Rendering Accessories

Adam Celarek

Research Unit of Computer Graphics
Institute of Visual Computing & Human-Centered Technology
TU Wien, Austria

Hi and welcome to this lecture about rendering accessories. By accessories I mean things, that are important, but don’t really fit into any of the other lectures.

Overview

- Sampling theory and filtering
- Parallelisation
- Post-processing
- Measuring Error
- Stratification and sampling patterns

We will start with a bit of sampling theory and learn how to use filtering to prevent aliasing.

Next we'll talk about parallelisation, and how to deal with filtering in practise.

Post-processing is at the end of the rendering pipeline, we will learn about how to map the high-dynamic range data to computer displays and files.

Then we'll have to talk about measuring error a bit, and last but not least we'll take a step back and see how to reduce error by using more uniform samples.

Let’s start with sampling theory and filtering, the goal of which is to reduce or prevent sampling artefacts, or in other words aliasing.

By the way, that picture is from Finland, the home of some very good computer graphics people..

Sampling theory and filtering

With filtering

Without filtering

source: Table scene designed by Olesya Jakob rendered with

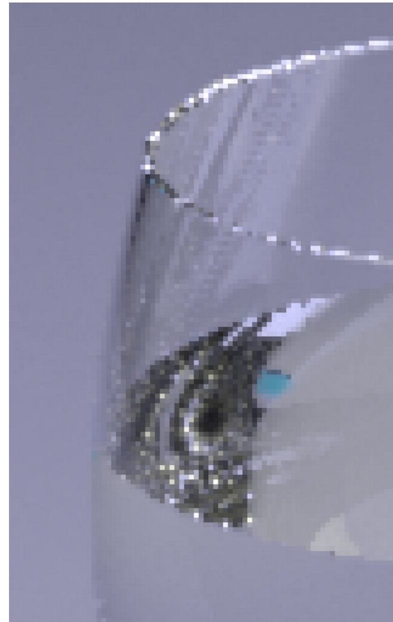
We begin with an example..

On the left filtering was used, on the right not.

The left looks a bit better when you compare **point** and **point**, but we can zoom in to see the difference more clearly..

Sampling theory and filtering

With filtering



Without filtering

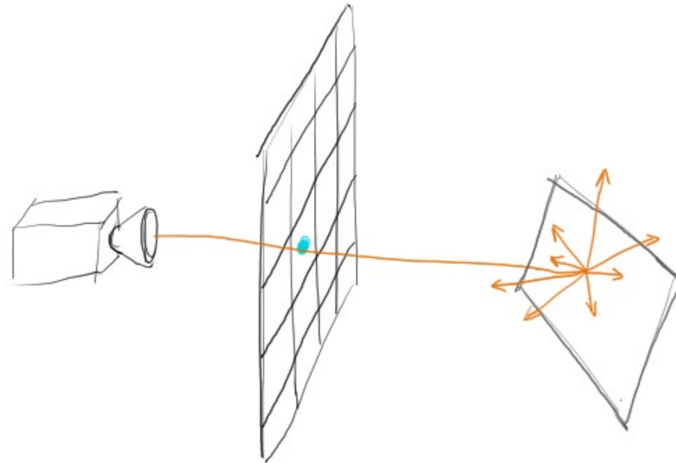
source: Table scene designed by Olesya Jakob rendered with

There are these bright pixels **point** in high frequency areas, that clearly go away when we filter..

We have to look into it :)

Sampling theory and filtering

- We are in the main rendering loop, where we loop over pixels and the number of samples N, create a camera ray and then start tracing it.

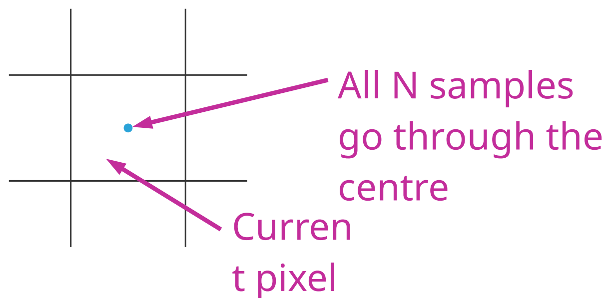


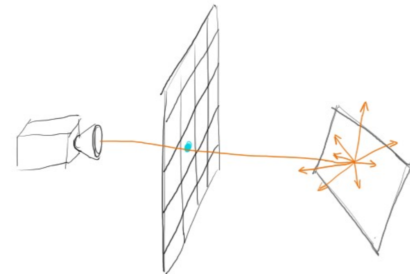
Let me give some context about where filtering happens..

It is applied in the image coordinate system, where we see the pixels. That means that it lives in the main rendering loop, where we walk over pixels – in contrast to where we lived in the first part of the path tracing lecture, the recursive part.

Sampling theory and filtering

- We are in the main rendering loop, where we loop over pixels and the number of samples N, create a camera ray and then start tracing it.
- Until now, every ray was shot through the pixel centre



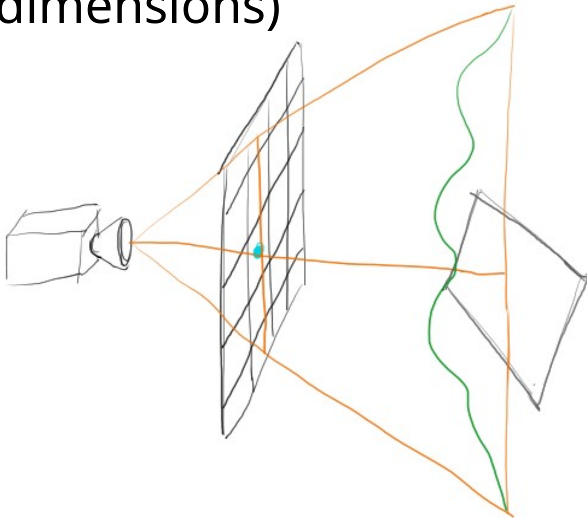


Okey, this is for context: We are in the main rendering loop, where we go over all pixels and the number of samples N, create a camera ray and start tracing it.

Until now, we shot every pixel contribution ray through the centre. But this is just an arbitrary choice. The underlying integral and code works for all floating point positions in the pixel plane.

Sampling theory and filtering

- Forget one screen dimension (works the same for higher dimensions)



$$f(x) = \int_{\Omega} f_j(\bar{x}) \, d\mu(\bar{x})$$

Next, we'll forget about one of the screen dimensions and start to think about a 1d function f(x). This is to make the explanation easier. All of the results also apply to the 2d screen we have in rendering.

The green wavy line *point here and here*, is now the function that we want to sample and reconstruct as pixels on the screen.

Here **point** you see the path integral formulation of light transport, because it's shorter.

It's the integral *point f* of the light transport function *point fj* over all transport paths. You already learned in the first part of this lecture how to estimate the result using Monte Carlo. It's the same process no matter the notation.

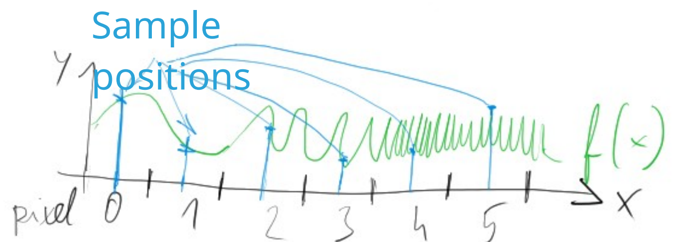
But we can also use the recursive formulation. In that case, we would say that we have a function f that eats screen coordinates. The definition of the function tells to shoot a ray through that position, and compute the light recursively. That recursive computation returns the brightness at that position of the screen. Until now it was always the pixel centre, but it's valid for all screen positions.

For now we assume that f(x) is a continuous function – as opposed to a stochastic variable like in Monte Carlo Integration. That means that it always returns the same value for a given position, and positions are not discrete like monitor pixels.

Sampling theory and filtering

- Forget one screen dimension (works the same for higher dimensions)
- This is a sampling problem, much like with an analogue audio signal.
- We want to reconstruct the original signal after sampling

Sample positions

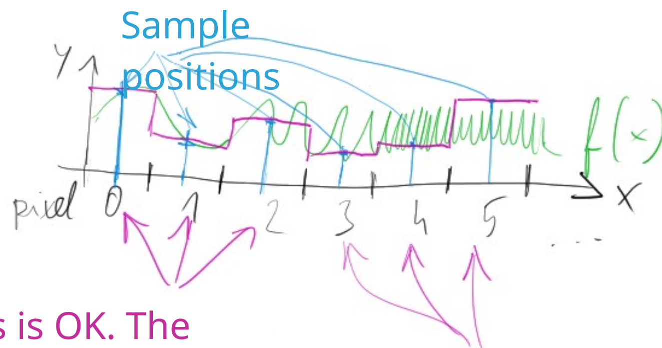


Now – we have that function *point f*, but only discrete pixels *point*, where each of them can display a single value.

This is a sampling problem, very much like when you have to convert an analogue audio signal to a digital signal, and then play it.

Let's say, we go the simple and naive way and sample the signal at the centres of the pixels..

Sampling theory and filtering



This is OK. The reconstruction is faithful!

Here not so much!

After sampling we have to reconstruct the signal, that is, draw it on the screen. As said, every pixel can have only one value, hence the boxy **point** reconstruction.

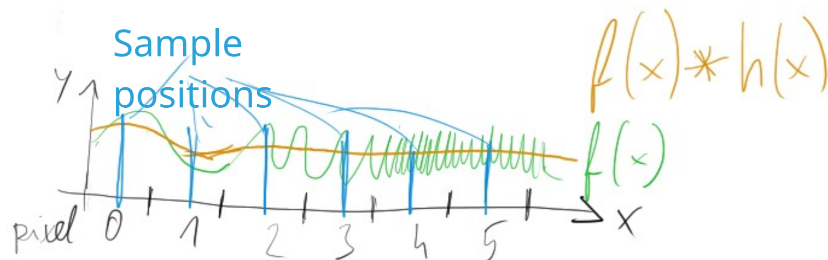
This gives us a faithful representation of f if the frequency is low **point** on the left. By faithful I mean here, that the pixel values are close to the signal f, and that they would not change much if we shift the sample positions by a tiny bit (less than a pixel).

So sampling the pixel centres works if the signal f has a low frequency, but it breaks down when the frequency becomes too large. The problem is called aliasing, you probably already heard about it. It happens when the sampling frequency is below the Nyquist frequency, which is 2x the signal frequency.

There are 2 solutions: increase the sampling frequency or smooth the signal. We can’t really do the first, because it would change the rendering resolution, besides, when would we stop? But we can smooth the signal...

Sampling theory and filtering

Solution: Smooth the signal before sampling with a low pass filter h(x)



Smoothing the signals means to apply a low pass filter h. We are filtering out high frequencies – hence the name of this section.

Look at that, if we sample the signal f*h **point** at the centres of the pixels, it will give us faithful values to be used for the reconstruction as a pixel signal.

You might think that we loose data, but that is not really the case as we couldn't reproduce it anyway.

Let's look now into choices for the filter (or convolution) kernel h

Sampling theory and filtering

What are the choices for smoothing kernel h(x)?

Well, first, what properties do we need?

- It should integrate to 1,
  otherwise the result would be brighter or darker than it should be.
  Well, actually, that is only needed theoretically, because our implementation will use normalisation anyways.
- The smoothing should be the right size
  - Too small and too many of the large frequencies are retained
  - Too large and we get visible blurring
- A shape that reduces artefacts (will not go into detail, see reading material).

But first, let's look at the properties that we need.

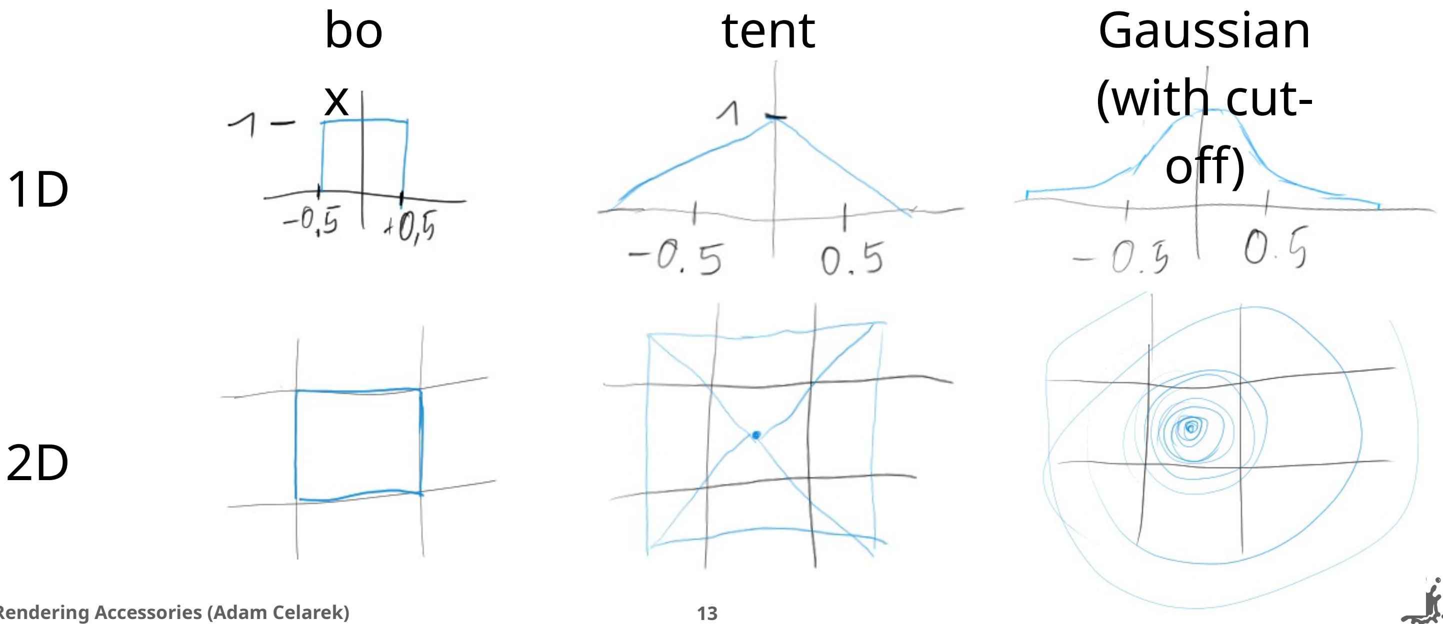
First, it should integrate to 1. This is because f(x) convoluted with h(x) should not be brighter or darker than the original signal f. This is more of a theoretical constrain, as any convolution kernel can be scaled and in fact we do that as part of the equation that we use.

Obviously the kernel shouldn't be too small or too large. In the first case we wouldn't fix the reconstruction problem, or in the second case we would loose too much details.

And finally, we need a good shape. Depending on the shape, some artefacts might appear. Different shapes can create different artefacts, and it's generally a trade-off between these artefacts. But we won't go into much detail there, because we would need much more sampling and reconstruction theory including the Fourier transform (not going there, not this time :)

Sampling theory and filtering

What are the choices for smoothing kernel h(x)?



Here we see some choices for the kernel.

The first row is the 1d case and the cross section through the centre of the 2d case at the same time.

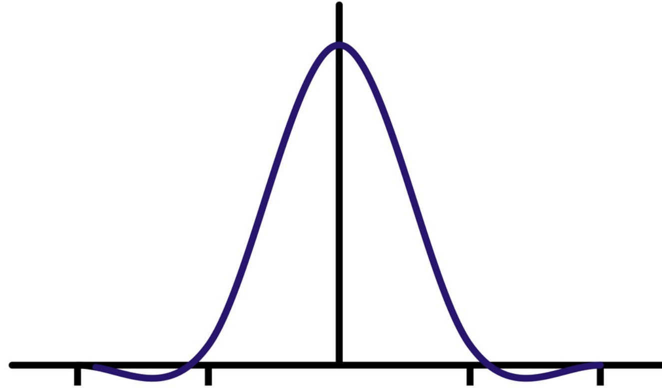
The simplest is the box filter, which is super simple to implement. You just take the average of all pixel samples (we'll talk more about how to implement later). It is common in super simple renderers, in case the author is too lazy to implement something better. It allows high-frequency sample data to leak into the reconstruction (PBR 7.8.1).

The tent filter is slightly better

And the Gaussian filter is reasonably good, but smooths the result.
Did you see that it is clamped **point**. This is to limit it's support.

Sampling theory and filtering

What are the choices for smoothing kernel h(x)?



Mitchell-Netravali-Filter with parameters (⅓, ⅓)

source: Vierge Marie, Wikipedia (public domain)

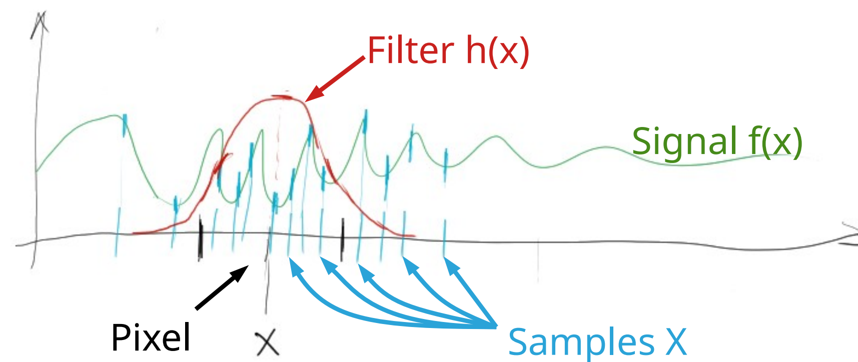
There is a family of filters called Mitchell-Netravali. You can see that they can be negative **point**. This sharpens the result somewhat, but can cause an artefact caused ringing (depending on the parameters).

The Mitchell-Netravali filter is considered the best by some, and I think it is a bonus task in our assignments.

Anyway, we will not go into further details in the differences. In my opinion, they are minute between Gaussian and Mitchell-Netravali.

Sampling theory and filtering

How?



Ok, we have seen why, we have seen the filters, but not how that works in the context of rendering. So here we go:

You see the signal f(x) and the filter kernel h(x) here. Before we said that we compute the convolution between f and h in order to get a smooth signal.

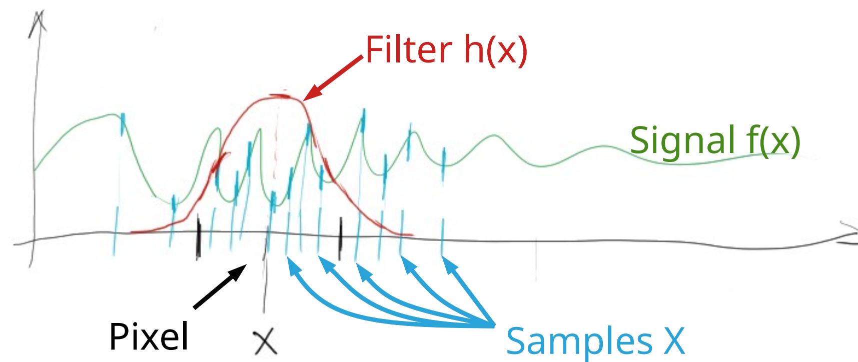
But we do not have the signal f as a function, we can only sample it. And so we need to change the approach somewhat.

We see that the centre of the filter kernel is placed over the centre of the pixel. This is like we wanted to compute the value of f convoluted with h on that position. And that makes sense, because we later want to reconstruct f(x) by sampling f*h(x) at the pixel centres.

And now we take samples of f times that shifted kernel h. Samples close to the centre of the pixel will have a large kernel weight and therefore a larger contribution to the pixel value.

Sampling theory and filtering

How?



$$I(x) = \frac{\sum_i^N \mathrm{h}(x - x_i)\,\mathrm{f}(x_i)}{\sum_i^N \mathrm{h}(x - x_i)}$$

h*f

Normalisation (box: divide by N)

Here you see the equation for that process.

In the numerator **point** you see exactly what we said before: the sample value of f is multiplied with the value of the kernel centred at the pixel we want. This is the same as convolution.

Remember we said that h should integrate to 1? So in theory we would use N instead of the sum in the denominator **point**. The sum results in N on average. However, using the formula as it is reduces noise.

Sampling theory and filtering

How?

- In 1d:

$$I(x) = \frac{\sum_i^N \mathrm{h}(x - x_i)\,\mathrm{f}(x_i)}{\sum_i^N \mathrm{h}(x - x_i)}$$

- In 2d:

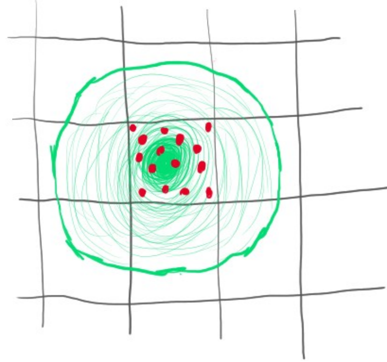
$$I(x, y) = \frac{\sum_i^N \mathrm{h}(x - x_i, y - y_i)\,\mathrm{f}(x_i, y_i)}{\sum_i^N \mathrm{h}(x - x_i, y - y_i)}$$

And here you see the equation for the 2d case, that we actually have in case of rendering. It works the same way.

Now, let’s have a graphical view of the kernel again..

Sampling theory and filtering

How?

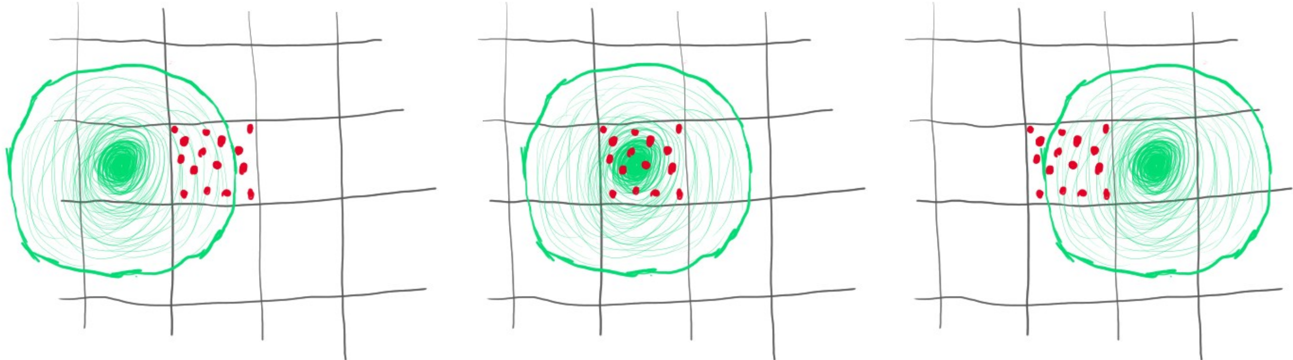


The green circle **point** is the filter, the red dots **point** are the samples for the current pixel.

The filter kernel is larger than the current pixel...

Sampling theory and filtering

How?



.. and so pixel samples also contribute to neighbouring pixels.

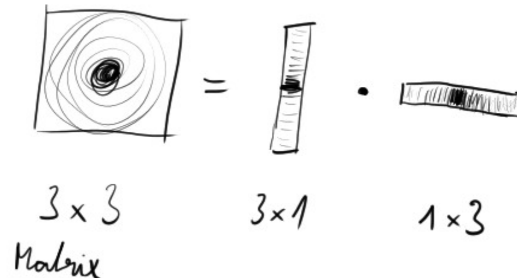
You can see that here on the left, where these samples **point** would also contribute to that pixel **point**.

Similarly, samples in this pixel **point neighbouring pixel in the centre image** also contribute to that pixel **point at current pixel in centre image**.

Etc, this applies too all the neighbouring pixels.

OK. I hope that is more or less clear, and we are ready to look at another thing concerning filters..

Separable filter kernels



Works for: Gaussian, tent, box, Mitchell-Netravali, ..

Many discrete smoothing filters are separable. That means that the matrix **point** used for filtering can be produced from the outer product of two vectors **point**.

This applies for instance to the gaussian, tent, box and Mitchell-Netrevali filters. And since these filters are symmetric, the two vectors have the same values.

Separability allows us to implement filtering more efficiently, as we need to store only one vector instead of the whole matrix. The filter values can be computed on the fly. In case you work on the last assignment at TU Wien, this is the code that you have to complete.

Sampling theory and filtering

We know

- Why → Anti-aliasing
- Which filters → low-pass, box, tent, Gaussian, others
- How → Convolution + normalisation
- That a sample can also affect neighbour pixels, depending on the filter.

Alright..

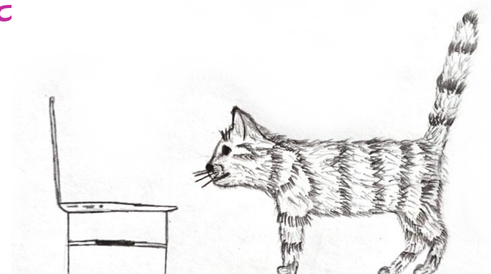
We learned that filtering fixes aliasing problems. Low pass filters are used, and the process includes convolution and normalisation.

Most filters are larger than a single pixel, and so most samples affect more than a single pixel.

Sampling theory and filtering

Reading

- Pharr, et al., Physically Based Rendering (Chapter 7)
- Smith (1995), A Pixel Is Not A Little Square!
- Mitchell & Netravali (1988),
  Reconstruction filters in computer-gra
- Search for sampling theory



Here are some reading links

Check for instance our course book. It covers sampling and reconstructing quite extensively, including more sampling theory and comparisons between the different filter types.

A classic read is “A pixel is not a little square”

You can also ready the original paper by Mitchell & Netravali

---

And finally I’ve seen, that there are many youtube videos about sampling theory, and even more text resources on the net.

That concludes our section about sampling theory and filtering, although we'll see it again in the next section. Filtering prevents or at least reduces sampling artefacts by low-passing the image signal and then sampling in the pixel centres. These samples are then used as the pixel colour.

And now, let’s have a quick look at parallelisation.
The goal is clearly to harvest the performance of modern multi-core systems.
That is already crucial today, but it becomes more and more important as the core counts increase.

Btw: If you read papers from the 90ies, that won’t be the case.

This picture is also from Finland, they have a lot of amazing lakes and forests, and summer at a cabin is super relaxing :)

Parallelisation

- Each sample $X_{j,i}$ is independent (j.. the pixels, i.. sample number)

$$I_j \approx \frac{1}{N} \sum_{i=0}^{N} \frac{\mathrm{f_j}(X_{j,i})}{\mathrm{p}(X_{j,i})}$$

- Could start a new thread for each sample
  - Synchronisation at the sum expensive
  - Overhead
- A new thread per pixel
  - Still some overhead
  - Synchronisation between pixels because of overlapping filters
- → Render blocks

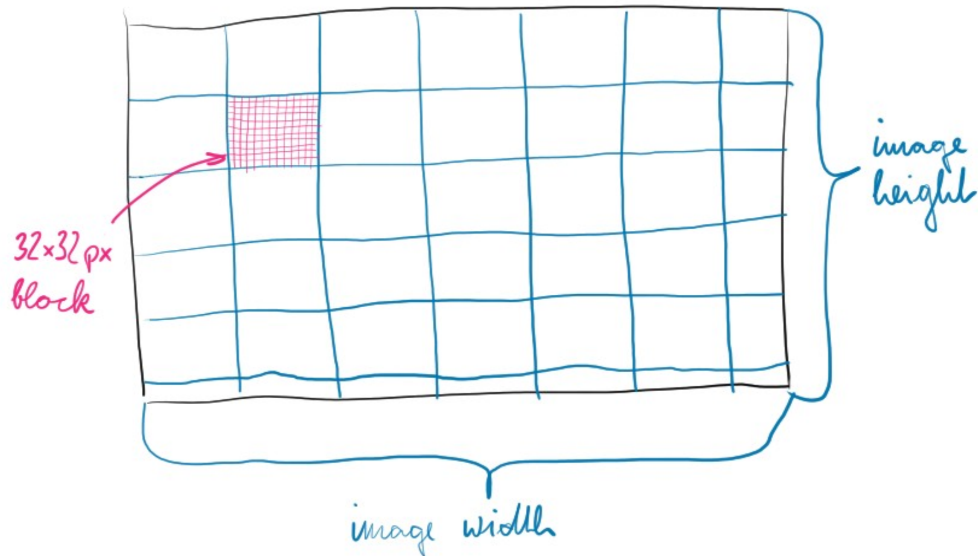
Monte Carlo rendering is embarrassingly parallel, as every sample of the integrand is independent. We are talking about the paths that start or end at the camera, not the hemisphere samples, although these would be independent as well.

So we could start a new thread for each sample. But that would require synchronisation at the sum and probably unneeded overhead for threading. That might pay off for GPUs, but probably not even that. Certainly not for CPUs.

We could start a new thread per pixel – more like GPU style, for the CPU that is still too much overhead + we would actually need synchronisation because of overlapping pixel filters (but that is doable).

So for CPUs we use blocks of pixels..

Parallelisation

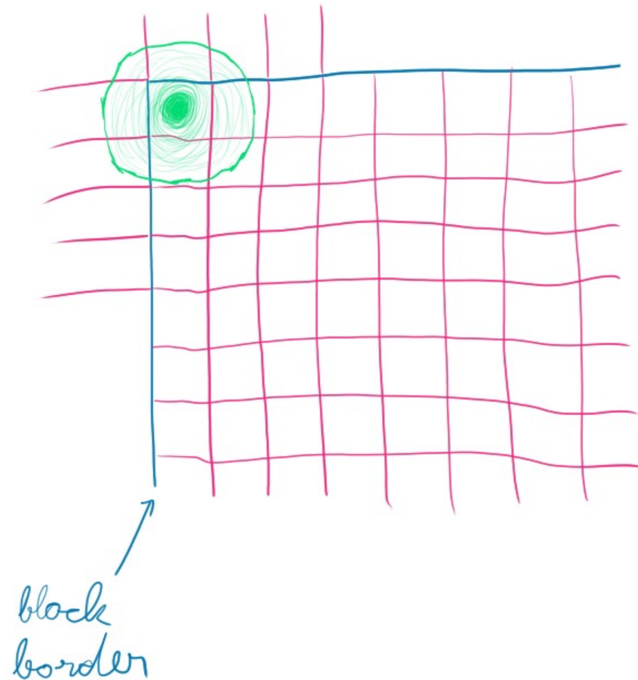


Such blocks can have a size of for instance 32x32 pixels.

Another advantage is that the first few bounces will have a high probability of hitting the same geometry, and so the caches of the CPU have fewer misses.

Blocks at the border of the image **point** are often smaller.

Parallelisation



Samples are taken for all pixels inside a block.

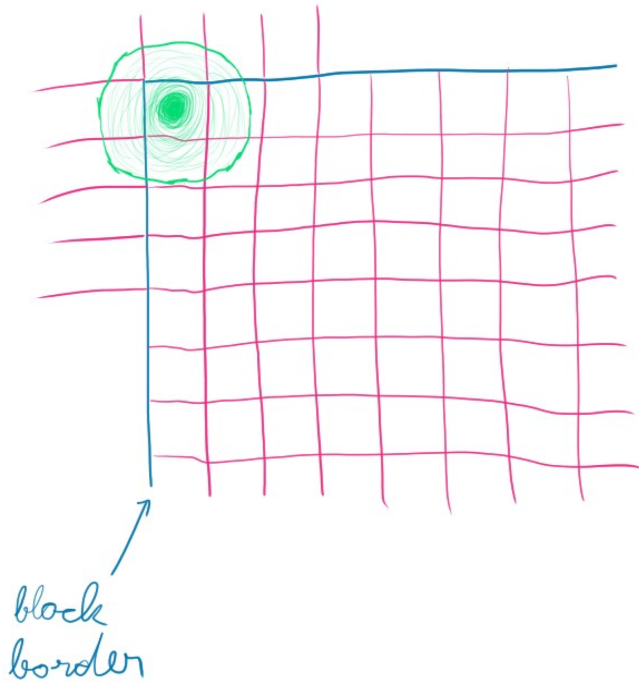
But as said, due to filtering, each sample can contribute to more than one pixel – including pixels outside the block.

Therefore, each block needs an additional border to store that contribution.

And so, when a block is handed to a thread, it must be slightly larger than the extent of the sampling area.

This means, that blocks can overlap. We will now see, how that is handled in a rendering system like nori.

Parallelisation



$$I(x) = \frac{\sum_i^N \mathrm{h}(x - x_i)\,\mathrm{f}(x_i)}{\sum_i^N \mathrm{h}(x - x_i)}$$

- Use a separate sum for the numerator and denominator.
- Per render block, store these values for a large enough border around the block as well.
- When a block is finished, add to a global image using blocking operations.

When we look back at the equation for filtering, we see that there is a sum in the numerator and the denominator. We have to complete the sum before the division. That means that we need to do the division after merging the blocks to the complete image!

And that is actually quite easy, we just keep track of numerator and denominator separately, add the blocks to the final image using blocking or atomic add operations, and do the division once all threads are finished.

Parallelisation

$$I(x) = \frac{\sum_i^N \mathrm{h}(x - x_i)\,\mathrm{f}(x_i)}{\sum_i^N \mathrm{h}(x - x_i)}$$

Vec4(red, green, blue, weight)

- Use a separate sum for the numerator and denominator.
- Per render block, store these values for a large enough margin around the block as well.
- When a block is finished, add to a global image using blocking operations.

Nori solves that by putting the 3 values of the numerator for the 3 colours and the filtering weight into a 4 element vector.

The blocking add operation doesn't even need to know what the data is.

As a last step we simply divide all pixels by their 4th component.

Parallelisation

- Divide the image in small blocks, and let 1 thread work on one block
- Some other rendering algorithms might require to hold the whole image per thread (BDPT, MLT)
- Rendering very easy to parallelise, even to multiple machines
- When using 32bit float format, we can even compute the mean of independently rendered images.
- Some years ago I rendered stuff on a super computer in Finland and used several thousand CPUs and more than a terabyte of RAM at once.

For path tracing we can use this method without any problems.

But some other algorithms require access to the whole image.
For instance, that would be the case if we started tracing from the light sources, and wouldn't know which pixel the ray ends up.

Rendering is very easy to parallelise even to multiple machines. This works by computing multiple images with different random seeds. They can be added up after finishing rendering using 32 bit floating point image formats.

That way I occupied several thousand CPUs and more than a terabyte of RAM during my master thesis at Aalto, Finland.

Some bottlenecks start to appear once your scenes become very large, for instance in large film productions. Sometimes the scenes don't even fit into RAM any more and special architectures need to be employed.

That was a short section about parallelisation.

Many renderers implement threading by rendering in blocks.

These blocks need to include a small border to account for filtering.

And it is relatively easy to parallelise even to multiple machines.

That is my cherry tree in Poland.

But we'll look at post processing now. That is, what do we do before storing or showing our computed radiance values.

Let's start with a bit of context..

Post-processing

Dynamic Range Examples	Device / Setting
Real world	
800 000 : 1	Surface illuminated by sun : moon
100 : 1	Diffuse white : black surface
80 000 000 : 1	Expected real world dynamic range
Human vision	
100 : 1	Photoreceptors
10 : 1	Pupil size
100 000 : 1	Neural adaptation
100 000 000 : 1	Dynamic range of human vision
Technology	
1 000 : 1	Displays
256 : 1	8 bit image files
10^{76} : 1	32 bit float

source: Thomas Auzinger, Rendering lecture 2019

The first thing to realise is, that the dynamic range of light in the real world is considerably larger than what we can show on displays or store in normal image files.

Take for instance the difference between illuminating a piece of black and white paper by the sun versus the moon. The sun's illumination is 80 million time stronger.

The human vision system can adapt and process a difference of up to a 100 million. Adaptation takes some time, but still.

In technology, displays can have a difference between the brightest and darkest pixel in the ballpark of thousands. Modern displays, that can partly dim their backlight can do more, but that requires special drivers and especially authored content. The default are 8 bits per colour image formats with 256 levels of brightness – seems almost archaic, but that's the deal for most cases. Our computations during the light simulation use 32 bit float, the range is large enough.

Post-processing

Tone mapping

- We compute radiance in 32 bit floating point precision
- However, most displays and image formats use an 8 bit unsigned integer format
- Map high dynamic range (HDR) image data to 0 – 1 (8 bit) for display
- We need to apply range compression
- Nowadays there are also 10 or even 12bit displays, which would need different processing (not covered)

source: Thomas Auzinger, Rendering lecture 2019

So, tone mapping is about reducing the dynamic range from our light simulation to a 0 – 1 range, that can then be mapped to an 8 bit integer.

We need to compress the range.

And on a side note, there are 10 or even 12 bit displays today, but these require different processing that probably happens in special software or in the driver anyway and we will not cover it. It is easy to output 16 or 32 linear float data for that purpose.

Post-processing

Example

(tone mapped, we don't know how to do that yet)

source: Thomas Auzinger, Rendering lecture 2019 (from http://dcgi.felk.cvut.cz/home/cadikm/tmo/)

Let's take this image as an example.

It is tone mapped in some way. For the sake of the argument, let's assume that in the light **point** right here it has a brightness of 35.5 (that's a float value that could arise from rendering). On that white piece of paper it could have a brightness of, say, 1.2, and somewhere in the dark it could be 0.1.

How to map that range to 0 – 1, so that we can then convert it to 8 bit?

Example

Division by maximum

$I_{out} = I_{in}/max(I$

source: Thomas Auzinger, Rendering lecture 2019 (from http://dcgi.felk.cvut.cz/home/cadikm/tmo/)

One option could be to divide by the largest brightness value.

That is a valid method, and – don't laugh, I used it at some point in time. If the scene has a low dynamic range, that is pretty OK. But it's certainly not a good method in terms of quality and stability.

Example

Clamp

$I_{out} = \min(I_{in}, 1$

source: Thomas Auzinger, Rendering lecture 2019
(from http://dcgi.felk.cvut.cz/home/cadikm/tmo/)

Our next attempt could be to clamp the image to a range of 0 – 1.
That is what often happens before tone mapping is implemented. With most scenes that is not that bad either.

Post-processing

Example

Exponential mapping

$I_{out} = \log(I_{in})$

source: Thomas Auzinger, Rendering lecture 2019
(from http://dcgi.felk.cvut.cz/home/cadikm/tmo/)

Taking the logarithm is a very basic tone mapper. I used it to display some scientific data in another domain. Simple, but not really proper – we would like to adjust the parameters, give more light to the shadows etc..

Post-processing

Example

Reinhard tone mapper

source: Thomas Auzinger, Rendering lecture 2019 (from http://dcgi.felk.cvut.cz/home/cadikm/tmo/)

And right here we now have a proper tone mapper – Reinhard!

Post-processing

Tone Mapper Taxonomy

- Global:
  Mapping function is uniform on the whole image
  - + Fast
  - - But loss of detail
- Local:
  Mapping function depends on the values of the pixels in the neighbourhood of the currently mapped one.
  - - Slow
  - + Local contrast enhancement

source: Thomas Auzinger, Rendering lecture 2019

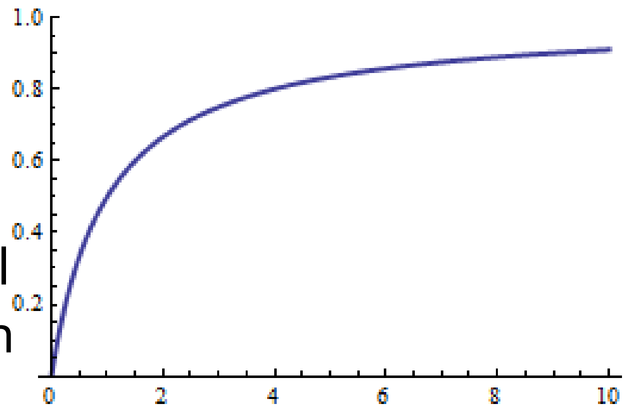
Before we continue, let me say, that there are global and local tone mapping algorithms. The global ones apply the same function to all pixels, while the local ones are applying different parametrisations depending on the local neighbourhood of a pixel.

Accordingly the local ones are slower, but they can retain more details. We'll see an example in two slides..

Post-processing

Reinhard Tone Mapper

- Reinhard et al., Photographic tone reproduction for digital images, Siggraph 2002
- Widely used
- Global and local variant
- Listing processing steps woul be tiresome,many sources on the internet (or the paper)



source: Thomas Auzinger, Rendering lecture 2019

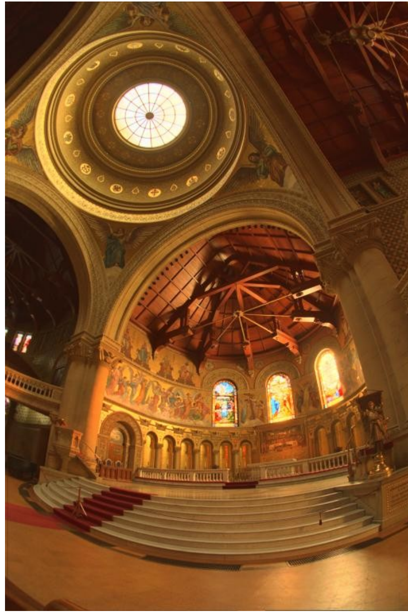
The Reinhard tone mapper is widely used and there is a global and local variant.

We will not cover the processing steps, it would be tedious and you would have to look them up when you implement anyway. Refer to the paper.

The goal of that process is to find a transfer function, like shown here. The exact shape of that function depends on the whole image in case of the global variant, and on the local neighbourhood in case of the, well, local variant.

That shape is somewhat similar to a logarithm, hence logarithms are also usable.

Post-processing



Global

Local

(from http://cybertron.cg.tu-berlin.de/pdci08/tonemapping/tone_mapping.html)

source: Thomas Auzinger, Rendering lecture 2019

Here are the examples I promised.

On the left you see a global mapping and on the right a local one.

If you look closely, you can see that some of the darker areas on the right are darker. For instance next to the wood on the roof ** point**. The window in the roof **point** is a bit – hm, blurred, there is some sort of bloom. It’s not as crisp as on the right.

Let’s zoom in a bit..

Post-processing

Global Local

(from http://cybertron.cg.tu-berlin.de/pdci08/tonemapping/tone_mapping.html)

source: Thomas Auzinger, Rendering lecture 2019

The global version is less crisp. Look at here **point saints and arches** and the wooden parts **point** are clearly darker in the local version.

Post-processing

Other Tone Mappers

- Bilateral filters
  F. Durand and J. Dorsey, Fast bilateral filtering for the display of HDR images, Siggraph 2002
- Gradient processing
  Fattal et al., Gradient domain HDR compression, Siggraph 2002
- Etc..
  Wikipedia lists over 20

source: Thomas Auzinger, Rendering lecture 2019

Reinhard is not the only tone mapper.

Here is a reading list, but there are more on Wikipedia.

Enough of tone mappers :)

Let’s turn our attention to gamma encoding, something slightly different..

Post-processing

Gamma Encoding

- Human perception of brightness follows an approximate power function.
- Greater sensitivity to relative differences between darker tones.
- Therefore, gamma encoding is used to optimise the usage of bits when storing or transmitting an image.
- It's applied after tone mapping, $I^{1/2.2}$, check online sources for more info.
- You might need to decode gamma when reading textures (most image formats store gamma encoded, but light simulation needs linear intensities)

source: Wikipedia

*read slide

Alright.

Post-processing

Mission: Store or display renderings

- Tone-mapping to map from HDR to 0.0 – 1.0 floating point.
- Gamma encoding for storage
- Not needed when using HDR formats

source: own

To sum up,
...

The next topic is measuring error.
And I know that it is a bit boring, but it’s important none the less!
It is crucial to be able to compare algorithms, but we can do even more as we will see.

The picture shows a former eucalyptus forest in Australia. The error there was to put Koalas in the trees. Too many Koalas and too few trees and of the wrong type.

Measuring Error

- Estimators estimate values – for instance the colour of a pixel
- Often denoted with a hat: $\hat{I} \approx I$
- Error of an estimator: $\mathcal{E} = \hat{I} - I$
  - Alternatively, amplitude of the error, $\mathcal{E} = \left\| \hat{I} - I \right\|$
  - relative error, $\mathcal{E} = (\hat{I} - I)/I$
  - or (mean) squared error for arrays of estimators.
- Real I not available, so use a high quality reference estimate

So estimators, well, they estimate values – for instance the colour of a pixel. As such, they don't give the real value, and therefore there is a certain error. Many times it's useful to compute the error before any post-processing, and that's what we use here.

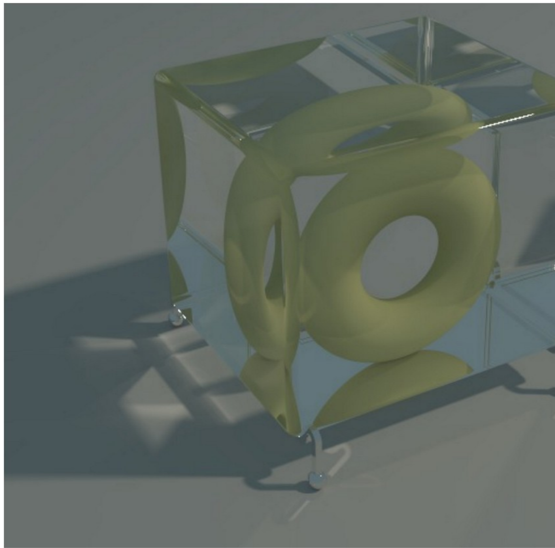
Estimators are denoted with a hat (that is a general statistician thing). And the error can be computed as the estimate - the real value. That is the absolute error as opposed to the relative error which you can see below **point** (so divided by the real value). This error **point** also contains a sign. Sometimes the unsigned error, or amplitude error is computed **point**.

The nomenclature is fuzzy, and it is also called absolute error. When we want to compute the error for the whole image, we can take the mean of all errors. But in that case we have to compute the amplitude, or better square of the error, otherwise the positive and negative errors would cancel out.

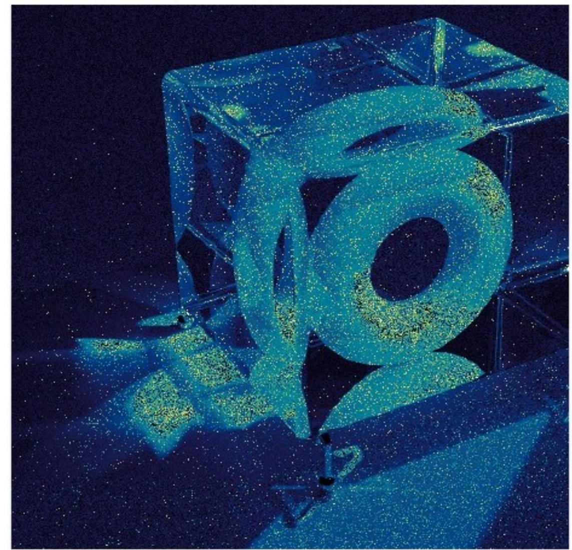
The real value I is usually not available, therefore high quality reference estimates (also called ground truth) are used.

We will now see some examples of such errors

Measuring Error



Reference



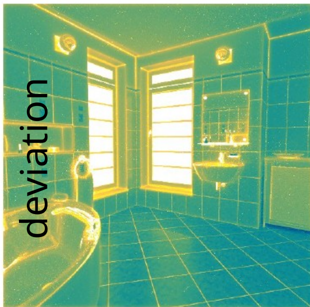
Absolute Error

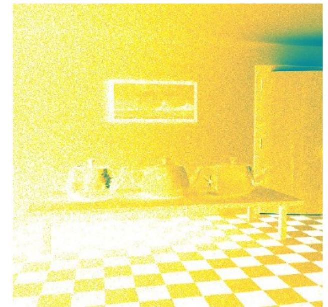
On the left you see a high quality reference and on the right the absolute error of a path tracing algorithm.

Measuring Error

References

Standard deviation





(a) absolute (b) relative (c) absolute (d) relative

In here you can get an idea of the difference between absolute and relative error.

The error shown is the standard deviation per pixel (something that we'll get into in the next few slides), but you can see the difference anyways.

In the first example on the left, it seems at first that the relative error is better, because in very bright areas we won't see it anyways.

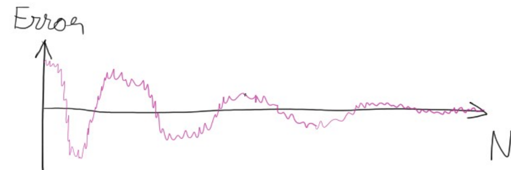
But on in the example on the right we can see, that the error can become very large in dark areas. In fact, it's needed to add an epsilon to the reference, so that it doesn't produce NaN.

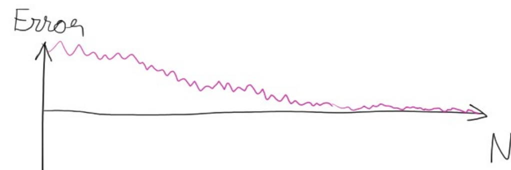
But enough of that, let's now look at...

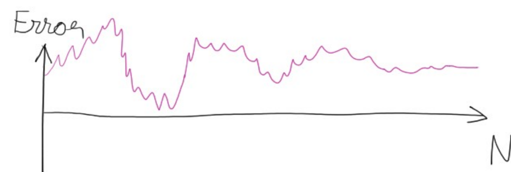
Measuring Error

Classes of estimators

- Unbiased
  $I = \mathrm{E}[\hat{I}]$
- Biased but consistent
  $I = \lim_{n \to \infty} (\hat{I}_n)$
- Biased
  $I = \mathrm{E}[\hat{I}] + \mathcal{E}$







… classes of estimators.

On the right you see plots of signed error (\hat{I} – I) against the number of samples on the x-axis. You won't see such plots in papers or websites, as other error measures are used most of the time. But they are good for visualising the concept here.

Unbiased estimators, like for instance Monte Carlo integration methods have the expectation of the estimator equal to to true value **point**. The error plot of a single estimate oscillates around zero and the amplitudes become smaller as N increases **point**.

Biased but consistent estimators at no point have the expectation of the true result. But the error becomes smaller with larger sample size. The estimator converges to the ground truth. The error can be negative as well, this here **point** is just an example.

And finally, there are biased algorithms, that never converge to the true solution. That usually happens when you trade noise for bias. You can get a nicer picture with less noise, that doesn't match the physics of light for instance. A typical example would be clamping the contribution (clamping fireflies as you saw them in the first part of the path tracing), limiting the recursion depth, or just not render certain effects, like for instance caustics.

Recap: Monte Carlo Integration

I, the integral of f(x) over the domain D is

$$I = \int_D \mathrm{f}(x)\,\mathrm{d}x = \int_D \frac{\mathrm{f}(x)}{\mathrm{p}(x)}\,\mathrm{p}(x)\,\mathrm{d}x = \mathrm{E}\left[\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right]_{\mathrm{p}(X)}.$$

But this is no estimator that we can use, all just theoretical.

$$\mathrm{E}[X] = \lim_{n\to\infty}\left(\frac{1}{n}\sum^{n} X\right)$$

=> Law of large numbers

$$I = \mathrm{E}\left[\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right]_{\mathrm{p}(X)} = \lim_{n\to\infty}\left(\frac{1}{n}\sum^{n}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right) \approx \frac{1}{N}\sum^{N}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}$$

We are focusing on unbiased Monte Carlo Integration, and at this point I would like to make a super brief recap.

I is the integral of f over a certain domain D. We can multiply f by p/p, which is one. And turn that into an expectation using a sampling strategy that creates samples according the the pdf p.

That is not an estimator as we have no procedure for estimation yet. But we can use the law of large numbers to turn that into an estimator.

The law of large numbers tells us, that the average of a ever increasing number of samples X is the expectation.

When we apply that to our stochastic variable f over p, we get the Monte Carlo integral estimator **point 1/n sum f/p

Super Short Probability Primer

- Variance (variability)

$$\mathrm{Var}(X) = \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{E}[X^2] - \mathrm{E}[X]^2$$

- Variance of the following expression of **uncorrelated** X

$$\mathrm{Var}\left(\sum^{N} aX\right) = a^2 \sum^{N} \mathrm{Var}(X)$$

- Central limit theorem (CLT)

$$\lim_{n\to\infty}\left(\frac{1}{n}\sum^{n} X\right) \to N(\mathrm{E}[X], \mathrm{Var}(X)/n)$$

Now we need another super short recap about probability – and the variance in particular.

The variance of a stochastic variable gives us a measure of how far spread out the samples will be. And you see two equivalent methods to compute it.

The next result tells us, how the variance of a constant times the sum or uncorrelated variables is computed. The constant a can be squared and moved out, and the variance of a sum is the same as the sum of variances. These equations are taken from wikipedia :)

Finally, the central limit theorem. It is usually written in a slightly different way, but this form is equivalent and more useful to us. The average of a ever increasing number of samples of a stochastic variable X tends towards a normal distribution with expectation and variance given **point**.

Measuring Error

Central limit theorem (CLT) $\lim_{n\to\infty}\left(\frac{1}{n}\sum^{n} X\right) \sim \mathrm{N}\left(\mathrm{E}[X], \mathrm{Var}(X)/n\right)$

Our estimator $I = \lim_{n\to\infty}\left(\frac{1}{n}\sum^{n}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right) \approx \hat{I} = \frac{1}{N}\sum^{N}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}$

The Error $\mathcal{E} = \hat{I} - I = \hat{I} - \mathrm{E}[\hat{I}]$

$$\mathrm{E}[\mathcal{E}] = \mathrm{E}\left[\hat{I} - \mathrm{E}[\hat{I}]\right] = \mathrm{E}[\hat{I}] - \mathrm{E}[\hat{I}] = 0$$

$$\mathrm{Var}(\mathcal{E}) = \mathrm{Var}(\hat{I}) - \mathrm{Var}(\mathrm{E}[\hat{I}]) = \mathrm{Var}(\hat{I})$$

and $\mathcal{E}$ also tends towards a normal distribution.

I've put the CLT and our estimator next to each other. We see that the estimator is pretty close to the CLT with the stochastic variable being f over p. That means, that, in the limit, our estimator "I hat", a stochastic variable, becomes normal distributed.

Now let's look at the error "I hat" minus "I". We know that the expectation of "I hat" is I, and so we can substitute it. Now we quickly see, that the expectation of the error is 0. That's logical, since it's unbiased.

We can also look at the variance of the error. We saw that the variance of a sum of stochastic variables is the sum of variances. So we replace that **point**, and the variance of the expectation (a constant), is 0. Therefore the variance of the error is the variance of the estimator.

And finally, without steps, the distribution of the error also tends towards a normal distribution. You should be able to check that without problems on your own.

Measuring Error

TU WIEN

Monte Carlo estimator

$$I \approx \hat{I} = \frac{1}{N}\sum^{N}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}$$

Variance of that estimator

$$\mathrm{Var}(\hat{I}) = \mathrm{Var}\left(\frac{1}{N}\sum^{N}\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right) = \frac{1}{N^2}\sum^{N}\mathrm{Var}\left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right)$$

OK, the variance of the estimator is important. What can we find out about it?

On top again the estimator (denoted with a hat). And, as said, the estimator is a stochastic process, therefore the result is a stochastic variable again.

We can compute the variance of that estimator. The result will tell us, how far spread out the estimates will be. That means, if we run the estimator several times, how far spread out the results will be – at the same time, how far spread out the error will be.

Using the equation from the probability primer before, we can pull out the 1/N (it'll become 1/N^2), and make it a sum of variances.

Measuring Error

Monte Carlo estimator

$$I \approx \hat{I} = \frac{1}{N} \sum^{N} \frac{\mathrm{f}(X)}{\mathrm{p}(X)}$$

Variance of that estimator

$$\mathrm{Var}(\hat{I}) = \frac{1}{N^2} \sum^{N} \mathrm{Var}\left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right) = \frac{1}{N^2} N \,\mathrm{Var}\left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right)$$

$$= \frac{1}{N} \mathrm{Var}\left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right)$$

Variance of the estimator becomes smaller with a larger N

Now, the Variance of f/p is a constant (a theoretical value, we don’t know it, but we know that it exists). And a sum of N values is N times the value.

So we can replace the sum by a multiplication with N.

And two of the N cancel out.

So the result here is, that the variance of the estimator (and the error) is 1/N times the variance of a single sample.

Neat!

Measuring Error

- The distribution of error of a Monte Carlo estimator tends towards a normal distribution as N increases.
- Its expectation is 0 (yes, because it's unbiased).
- The variance of the estimator is also the variance of the error.
- The variance is in strict relation with the number of samples: double N → half variance

Variance, variance, what do we know about it?

To summarise..

The distribution of error of a Monte Carlo estimator tends towards a normal distribution as N increases.

Its expectation is 0 (yes, because it's unbiased).

The variance of the estimator is also the variance of the error.

And finally, the variance is in strict relation with the number of samples, double N will result in half the variance.

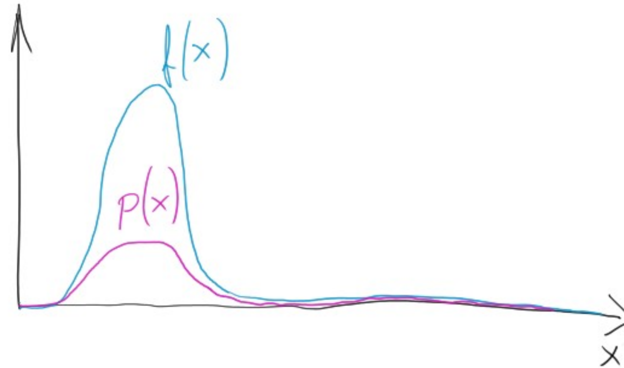
Variance, variance, what do we know about it? Can we compute it somehow?

Measuring Error

Importance sampling

$$I \approx \hat{I} = \frac{1}{N} \sum^{N} \frac{\mathrm{f}(X)}{\mathrm{p}(X)}$$

$$\mathrm{Var}\left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)}\right)$$



We heard about importance sampling before, and we learned that it can be used to reduce the variance / error. It would be even better with MIS.

Can we compute the variance out of it?

Well, no. Or at least I'm not aware of a method to do that. It's kinda a dead end.

One of our researchers is working on a method to use the information gathered by the path samples to get a better estimate of the variance. But that is only an improvement to what we'll see now..

Measuring Error

We can measure sample variance easily, and derive an estimator for the estimator variance!

$$\mathrm{Var}(\hat{I}) \approx \frac{1}{N^2} \sum^{N} \left(\frac{\mathrm{f}(X)}{\mathrm{p}(X)} - \hat{I} \right)^2$$

We don‘t even need a reference solution!

It is even possible to do that „online“, which means that we do not need to compute the average of all samples before we compute the variance. Search for the Welford algorithm!

When we want to do that for path tracing, we would compute the per-pixel sample variance.

We can measure the sample variance. It’s simple, you have a bunch of values (the f over p), and compute their variance.

In order to get an estimate of the estimator variance, we need to divide by N (so division by N squared in total).

This process does not even require a reference solution!

---

read the rest from the slide

Time for some examples..

Measuring Error

Bathroom scene

Standard deviation per pixel

source: own

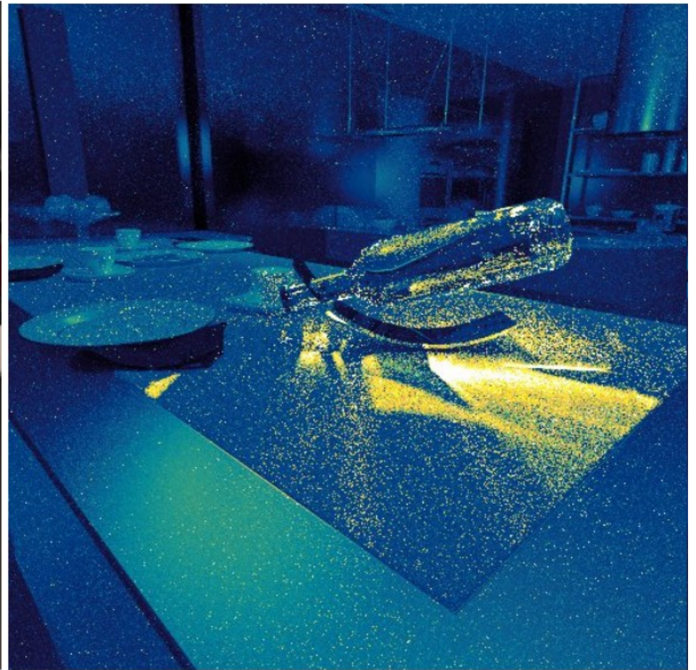
On the left you see a rendering of a bathroom scene.

On the right there is the standard deviation per pixel.

In my opinion standard deviation is preferable because of the linear scaling (variance has a quadratic scaling).

Measuring Error

Bottle scene



Standard deviation per pixel

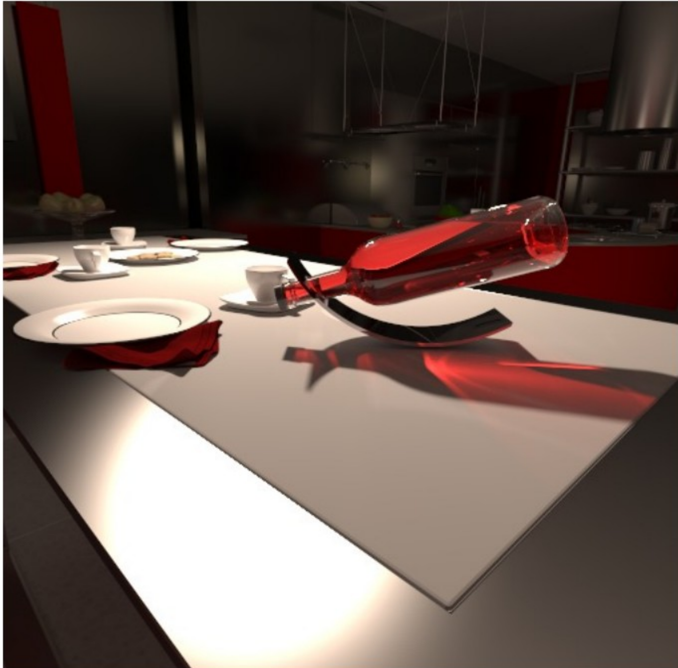
source: own

Here we see another example.

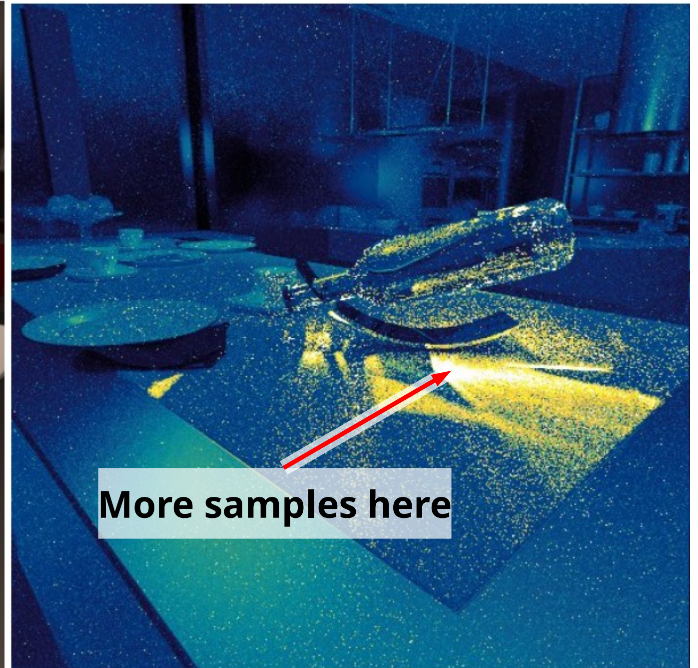
And we can clearly identify that the caustic is problematic for the path tracing algorithm – because the standard deviation (or variance) is high.

Alright, what can we do with that information? ...

Measuring Error



Bottle scene



Standard deviation per pixel

source: own

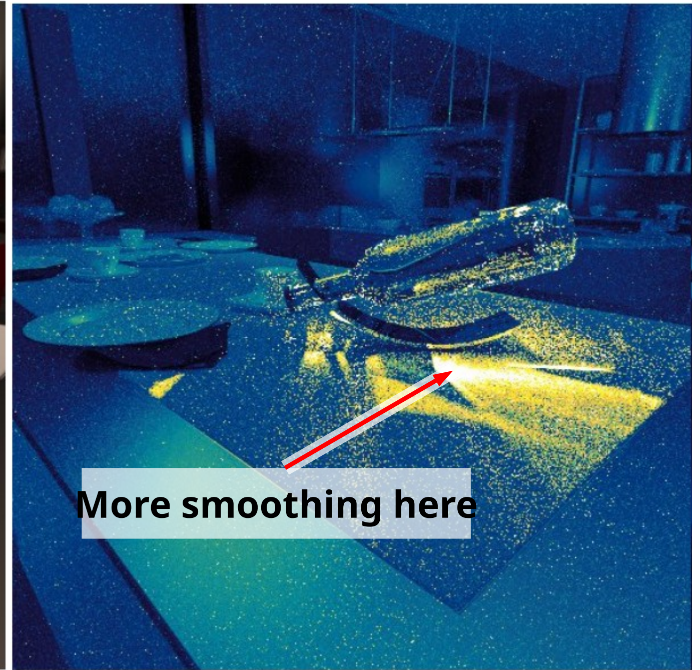
One thing: we know that the error in that area is large. But we also know that doubling the number of samples reduces the estimator variance by half.

Erm, so through more samples at these problematic areas!

This is called adaptive sampling. And there are many methods to do that in intelligent ways, but that would be too much for this lecture. Look for papers on your own :)

Measuring Error

Bottle scene



Standard deviation per pixel

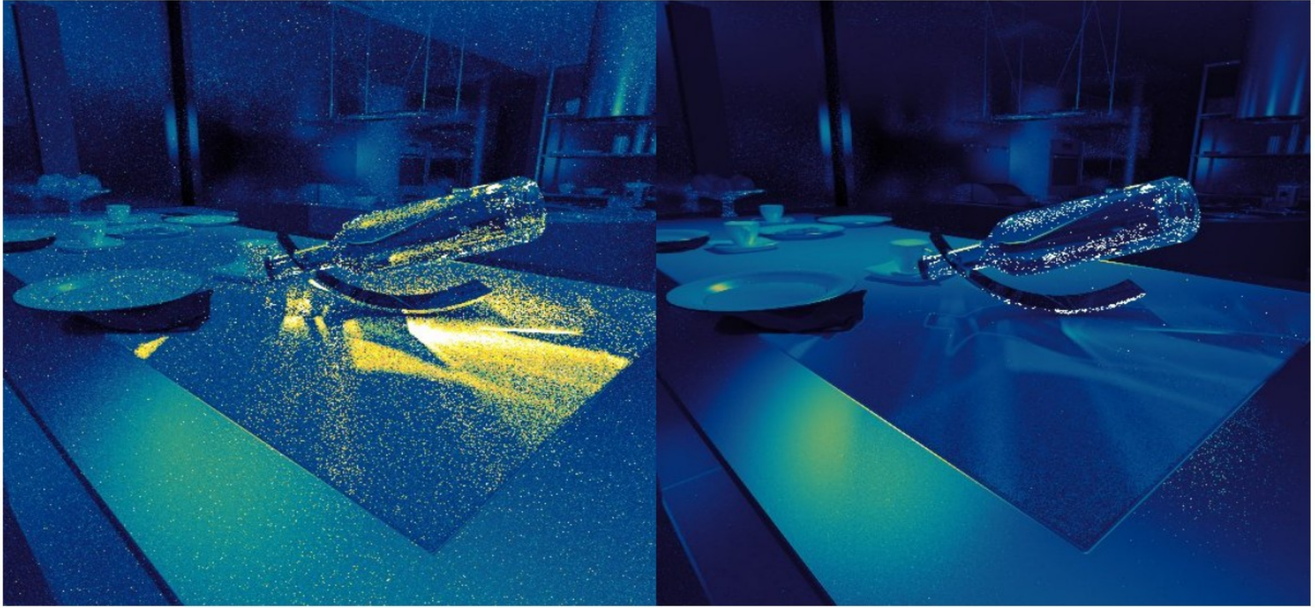
source: own

Another method to combat noise in renderings is smoothing. Adaptive smoothing is used often, which means that edges are not blurred for instance. Most noise reduction methods are biased. however, a little bit of bias is often better than a noisy image.

If we know that there is a lot of error in a certain part – in other words objectionable noise – we can apply more smoothing. This would prevent unnecessary smoothing of diffuse textures for instance.

And finally...

Measuring Error



(e) PT (f) BDPT

source: own

We can compare different algorithms.

On the left path tracing, it contains a lot of error in the caustic.

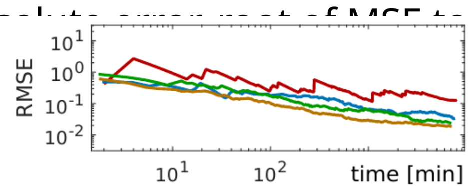
Bidirectional path tracing on the right shows less error in the caustic, but it looks like there is more error or noise in the refractions inside the bottle.

So generally, if you develop a new algorithm, it is good to know its strengths and weaknesses, and this is a good method to do that. In my opinion better than comparing ground truth to real images like we saw in the intro.

Measuring Error

What if we want a metric for the whole image (e.g., for ranking algorithms):

- Mean square error (MSE), per pixel mean of (rendering - reference)^2
  - Very popular in literature
  - many variation (relative to brightness, mean absolute error, root of MSE to be linear..)
  - However, sensitive to outliers
- I'm advocating to use root mean variance and also report the standard deviation of it.



(a) RMSE over time

Finally (now for real), we often want to rank algorithms. E.g., say that algorithm a is better than b according to some metric.

Literature often uses Mean square error to do that. That is a per pixel mean of the squared error between a rendering and a ground truth.

There are many variations of that, for instance computing the error relative to the brightness of the reference, or computing mean absolute error, which weighs outliers less, or computing root mean square error, which is linear like standard deviation.

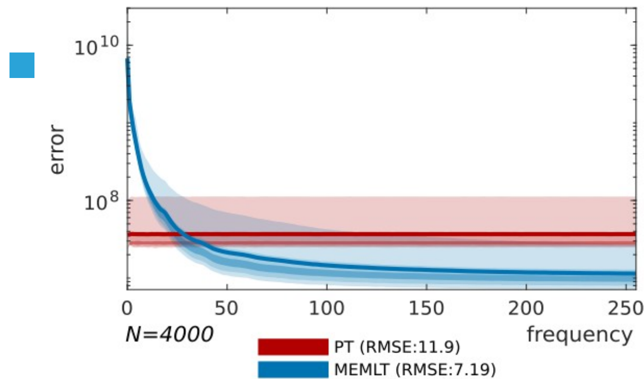
All these methods are sensitive to outliers (fireflies in the case of path tracing). **Point** here you see the RMSE error plotted over rendering time. The jumps are outliers, and so it would be hard to choose a fair rendering time for comparison.

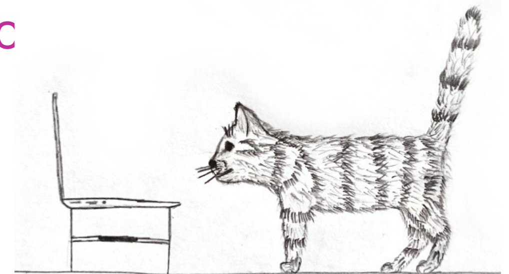
So I'm advocating to use the root of mean variance, where the variance is estimated per pixel.

Measuring Error

Reading

- Celarek (2017), Quantifying the Convergence of Light-Transport Algorithms
- ight Transpc

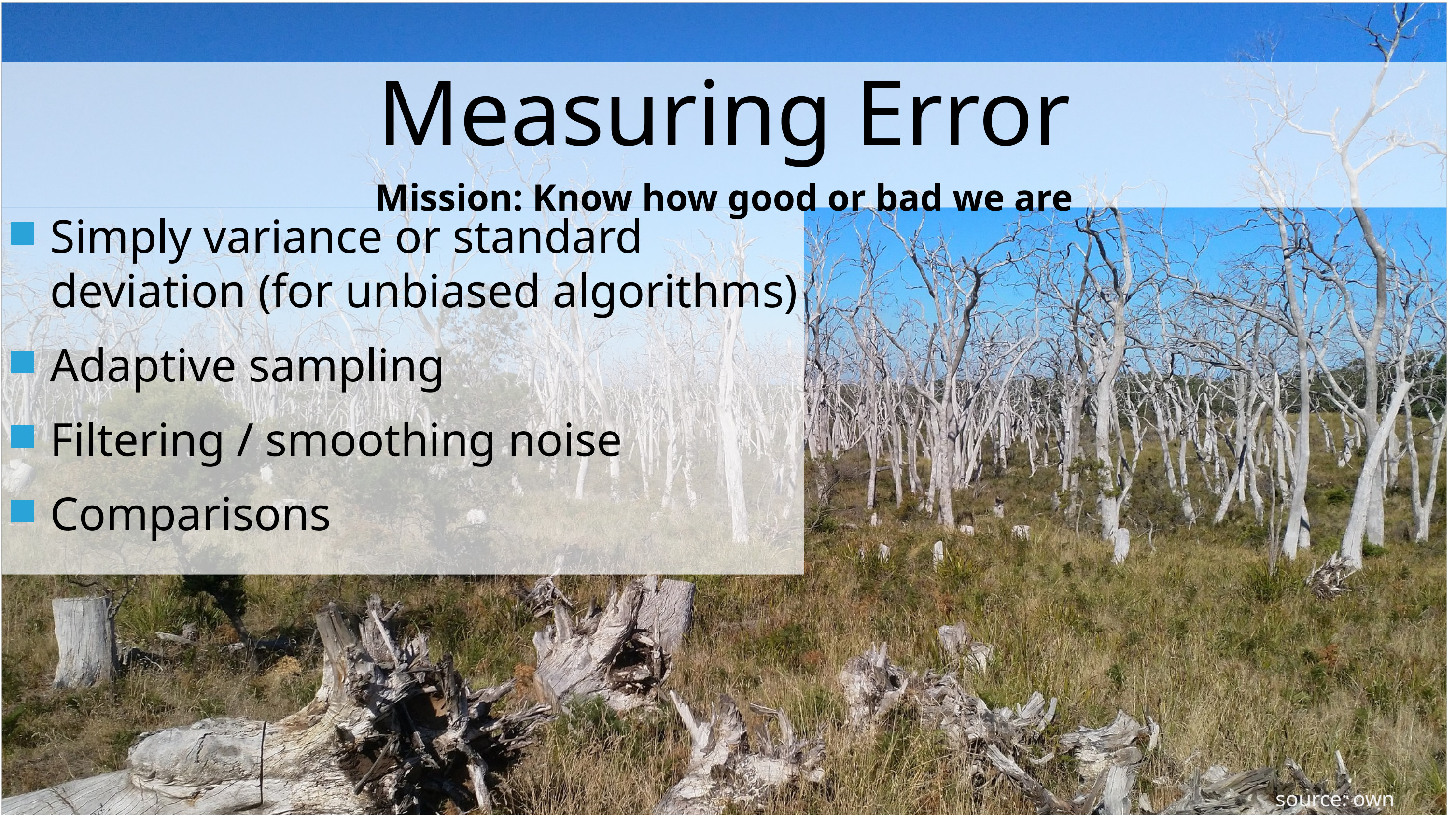




Here are some reading links

The first one is a diploma thesis and the second one a paper.

Both roughly cover the same methods, which also includes a plot of error over frequencies, but the first one being more extensive.



And here comes the summary slide:

We have our mission accomplished, we know how to judge the quality of our rendering work.

For unbiased algorithms like path tracing that is simply measuring the variance or standard deviation.

We can use that knowledge to through more samples at problematic areas, smooth out noise in that areas and for comparisons between algorithms and sampling strategies.

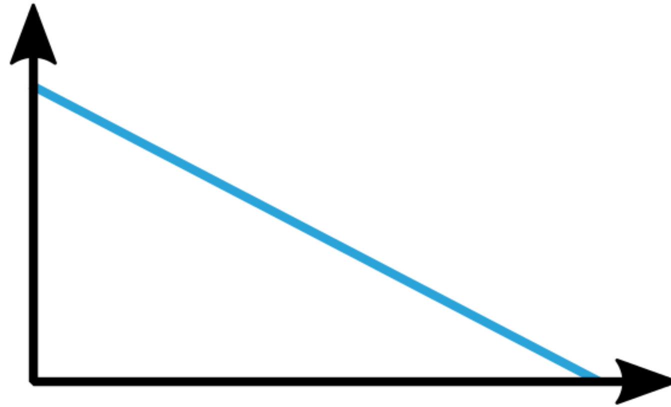
I don’t remember where I took the picture, probably somewhere in France. It’s a facade, looks a bit like cosine hemisphere samples projected on the tangent plane ;)

–

We are now making a jump from looking at the result of rendering, back to the basic concept of integration and it’s implementation.

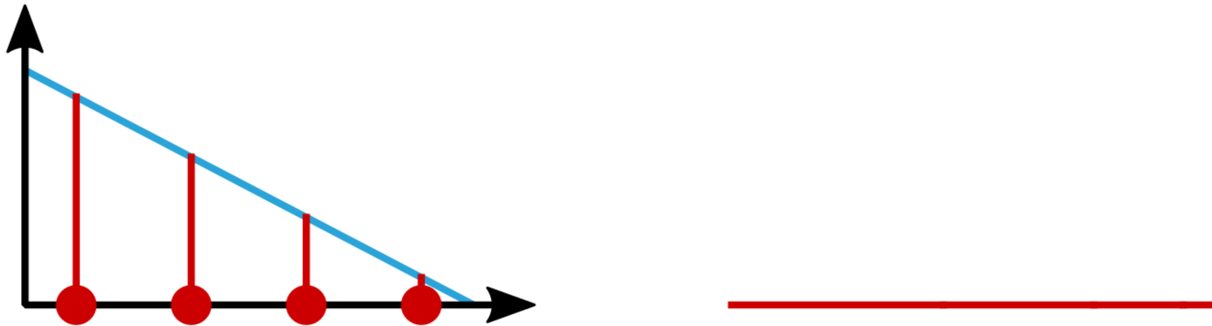
Sampling patterns are a detail of Monte Carlo integration, so this is an addendum to the Monte Carlo and sampling lectures.

Bad random numbers cause noise



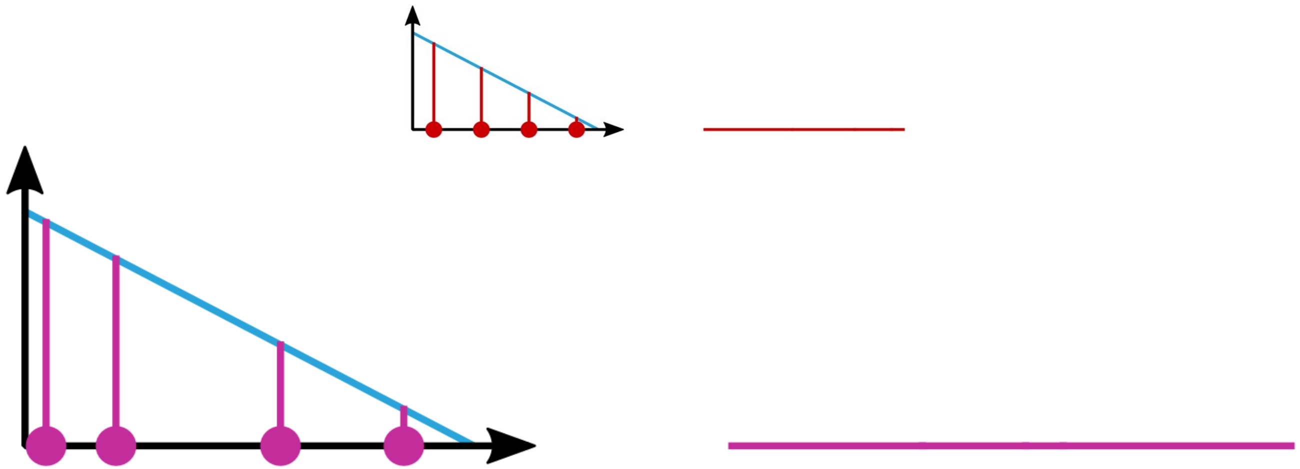
First I'll explain the problem using a 1d example.

Bad random numbers cause noise



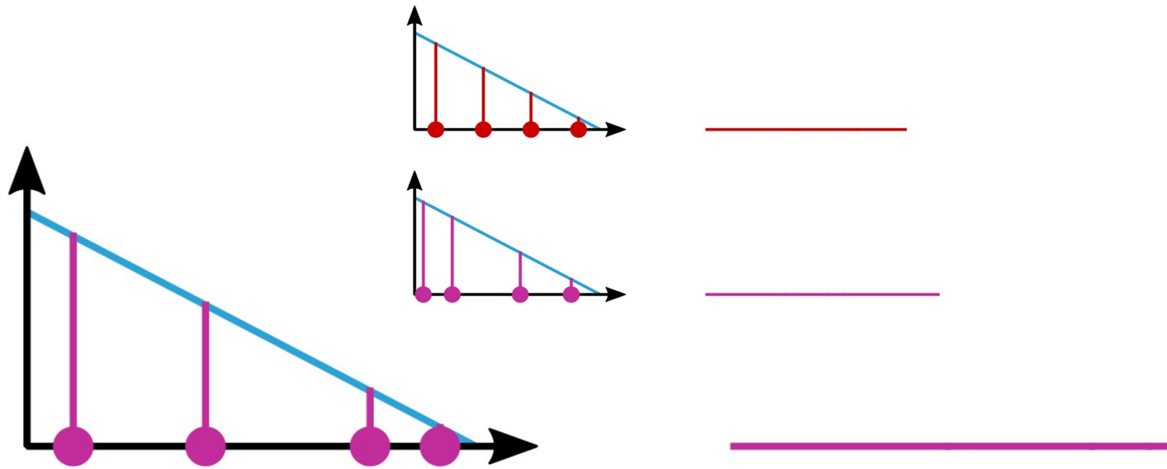
Let’s assume equidistant samples, which is basically a trapezoidal integration rule. On the right side you see the sum of the sample values. Dividing it by the number of samples would give us the integral of the blue function.

Bad random numbers cause noise



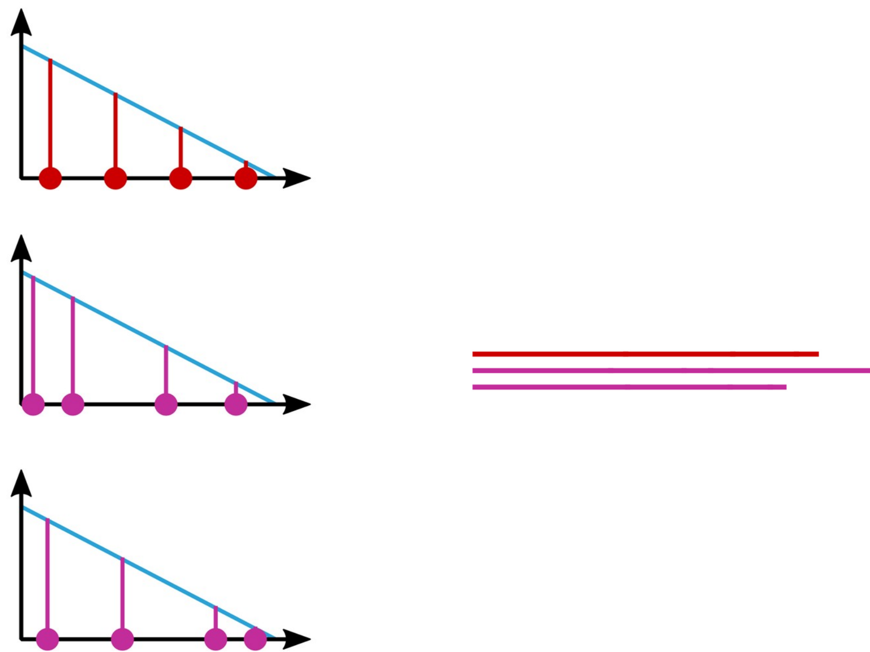
But since we are doing Monte Carlo, the samples will not be equidistant. Sometimes they will cluster in one area,

Bad random numbers cause noise



And sometimes in another.

Bad random numbers cause noise



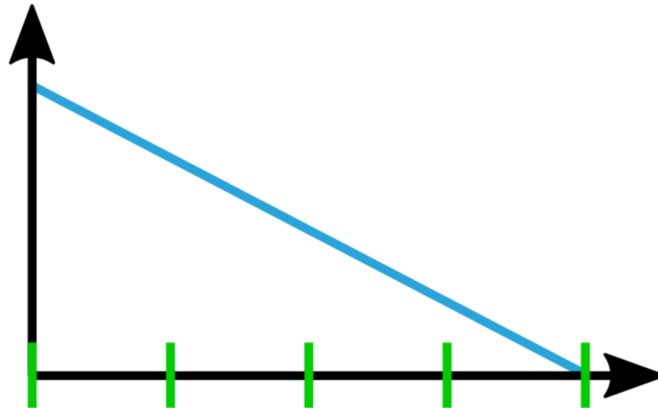
Which gives different results.

This is variance that can’t really be explained by looking at the probability densities. I mean yes, you introduce variance by the fact that there is a PDF, and sampling and everything, but that variance feels unnecessarily excessive.

We want randomness, but, that is too much..

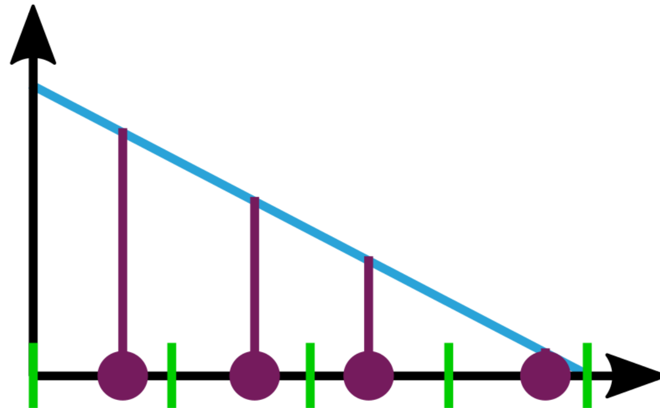
Obviously, I wouldn’t bring this up without a solution^^

Bad random numbers cause noise



Let’s divide the integration domain into N equidistant areas and take one random sample in each

Bad random numbers cause noise

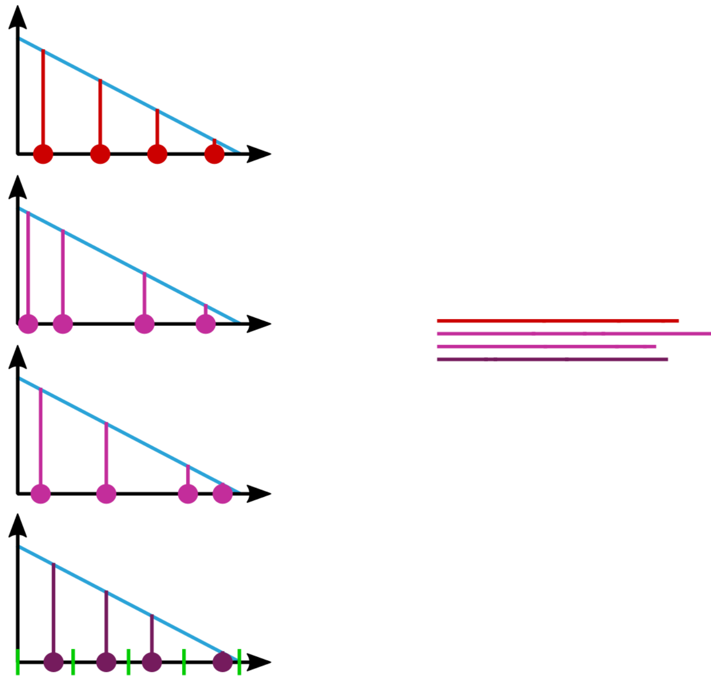


Like here..

This is called stratification, or stratified samples.

The samples are still random, but more regular!

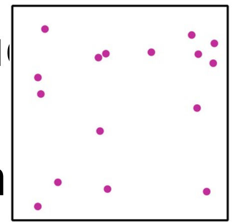
Bad random numbers cause noise

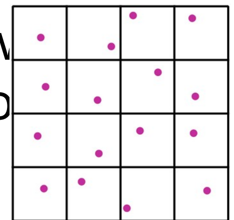


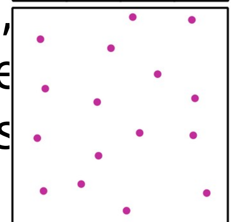
When we compare the results, the stratified variant has a smaller error.

Stratified sampling

- With random uniform sampling, we can get unlu
  e.g. many samples clump in one area.
- However, we can subdivide the integration doma
  using a grid. Each grid cell is called a *stratum.*
  Then, randomly select a stratum from the ones w
  lowest number of samples, and pick a new rando
  sample in it.
- We are working in the primary sampling domain,
  before importance sampling. Therefore this gene
  more regular samples even for things like a gloss
  BSDF.







Here we have another example, but in 2d. On the top it's random sampling and you can see at the bottom that the samples are more regular, when we use stratification.

Well, well, but let's think whether this can scale to the 4, 6, 10, and more dimensions of a path-space integral…

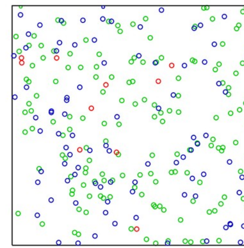
The curse of dimensionality kicks in again – we can't have one sample per grid cell, same as with the trapezoidal rule.

However, there is a simple solution. From the cells with least number of samples, pick one randomly and and sample in it. That way the cells stay relatively balanced (the maximum difference of samples per cell is 1).

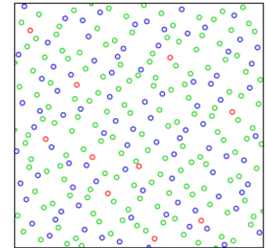
This approach should be implemented directly in the random number generator, before using these numbers in BSDF samples etc., so before any warping is applied. The benefits translate to the the warped distribution as well. The domain before warping is called the primary sample domain.

Low-Discrepancy Series

- Replace the built-in RNG with a sample generation algorithm that sacrifices randomness for good spatial distribution
- In other words: instead of stratification, use an „*hacked*“ random number generator, that produces stratified samples in the first place
  - Numbers are not random, it's a simp called *Quasi Monte Carlo*
  - Can cause artifacts if it's not done ri
  - E.g., Halton Sequence



Default RNG



2,3 Halton Sequence

Source: Wikipedia, User Jheald (CC-BY-3.0)

Low-discrepancy series are a way to achieve uniform sampling without using stratification.

Basically, the random number generator is replaced with an algorithm that generates samples according to a mathematical formula, guaranteeing a relatively uniform distribution at all sample counts.

We kinda hack the random number generator to produce stratified samples ;)

The numbers are not random, hence Monte Carlo turns into Quasi Monte Carlo.

The Halton sequence in the picture is one example for such an algorithm. The whole thing is a bit tricky to get right, best follow the course book, we have references at the end of this section.

Quasi Monte Carlo

- Often applied only to the first few dimensions (e.g. direction of the first bounce, surface sampling of a light)
  - In fact, having the same number of samples for each pixel is already stratified sampling. Could choose position in the pixel plane randomly.
  - Pointless in higher dimensions, because it's cursed
  - Most of the error often in high dimensions (e.g. caustics, deep refractions in glass).
- Convergence rate for low dimensions can be actually better than O(1/N)
  [Pilleboue, Singh et al.: Variance Analysis for Monte Carlo Integration]
  - Simpson's and other higher order quadrature rules also have a better convergence rate
  - They also don't work for higher dimensions

Quasi Monte Carlo is often applied only to the first few dimensions of the integral, for example the direction of the first bounce, surface samples of a light etc.

In fact, using the same number of samples per pixel is already stratification!

Any of these methods become pointless in higher dimensions, because sampling will be very sparse.

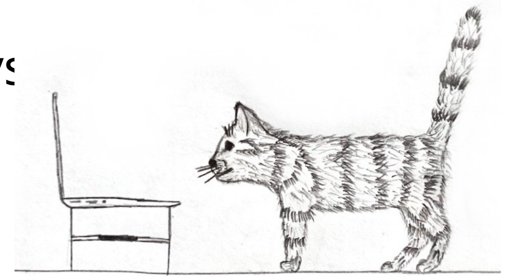
Worse, in many scenes most of the error is in higher dimensions, for example caustics, deep refractions in glass and so on. However, it's still used, because it's cheep, and diffuse surfaces are cleaner.

It's also a bit funny. Previously we learned, that the convergence rate is 1/N. Using sample patterns can be even better than 1/N, that is, error in low dimensions converges asymptotically quicker than in high dimensions! That's similar to Simpson's and other higher order quadrature rules, and they also don't work for higher dimensions.

Quasi Monte Carlo

Reading

- Pharr, et al., Physically Based Rendering (Chapter 13.8 and 7.4)
- SIGGRAPH 2012 Course: Advanced (Quasi-) Monte Carlo Methods for Image Synthesis
- Pilleboue, Singh et al.: Variance Analys
  Integration



Here are some reading links

Check our course book again. There is also a SIGGRAPH course. And the paper by Pilleboue, singh et al. shows a way to analyse sampling patterns, and gives exact numbers on convergence.

That concludes our section about sampling patterns.
We learned that having regular samples reduces error in low dimensions.

That's it for today

We covered a whole bunch of topics :)

I wanted to include that image in my diploma thesis, a glass of water with an oat drink, which shows participating media, and a nice caustic.

Thanks for your attention...