Kind of Quick Ray Tracing

Greg Yauney*
Advanced Computer Graphics
Spring 2012

Abstract

Adaptive ray tracing is a way to accelerate the traditional ray tracing we implemented in homework three. The idea, at its most fundamental, is to apply a test to each pixel and point in the scene which determines if the final color of the pixel/point would be altered if a distributed ray tracing technique is applied to it. In this project, I implemented such tests for antialiasing, soft shadows, glossy reflections, and motion blur, along with alternative methods of testing and subsequently shooting the distributed rays once/if said tests are passed. Some of these methods is adaptive on the other end—it dynamically adjusts how many rays are shot into the scene after a pixel or point passes its corresponding test. This paper is basically a description, analysis, and comparison—of both efficiency and quality—of the original methods for these techniques and my newly implemented adaptive methods.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing;

Keywords: adaptive, ray tracing, kind of cool

1 Introduction

I'm going to pretend that my primary motivation for undertaking this project is not that I hemmed and hawed for so long that I needed to finally settle on a final project that was doable in two weeks, but rather that I am all sorts of intrigued by just how slow homework three is when there're *a lot* of distributed rays. The main idea behind such distributed ray tracing is to send out multiple rays per pixel/point so as to approximate Real Life effects like soft shadows [Cutler 2012]. The reason, it turns out, for the aforementioned colossal waste of time is that in homework three, distributed rays are computed/sent/shot for *every single pixel*, even when they won't contribute any extra information to the final image. With this project, I devised and implemented tests that resulted in shooting only as many rays as is needed to produce a high-quality image, which of course results in a far faster rendering process, although there is a trade-off with the quality of the finished image.

2 Related Work

The primary works that form the basis of this project are the lecture notes on ray tracing and homework three. As for work related to the adaptive stuff I'm actually doing: aside from first reading about adaptive antialiasing in Wright and Lynn's final project [Wright and Lynn 2011], I mostly wanted to see if I could devise any of these methods on my own (which was probably not the best idea in retrospect, although I did learn more than if I had merely rotely implemented e.g. [Hachisuka et al. 2008]'s methods). If you want a refresher on ray tracing, or if you slept through that lecture, definitely check out [Cutler 2012], or if you'd rather get it directly from the actually pretty old by now source, read [Whitted 1980].

*yauneg@rpi.edu

3 Antialiasing

The original antialiasing method described by both Cook et al. way back in the original distributed ray tracing paper [Cook et al. 1984] and then this semester by Professor Barb Cutler works pretty simply: for each pixel in the image plane, shoot out into the scene a fixed number of rays through a random point in the pixel, then average the colors returned by each ray.

3.1 Stratified Quadrant Method (SQM)

This method was originally described by Genetti and Gordon [Genetti and Gordon 1993] and implemented last year by Wright and Lynn in their own final project for this class. For each pixel, one ray is shot into a random location in each quadrant (for a total of four rays). The returned colors are subsequently averaged, and if any quadrant's color is different from the average by more than some variance threshold ϵ, then that quadrant is split into subquadrants and recursed upon. This process continues until either the maximum number of recursions for each quadrant/subquadrant is reached or the variances in all quadrants are below ϵ. It's worth noting that this quadrantization ensures that the random points on the pixel are picked using stratified sampling (hence the name SQM) while all the other methods pick points uniformly at random (for the difference between stratified and random sampling, see lecture 11 [Cutler 2012]).

3.2 Full Uniform Method (FUM)

This is a more naive method: it shoots four rays through random points of the pixel. If the average of the colors returned by these four rays differs from the color of a single ray through the center, then the user-specified number of rays are sent out through uniformly random points in the pixel, just like in the original non-adaptive method.

3.3 Thresholded Uniform Method (TUM)

This is initially similar to the previous method: four rays are shot through random points of the pixels, and if they differ from the color of the ray of the center, then four more rays are sent through random points (for a total of eight rays). If the new average differs from the four-ray average by more than ϵ, then eight more rays are shot out (for a total of sixteen). This process of doubling the number of rays repeats until either a maximum number of rays is reached or the average color varies from the average on the previous iteration by less than a threshold ϵ. In this way, the number of rays shot per pixel which passes the test is dynamic (unlike in the previous method).

3.4 Comparison

Check the heck out of figures 3 and 4 for a thrilling comparison of all these.

The reason why TUM is way faster than FUM is that it doesn't shoot nearly as many rays for the pixels that passed the test, but the speed difference between FUM and SQM is a bit harder to explain—it turns out that in my current implementation of SQM,

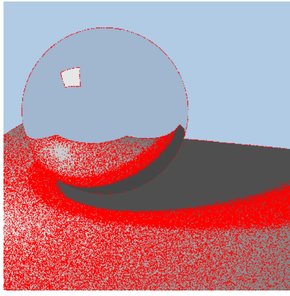


Figure 1: *RIM with* $\epsilon = 0.01$

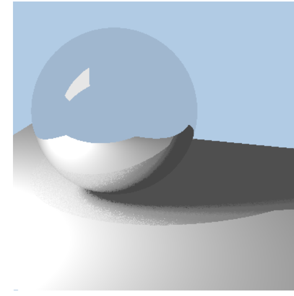


Figure 2: *A bigger light source causes more banding when using the soft shadows SQM with* $\epsilon = 0.01$

the values from each previous iteration are disregarded, meaning that although the maximum number of recursions in FUM is three, the maximum number of rays per pixel isn't 256, it's $4 + 16 + 64 + 256 = 340$. This problem would of course be amplified by more recursions.

FUM and TUM produce slightly noisier/grainier images than both the original method and SQM due to their use of uniform random samples since there's far more variance in where each of the four initial test rays lands.

3.5 The Importance of ϵ

ϵ directly governs how many pixels pass the test which determines if more rays are sent at them. Figure 1 shows an example of FUM with $\epsilon = 0.01$—way, way more pixels need to have the maximum number of rays sent through them than in Figure 4d where $\epsilon = 0.05$. Unfortunately, I haven't yet figured out a system for adaptively choosing epsilon, if that's even possible; it was all trial and error. The guiding principle, though, is that the greater ϵ is, the less noise will be in the final image, but the longer it'll take, which can be chalked up to more pixels passing that initial variance test. This still holds true when these methods are applied to all the other distributed effects.

4 Soft Shadows

The original soft shadows algorithm we implemented in homework three works thusly: when a ray emanating from the eye collides with an object in the scene, a specified number of rays (regardless of whether these rays provide any extra contribution to the final pixel in the image) are created from the original intersection point and are directed to random points on the area light source. If a ray isn't occluded by any objects in its path, then the total color is increased by the color of the light emanating from the source. The final color of the pixel is the result of dividing this total by the number of rays. As for the adaptive methods, I implemented variations of all three of the above antialiasing strategies.

4.1 SQM

This is identical to the antialising SQM except that instead of stratificatively sampling the quadrants of a pixel, it samples the quadrants of a light source. The quality of the resulting image, however, is far noisier than expected, especially when considering that the results of the antialiasing SQM aren't significantly lamer than their non-adaptive result. I've come to the conclusion that the sheer size of the light source compared to the size of each pixel is what's to blame. Since antialiasing involves sampling pixels, and pixels are much smaller than the light source, there's a lot less variability in the paths of the antialiasing rays. As you can see in figure 2, increasing the size of the light source causes significantly more noise and banding.

4.2 Full Penumbra Method (FPM)

This solves the noisiness/banding problem by sending the initial four rays (which, remember, test whether or not the point needs more shadow rays) to the four corners of the light source (so far I've only used quads as lights, but this is easily extensible to light sources of other shapes). If none of the rays intersect any objects, then the point's color receives the full contribution of the light since there's no way it can be in shadow, and conversely, if all corners are obscured, the point is fully in the shadow's umbra, meaning that the light in question doesn't contribute even a bit toward its color. If only some of the corners are obscured while others aren't, that means the point in question is in the penumbra (i.e. the part of the shadow that's soft). This method then naively sends the full specified number of rays. It produces an image of the exact same quality as the full non-adaptive method since it's guaranteed to only send the extra shadow rays for points in a shadow's penumbra. However, the speed-up is only in terms of how many points need these extra rays, not in terms of how many shadow rays are shot for points which pass the test.

4.3 Thresholded Penumbra Method (TPM)

This method solves both the noisiness problem of SQM *and* the still-way-too-many-rays problem of FPM by using the same initial penumbra test but then only iteratively shooting as many rays as are necessary to get the final color below some variance away from the color on the previous iteration, just like the antialiasing TUM.

4.4 Comparison

Looks like I've sufficiently compared these methods in the above sections. I will re-emphasize, however, that FPM easily produces the best image compared to the other adaptive methods. Also, I can't really think of any situation which would be better off with SQM rather than FPM or TPM. Figures 5 and 6 should help you see the differences between these methods in a hopefully not too confusing manner. It'll also give you a quantitative comparison of the runtimes, which makes intuitive sense: the more points need shadow rays, the longer it's going to take.

5 Glossy Reflections

Glossy reflections are an extension of the standard perfect reflection from homework three. Since we weren't required to implement glossiness), here's a primer: instead of always perfectly calculat-

ing the reflection direction, the direction is perturbed based on the roughness of the reflective material. Multiple such samples are sent from each reflective point and then averaged together.

5.1 Adaptive Glossiness

This sends four test rays which are perturbed from the perfect direction in the same manner as the normal method; if any of them differ from their average by more than ϵ, then all the rest of the glossy rays are sent and averaged. It's basically a modification of the antialiasing TUM, sending more rays until the current average color is less than the threshold away from the last iteration's average color.

The first obvious improvement is to maybe stratificatively sample a segment of the hemisphere (with a size determined by the roughness) around the point that points instead of merely perturbing the reflected direction. That would probably cut down on the graininess that you can see in figure 9c, which is a result of some pixels passing the test that shouldn't've thanks to randomness, and vice versa. Oh also, figures 7 and 9 are a complete comparison of these two methods.

6 Motion Blur

Since we also weren't required to implement motion blur in homework three, my first task was to get basic motion blur working. The basic idea is to supersample each pixel temporally. What that means is that multiple rays are sent per pixel, each at different times, and then averaged. An exposure time and the desired number of temporal samples are specified by the user. The timestep is computed by dividing the exposure by the number. So that means all the temporal samples are evenly spread out along the exposure time.

6.1 Adaptive Motion Blur

Instead of sending all the temporal rays for each pixel like in the standard method, five initial test rays are sent out, one each at the beginning, a quarter, half, three quarters, and all of the way through the exposure (so the timestep is 0.25 times the exposure). If any of the samples are different from each other, that means the pixel experiences an object moving through it during the exposure, and the full number of temporal rays are sent at it.

A serious limitation of my implementation is that if an object is sufficiently small and moving sufficiently fast, the initial test will actually miss the object entirely—it would probably be best to adaptively choose how many initial test samples (at a correspondingly adaptively-chosen timestep) are compared based on the length of the exposure time and the size of each object. In my example image, though, I made sure the objects are large enough so this isn't an issue.

My implementation also only deals with constant velocities. It would be cool (and I would totally do so if I had more time / if I had better planned my time) to extend this to handle accelerations and then dynamically adjust the placement of samples along the exposure so that there are more where the object is moving faster (which I'm pretty sure would be a type of importance sampling).

As usual, q.v. figures 8 and 10.

7 Putting Some Of All Of It Together For At Least A Couple Kind Of Impressive Images

Check 'em out: figures 11 and 12.

8 Conclusions and Further Work

So ultimately, what I've learned from this project is that adaptivity *greatly* speeds up distributed ray tracing while, if you're not careful, sacrificing at least a little bit of quality.

There's always implementing other ray tracing effects, e.g. depth of field and refraction, plus the adaptive tests can of course use refinements and improvements—soft shadows especially—and I'm of the humble opinion that this project is definitely a foundation for me to explore these. Ultimately, though, it turns out that a way, way more advanced version of adaptive ray tracing was already explored by Hachisuka et al. with way, way more rigor and technicality and to far more impressive effect in [Hachisuka et al. 2008].

As for what I'd like to do from here, this project has definitely made me more interested in ray tracing in general, especially A) how to make the ray tracer work quickly with larger scenes. According to lecture 11 [Cutler 2012], the real bottleneck in bigger scenes is the massive number of objects (in our system, triangles and quads and spheres) that need to be tested against every ray in order to determine the closest intersection, but I've never run into this because the scenes I've dealt with only have a couple objects in them. For larger numbers of objects, spatial data structures like the k-d tree can greatly reduce this number of tests (about which we learned in lecture 5 [Cutler 2012]), as described in [Fussell and Subramanian 1988]. B) I'd also really like to tackle a real-time ray tracer like the ones we discussed in lecture. I know these wouldn't be impressive achievements since they've already been done before, but I'm definitely interested in them now that I've taken this class.

9 Acknowledgments

It's always fun to thank people—especially in such an official way—so here I'll sappily thank Professor Barb Cutler for teaching this class (definitely not in the hopes of getting a better grade on this project, I assure you); it's been phenomenal not only because the subject matter is all sorts of interesting, but because Barb teaches it in such an engaging way, something I've nowhere near experienced with other technical classes. I had fun when I wasn't falling asleep during the really long powerpoints.

References

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *ACM Transactions on Graphics 18*, 3 (July), 137–145.

CUTLER, B., 2012. Ray tracing & distributed ray tracing. Advanced Computer Graphics Lecture Notes, Jan./Feb./Mar./Apr./May.

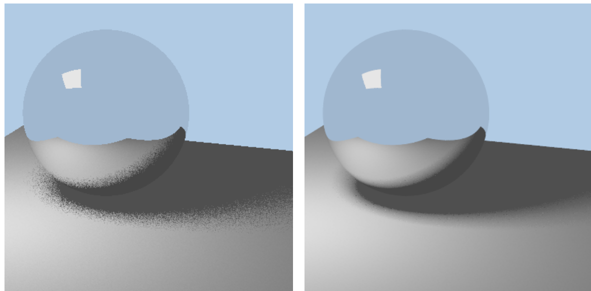
FUSSELL, D., AND SUBRAMANIAN, K. R. 1988. Fast ray tracing using k-d trees. *Department of Computer Sciences, The University of Texas at Austin* (Mar.).

GENETTI, J., AND GORDON, D. 1993. Ray tracing with adaptive supersampling in object space. *Graphics Interface*, 70–77.

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R. P., DALE, K., HUMPHREYS, G., ZWICKER, M., AND JENSEN, H. W. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics 27*, 3 (Aug.).

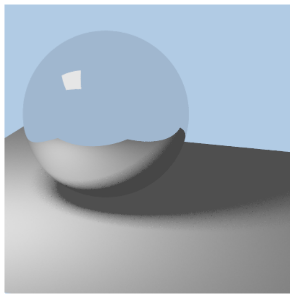
WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

WRIGHT, G., AND LYNN, Z. 2011. Feature complete ray tracer. *Advanced Computer Graphics Final Project* (May).

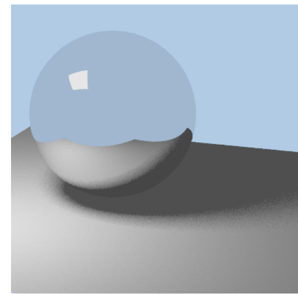


(a) *No antialiasing*
2.02 seconds

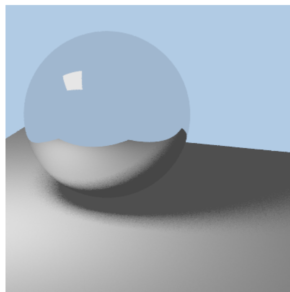
(b) *256 rays per pixel*
512.85 seconds



(c) *SQM*
60.89 seconds

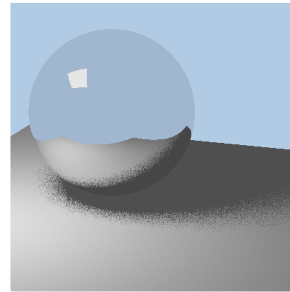


(d) *FUM*
54.6 seconds



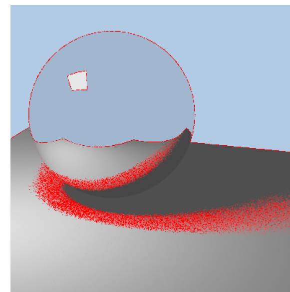
(e) *TUM*
20.89 seconds

Figure 3: *Comparison of antialiasing methods. There are also four soft shadow samples per pixel (to give it something to antialias).* $\epsilon = 0.05$. *All of them have the maximum number of rays per pixel set at 256 so as to stay consistent with the non-adaptive method.*

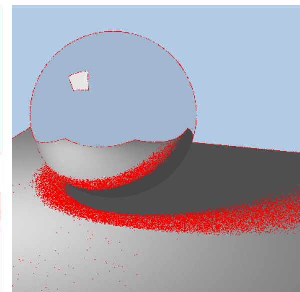


(a) *No antialiasing*

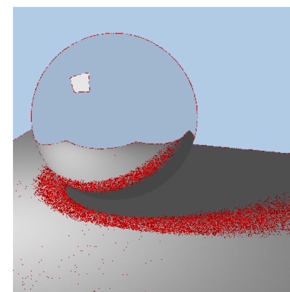
(b) *256 rays per pixel*



(c) *SQM*

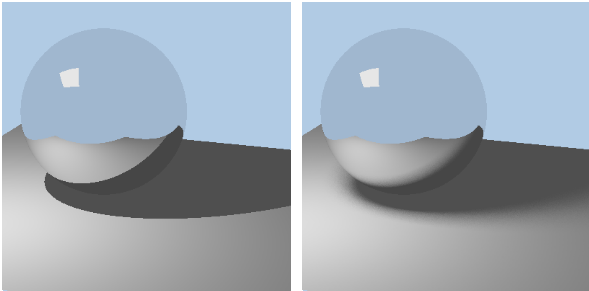


(d) *FUM*



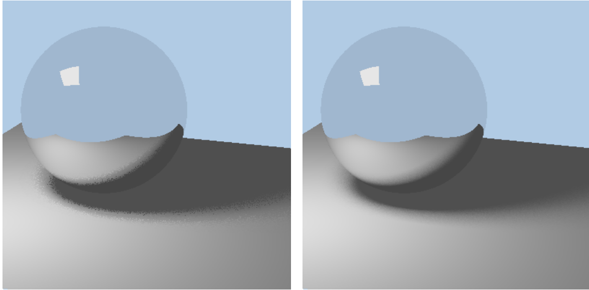
(e) *TUM*

Figure 4: *The same as the previous figure, except the pixels affected by each antialiasing method are in red—the brighter the red, the more rays were used for that pixel.*



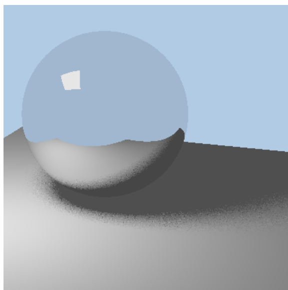
(a) *Perfect shadows—no soft shadows*
1.01 seconds

(b) *256 shadows rays per point*
67.73 seconds



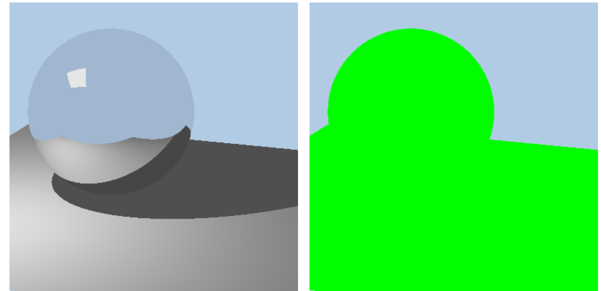
(c) *SQM*
4.42 seconds

(d) *FPM*
11.15 seconds



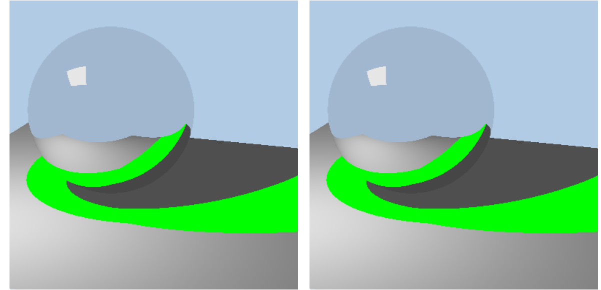
(e) *TPM*
4.24 seconds

Figure 5: *Comparison of soft shadowing methods.* $\epsilon = 0.01$. *The last three all have 256 as the maximum number of rays per point so as to stay consistent with the non-adaptive method.*



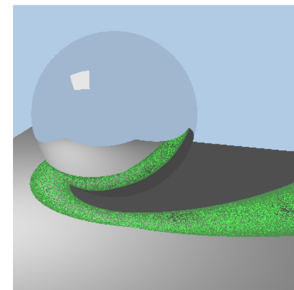
(a) *Perfect shadows—no soft shadows*

(b) *256 shadow rays per point*



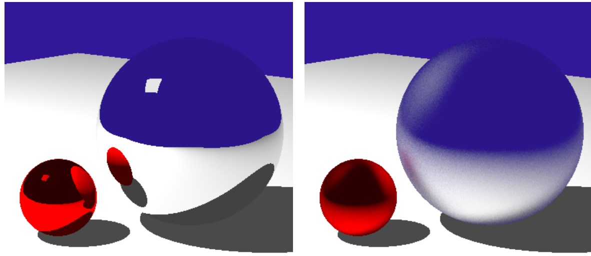
(c) *SQM*

(d) *FPM*



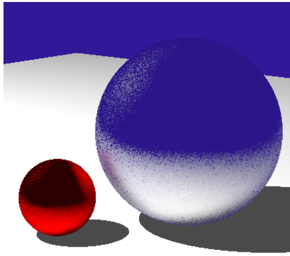
(e) *TPM*

Figure 6: *The same as the previous figure, except the points affected by each soft shadowing method are in green—the brighter the green, the more shadow rays were used at that point.*



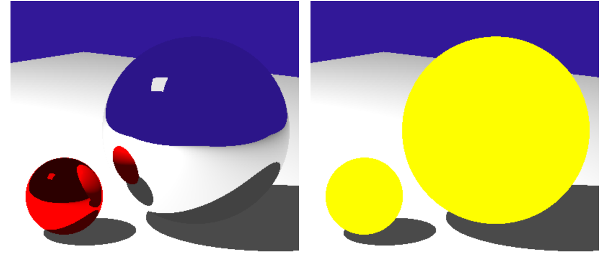
(a) *Perfect reflections—no glossiness*
1.75 seconds

(b) *256 glossy rays per reflective point*
67.49 seconds



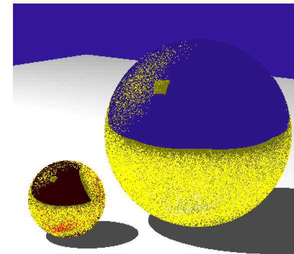
(c) *Adaptive*
29.77 seconds

Figure 7: *Comparison of glossy reflection methods.* $\epsilon = 0.05$. *256 is the maximum number of reflected rays per point so as to stay consistent with the non-adaptive method.*



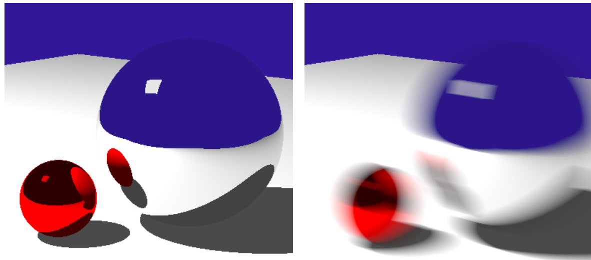
(a) *Perfect reflections—no glossiness*

(b) *256 glossy rays per reflective point*



(c) *TRIM*

Figure 9: *The same as in figure 7, except the points affected by each glossy reflection method are in yellow.*

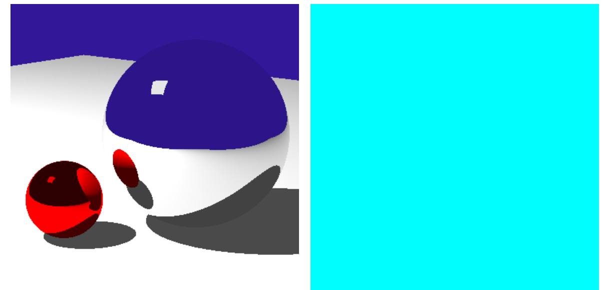


(a) *No motion blur*
1.71 seconds

(b) *256 temporal rays per pixel*
340.19 seconds

(c) *Adaptive*
163.89 seconds

Figure 8: *Comparison of motion blur methods. 256 is the maximum number of temporal rays per pixel so as to stay consistent with the non-adaptive method.*

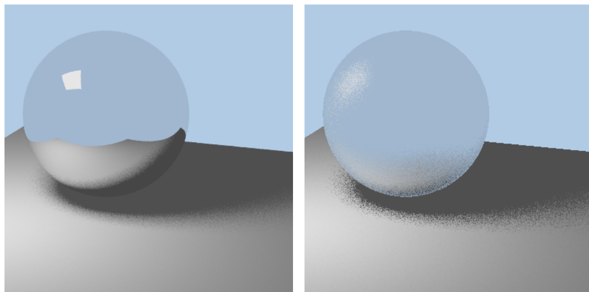


(a) *No motion blur*

(b) *256 temporal rays per pixel*

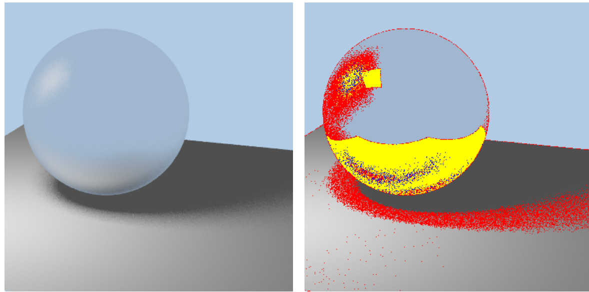
(c) *Adaptive motion blur*

Figure 10: *The same as in figure 8, except the points affected by each motion blur method are in cyan.*



(a) *Only antialiasing TUM 11.16 seconds*

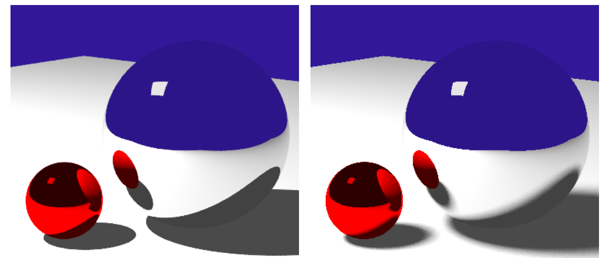
(b) *Only adaptive glossiness 4.12 seconds*



(c) *Both antialiasing TUM and adaptive glossiness 25.12 seconds*

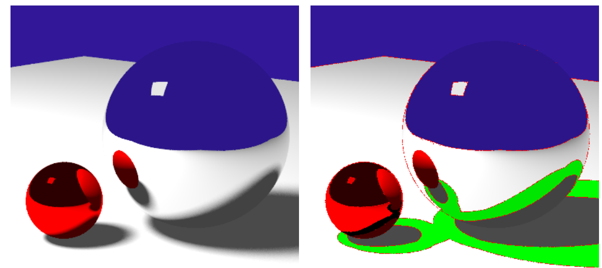
(d) *Visualization of the adaptive methods. Red: solely antialiased, Yellow: solely glossied, Blue: both.*

Figure 11: *A combination of antialiasing and glossiness (with a 4 constant non-adaptive soft shadow samples per point).*



(a) *Only antialiasing FUM 10.38 seconds*

(b) *Only soft shadowing FPM 12.59 seconds*



(c) *Both FUM and FPM 132.04 seconds*

(d) *Visualization of the adaptive methods: pixels that get antialiased are in red while soft-shadowed pixels are in green*

Figure 12: *A combination of antialiasing and soft shadows.*