Abstract

We describe a simple extension for MSAA to ray tracing and describe how to integrate motion blur with MSAA. We show that MSAA extends naturally to a ray tracer, but it does not integrate well with motion blur. To solve this problem, we add a cache to help reduce unnecessary shading.

We then show that the quality and performance of our MSAA + motion blur solution makes it worthwhile for many ray tracing workloads.

1 Introduction

Consider the following two functions from a generic ray tracer:

```c
// get closest intersection.
TriangleId intersect(Ray r);

// get color of the closest intersection.
Color shade(Ray r);
```

In general, the `shade` function is considerably more expensive to compute than the `intersect` function. Even in the simplest shaders, `shade` will at least call `intersect` dozens of times. However in most common physically based rendering techniques the `shade` function will be vastly more complex. Therefore, if at all possible we would like to reduce the number of times we shade as much as possible. However, we assume that it is relatively cheap to compute the nearest intersections.

Coupled with the assumption that luminance varies relatively slowly over the surface of a geometry, as well as time, we can adapt a reduced shading anti-aliasing technique known as multi-sample anti-aliasing (MSAA). The idea of MSAA is to reduce the number of shades we perform per pixel. Added complexity comes when we want to add MSAA to motion blur. In order to deal with this we investigate the performance of caching.

In this paper we extend a ray tracer with MSAA and motion blur and then evaluate the performance.

We largely extend the ideas presented in [Ragan-Kelley et al. 2011], which explored rendering distributed effects with MSAA in a rasterization setting. [Burns et al. 2010] also propose a similar system, which essentially reorders the shading computations in order to save the amount of shading, again in a rasterization setting.

[Apodaca and Gritz 1999] seems to be the only ray tracer to incorporate motion blur and multi-sample anti-aliasing (MSAA) and Pixar have used it while rendering the movie ‘Cars’ [Christensen et al. 2006].

2 MSAA and Motion Blur

First we describe a simple extension of MSAA to ray tracing, and then we describe how to extend MSAA to support motion blur.

2.1 Extending MSAA to Ray Tracing

In Algorithm 1 we show the basic pseudo-code for implementing MSAA in a ray tracer.

```plaintext
Algorithm 1: Extension of MSAA for ray tracing.

input : pixels: image sample points, rays: set of sample points
output: MSAA anti-aliased image.
begin
for (i, j) ∈ pixels do
  weights ← map from primitiveId to int.
  count number of intersects with each primitive.
  for ray ∈ rays do
    primitiveId ← intersect (ray)
    weights [primitiveId ] ← weights [primitiveId ] + 1

  pixels [i, j] ← Black
  color one of the intersection points and weight by the number of intersections.
  for (id, numIsects) ∈ weights do
    pick a ray that intersects ‘id’.
    ray ← randomRay (id)
    color ← shade (ray)
    weight ← numIsects / len (rays)
    pixels [i, j] ← pixels [i, j] + weight × color
end
end
```

Figure 1 shows the quality difference between super-sampling at 8 shades per pixel, versus MSAA at less than 1.4 shades per pixel. The loss in quality is minimal, despite the significant reduction in shading rate.

We would like to get this shading rate when combining MSAA with motion blur, which turns out to be a non-trivial modification.
2.2 Motion Blur

Briefly, we will describe a basic algorithm for implementing motion blur.

The basic idea is to apply stochastic sampling to time as well as space.

Consider rendering a frame that is open from time $t_0$ to $t_1$. Each time we perform an intersection with a geometry we pick a time point $t$, with which we use to shade. To get $t$, we pick a random sample $\delta \in [0, 1)$ and let $t = t_0 + \delta(t_1 - t_0)$.

2.3 Combining MSAA with Motion Blur

The pseudo-code for our algorithm is presented in Algorithm 2.

Algorithm 2: Implementation of Cache for ray tracing.

```plaintext
input : scenes: the 3D scene, rays: set of sample points
output: MSAA anti-aliased image.
begin
primitiveId ← intersect(ray)
point ← Point of intersection on the differential geometry
binId ← getBinIndex(point, primitiveId, rayDiffXY)
if binId in Cache then
    color ← interpolate(point, rayDiffXY)
else
    color ← shade(ray)
    Add color to cache
end
end
```

While the basic algorithm described in the previous section works well on scenes with static geometry, the algorithm breaks down once a point on a geometry is able to move from one pixel to another in the same frame.

The reason this is not handled by MSAA is because nearby points on the geometry can move from one pixel to another. This means that we might shade one point, $p$ in pixel $a$ and then move to a nearby pixel, $b$ and get an intersection with a point $q$ that is very near $p$. This means that we are essentially wasting a shade. We would therefore like a way to capture such intersection cases and make sure that we reuse shading. This problem is shown clearly in Figure 3. Notice that the naive MSAA algorithm performs very poorly at reusing shades, even with very modest motion.

In order to make use of the MSAA assumptions listed in the introduction, we added a caching mechanism behind the basic MSAA algorithm, which collects misses from motion blur.

The basic idea behind the algorithm is to split each primitive into blocks. A block is a small subregion of a primitive, over which we will ensure that we do not shade more than once. The size of each block is dependent on the ray differentials of the particular ray we are tracing [Igehy 1999].

For each ray shade request, we first check whether the block has been shaded or not.

If the block has been shaded then we take a weighted interpolation of the surrounding shade regions (the actual interpolation scheme is not all that important) and let the shade color be set to the interpolated color. It turns out this scheme works well in practice and does not result in much loss in quality. This is at least partially explained by the fact that as blur increases (longer blurs), the number of cache requests we get will also increase - which would
result in loss of accuracy due to the fact that we are not computing as many shade points. However, this is offset by the general rule that as the objects get blurrier, it gets harder to spot inaccuracy.

If the block has not been shaded then we shade and store the point in our cache.

In our project we did not investigate how large our cache should be. We only checked that the cache did not overflow the amount of free memory available on our system. As shading becomes more and more expensive it may well be useful to actually have boundless caches in order to make sure that only in the very worst case do we ever miss a shade. Cycle counting the average and standard deviation for a set of typical shades would give a good indication of how much space to dedicate to the cache, and which level of the memory hierarchy you are willing to spill the cache to.

3 Evaluation

To evaluate our workload, we initially started off by varying the thread task size and investigated its impact on the cache performance. Fig 4. shows the impact of thread tasks of sizes 4, 128 and 512 with the lighter color meaning more shading. If the task width increases per thread, we see a speedup due to decrease in the number of overlapping shading. Since the blur is generally within the same grid region, it results in better performance as we have a cache hit per core. On the other hand, if the tasks become too large, then we see a steady increase in the render times because of the high variability in the workload. This theory is collaborated by the diagram in Fig. 4. In general we hope that each cache should be disjoint, in that there should be as little redundancy in the caches over multiple cores. Figure ?? shows that in general, for large tile sizes this is generally true. The sweet spot of 256 width of thread task / pixels was found to be the ideal estimate for the most of the scenes.

The maximum speedup of our implementation is achieved in Bunny. To further enhance the performance, we initially use mutex locks to atomize cache access for each thread but this resulted in a dramatic drop in performance. This was primarily because now each pixel was shaded sequentially. To overcome this problem, we made the cache local to each thread as we did not have to worry about two threads reading and writing data at the same time. This approach resulted in the speedup that is seen in Fig. 6.
4 Conclusion

The main idea behind this paper was to showcase that we can combine MSAA and motion blur and integrate it with a ray tracer. We solve the problem by using cache to speedup the rendering process by shading as less as possible and storing the result. Finally, we demonstrated a significant performance increase achieved by using our method over the more traditional supersampling approach.

References


