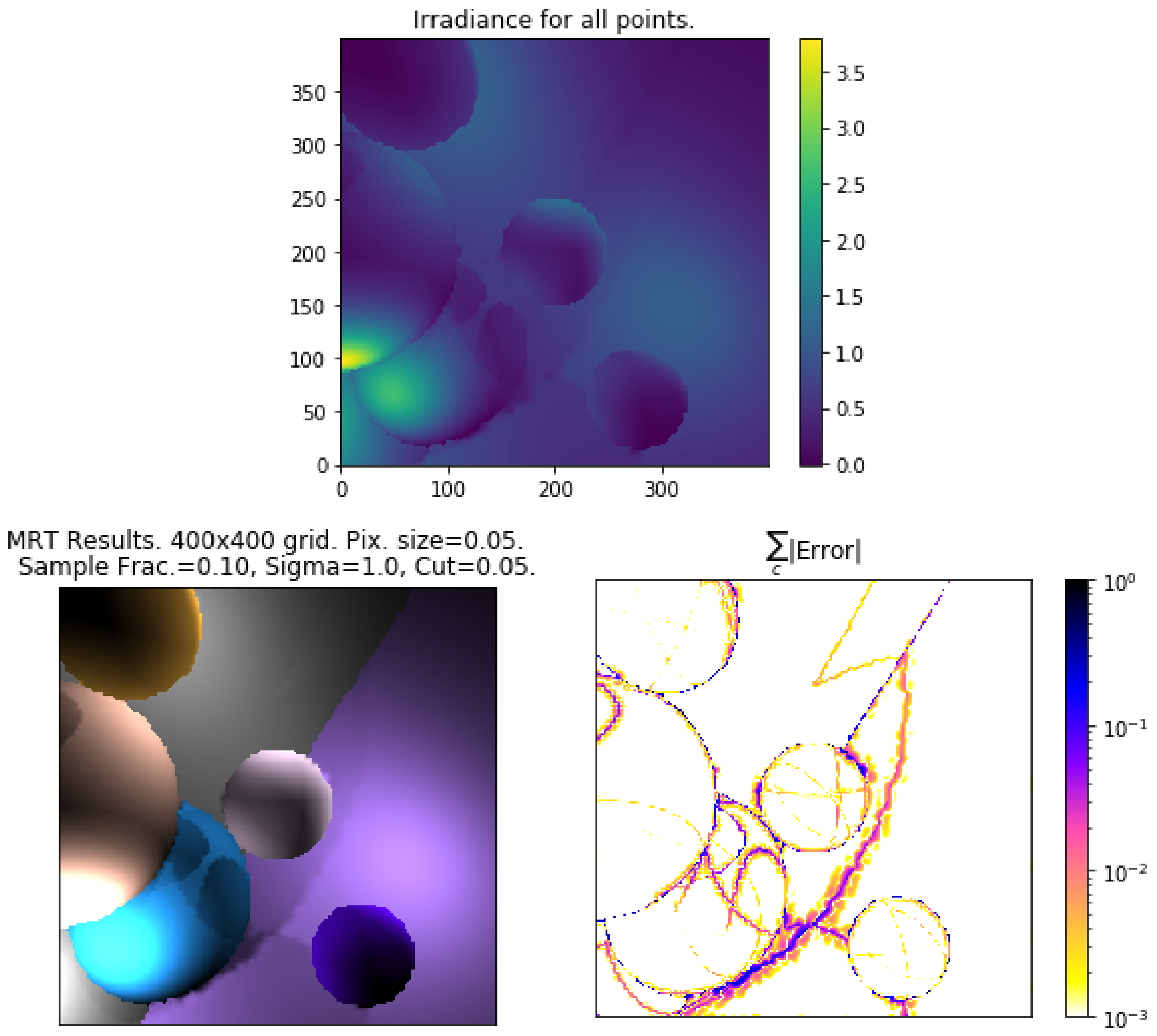


Fig. 5. Top: The interpolated irradiance of all pixels within the image. **Bottom Left:** Rendering (using the Modified Ray-Tracer) of a scene containing 5 spheres, 2 planes, and 4 point light sources. Technical details of the scene (e.g., pixel size) are given in the title. **Bottom Right:** Error per pixel in the MRT result (calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3).

these capstone pixels, we are reducing the error produced by RBFs at boundaries. We then add in a grid-based sample of "secondary" pixels, which are extracted from the set of known, non-capstone pixels with the given id. We include secondary pixels *only up to a maximum amount*. In other words, we directly prevent the RBF from using too many pixels, thus reducing the RBF interpolator's runtime. For this report, we use the 10 brightest and the 10 darkest pixels as part of the capstone pixels, and we include up to 2000 secondary pixels beyond that set.

We "train" the RBF interpolator on the capstone pixels and secondary pixels, and then we use the interpolator to interpolate all unknown pixels with the same id. We do this for each id in the image, until we have eventually interpolated the entire image. Figure 5 shows the interpolated irradiance for the entire image, the scene from Figure 1 as rendered by the MRT, and the absolute error between the SRT and MRT results. We evaluate the MRT's performance in Section 3.

3. Analysis

We now use the example from Figures 1 and 5 and a series of test cases to evaluate the performance (i.e., accuracy and runtime) of the MRT relative to the SRT. Please note that the example was rendered using the provided "example_MRT.ipynb" script, while all test cases were rendered and timed using the provided "test_MRT.ipynb" script.

A. General Performance. From the error presented in Figure 5, we can see that the SRT and MRT results are nearly identical. By eye, we cannot detect any difference in the colored inner surfaces of the objects, or in the colored inner regions of shadows. However, we see immediately that there are large errors along the edges of the objects and shadows. We can clearly see that the edges of regions of shadow in the MRT image are not fully smooth, and the edges of the spheres are notably warped. These errors show that more work can still be done to improve the edge-finding component of the MRT algorithm. We discuss possible improvements in Section 4.

B. Performance vs Sampling Fraction. We first test the dependence of performance on the sampling fraction (i.e., the number of fully ray-traced pixels that we use to train our interpolator). Figure 6 plots the MRT error and runtime as a function of sampling fraction. Snapshots of performance are shown in both Figure 6 and Figure A.

The MRT error is highest for the smallest sampling fraction considered, which is what we would intuitively expect. Error decreases sharply up until a fraction of $\sim$0.05. Surprisingly, the errors for fractions $\geq$0.05 are roughly constant. This result implies that the underlying RBF interpolator is able to interpolate fairly well even from fairly small samples of data.

The MRT's relative runtime (i.e., (MRT runtime / SRT runtime)) increases with sampling fraction. There are notable breaks in the trend, such that the slope of the trend becomes shallower past certain values of sampling fraction. These breaks may be due to the maximum limit placed on the number of secondary pixels allowed. As the sampling fraction increases, the number of pixels assigned to a particular id also increases. However, the maximum limit prevents the underlying RBF interpolator from using more than a specified number of secondary pixels, Therefore the RBF interpolation runtime per id becomes fixed once this maximum limit is exceeded. We thus expect that the slope of the trend in runtime with sampling fraction will become shallower and shallower, eventually becoming constant as each collection of secondary pixels exceeds that limit.

We use a sampling fraction of 0.1 for the remainder of this report.

C. Performance vs Filter Parameters. We next test the dependence of MRT performance on the Gaussian filter parameters, namely sigma and the normalized cutoff. Figure 7 plots the MRT error

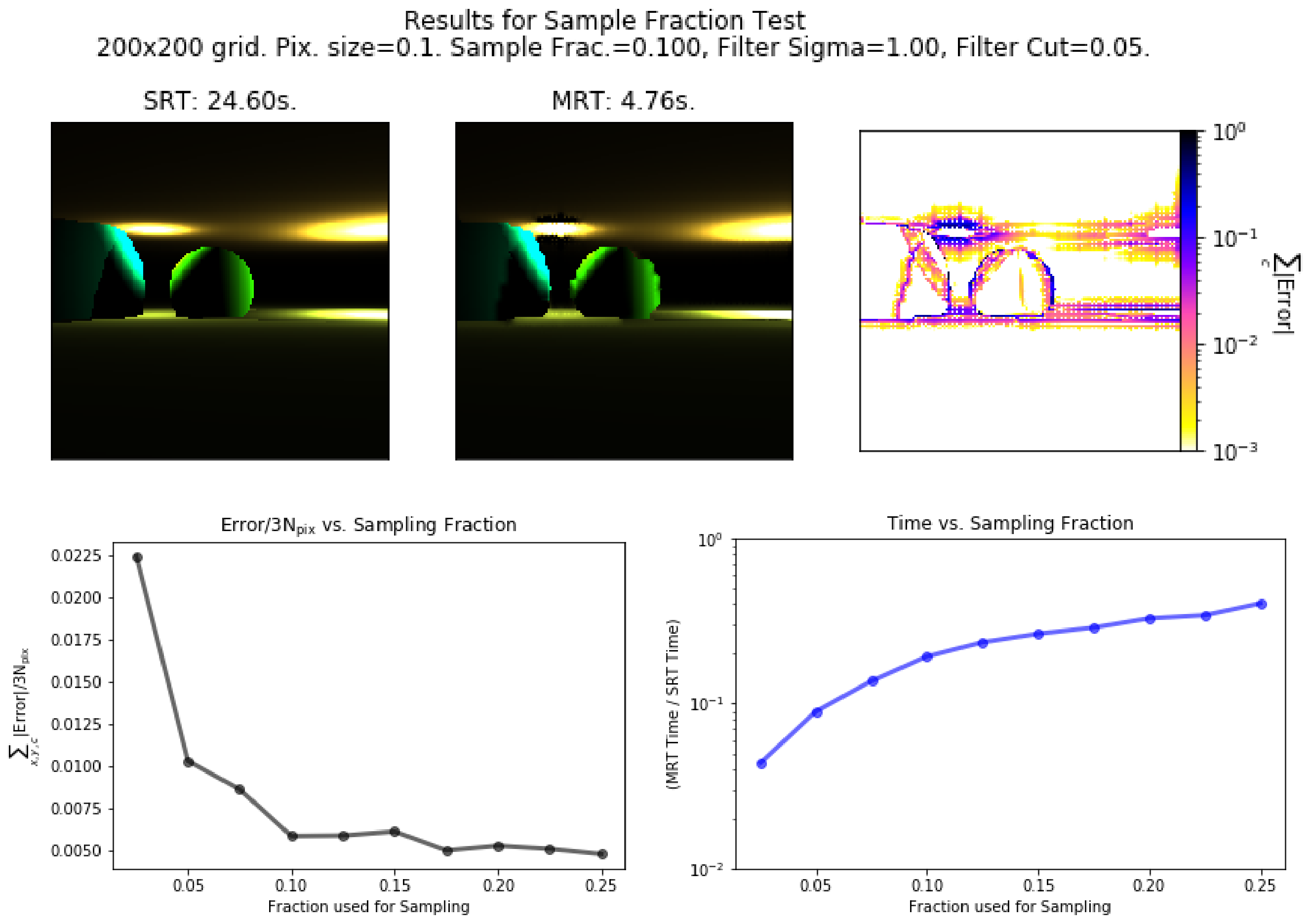


Fig. 6. Top: A snapshot of MRT accuracy for a sampling fraction of 0.1. Other snapshots are shown in Figure A. The snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 2 spheres, 2 planes, and 3 point light sources, was constructed specifically to have long, narrow regions of irradiance. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles. **Bottom:** MRT accuracy (left) and relative runtime (right) as a function of sampling fraction.

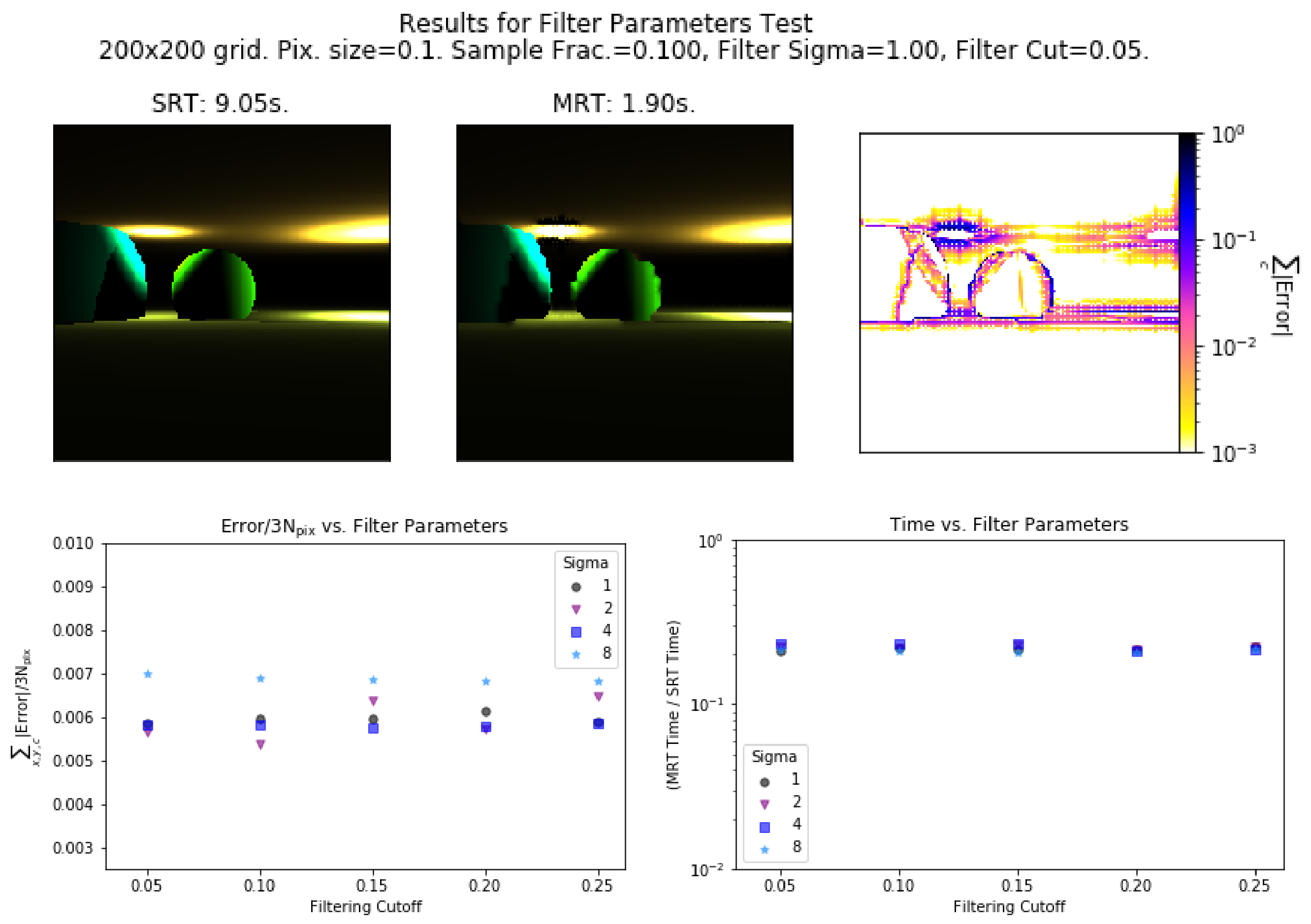


Fig. 7. Top: A snapshot of MRT accuracy for a Gaussian filter sigma of 1 and a normalized cutoff of 0.05. Other snapshots are shown in Figure B. The snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 2 spheres, 2 planes, and 3 point light sources, was constructed specifically to have long, narrow regions of irradiance. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles. **Bottom:** MRT accuracy (left) and relative runtime (right) as a function of filter parameters.

and runtime as a function of Gaussian filter parameter, and snapshots of performance are shown in both Figure 7 and Figure B.

We note that error tends to increase as sigma increases. Changes in cutoff appear less significant; error either stays roughly constant or slightly increases as the cutoff increases. The relative runtime does not seem to depend on either sigma or the cutoff, because it does not change significantly as either factor increases. We set the sigma to 1 and the cutoff to 0.05 for the remainder of this report.

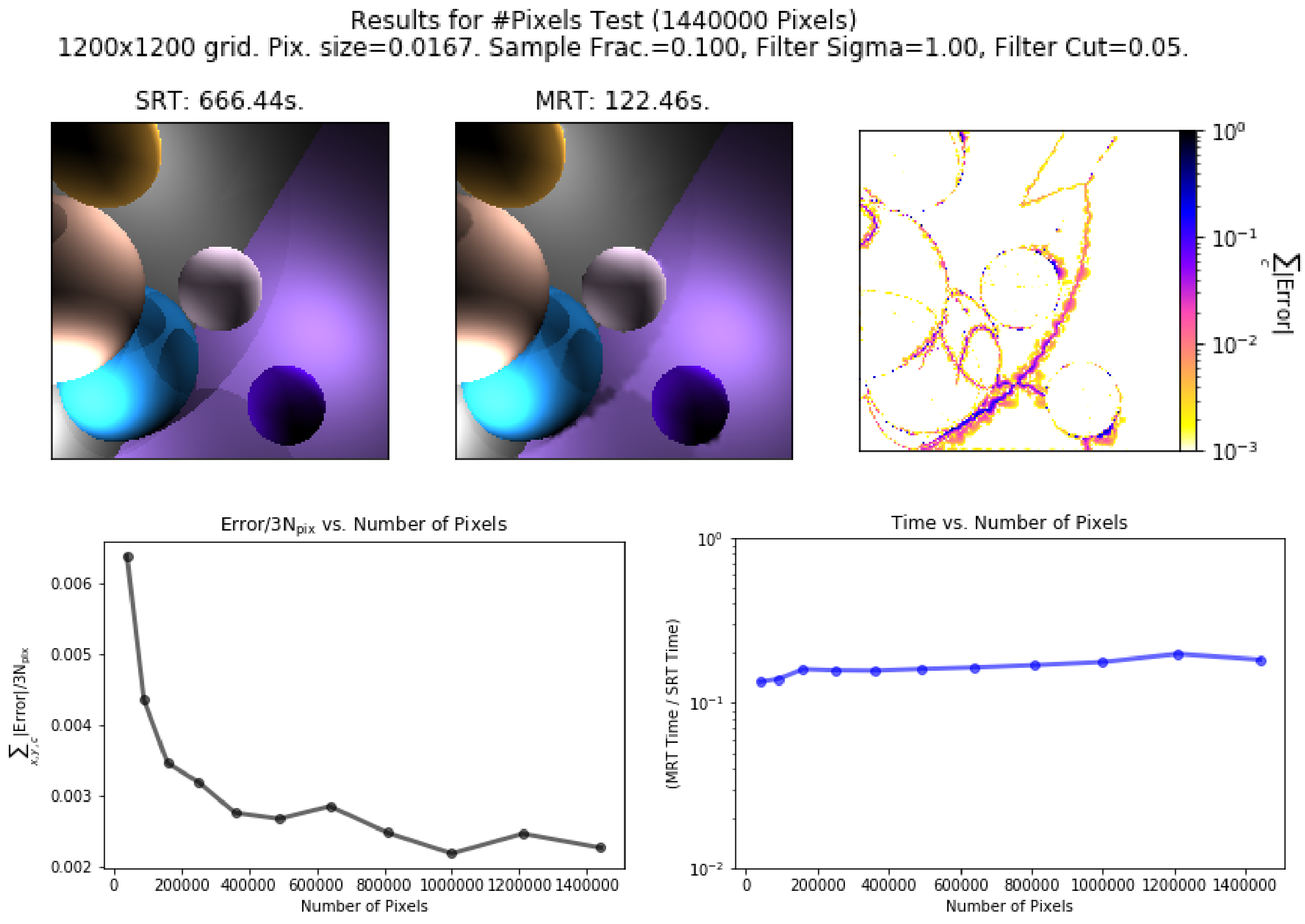


Fig. 8. Top: A snapshot of MRT accuracy for a 1,200×1,200 grid, the highest resolution considered. Other snapshots are shown in Figure C. The snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 5 spheres, 2 planes, and 4 point light sources, was constructed specifically to have complex irradiance patterns and overlapping regions of shadows. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles. **Bottom:** MRT accuracy (left) and relative runtime (right) as a function of the number of pixels.

D. Performance vs Number of Pixels. Here we test MRT performance with respect to the number of pixels in the final image. Figure 8 shows the MRT error and runtime as a function of the number of pixels, while both Figure 8 and Figure C show snapshots of MRT performance.

This test highlights two crucial traits of the MRT. Firstly, the MRT relative runtime is fairly constant as the number of pixels increases. Therefore we are not subject to any new computational penalties for changing the resolution of the image. Secondly, the MRT error decreases as the number of pixels increases. Therefore we actually benefit from using high-resolution images instead of low-resolution images, and high-resolution images are more likely to be used in modern contexts.

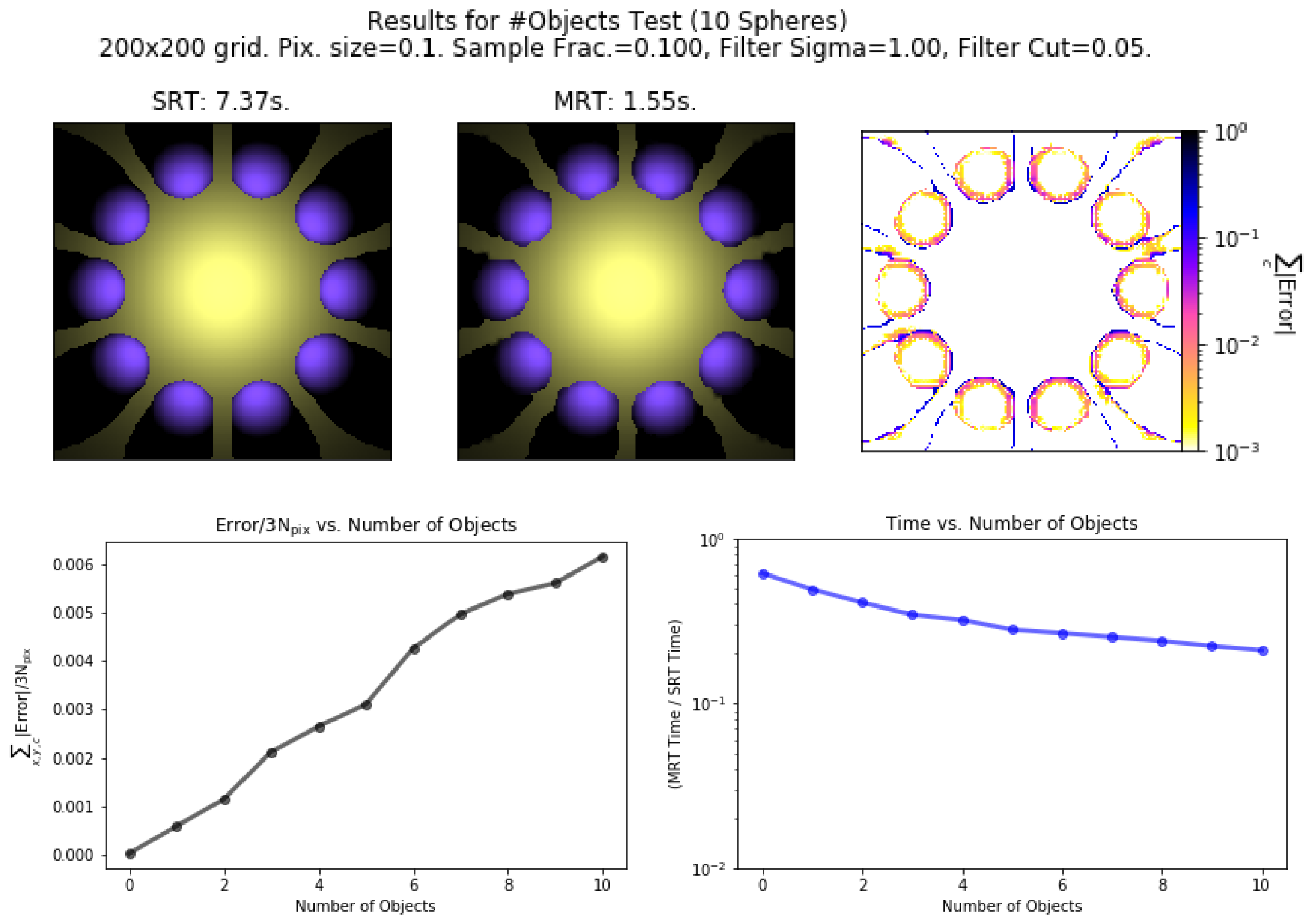


Fig. 9. Top: A snapshot of MRT accuracy for a scene containing 10 spheres. Other snapshots are shown in Figure D. The snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles. **Bottom:** MRT accuracy (left) and relative runtime (right) as a function of the number of spheres.

E. Performance vs Image Complexity. We now test MRT performance with respect to the complexity of the image. Figures 9 and 10 show the MRT error and runtime as a function of the number of objects and the number of lights, respectively. Both Figure 9 and Figure D show snapshots of MRT performance relative to the number of objects, while both Figure 10 and Figure E show snapshots of MRT performance relative to the number of lights.

We find that error increases as we introduce more objects and/or regions of shadow into the image. This is because the MRT's current ultimate weakness is its inability to exactly mimic and interpolate the edges of objects and shadows in a scene. The MRT runtime, on the other hand, decreases as the image complexity increases. This result highlights an important characteristic of the MRT. Notably the computational intensity of the SRT increases as the number of lights and objects increases, because each ray from each pixel must iterate through all objects and lights to check for, and then calculate, intersections. However, because the MRT relies on surface-level interpolation for the majority of the pixels, the MRT does not have to spend additional runtime explicitly tracing those more complex rays. We thus infer that the MRT's relative runtimes derived here for a simple underlying ray-tracer are an upper limit for more advanced ray-tracers. We predict that the MRT's relative runtime will likely decrease when used for more complex, advanced ray-tracers.

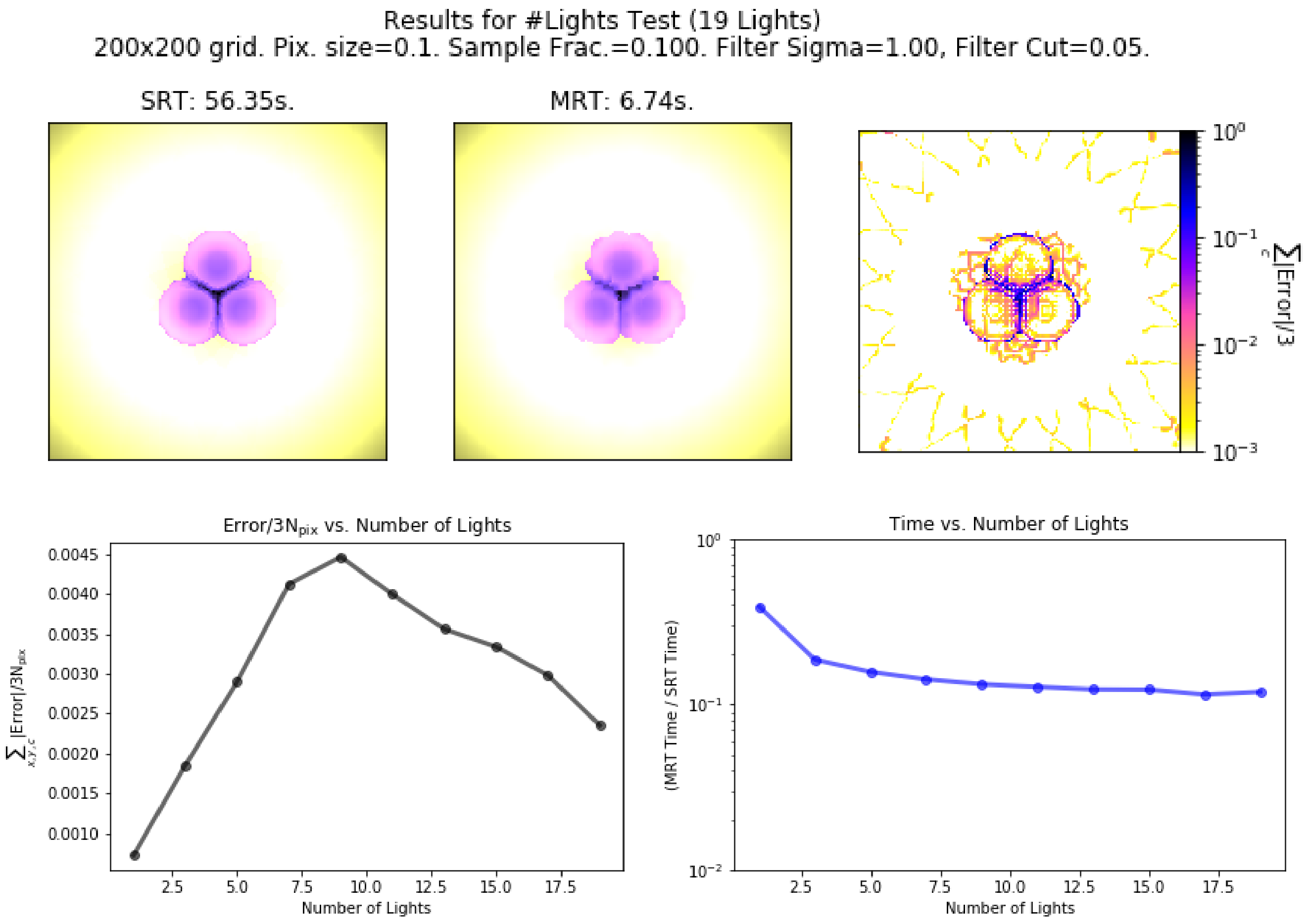


Fig. 10. Top: A snapshot of MRT accuracy for a scene containing 19 point light sources. Other snapshots are shown in Figure E. The snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles. **Bottom:** MRT accuracy (left) and relative runtime (right) as a function of the number of lights.

4. Recommended Future Work

From our analysis of the modified ray-tracer (MRT) performance, we have determined four key benefits and the ultimate shortcoming of the MRT relative to the simple standard ray-tracer (SRT):

1. **Benefit:** When sampling and ray-tracing 10% of the full image, the MRT runtime for approximating the full image is ∼10-20% of the SRT runtime.
2. **Benefit:** The MRT accurately approximates the inner surfaces of objects and shadows when producing an image.
3. **Shortcoming:** The MRT poorly approximates the edges of objects and shadows when producing an image.
4. **Benefit:** The MRT provides better performance for higher-resolution images.
5. **Benefit:** The runtime of the MRT relative to the SRT will likely *decrease* as the image complexity increases.

In light of these key points, we present possible ways of improving MRT performance, as well as recommendations for adapting the MRT algorithm for more complex, advanced ray-tracers.

A. Improving Edge-Finding and Edge-Interpolation. The major shortcoming of the MRT is its inaccuracy in (1) finding the edges of objects and the edges of shadows in an image, and (2) interpolating the irradiance near those edges. The first problem is rooted in our edge finder. Currently we use a Gaussian filter to "smooth" the jagged edges produced by our recursive space partitioning algorithm. However, this filter could certainly and easily be replaced with a filter better suited for this purpose, such as a filter that is optimized for smoothing edges in a binary matrix.

The second problem is rooted in the sampling we perform for our interpolator. Given a large object in an image spread over many pixels, our interpolator only uses a subset of capstone pixels and secondary pixels as a base for interpolating the remaining pixels associated with that object. Therefore, the secondary pixels may exclude key pixels in the image that make up the overall irradiance pattern. This is especially true for large objects that have sharp, steep gradients in irradiance, such as those seen in Figure A. In these example cases it is statistically less likely for the interpolator to adequately sample the very small regions of irradiance on the planes. This problem is tricky to mitigate without increasing the relative runtime of the MRT. That being said, one likely solution is to improve the capstone pixels used by the interpolator. For this report, the capstone pixels for a given id consist of the 10 brightest known pixels, the 10 darkest known pixels, and all known pixels that form the convex hull of that collection. In the future, this selection could be improved by including, for example, the 10 median known pixels, which would ensure that a more diverse spread of the irradiance pattern is sampled by the interpolator.

B. Implementing the MRT for more Advanced Ray-Tracers. We highly recommend that the MRT algorithm be applied to more advanced standard ray-tracers (ARTs). ARTs have many enhancements compared to our simple standard ray-tracer (SRT), leading to more complex irradiance patterns. We predict that the MRT algorithm would not only be able to approximate these more complex patterns accurately, but in certain cases would also provide even better, faster relative runtimes compared to what we have achieved in this report.

Firstly, our SRT assumes all objects have a dull non-reflective material. ARTs, on the other hand, often support more object materials, including reflective, refractive, and luminous ones. Ray-tracing such materials requires far more computation to compute the irradiance. Ray-tracing reflective materials (such as mirrors) is particularly expensive, because it requires additional recursive ray-tracing to do accurately. We predict that our MRT algorithm would accurately approximate reflective surfaces and other materials. In particular, the underlying edge finder should be able to split a reflective surface into different components corresponding to the reflected objects/shadows, and then perform surface-level interpolation using the known pixels of reflected irradiance within each component.

Secondly, our SRT assumes all objects are either monochromatic planes or monochromatic spheres. ARTs, however, can support a variety of multicolored three-dimensional objects, such as cubes, prisms, and blobs. The underlying edge finder and surface-level interpolator built into the MRT algorithm are constructed such that the MRT does not care about the types of objects present in the scene. The MRT will approximate their irradiance regardless. Approximating multicolored surfaces, however, would be trickier. Since the current MRT ray-traces only a subset of pixels in the image, and then interpolates the irradiance rather than the color, the MRT loses spatial, point-specific information about the color on each object. One possible solution would be to modify the MRT algorithm so that it interpolates each of the red-green-blue (RGB) components separately, rather than the irradiance. This would increase the number of regions that the MRT must interpolate over by a factor of three, and thus would increase the MRT's final relative runtime. That being said, this additional runtime cost might not be too severe, as long as the underlying interpolator is efficient and fast.

5. Conclusion

We have demonstrated that the modified ray-tracer (MRT) accurately approximates ray-traced surfaces, and that the full MRT runtime is a fraction of what is needed for the simple standard ray-tracer (SRT). We have noted that the MRT's accuracy is poor along the edges of objects and shadows in the final image, but we have suggested ways to improve that aspect of performance in the future. We highly encourage others to try applying the MRT algorithm to more advanced standard ray-tracers, and to let us know of the subsequent MRT performance.

ACKNOWLEDGMENTS. I first thank Suffern (2007) for being a truly awesome reference for ray-tracing. The textbook provides in-depth description, calculations, and discussion of how ray-tracing works, all while being completely and utterly friendly to ray-tracing beginners. Thank you for teaching me how ray-tracers work!

I also definitely want to thank my primary AC299R adviser for many helpful meetings and checkpoints during this semester project. I also gratefully thank my secondary AC299R adviser for helpful advice and support for this process. I additionally want to thank my previous class, AM205; I planted the seed for this project during my final project for AM205, and I am so grateful that I've been able to see that seed grow.

Finally, I want to thank Python. Python's `numpy`, `matplotlib`, `timer`, and `scipy` packages were used heavily in this project, from plots, to arrays, to matrices, to timers, to a fast implementation of RBF interpolation (which was way faster than what I'd coded by hand).

Thank you!

References

Suffern, K. (2007). *Ray Tracing from the Ground Up.* CRC Press LLC, Natick.

Appendix

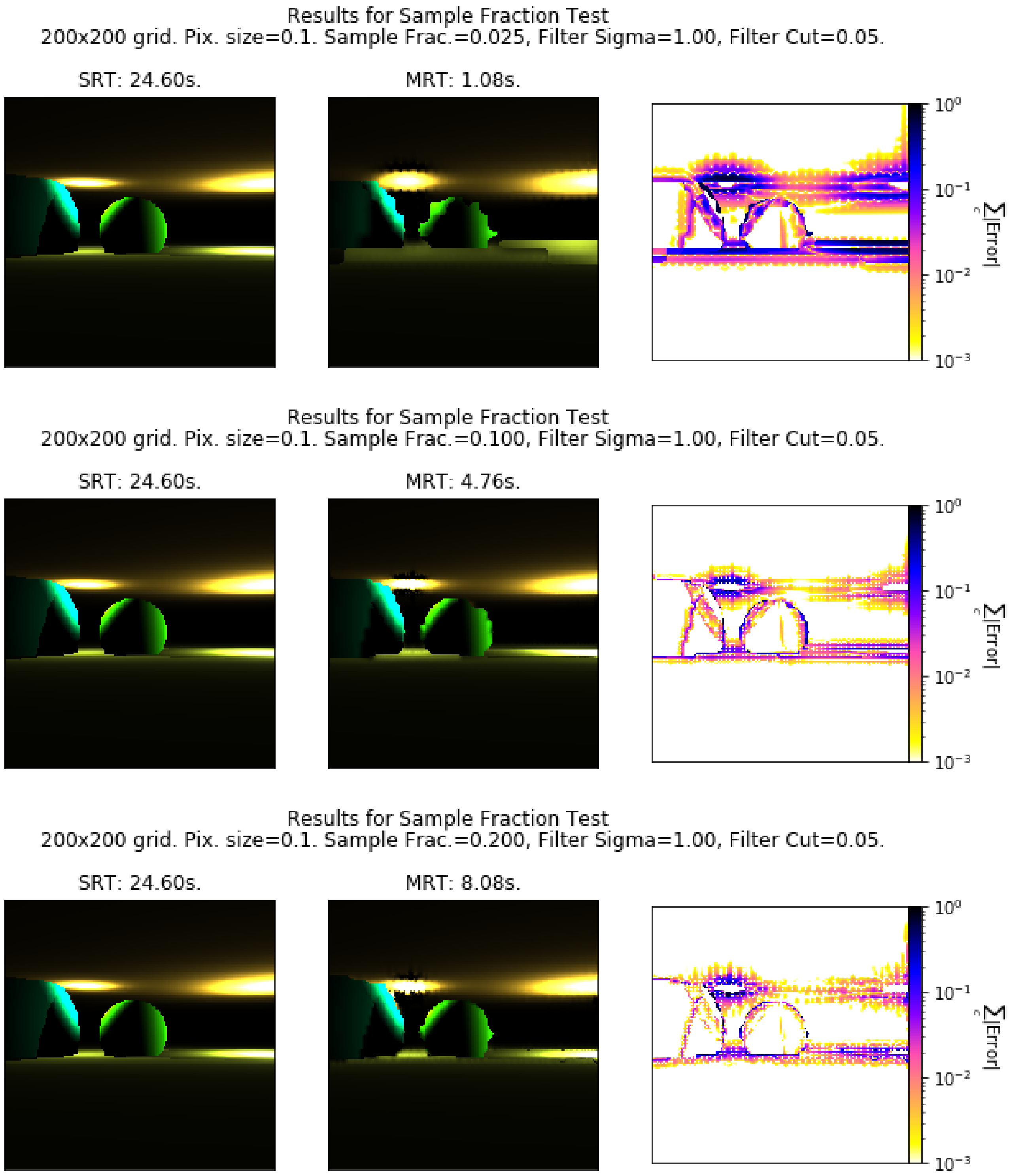
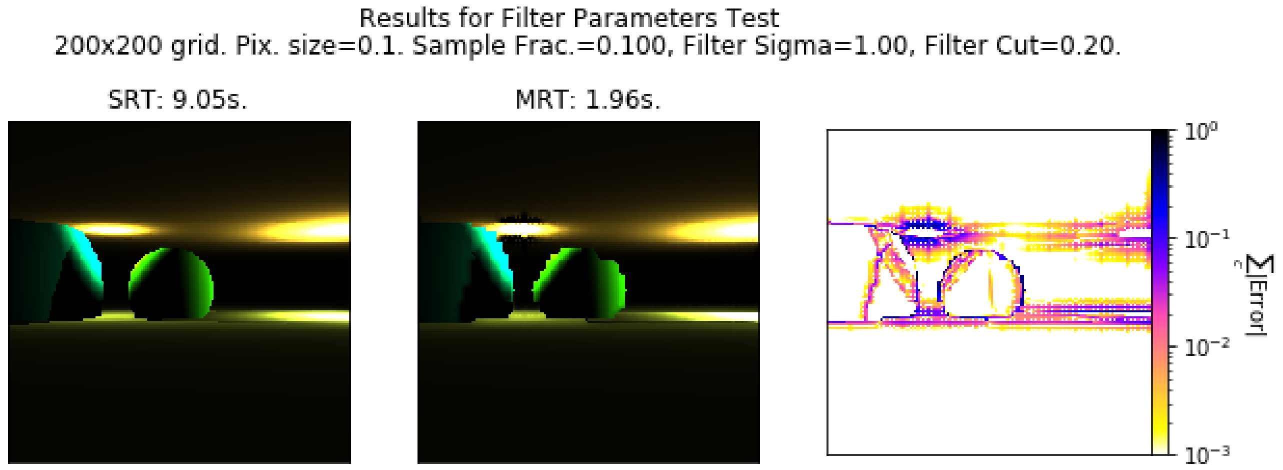
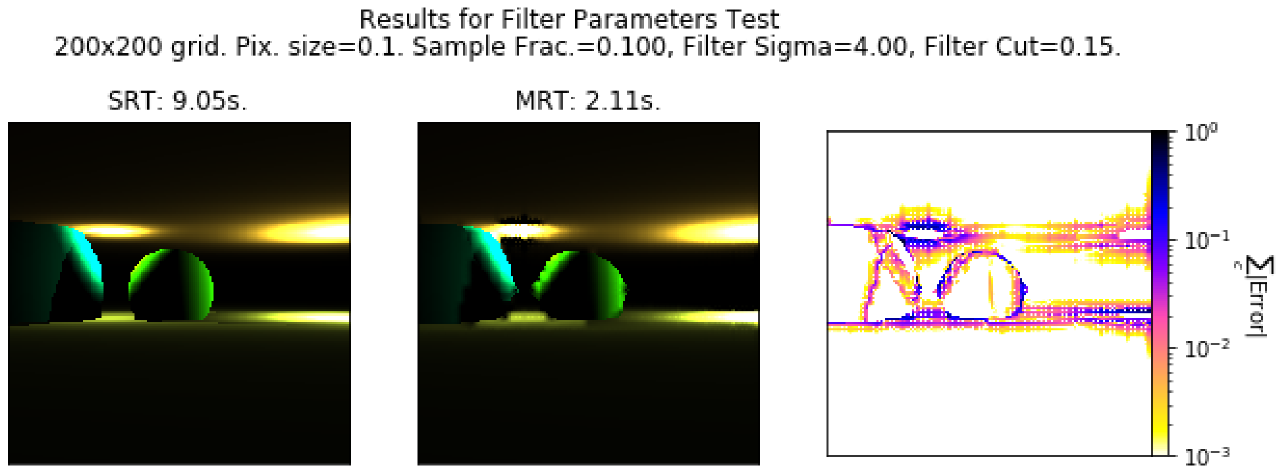

Results for Sample Fraction Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.025, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 24.60s.
MRT: 1.08s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c|\mathrm{Error}|$
Results for Sample Fraction Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.100, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 24.60s.
MRT: 4.76s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c|\mathrm{Error}|$
Results for Sample Fraction Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.200, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 24.60s.
MRT: 8.08s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c|\mathrm{Error}|$

Fig. A. Snapshots of MRT accuracy for different sample sizes. Each snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 2 spheres, 2 planes, and 3 point light sources, was constructed specifically to have long, narrow regions of irradiance. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles.


Results for Filter Parameters Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.100, Filter Sigma=1.00, Filter Cut=0.20.
SRT: 9.05s.
MRT: 1.96s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$


Results for Filter Parameters Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.100, Filter Sigma=4.00, Filter Cut=0.15.
SRT: 9.05s.
MRT: 2.11s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$

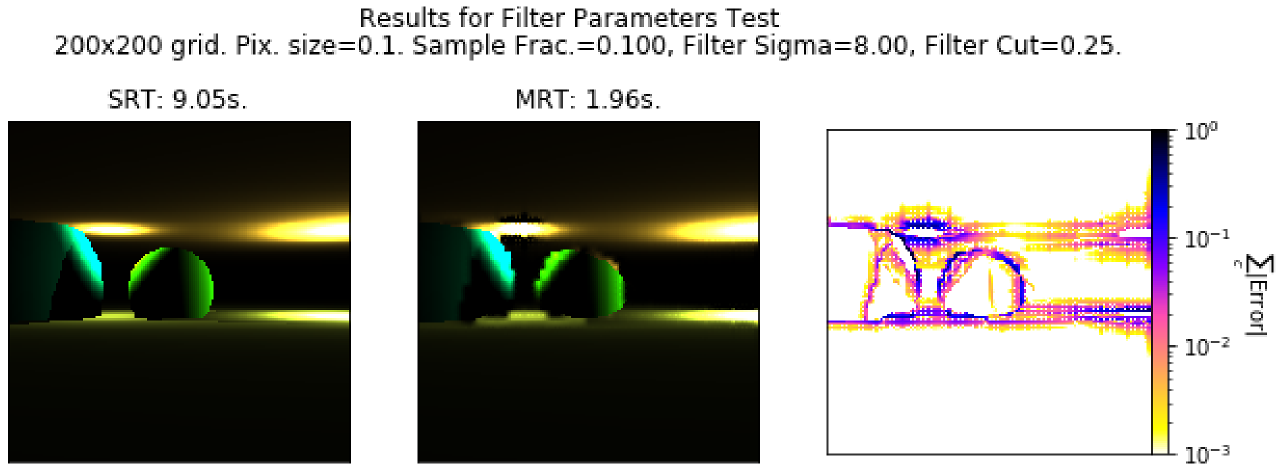
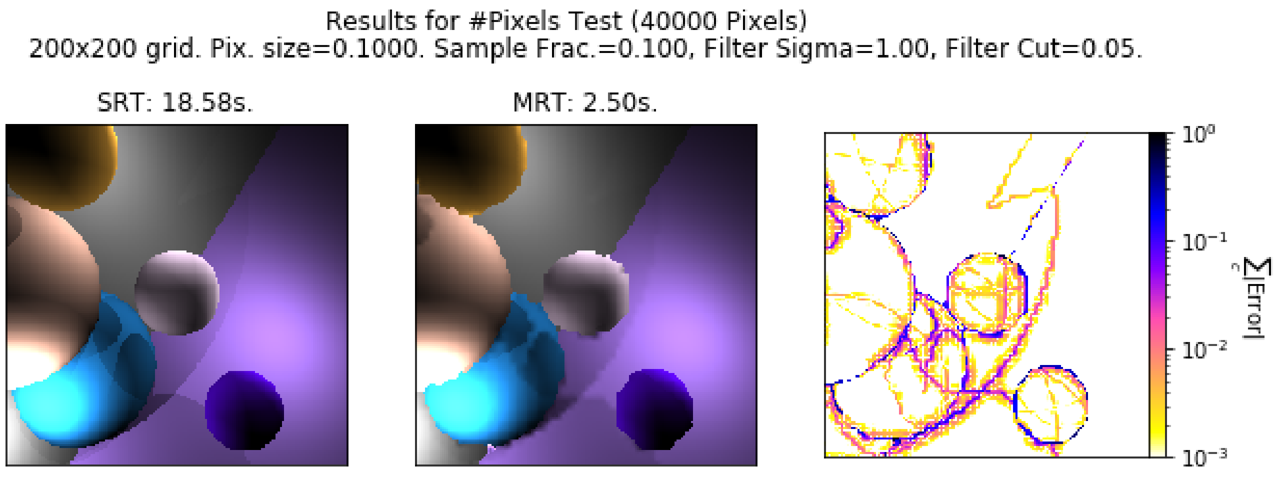
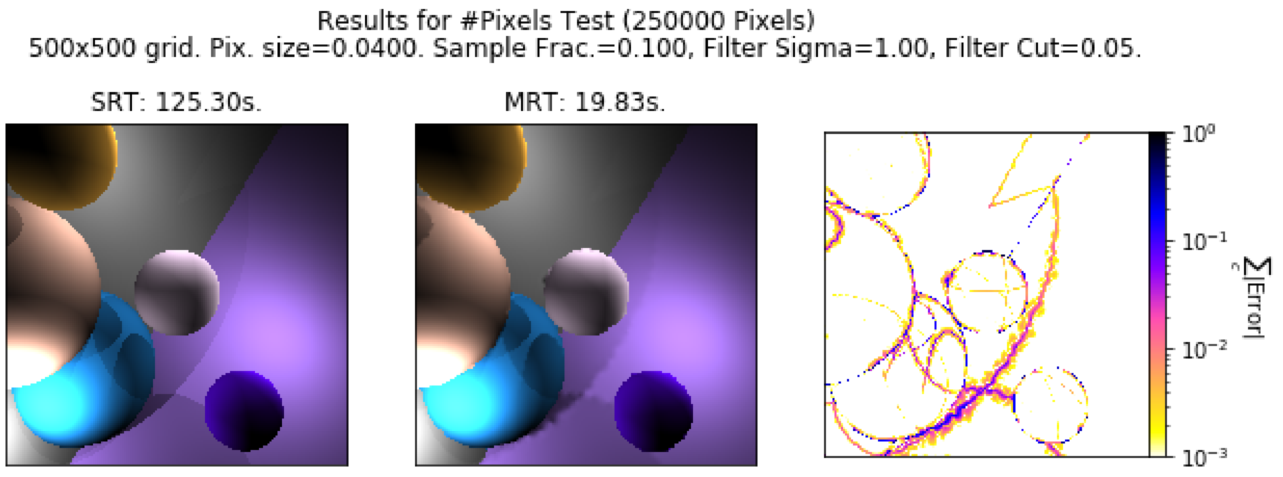

Results for Filter Parameters Test
200x200 grid. Pix. size=0.1. Sample Frac.=0.100, Filter Sigma=8.00, Filter Cut=0.25.
SRT: 9.05s.
MRT: 1.96s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$

Fig. B. Snapshots of MRT accuracy for different Gaussian filter parameters. Each snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 2 spheres, 2 planes, and 3 point light sources, was constructed specifically to have long, narrow regions of irradiance. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles.


Results for #Pixels Test (40000 Pixels)
200x200 grid. Pix. size=0.1000. Sample Frac.=0.100, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 18.58s.
MRT: 2.50s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$


Results for #Pixels Test (250000 Pixels)
500x500 grid. Pix. size=0.0400. Sample Frac.=0.100, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 125.30s.
MRT: 19.83s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$

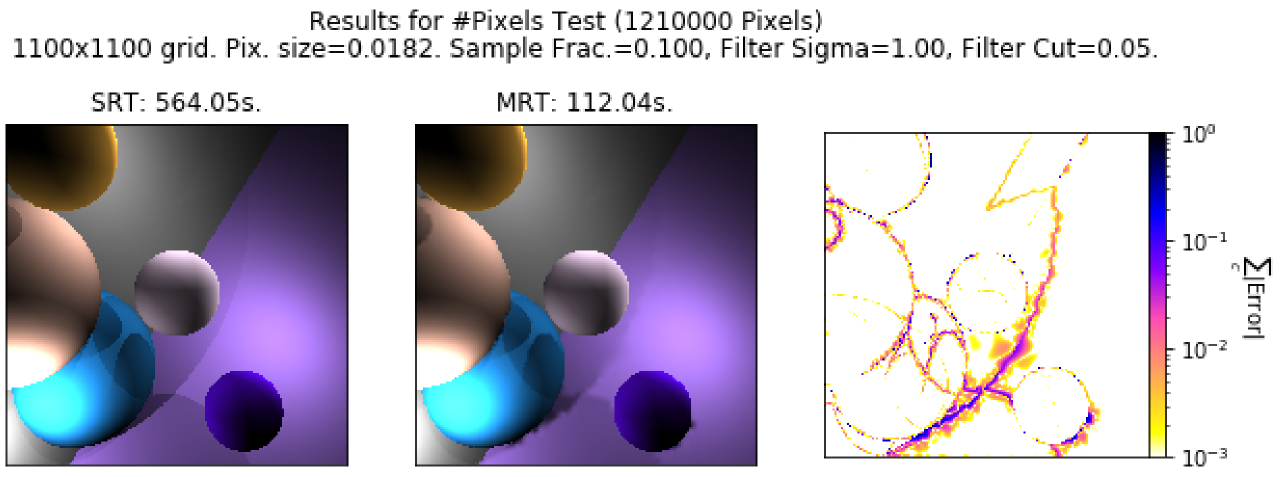

Results for #Pixels Test (1210000 Pixels)
1100x1100 grid. Pix. size=0.0182. Sample Frac.=0.100, Filter Sigma=1.00, Filter Cut=0.05.
SRT: 564.05s.
MRT: 112.04s.
10^0
10^{-1}
10^{-2}
10^{-3}
$\sum_c |\text{Error}|$

Fig. C. Snapshots of MRT accuracy for different numbers of pixels. Each snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. The scene, which contains 5 spheres, 2 planes, and 4 point light sources, was constructed specifically to have complex irradiance patterns and overlapping regions of shadows. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles.

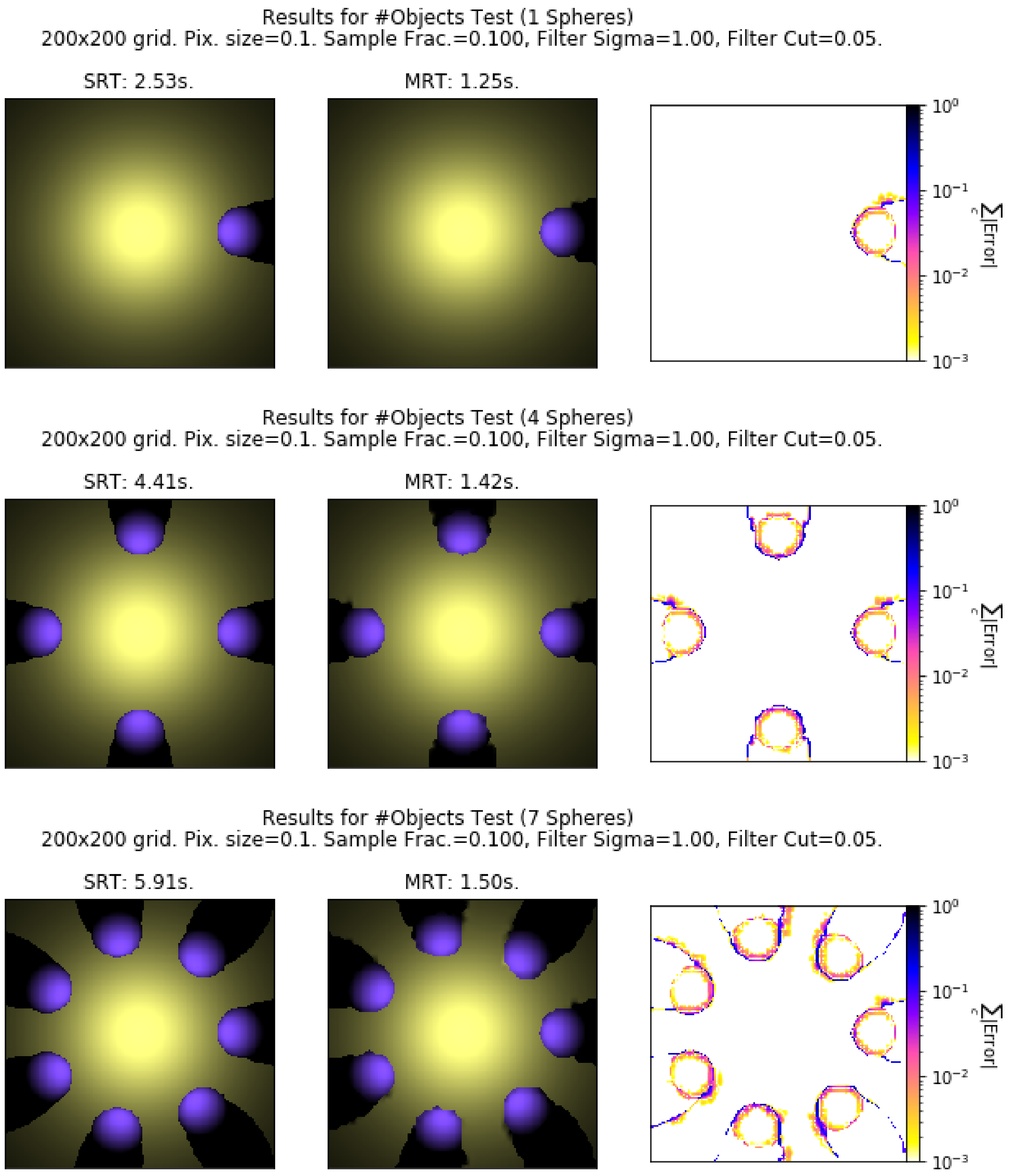


Fig. D. Snapshots of MRT accuracy for different numbers of spheres. Each snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles.

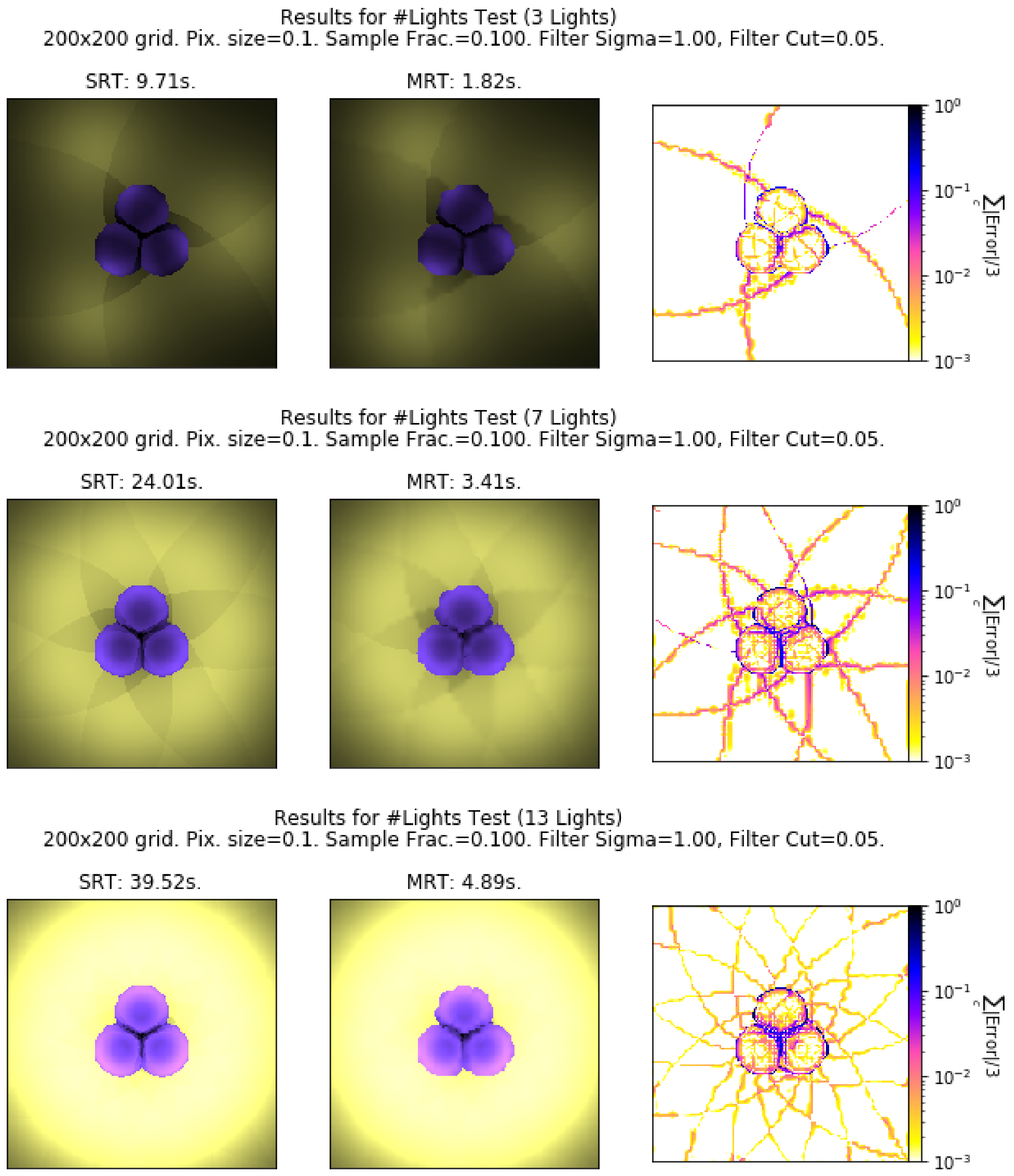


Fig. E. Snapshots of MRT accuracy for different numbers of lights. Each snapshot shows the SRT result (left), the MRT result (middle), and the error (right). The error is calculated as the absolute difference in each pixel's final 3 red-green-blue (RGB) color values relative to the SRT result, divided by 3. Technical details and parameters of the scene (e.g., pixel size and sample fraction) are given in the titles.