



Simple Geometric Image Denoising  
for **Real-time Path Tracers**

While developing [Final Stage](#) I created a **basic image denoiser that was surprisingly effective** given its simplicity.

Now, there's a lot of research in this area, and several robust solutions are available that leverage artificial intelligence and GPU acceleration to produce [stunning results](#).

*So why didn't I use them, and instead decide to reinvent a wheel?*

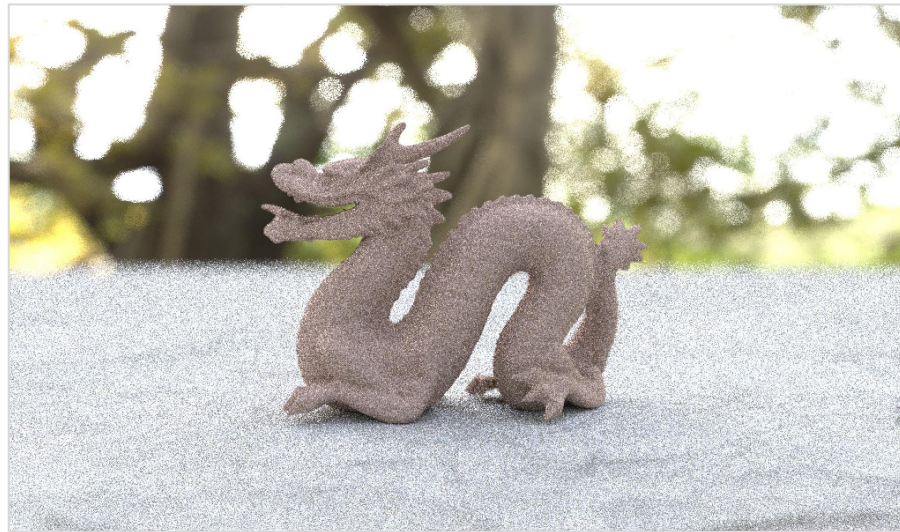
- **Final Stage is a fun side project.** *Reinventing the wheel is part of the fun!*  
This lets you see how far you can get by yourself and learn a bit about wheel construction.
- **I wanted a simple approach** that I could drop into Final Stage, without requiring special hardware or closed/third-party software.

## THE PROBLEM

Photorealistic rendering algorithms often rely on **random sampling** to render an image of a scene. This approach presents results quickly, but produces highly noticeable image noise.

Over time, this noise will become less visible as the pixel samples converge on their final values, but this can take a very long time.

Image denoising is a process integrated into the path tracer that helps us **quickly improve our image quality by filtering and smoothing out grainy artifacts.**



*Final Stage render using 50 samples per pixel (spp). Due to the low sample count, this image suffers from significant grainy noise.*

**Image denoising is actually a broad collection** of very useful processes that are used not only in path tracers, but also in video compression, computer vision, video capture pipelines, and much more.

Anywhere that noise may find its way into an image, denoising filters are there to help clean things up. There are many different kinds of noise, and fortunately there are equally many ways of correcting it.

Before we dive into the process, let's take a quick look at some results to provide context



no denoise filter



geometric denoise filter



no denoise filter

geometric denoise filter



**Denoising filter introduces blur and gradient artifacts**, *but removes graininess*.  
From a visual perception standpoint, the filter appears to roughly improve image quality equivalent to that of a 300% increase in sample count.

Image improvement more readily noticeable on smooth diffuse surfaces, less so on reflective or complex surfaces.

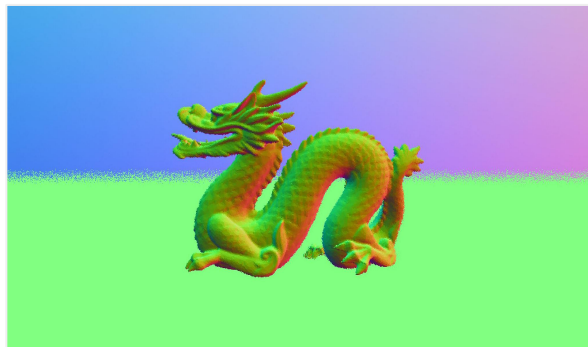
1

**Trace the scene as normal, but keep track of the following information for each pixel:**

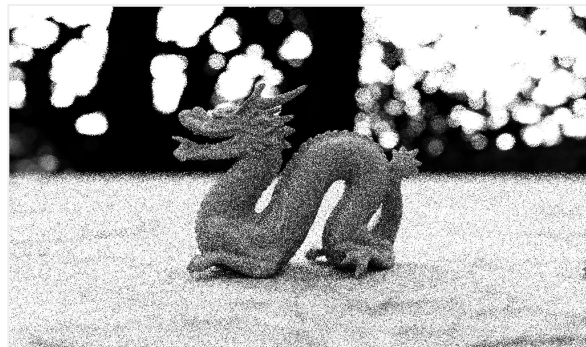
- Distance to the nearest object (i.e. depth map)
- Normal vector at the nearest object (i.e. normal map)
- Material id of the nearest object (just a random number generated for each material type)
- Mean color value of all the samples collected thus far
- Variance observed thus far



depth map



normal map



variance map (scaled)

2

**After each frame completes, apply a smoothing filter to the image:**

```
for each pixel  $p$  in the image:
    color_total = color at pixel  $p$ 
    kernel_size = pixel_variance * 2.0

    for each neighboring pixel  $n$  within a kernel_size radius:
        if material id at  $n$  != material id at  $p$ :
            skip pixel  $n$ 
        if depth at  $n$  is > depth_threshold away from depth at  $p$ :
            skip pixel  $n$ 
        if dot(normal at  $n$ , normal at  $p$ ) is > normal_threshold:
            skip pixel  $n$ 

        weight = (1.0 - distance of  $n$  to  $p$ ) / kernel_size
        weight_total += weight
        color_total += (color at  $n$ ) * weight

if color_total > 0:
    color_total /= weight_total
```

2

After each frame completes, apply a smoothing filter to the image:

```

for each pixel  $p$  in the image:
    color_total = color at pixel  $p$ 
    kernel_size = pixel_variance * 2.0

    for each neighboring pixel  $n$  within a kernel_size radius:
        if material id at  $n$  != material id at  $p$ :
            skip pixel  $n$ 
        if depth at  $n$  is > depth_threshold away from depth at  $p$ :
            skip pixel  $n$ 
        if dot(normal at  $n$ , normal at  $p$ ) is > normal_threshold:
            skip pixel  $n$ 

    weight = (1.0 - distance of  $n$  to  $p$ ) / kernel_size
    weight_total += weight
    color_total += (color at  $n$ ) * weight

if color_total > 0:
    color_total /= weight_total

```

Arbitrarily multiplying pixel variance to get kernel size.

Determined by trial and error, 2.0 seems to work well.

2

After each frame completes, apply a smoothing filter to the image:

```

for each pixel  $p$  in the image:
    color_total = color at pixel  $p$ 
    kernel_size = pixel_variance * 2.0

    for each neighboring pixel  $n$  within a kernel_size radius:
        if material id at  $n$  != material id at  $p$ :
            skip pixel  $n$ 
        if depth at  $n$  is > depth_threshold away from depth at  $p$ :
            skip pixel  $n$ 
        if dot(normal at  $n$ , normal at  $p$ ) is > normal_threshold:
            skip pixel  $n$ 

    weight = (1.0 - distance of  $n$  to  $p$ ) / kernel_size
    weight_total += weight
    color_total += (color at  $n$ ) * weight

if color_total > 0:
    color_total /= weight_total

```

Our kernel size scales with our uncertainty about the true pixel value.

Less certainty → accumulate more pixels.

2

After each frame completes, apply a smoothing filter to the image:

```

for each pixel  $p$  in the image:
    color_total = color at pixel  $p$ 
    kernel_size = pixel_variance * 2.0

    for each neighboring pixel  $n$  within a kernel_size radius:
        if material id at  $n$  != material id at  $p$ :
            skip pixel  $n$ 
        if depth at  $n$  is > depth_threshold away from depth at  $p$ :
            skip pixel  $n$ 
        if dot(normal at  $n$ , normal at  $p$ ) is > normal_threshold:
            skip pixel  $n$ 

        weight = (1.0 - distance of  $n$  to  $p$ ) / kernel_size
        weight_total += weight
        color_total += (color at  $n$ ) * weight

if color_total > 0:
    color_total /= weight_total

```

Only include nearby pixels that are likely from the same object and lit under similar circumstances

2

After each frame completes, apply a smoothing filter to the image:

```

for each pixel  $p$  in the image:
    color_total = color at pixel  $p$ 
    kernel_size = pixel_variance * 2.0

    for each neighboring pixel  $n$  within a kernel_size radius:
        if material id at  $n$  != material id at  $p$ :
            skip pixel  $n$ 
        if depth at  $n$  is > depth_threshold away from depth at  $p$ :
            skip pixel  $n$ 
        if dot(normal at  $n$ , normal at  $p$ ) is > normal_threshold:
            skip pixel  $n$ 

        weight = (1.0 - distance of  $n$  to  $p$ ) / kernel_size
        weight_total += weight
        color_total += (color at  $n$ ) * weight

    if color_total > 0:
        color_total /= weight_total

```

Weighted average of  
neighbors by distance  
to the current pixel ( $p$ )

An important detail — when computing our variance it's important that we use an ***incremental approach*** that's *quick* and has a *constant cost*.

We may need to filter a high resolution image with millions of samples per pixel, and computing variance can quickly become a prohibitively expensive operation.

*My solution:* compute an ***incremental mean*** and an ***incremental approximation to variance***. This requires only a minimal amount of work per frame, and avoids having to re-analyze the full sample set.




**Mean**

```
vector3 ComputeMean(list_of_samples, sample_count):  
  for each sample s in list_of_samples:  
    mean += (1 / sample_count) * s  
  return mean
```

**Variance**

```
vector3 ComputeVariance(list_of_samples, sample_count, mean):  
  for each sample s in list_of_samples:  
    mean_delta = (s - mean).length()  
    variance += (1 / sample_count) * (mean_delta * mean_delta)  
  return variance
```




**Mean**

```
vector3 ComputeMean(list_of_samples, sample_count):  
  for each sample s in list_of_samples:  
    mean += (1 / sample_count) * s  
  return mean
```

**Variance**

```
vector3 ComputeVariance(list_of_samples, sample_count, mean):  
  for each sample s in list_of_samples:  
    mean_delta = (s - mean).length()  
    variance += (1 / sample_count) * (mean_delta * mean_delta)  
  return variance
```



This approach requires us to re-examine all of our samples each frame, ***which is a non-starter.***

## INCREMENTAL APPROACH

### Incremental Mean

```
vector3 ComputeIncrementalMean(previous_mean, sample_count, incoming_sample):  
    return (current_mean * sample_count + incoming_sample) / (sample_count + 1)
```

### Incremental Variance (approximation!)

```
vector3 ComputeIncrementalVariance(  
    previous_variance, sample_count, incoming_sample, incremental_mean):  
    mean_delta = (incoming_sample - incremental_mean).length()  
    return (sample_count * previous_variance +  
            (mean_delta * mean_delta) / (sample_count + 1))
```

By retaining the previous frame's mean and approximated variance, and incorporating new samples directly, we avoid having to re-evaluate the entire sample set.

## INCREMENTAL APPROACH

### Incremental Mean

```
vector3 ComputeIncrementalMean(previous_mean, sample_count, incoming_sample):  
    return (current_mean * sample_count + incoming_sample) / (sample_count + 1)
```

### Incremental Variance (approximation!)

```
vector3 ComputeIncrementalVariance(  
    previous_variance, sample_count, incoming_sample, incremental_mean):  
    mean_delta = (incoming_sample - incremental_mean).length()  
    return (sample_count * previous_variance +  
            (mean_delta * mean_delta) / (sample_count + 1))
```

**Incremental variance != variance**  
Output is only loosely correlated!

By retaining the previous frame's mean and approximated variance, and incorporating new samples directly, we avoid having to re-evaluate the entire sample set.

Simple technique that yields reasonable results, but approach is ***highly dependent*** upon hand tuned thresholds (depth\_threshold, normal\_threshold, kernel size multiplier). In theory this could be automated, but for lower level of effort I would instead opt to integrate a commercial solution.

Properly tuned thresholds appear to roughly **improve image quality equivalent to that of a 300% increase in sample count**. Poorly tuned thresholds will result in excessive blurring artifacts, which may be worse than the noise we are attempting to correct!

Overall a quick and fun project with lots of room for experimentation.



**Thanks for listening (or reading)!**

Don't forget to check out the source for [Final Stage 2.0](#), which demonstrates the concepts discussed in this talk.