FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Kubíček

An Efficient Load-balancing Image Sampler for Path Tracing

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: doc. Alexander Wilkie, Dr.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedicated to my mother who has supported me on my journey in higher learning, my loving girlfriend who has stood by me when I was feeling down, all my friends who have helped me along, and to my supervisor who has shared his experience with me and without whom this thesis would not be half as good as it is.

Title: An Efficient Load-balancing Image Sampler for Path Tracing

Author: Tomáš Kubíček

Department: Department of Software and Computer Science Education

Supervisor: doc. Alexander Wilkie, Dr., Department of Software and Computer Science Education

Abstract: In this thesis, we go through the process of improving a research renderer called ART. We focus primarily on the image sampler. We analyse the shortcomings of the stochastic sampler that was present before this thesis was written.

Those being the huge memory consumption, lack of real-time visualization, and a limited communication channel between the renderer and the application that is supposed to control it.

We then design two samplers that could serve as a replacement. We choose the one which seems to be better suited for the task.

We then run a series of tests which serve to finetune parameters of the new sampler and also serve as a comparison between the new and the old sampler. By using the data from the tests we show that we have reduced the memory foot-print while maintaining the speed at mostly the same level as before while enabling working with greater number of cores.

Keywords: path tracing, image sampling, parallel programming, computer graphics, rendering

Contents

Introduction

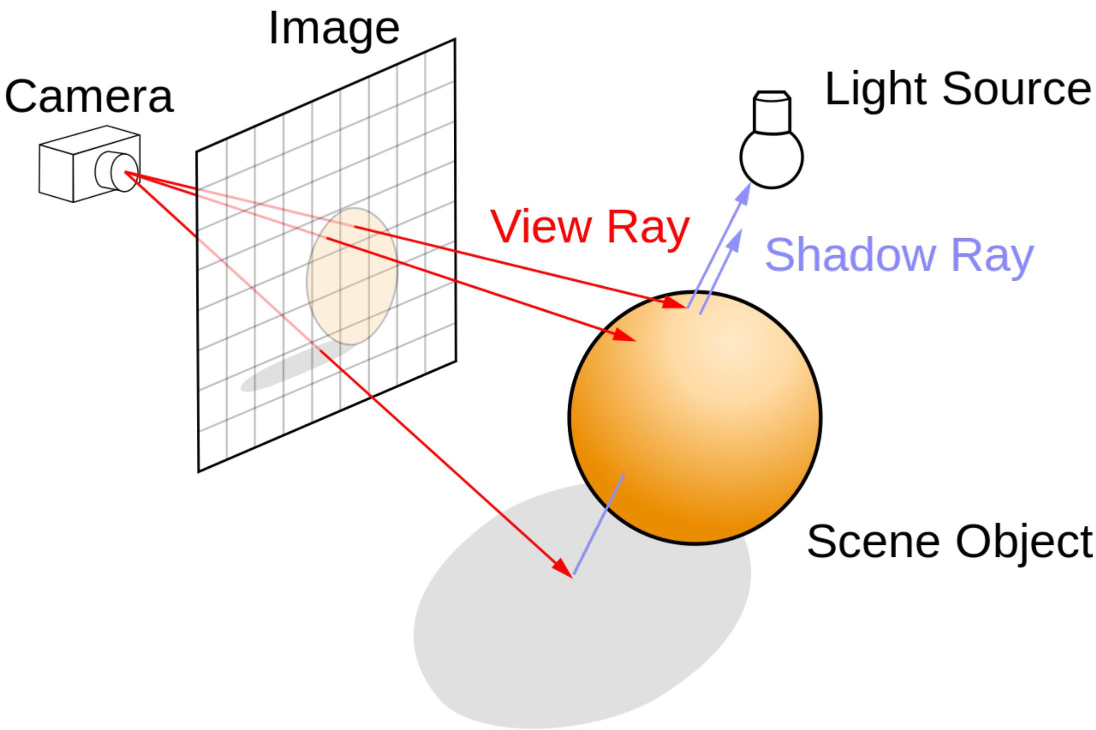


Figure 1: A motivational picture about raytracing. Taken from wikipedia article about Ray Tracing [1].

Rendering is a process in which a program, usually called a renderer or a rendering engine, produces a representation of a scene from its description. The rendered representation is usually an image. It could be a regular RGB image; however, we will be more interested in spectral images, which will be described shortly. Such a scene description can contain many things e.g., models and their state, lights and cameras. So, the goal of the whole effort is to compute what goes into the pixels of the result image so that it shows exactly what the scene would look like.

Rendering can be done in any number of ways, and a full discussion of it is beyond the scope of this thesis. However, a good book about this topic, raytracing in particular, would be Physically Based Rendering: From Theory to Implementation by Pharr, Jakob, and Humphreys [2]. In this introduction we will talk only about the basics of raytracing 1 and details connected to parallelization and image sampling.

Brief Overview of Image Synthesis

Our goal is to synthesize an image of the scene. Let us start by defining such a scene, and then we will iteratively describe an algorithm, which should give us an intuitive understanding of important raytracing concepts.

Imagine us having a scene 2 which contains some emissive object. In a normal world it would be, for example, a very hot piece of wire in a lightbulb which would

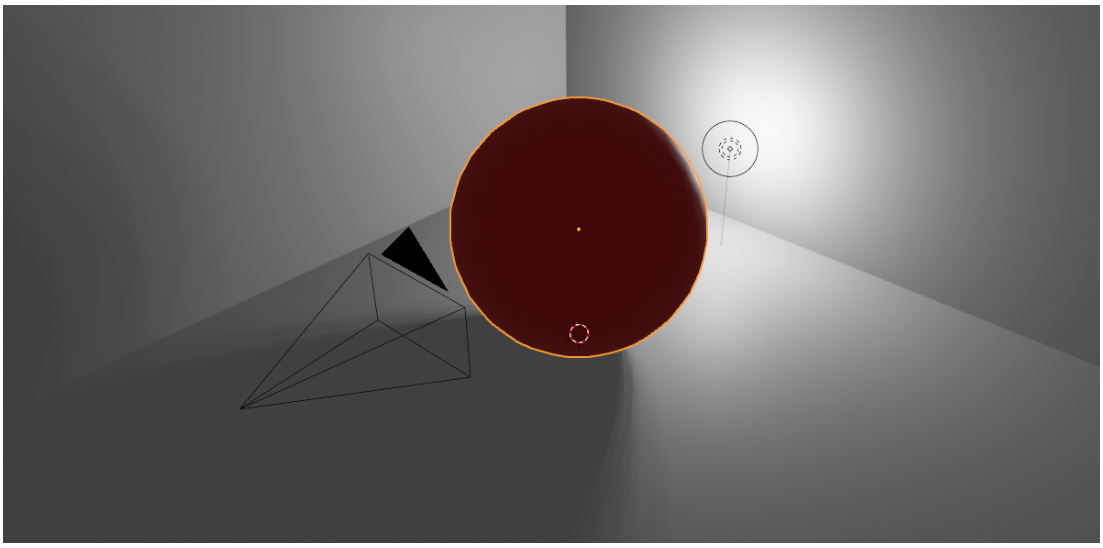


Figure 2: A very simple representation of the example scene done in Blender.

radiate light, but we will simplify this by only considering a uniform pointlight, i.e., a light which shines in every direction from a single point in space. For the sake of simplicity, we say that it has a white colour. We will place this light behind a red ball. In front of this ball, we will place a camera. It, of course, will not be a real camera, but rather a representation of the camera made up of the plane where the pixels are located. All this enclosed in a room with white walls.

So, what exactly are we computing at each pixel? We are computing the radiance. That is, the amount of light from the scene that passes through the pixel. If we somehow computed this per pixel value, we would end up with an image that could be turned into a black-and-white image.

If we would like a colour image instead of a black-and-white image, we would have to extend what exactly we are computing at each pixel.

This brings us to the spectra of light. In elementary physics we learn that light comes in many forms. It has discrete wavelengths that our eyes or cameras interpret as different colours. So, if we want a coloured image, we have to somehow represent this phenomenon. For us, the most important band of light is the so-called visible light going from 380 nm to 700 nm. We can split this band into buckets. How many are there and if they are uniform is up to us.

In general, having more fine-grained buckets brings greater fidelity at the cost of increased memory consumption, and uniformly spaced buckets are easier to work with. It might be a good thing to consider covering even an increased range of light than just visible, ultra-violet, and infrared light, because even though these bands of light are not visible, there are modellable interactions of these bands which produce visible light. In the figure 3 we can see how such spectra could be partitioned.

We have therefore moved from computing the radiance to computing the spectral radiance at each pixel. Before we even begin to talk about how we are going to compute this spectral radiance we should look at how we would convert the spectral radiance into an RGB image which could then be shown on our displays.

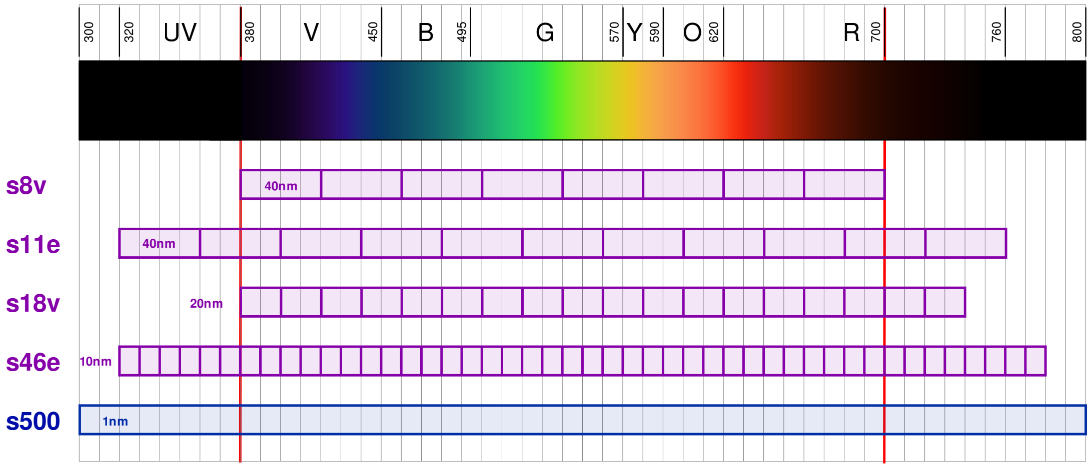


Figure 3: Examples of spectral representations, taken from the ART handbook [3]. Unlike RGB values, which are completely defined by three float numbers, spectral quantities are theoretically continuous, and require some form of sampling to be represented in memory. How many samples one needs depends on the purpose of the rendering software: for visual accuracy (that is, to have enough samples to satisfy the human visual system) around 15-20 samples are usually sufficient. For hyperspectral applications (e.g. for thermal simulations), or for specific scientific problems (e.g. astronomical radiative transfer simulations), substantially denser spectral sampling can be necessary.

Conversion of spectral data to RGB

There has to be a transformation from the spectral data to RGB. But to construct such a transformation, additional information about the camera is needed.

Cameras and eyes, for that matter, have multiple types of receptors. These receptors are sensitive to each wavelength differently. In the figure 4 we can see spectral sensitivity of the human eye receptors to different wavelengths. From this integrating (or in our case discretely summing) the sensitivity multiplied by the intensity of that wavelength produces the final strength of that signal from that one receptor. In practice, this is not directly useful, though, as this only gives one the raw intensity of the sensor cells. For signals that carry on into our brain, one also needs to take the neural processing that happens in the retina into account, which is why spectral to colour conversions use direct, cumulative models like those described in the subsequent paragraphs.

In practice, this is done in one of two ways. If we are trying to simulate a concrete camera for the purposes of matching that camera in a scene, we have to measure such a response curve of the camera. Then our RGB values should be exactly the same as if the picture were taken with that camera.

If we, however, just want to transform our image to be viewed on a display, then it is done by first transfering it to a coordinate system called CIE XYZ. This is done via sensitivy curves shown in the figure 5. The transformation of this coordinate system to the colour space of the displays is then a simple multiplication by a 3×3 matrix.

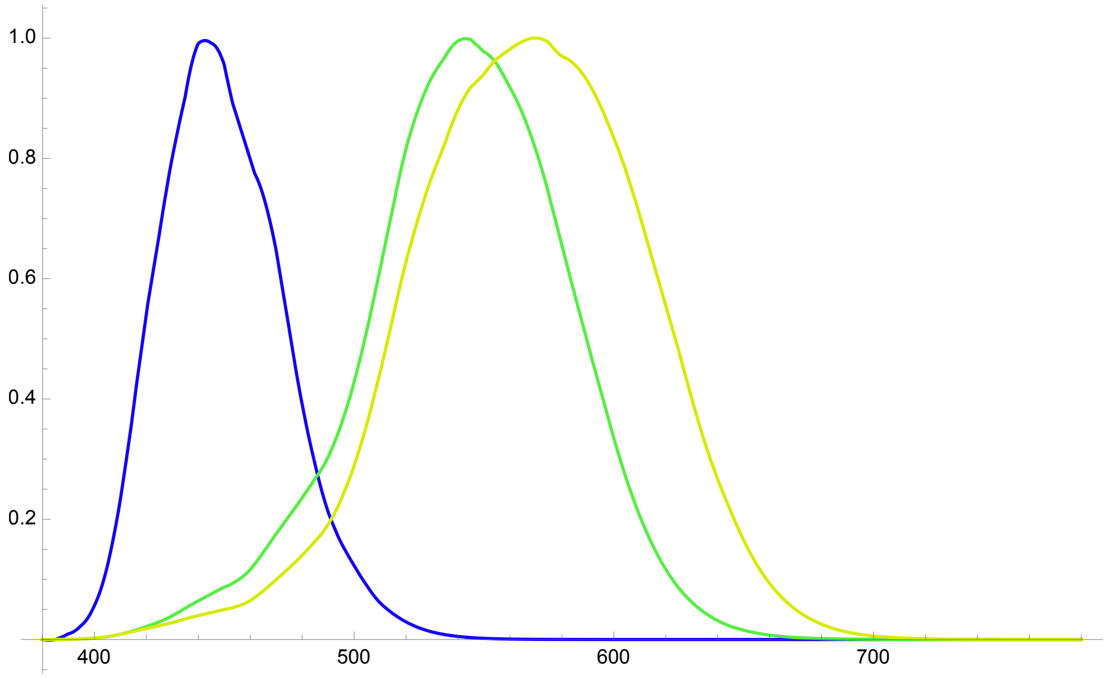


Figure 4: Response curves of the receptors of the human eye taken from [4]. Note that our perception is not RGB, but that our ability to perceive red rather results from the subtraction of the yellow-green and green channels we actually have in our eyes. These sensitivity curves are not directly used in practice, as described in the text, for transforming spectral data.

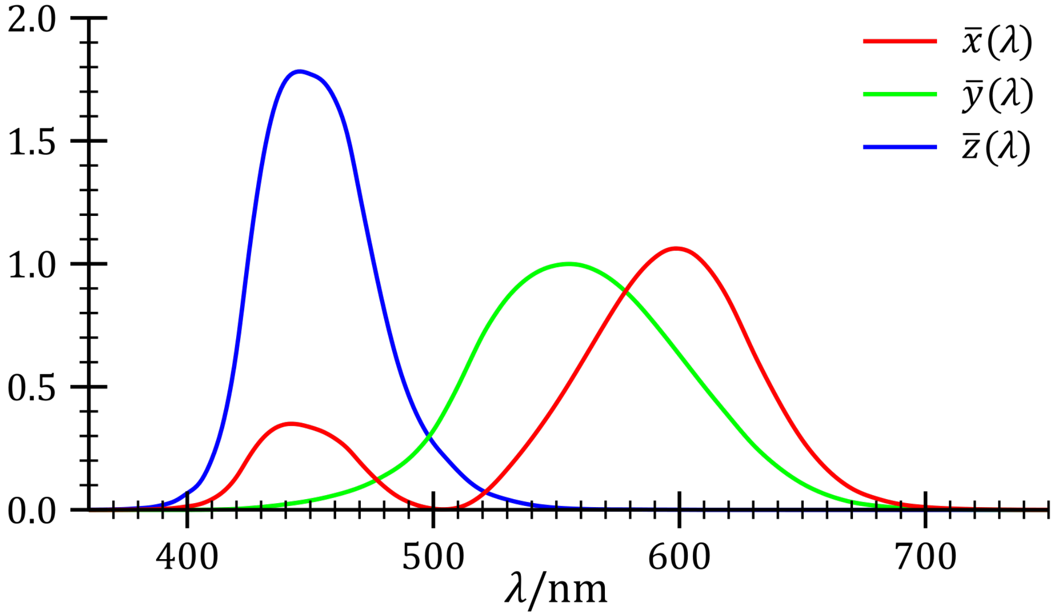


Figure 5: Response curves of the standart observer used in transforming from spectral data to CIE XYZ taken from [5].

Tonemapping

Figure 6: Tonemapped HDR image taken from [6]. This image shows a lot of detail in both the dark and lit parts of the image.

When we have transformed the spectral image to RGB we are faced with one more challenge. In the real world there is really no upper limit on how much light can shine on the pixel and therefore our values can span from miniscule amounts of light to very large values, but our displays have an upper limit of how much light they can produce at a given pixel.

We would then have to map the values obtained by rendering them into the upper and lower limits of light that can be emitted. But oftentimes a trivial mapping does not give us the desired effect, and more care has to be put into selecting or constructing an appropriate tonemapping operator. We can see a tonemapped image in the figure 6.

Simple integrator

We will now come up with a method for computing the spectral radiance. In general, such methods are called integrators, and we will construct one such method.

We are interested in how much spectral radiance is there at a given pixel. We could estimate this quantity by tracing the path packets of light with some specific wavelength take in our scene and modeling their interactions with objects. We would then average the paths that reach the pixel we are estimating.

To do such a tracing, we will need to describe a mathematical model of computing these paths.

Ray casting

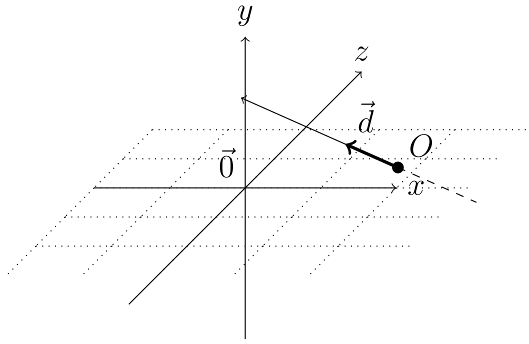


Figure 7: An example of a ray. The dashed line is the opposite ray.

Let us start by defining the ray R. It is an affine half-space of dimension 1 in $\mathbb{R}^3$ that can be easily parametrized by its starting point O and a free vector $\vec{d}$ as $O + t \cdot \vec{d}$ for $t \in \mathbb{R}_0^+$. An example of how a ray can look can be seen in 7.

Ray casting is then the act of finding the intersections of a ray and objects in a scene. This is a computationally very intensive task and any renderer utilizing this technique will spend most of the time doing ray casting. Exactly how these intersections are found in a real production system is a complex topic, but we will illustrate the concept with a simple example.

Intersection example

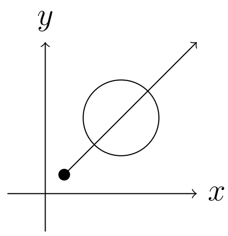
Let us move for the sake of brevity from $\mathbb{R}^3$ to $\mathbb{R}^2$ but the described process would work exactly the same for a similar task in $\mathbb{R}^3$. The definition of a ray is then very similar to the original with the sole exception of it being a sub-halfspace of $\mathbb{R}^2$. Our task will then be to find intersections of a ray and a circle. Let us remember that a circle is a set of points in $\mathbb{R}^2$ satisfying this implicit equation:

$$(x - S_x)^2 + (y - S_y)^2 = R^2$$

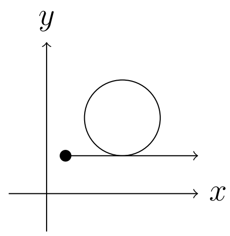
where the constants S_x and S_y represent the components of the centre of the circle and R represents the radius. This equation is equivalent to saying that points on the circle are those that are at exactly the distance R from the centre of the circle.

From the parametrization $O + t \cdot \vec{d}$ of the ray we can substitute its components into the implicit equation of the circle and find the points which satisfy both. The computation looks as follows:

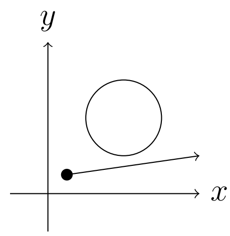
$$\begin{aligned}
(x - S_x)^2 + (y - S_y)^2 &= R^2 \\
\left(O_x + t \cdot \vec{d}_x - S_x\right)^2 + \left(O_y + t \cdot \vec{d}_y - S_y\right)^2 &= R^2 \\
2 \cdot t \cdot O_x \cdot \vec{d}_x + t^2 \vec{d}_x^2 - 2 \cdot t \cdot S_x \cdot \vec{d}_x + O_x^2 - 2 \cdot O_x \cdot S_x + S_x^2 & \\
+2 \cdot t \cdot O_y \cdot \vec{d}_y + t^2 \vec{d}_y^2 - 2 \cdot t \cdot S_y \cdot \vec{d}_y + O_y^2 - 2 \cdot O_y \cdot S_y + S_y^2 &= R^2
\end{aligned}$$



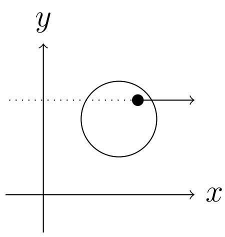
(a) A ray with two intersection points. There would be two distinct and non-negative values of t.



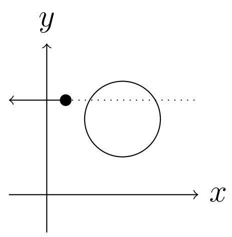
(b) A ray with one intersection point. There would be exacly one value of t and it would be non-negative. Also notice that the ray is tangent to the circle.



(c) A ray with no intersection point. Discriminant would be negative.



(d) A ray starting inside the circle. There would be two values for t but only one would be non-negative.



(e) A ray which has no intersection but the opposite ray has two. There would be two values of t but both would be negative.

Figure 8: All configurations of a circle and a ray.

$$
\begin{aligned}
t^2 \cdot \left(\vec{d}_x^2 + \vec{d}_y^2\right) \\
+t \cdot 2 \cdot \left[\vec{d}_x \cdot (O_x - S_x) + \vec{d}_y \cdot (O_y - S_y)\right] \\
+\left[O_x^2 + S_x^2 + O_y^2 + S_y^2 - R^2 - 2 \cdot (O_x \cdot S_x + O_y \cdot S_y)\right] &= 0 \\
t^2 \cdot \|\vec{d}\|^2 + t \cdot 2 \cdot \langle \vec{d}, O - S\rangle + \left(\|O - S\|^2 - R^2\right) &= 0.
\end{aligned}
$$

We have suggestively ended our equation in a form that should remind us of the quadratic formula. We can now compute the discriminant as follows:

$$
\begin{aligned}
D =& 4 \cdot \langle \vec{d}, O - S\rangle^2 - 4 \cdot \|\vec{d}\|^2 \cdot \left(\|O - S\|^2 - R^2\right) \\
=& 4 \cdot \left[\langle \vec{d}, O - S\rangle^2 - \|\vec{d}\|^2 \cdot \left(\|O - S\|^2 - R^2\right)\right].
\end{aligned}
$$

The value of the discriminant can be either negative, zero, or positive, and from this we can deduce whether there are none, one, or even two intersections, respectively. If there are any we can then carry on with the quadratic formula and compute the value of t this way

$$
t_{1,2} = \frac{-\langle \vec{d}, O - S\rangle \pm \sqrt{\langle \vec{d}, O - S\rangle^2 - \|\vec{d}\|^2 \cdot (\|O - S\|^2 - R^2)}}{\|\vec{d}\|^2}.
$$

We have now explicitly computed the value of t. If we would now want the point at which the intersection happens, we would simply plug t into the parametric equation of the ray. In the figure 8 we can see all the configurations of a ray and circle. Also note that if t is negative we have computed an intersection which lies in the opposite half-space and not on the ray.

There are two types of use of ray casting. Either a renderer is based solely on it or it is only used to enhance the fidelity of an image computed by different means. The second kind is of course much less intesive in terms of computing power and is the one generally used when ray casting and real-time rendering are involved together as is the case in games nowadays. Renderers which use only raytracing are at this moment in time unable to compute an image for real-time rendering purposes. Rendering an image or a single frame of a video can take hours or even days. But that does not make them less useful. They are widely used in, e.g., film animation and architectural software.

Integrator

We now have the basic concept of ray casting and we can now move onto designing the integrator itself.

One approach to this task is to shoot many rays originating at the lightpoint with every wavelength we are interested in and in every possible direction (meaning some uniform sampling of the 3D sphere around the light). For each ray, we would compute whether it intersects a pixel of that camera. We would then average out the intensity of the light rays going through the pixel.

If we were to do it really like this, the result would not be a good representation of the scene. It would produce a white image.

Why are we seeing a white image? Well, take a ray shot directly from the light to the camera. It will successfully intersect some pixel, and we will count it in our algorithm, but when you look at the description of the scene, it should have been blocked by the ball. Therefore, we have to extend our integrator. Apart from finding which object we intersect we have to compute how far along the ray we have intersected that object. And then we say that we have intersected an object only if it is the closest intersection from the origin of the ray.

If we, with this new notion of intersections, run the algorithm again, we would be met with a slightly more faithful representation of the scene, white picture with a black circle in the middle. It could also be an all-black image. This would depend on the size and distance of the ball in relation to the image.

This is a progress, but we still have some work to do. In the real world, even the front of the ball facing the camera would be illuminated by the light that bounces around the room. This is called global illumination. Other rendering techniques would have to resort to some cheating to get this working, but we can slightly tweak our integrator and get our desired effect without cheating.

This leads us to the recursive nature of ray tracers. When light intersects an object apart from the pixels of our camera, it usually does not stop there. It interacts with the object material, is somehow changed and is shot in a new direction.

Interactions of light with solid objects can be mathematically modelled to varying degrees of realism and complexity. Suppose we come up with a model, we would then be able to send a new ray from the point of intersection in a new direction. This could then be repeated until the ray hits one of our pixels or until we have run out of patience.

This would again give us an image more in tune with reality. But there is more work to be done.

Firstly, doing raytracing from the light source is really inefficient because we would spend a lot of time raytracing a ray that would not reach any pixel in the end. Therefore, a better idea is to raytrace from the camera, which means shooting rays through the pixel into the scene and hoping that we eventually hit some kind of emissive material.

This again would be really inefficient in the same way because we would once again do a lot of raytracing without hitting any emissive material. But there are ways to make it converge faster. For example there is a technique which in essence sends rays towards lights that check whether they are obstructed while handling an intersection and if these light feelers are not obstructed we would count it as a hit. This produces a bias in an image, which would have to be handled.

This is enough to give us a good and intuitive understanding of how raytracing works without us being overly technical about it.

What is Image Sampling?

When rendering an image via raytracing we have described us shooting rays through each pixel to estimate the spectral radiance. This estimate would be very noisy if we were to send only a single ray per bucket at that pixel. We therefore have to send multiple rays per pixel to produce a noise-free estimate. This is called sampling that pixel and in effect sampling the whole image. Now, sampling the pixels in the same way would always produce the same results, which is not what we want.

We avoid this issue in two ways. First, a pixel is not a point-like structure; rather, it is a very small square on the image plane. Therefore, we can send the ray through a slightly different point on that small square. Second, during the ray-tracing process, there are many points where we have multiple options to choose from. We have omitted this from our explanation of ray tracing and hidden it in the point about modelling ray&intersection behaviour.

We choose these options at random. To achieve even more randomization, we can set the seed of the random generator to different values when we first shoot the ray through the pixel.

Rendering even one pixel to a reasonable standard of accuracy might take hundreds, thousands, or even tens of thousands of samples in that one pixel. This all means that computing the entire image will take an inordinate amount of time. Fortunately for us, modern computer architecture has an answer in terms of multicore CPUs and parallel programming. This allows us to compute multiple samples at the same time and thus decrease the rendering time, however, parallel programming has its own pitfalls: synchronization and data races. In the subsequent chapters, we will explore how all of this is done in more detail without falling into the pitfalls.

Kernel splatting

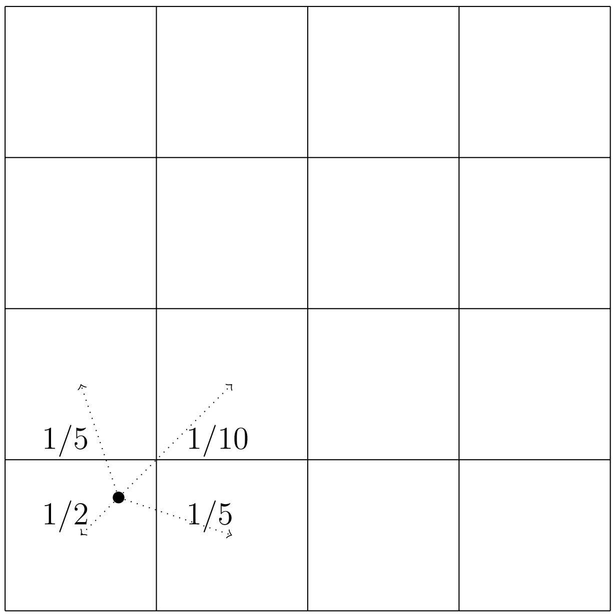


Figure 9: How splatting might look. Given a sample at the point the intesity is redistributed according to the splattine kernel. In this case this is an arbitrary splatting kernel loosely based on the distance from the centres of the pixels. Pixels which are not adjacent receive no contribution by default in this splatting kernel. This redistribution is shown as a fraction of the intesity of the sample.

There is one more technique related to image sampling that will be of great importance to us when designing and writing our image sampler. It is called kernel splatting. The motivation behind this technique is this: imagine that you sample a pixel in its corner and you get back an intensity at particular wavelength.

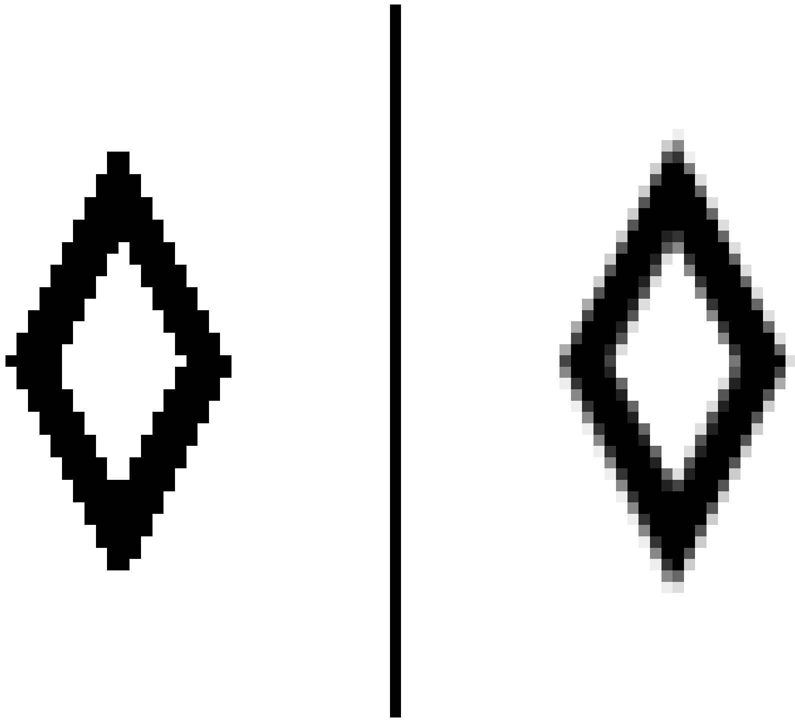


Figure 10: An example of anti-aliasing an image taken from [7]. To the left the picture is very pixelated and to the right the picture has blended and smoother edges.

Now, because we are at the corner of the pixel, it might be a good idea for this sample to also influence the pixels in the surrounding area and not just the pixel we are sampling. We do this by distributing the intensity and also adding partial samples to the nearby pixels. This can be seen in figure 9. The final sum of the partial samples is then used to average the intensities in a given pixel. How this is done is modeled by the splatting kernel.

The splatting kernel is a function that given a point in the image plane returns a fraction of the intensity which should be given to every pixel and how big the partial sample should be. An important property of this function is that it has a bounded support, meaning that the sample on every point in the image affects only a bounded pixel area. And it is also very good if the bound is very small.

This technique is a powerful tool to fight with artifacts produced by trying to show a high-resolution image at low resolution. In our case we are trying to render down a continuous scene to a grid of pixels. Such artifact fighting techniques are called anti-aliasing techniques and we can see them in action in figure 10. Kernel splatting is such an anti-aliasing tool, but it is also a possible source of data races when writing a parallel image sampler.

Parallelism and synchronization

In this section, we will introduce common and important concepts of parallel programming.

Cores, Hyper-Threads and Threads

Programs in general are run on a CPU. In the modern day and age most CPUs have multiple cores and those cores oftentimes have multiple hyper-threads.

Each hyper-thread in a core runs a separate sequence of instructions whilst sharing execution units which are, for example, an unit which can add two numbers together or access memory for reading or writing. Having multiple hyper-threads in a core (mostly only two) can oftentimes be a good idea because hyper-threads usually do not fully exhaust all capabilities of the execution units.

This, however, is also the reason that even if we double the hyper-thread count in a trivially parallelizable task, we should not expect halving the execution time. Because the hyper-threads, thought running separate instructions streams, will oftentimes collide in the use of the execution units thus one of them will be stalled.

The next concept is that of the thread (not to be confused with hyper-thread). Thread is an operating system abstraction which executes some user determined function. A program can create many threads and operating system schedules them to run for some amount of time on the hyper-threads.

They can be created for different purposes. They can be made because of a need to do multiple things at once. For example, to watch input from the user, simulate physics, etc., all at the same time. Or they can be created to exploit more cores or hyper-threads and therefore increase the speed of the program.

The term thread is often substituted for hyper-thread (and never the other way around).

Race conditions

While we are executing parallel programs, we have to be wary of race conditions which are unsynchronized accesses to the same memory location.

There is one unsynchronized access, which is always correct, the so-called Read-Read access when there are multiple threads reading from the same memory location.

The other three - Read-Write, Write-Read, and Write-Write - can produce a data race.

For example if two threads want to write to the same memory location we cannot be sure which one of them has written last to the location and we therefore cannot be sure about the final value stored there. This is the Write-Write case.

We in general should want to avoid these and thus we have to use mechanisms to avoid them. These mechanisms are called synchronization primitives.

Synchronization Primitives

The most common primitive is mutex. It is an object that a running thread can lock, then do some things, and then unlock. If multiple threads were to try to lock the same mutex, only one of them would succeed, and the rest would either have to sleep and wait for the lock, or they would be notified of their failure to acquire the lock immediately and could try to handle that somehow.

This can be used in many ways, such as protecting one piece of code from being executed by multiple threads at the same time or protecting multiple pieces of code from being executed concurrently.

A very similar thing is a semaphore. These can be configured at the start with a number. Threads can then wait on the semaphore and if the semaphore contains a positive number, they can carry on and the value is decreased. If the number is zero threads are either put to sleep until this changes or are notified of the failure. They can also post to a semaphore, thus increasing its stored number.

Next, there is a barrier. Barrier is configured at the start by a number. Threads can arrive at the barrier and sleep there until they are woken up. The barrier counts the number of threads that arrive there and, when enough of them arrive, it wakes them all.

Lastly, there is a conditional variable. These work in tandem with a mutex. Let us say that there is some variable x in our program. And we want to execute some code after some condition is fulfilled. We first acquire a mutex that is connected to this variable. Then we test the condition. If the condition was fulfilled, we do what we wanted to do, and then we release the mutex and we are done.

If, however, the condition is not fulfilled, we wait on the condition variable which puts us to sleep and simultaneously releases the lock.

The other threads can, while holding the mutex, update x and notify sleeping threads on the condition variable that the condition might now be fulfilled. This will notify them all or only one, depending on the method of notification.

One of them is woken first and at the same time acquires the lock. It can then do the same thing as before.

One thing of note is that neither the mutex nor the condition variable is intrinsically linked to the variable x. This is something that the programmer must keep in mind when using this primitive.

ART Renderer

We have introduced the necessary background information to be able to talk about the topic of this thesis. This thesis is about an improvement to the image sampler of a renderer called Advanced Rendering Toolkit commonly called ART [3].

ART is a renderer that started its development in Vienna at the Institute of Computer Graphics by Robert F. Tobler in 1996. Later on the fulcrum of development was transferred to the Computer Graphics Group at Charles University under Alexander Wilkie.

It is a spectral renderer not too dissimilar to the one we have constructed or any other like PBRT; however, it has features which are not supported at all or are only supported by few current production renderers. These are bispectral materials that facilitate the rendering of fluorescence and the rendering with polarized light.

Issues with Sampling in ART

Unfortunately, the ART sampling code is rather dated and has issues. There are one major and three minor issues. The major one is that it is prohibitively memory hungry and the memory consumption increases drastically per each core we want to utilize, and it also increases with higher resolutions. This presents a big problem when using the renderer for research purposes on university servers, and it surely has negative implications in terms of cache locality of the renderer, too. We will go more in-depth about this issue in section 1.1.

The following are minor issues, but we will solve them none the less.

The sampling code in ART was written to be highly extensible by subclassing. But this capability was never really used, and the price for it was over-engineered and hard to understand code. We will simplify the code and remove the unused capability for subclassing.

The image sampler is also an ideal place to put code which can visualize the rendering as it is progressing in real time. But this was never implemented.

It is also the place which controls the rendering process and therefore a place which also interacts with the outside world to, for example, stop the rendering. These commands from the outside were implemented, but in a way that allowed for almost no extension beyond what was already there. We will provide a way to extend these commands.

1. Sampling Techniques

Let us talk more in-depth about image sampling used by ART before this thesis and the techniques we considered when designing the replacement.

1.1 Original ART sampling

The folder `Source/ClassLibraries/ImageSampler/` contains the original implementation of image sampling in ART in these six files:

- `ArnImageSamplerBase.[m|h]`
- `ArnImageSampler.[m|h]`
- `ArnStochasticImageSampler.[m|h]`.

ART dodges the inherent issues of parallel programming, data races, and synchronization, by throwing a lot of memory at the problem. It keeps a separate spectral image per rendering thread, and each thread is given its fair share of the number of samples per pixel it should perform. The partial images are merged into a single output image when all threads are finished with rendering their share of samples or when an external command is given to output the current image. This has its benefits and detriments.

The obvious benefit is the absence of complicated synchronization schemes. This ease of implementation is probably the main reason why it was done this way in the first place. The lack of synchronization and the fact that every thread has its own sandbox means that there is no slowdown due to locking or other synchronization primitives.

But, as was already said, it consumes a lot of memory. Let us now explore how much it can take in a real word example. Imagine us having to render a 4K spectral image (resolution of 3840×2160) with polarisation. This is not an unreasonable request from a renderer because if ART were a production renderer used in the film-making industry it would be expected to handle at least this resolution if not 8K as these are the resolutions that the current generation of cameras shoot in, and renderers should match the resolution of the camera when adding VFX to the scenes.

ART provides multiple levels of fine-grainedness of the spectrum. For the sake of this example, let us use the most precise one, which has 46 spectral bands. In earlier versions of ART there was even a possibility of 450 spectral bands, every one nm, which was used for reference images for research but it is no longer supported. We suspect that the reason why this is the case will become clear when we complete this memory analysis. The code which implements these different spectra is, however, very easily extendable and there is no reason why spectra with 450 bands could not be reintroduced if the rendering with such a fine spectrum would be feasible.

To implement the polarisation something called a Stokes vector is used to represent one spectral bucket. How exactly this is done is not important to us. We only need to know that it uses 4 values. That is, we have 184 values per pixel.

We then have to add one more value to represent how many samples we did. In the end this gives us 185 values per pixel.

These values have a certain datatype. Ideally it would be a real number, however, that is impossible to represent on any digital device, so we have to satisfy ourselves with some sort of floating point datatype. ART uses doubles to represent them and those take 8 bytes per value.

Now to estimate the memory required we have to know how many cores we are trying to utilize. We are going to go with 48 in this example because that is the number of hyper-threads available on the machine with the greatest hyper-thread count from the CGG servers.

This server has around 123GiB of RAM, so if we exceed or even approach this number, the program would run out of memory during its execution.

This gives us a memory requirement just for the images:

$$3840 \cdot 2160 \cdot 185 \cdot 8 \cdot 48\text{B} \approx 549\text{GiB}.$$

As we can see, just the per-thread rendering images would take a little more than 4 times the memory that is available on that machine. And even if we were to drop the polarisation, we would still be at around the maximum RAM capacity. This means that at this moment this configuration would be unfeasible for ART.

The real memory footprint would be a bit bigger because the image into which the partial results are merged must also be allocated. And we did not consider any other part of the rendering pipeline.

1.2 Single-buffer sampling

We have described how ART works with memory, and now we have to talk about fixing that issue. If we want as small a memory footprint as possible, there is a practical limit of at least keeping one image in memory. (The theoretical limit is lower; for sure, an external memory algorithm could be designed and implemented but it would be unnecessarily complicated and more than likely very slow.)

If we had run the calculations we would find that the image in 4K would take about 12GiB and around 46GiB for the 8K image. This is very much manageable for the machine we are taling about. So we have succeeded in that regard.

But it is not obvious how we would go about sampling the image without causing a data race. Now the reason why the data races might prove to be an issue goes back to the technique we talked about in the introduction i.e. kernel splatting.

If there were no kernel splatting, a working implementation of this sampling technique would be very simple. Just assign regions of the image to different threads and let them do the work in place. But even this configuration has a problem. It is not adaptive. That means if one thread finishes its region early, it is done and does no further work, which is not desirable.

This could be solved. An approach that could work is the following. Have a single mutex. Then have the rendering threads iterate through the pixels. Each time a rendering thread would want to work on a new pixel, it would lock the mutex, increase the iteration variable, release the mutex, and then go ahead with sampling that pixel.

There are a few notes to point out at this time. The original sampler has an interesting property of sampling the image in a continuous manner, meaning it does not sample each pixel to the desired sampling count all at once, but rather passes over the image each time only doing a sample per spectral band. It is true that this is probably not optimal in terms of speed, but it has two benefits. First, it allows us to send a command to the image sampler and have it produce the current image, which is sampled all over instead of only at some pixels. Secondly, it allows for unlimited sampling. That means that you can start ART and let it run for a month, look at the result, decide whether the image is of high enough quality, and potentialy let it carry on sampling.

If we would like to support a similar behaviour, it would produce these two issues. There is a chance that in an environment with a lot of cores, the lock would be hotly contested, and it would be a bottleneck for this algorithm. This could be solved by changing the algorithm to do a number of samples, not just one. A sufficiently high number of samples would prevent the bottleneck, while leaving it as small as possible would provide more continuous sampling. This number of samples could be either fixed at the start of the run of the algorithm or be adaptively changed.

The next issue is more serious. If we are doing only a part of the samples, there is a chance that when we once again get to the same pixel the previous thread which had that pixel assigned has yet to finish its job. This is not very likely in a big image, but in a small one it would be possible.

A solution would be to introduce a per-pixel mutex. Then, have each rendering thread render to its own private variable. And only merge the private variable to the pixel under its lock. This would add some more memory overhead. If we take the standard pthread mutex, it takes 40B on a 64-bit machine. This is not an insignificant amount, but still very much manageable and it has the same impact as adding five more doubles to the per-pixel data, meaning that we would end with 190 values per pixel in the previous example. But even so, this is not ideal.

Extending this sampling technique to support kernel splatting would no longer be very difficult. The only difference would be that instead of a single pixel in an internal variable, we would have to have a small tile with an area of the upper bound on the support of the splatting kernel in use.

But instead of having collisions at per-pixel locks being a rare occurrence they would happen all the time because we are iterating pixels row by row, therefore, we are always sampling pixels which are near each other. This would indubitably slow down the rendering, and that is not desired.

It would then be possible to decrease this effect by providing a different iteration scheme. We could, for example, have a different vertical and horizontal stride based upon the bound of the splatting kernel instead of going row by row. But this would harm the readability of the code.

In the end, we have ended with a technique that certainly decreases RAM usage but would be either slower or harder to read and maintain. We have also introduced a great number of per pixel mutexes and even if we would solve the contestion issue they would still add memory and locking overhead.

1.3 Tiled sampling

The third technique which we will discuss is the technique we eventually decided to implement. It builds upon the single-buffer technique in many ways, while being different enough to not be considered the same by the author.

We started the single buffer sampler by wanting to have the practical minimum amount of memory and built from that idea. In this sampler we will allow ourselves to use a little more memory. We will keep the single image, into which we will merge the samples we make and we will also allocate memory for tiles. These tiles are gonna be the sandboxes where each thread is the only one with access to that memory, meaning there is gonna be no synchronization necessary when storing or reading samples.

We are gonna start by introducing two types of thread. The first we will call the merging thread, and the second one we will call the rendering thread. We are going to have one merging thread and several rendering threads up to the number of hyper-threads available on the machine. The rendering threads and the merging thread will communicate via two synchronized queue-like structures. Let us call them rendering and merging queues, respectively. This should remind us of the producer-consumer pattern.

These queues will consist of tasks. The task structure will contain its type, which will be more important later on when we are talking about fixing the minor issues presented in the introduction. They will also contain any additional information necessary to complete the given task type.

We now define two types of tasks. A rendering and a merging task. Both of these will contain similar information. For both of them, there are going to be the start and end points in the image where the sampling should happen, we are going to call this a rendering window, and also a pointer to the tile where the sampling data should either be stored or read. The rendering task will need two more parameters. How many samples should be done per pixel and how many samples were already scheduled in that rendering window.

The job of the rendering thread is very simple. It gets a new task from the rendering queue, resets the values in the tile, samples the rendering window according to the task, stores the result in the tile, and then pushes a new task with the same parameters as the rendering task (apart from the type) into the merging queue.

The job of the merging thread is also quite simple. Get a task from the merging queue and then merge the data from the tile into the master image. And after all of that push new rendering task into the render queue with the next render window but with the same tile.

This is the essence of the tiled sampler. But there are a few notes we want to make about the differences and similarities to the single-buffer sampler.

Our tiled sampler at this point has only two mutexes which are hidden in the synchronized queues. This is in stark contrast to the single-buffer sampler which had a mutex per pixel in the image. This can save a lot of memory. Locking and unlocking are also a lot less frequent.

But the tiled sampler still has the same issue that the single buffer sampler had, a bottleneck. This bottleneck is located in the merging thread. We could theoretically deal with this bottleneck via having multiple merging threads and we could try to come up with a scheme to lock the rendering windows of the image when merging the tiles into the main image, similarly to how we locked down the pixels in the single buffer sampler.

But we decided to go with the route of adaptively increasing the amount of samples a rendering thread has to do as the rendering progresses. Increasing this number will increase the gaps between issuing the rendering task and merging the tile. This in turn will completely remove the bottleneck after few increases. It also decreases the number of merges, which is good for performance because we are simply doing fewer operations.

Starting with a low number of samples is still a desirable thing because it gives us a rough image very quickly.

The question is how should we decide whether to increase the samples? And when should we increase them?

To keep the sample count as uniform as possible, the only place where we can increase the sample count is at the start of the iteration over the rendering windows. We will call a single iteration over all rendering windows an epoch.

We could simply multiply the sample count by two at the start of every iteration and it would probably work well in most cases. But in the event of unlimited sampling, it could lead to a single tile, which would take a very long time to render. This would then mean that there would be a large difference in sample counts between the more recently sampled rendering windows and those that have not yet been rendered during that epoch. This is not desirable because we want efficient but continuous sampling.

We have to find a different way to decide whether we should multiply the sample count. To do this, we have to talk a bit more about the merging queue. Instead of it being a simple queue with a lock, it would be a good idea to have it consist of 2 queues. One of them is going to be active and one of them is going to be inactive. The rendering threads will add tasks only to the active queue. When the merging thread wants to get a new task instead of simply popping from the active queue, it will switch the active and inactive queues under a lock. It will then have the sole access to this inactive queue and no locking is going to be necessary to interact with the inactive queue.

Then the simplest way to implement the growth condition is to check the size of the inactive queue right after this switch. If there are more tasks than some constant, we will take a note of it and at the next available point increase the sample count.

We decided that the appropriate constant for this is two. Having the constant at one would mean that it could happen that two rendering threads would finish at almost the same time even though the adaptive samples are large enough. And having it any bigger feels to us like a missed opportunity to increase the effectiveness.

But let it be said that it might be a good idea to set the constant to one and count the number of times we exceed that constant per epoch and only increase the number of samples if it happens in a reasonable amount of cases per epoch.

2. Code Overview

In this chapter, we are going to go over all the code that implements the image sampler we have already designed and also the extended IPC capabilities and real-time rendering visualization we have yet to talk about.

We are also going to talk about the ART architecture and how exactly the sampler fits into it. This part will also serve as an acknowledgement that snippets of the sampler are either exactly the same or only underwent slight tweaks from the previous samplers present in ART.

2.1 ART Language and Architecture

When ART started its development cycle, its creator, Robert Tobler, was looking for OOP capabilities that simply do not exist in pure C code while maintaining very good optimization techniques of C compilers. The C++ language was considered, but was found to be not stable enough at the time, particularly in templates and the lack of standardization of parallel programming. In the end, a combination of Objective-C and pure C was decided upon. ART was also written to be multiplatform running both on Linux and MacOS.

All of this unfortunately leaves it in a state where it is neither fully OOP nor fully imperative. For example, it uses C-style containers even when Objective-C has a different style of containers present. But let us say that this is not necessarily bad; just that it makes coding for ART more diffcult than it could be. The mixture of the language sometimes requires ugly workarounds when interfacing with the kernel of the operating system, which makes it difficult to write well encapsulated code.

All ART application are based upon a sequence of `ArnAction`-s. These actions are executed in a linear fashion. Each action can store and get arbitrarily many data objects from a stack.

The action sequence construction and stack initialization is left to the user, who programs what exactly should happen in the `.arm` file. This means that the language for designing scenes and what should happen with rendering is written in Objective-C itself. A detailed description of this format and its use can be found in the ART handbook [3].

For us, the actions we are interested in are, of course, the different image samplers. These are necessary when the user wants to render something. These image samplers do have to have a common interface, and therefore our own image sampler has to obey this interface.

It also means that when writing the sampler, we had to follow this interface even when it might be a bit unnecessarily convoluted.

2.2 ArcTevIntegration

The first part is going to be all about the real-time viewing. This was done via the external application called `tev` [8].

This program is capable of viewing and tonemaping HDR images. More importantly, for us, it provides an over-the-network API to create these images and send updates only to a small window in the image.

The class `ArcTevIntegration` is responsible for the communication between ART and `tev`. It can try to initiate the TCP connection with `tev` and also provides functions that implement all the commands described on github of `tev`.

All commands fill an internal buffer which increases in size if necessary. When the command is ready, it is sent over the network to `tev`.

An interesting point which makes this class a bit more interesting is the necessity to change everything to the little endian format when saving things to the buffer to be sent to `tev`.

The exact way this class is used when sampling the image will be described later in this chapter.

2.3 ArcMessageQueue

This part is about IPC. ART used to use only signals for communication between `impresario` and `artist`. The command-line application `artist` is the part of ART that is capable of interpreting `.arm` files. The `impresario` application allows the user to send commands to running instances of artist.

Using signals has inherent limitations because the POSIX standard allocates only two signals `USR1` and `USR2` for user-defined behaviour. Both of these signals were already used by the previous sampler. One was used for saving the current image and the second for tonemaping and opening the image in a predefined viewer. Also, the nature of signals does not allow for any more data to be sent along with the signal.

The addition of `tev` integration brought the need of more signals and also the need for additional information to be passed along. Namely, we need to be able to tell our sampler to try to connect to a running instance of `tev`. We also have to be able to tell the `artist` about the address or the port of `tev` if they are different from default. This capability of sending more information could prove to be useful in future versions of ART, which could implement a more complex communication to, for example, allow saving of the image to a new location.

There is one more requirement for the replacement of those signals. There can be multiple running instances of `artist`, and we have to be able to pick the instance which should receive a specific message.

All of requirements can be fulfilled by the standard IPC primitive on Linux and MacOS called `System V Message Queue`. The `ArcMessageQueue` is a class that allows both the `impresario` and `artist` apps to interact with such a queue.

When either of these apps is created, an `ArcMessageQueue` is instanced, too. This instance either connects to a particular message queue or, if necessary, creates that queue.

Each message in this queue has a type and content. A particular strength of this message queue as opposed to POSIX style message queue is that the listeners can request messages of a particular type.

In our case, we have the `impresario` app, which sends commands to the queue. And the `artist` instances which listen for the messages with their process id as the type. This emulates the capability of sending a signal to a particular `artist` instance.

In certain instances, the `impresario` is also the listener to clean the queue from messages to `artist` instances that no longer exist.

The author wrote the whole class `ArcMessageQueue` and only made changes to the `impresario` app to switch from the signals to `ArcMessageQueue`.

2.4 ArnTiledStochasticSampler

To start implementing the sampler, we had to first start by implementing the two types of synchronized queues, and to do that, we needed a normal queue. Unfortunately, ART has no such thing at the moment, and therefore we had to implement a simple circular queue, which can grow if it needs to.

2.4.1 Circular Queue

The queue is very simple. It has a pointer to the memory location where the data is stored in contiguous fashion, two offsets for the tail and head of the queue, and also the maximal and current length.

It has the usual peek, pop, and push semantics of a queue. A non-standard method of operation with this queue is prepending. This was added to the queue for high-priority tasks. Apart from it growing if necessary, the only interesting thing is that the whole data structure is macrofied to make it work with different types as is usual for pure C data structures. This is not used at the moment anywhere else, but it could prove useful down the line for ART.

Apart from this functionality, there are also initialization and destruction functions.

2.4.2 Merge and Render Queues

Using this circular queue, we wrote two wrappers, namely the merge and render queues. Both of these wrappers contain a single mutex and condition variable from the pthreads library.

With these synchronization primitives, we wrote functions that work with the underlying queues.

For the rendering queue, simple push and pop semantics were implemented. There are two notable things. The first is the use of the poison pill pattern. This pattern provides a way for graceful shutdown of the popping threads. That is, if the pop function encounters a special task (the poison pill), it returns this task, but does not pop it from the queue.

The second one is that if the queue is empty when trying to pop from it, the thread which called the pop is going to sleep and will be woken up when new tasks are added to the queue. It will then have another go at popping from the queue.

The merge queue keeps two circular queues inside it and it has special semantics. One of those queues is active, and one is inactive. Prepending and appending to the merge queue happens as we would expect under the included mutex and affects the active queue. There is no pop semantics that we would expect from a queue (the name queue might then be a bit misleading). Instead, we have swap semantics, where the pointers for active and inactive queues are swapped. This again happens under the mutex. It also has an interesting property similar to that of popping from the rendering queue. Meaning that if there are no tasks in active queue, the swapping thread is put to sleep and will be woken up when the active queue is inserted into.

2.4.3 Class methods

In this section, we are going to go through almost all methods of `ArnTiledStochasticSampler` and describe their function and if necessary comment on the implementation. This should provide us with a decent understanding of how this class was written. All code discussed here unless stated otherwise was written by the author.

`init`

This is the function that provides an interface for the user who wishes to use this sampler in the `.arm` file. Sets the provided parameters.

`code`

This provides serialization and deserialization support.

`setupInternalVariables`

This function is called in `init` or in `code` after deserialization. Sets the internal variables of the sampler.

`performOn`

This function is taken almost verbatim from the previous samplers. It looks through the data stack for objects necessary for rendering and then calls the next three functions in order.

`prepareForSampling`

This function allocates and prepares all the memory for the parallel sampling. This includes, but is not limited to computing the render windows, allocating all tiles, precomputing splatting kernels for sample locations, precomputing sampling locations and many more.

Some parts of this were taken from the previous samplers, but a lot of it was rewritten later.

`sampleImage`

Here we release the threads. We release all rendering threads, one merging thread, one that monitors the message queue, and one that handles the input and output of the interactive terminal.

After we release all these threads, we wait on a barrier for all the rendering threads to finish. Once they do we push a task which writes current image and then shuts down the merging thread, and after that we wait on a barrier for the merging thread.

`cleanupAfterImageSampling`

This function performs a cleanup of all the memory allocated by the sampler.

`renderThread`

This is the function that all the rendering threads perform.

Unless we have finished or ran into a poison task, we pop a new task from the rendering queue, clean the associated tile, subsequently we do the sampling, and then we insert the task with the changed type into the merging queue.

In the end, we arrive at the associated barrier.

`renderTask`

This task performs the sampling of the render window. It was heavily influenced by the previous sampler but has undergone non-trivial rewrite, which was required because of the switch to tiles.

This rewrite includes slightly changed initialization of random generators to provide seed-consistent rendering and a lot of coordinate recomputing based on the rendering window.

`mergeThread`

This is the function that the merging thread performs. It swaps the merge queues, checks the length for adaptive increase of sampling, and then goes through the tasks in the inactive queue.

The task is either a merge, in which case the `mergeTask` and then `tevTask` is called followed by generating a new rendering task and inserting it into the render queue if necessary.

Or it could be a write task that calls `writeImage`. Or write and tonemap task, which also calls `writeImage` and then creates a tonemapping thread that tonemaps that image and shows it to the user. The last one from these write themed tasks is a write and exit which is the same as the write task, but it also shuts down the merging and rendering threads.

It could also be a poison pill that would signal the shutdown of the merging thread.

And finally a tev connection task which tries to reconnect to tev and if successful repaints the image.

In the end, we arrive at the associated barrier.

`mergeTask`

This is a function which merges information from the tiles into the given rendering window in the master image.

`generateRenderTask`

This method generates new rendering tasks. If we have already finished generating those tasks, it generates a poison pill to gracefully shut down the rendering threads. Every subsequent call after generating the poison pill generates no task.

The return value says whether a new task was successfully generated (poison or rendering task).

`taskNextIteration`

This method advances the iterator over the rendering windows. If the end of the epoch has been reached, it resets the iterator, computes the new number of samples to be done in this epoch, and if that number is zero, it sets the flag that we have finished generating rendering tasks.

`tevTask`

This function performs a rudimentary conversion of the image into RGB values and sends this representation to tev through the `ArcTevIntegration`.

`repaintTevImage`

This function goes through all the rendering windows and calls the `tevTask` on them. This repaints the whole image.

`writeImage`

This function writes the current image to the output. It is very similar to the equivalent method in the previous sampler, and only small parts of it were a little simplified and some undefined behaviour regarding mutexes was removed.

`tonemapAndOpenThread`

This thread provides a simple tonemaping action sequence and runs it. It is virtually unchanged from the previous sampler.

`terminalIOThread`

This thread monitors the interactive terminal input.

It was based upon a similar code but greatly simplified and a bug regarding shutting down without returning the previous setup of the terminal was fixed. It was also extended to support the new tev commands.

`MessageQueueThread`

This thread monitors the message queue for new commands and resolves them either by interacting with the merge queue or by directly interacting with `ArcTevIntegration`.

`initTile`

Initializes objects and allocates buffers for the tile.

`cleanTile`

Sets everything in the tile to the default value.

`freeTile`

Releases objects and frees the buffers of the tile.

ArcSampleCounter

I have also slightly rewritten parts of this class to enable better information printing about the size of the samples.

3. Hyperparameters

In this chapter, we are going to talk about two hyperparameters present in the code. These are the size of the tiles and the tile multiplier.

The size of the tile affects the side length of the tile. The tile multiplier affects how many tiles are created during preparation for sampling.

There has to be at least as many tiles as there are rendering threads otherwise there would be rendering threads that would sit idle because there would not be any tasks with available tiles to work on. Rather than adding a fixed number to this baseline, we decided that a good approach would be to multiply it by the multiplier.

We will run our sampler on three scenes which are provided alongside ART. These are the CornellBox, ParkedPlane, and TransparentQuadricsBox.

We will consider these tile multipliers: 1, 1.5, 2 and 3. And these tile sizes: 2, 4, 8, 16 and 32.

We will run these tests on one of the CGG servers which has 20 cores and 40 hyper-threads. We will only run these tests with the maximum amount hyper-threads.

Each of these scenes uses the same splatting kernel which only affects adjacent pixels.

We are going to do 2048 samples in total in each scene. In the table 3.1 we

Tile Multiplier	Tile Size	Pixel count
1	2	640
1	4	1440
1	8	4000
1	16	12960
1	32	46240
1.5	2	960
1.5	4	2160
1.5	8	6000
1.5	16	19440
1.5	32	69360
2	2	1280
2	4	2880
2	8	8000
2	16	25920
2	32	92480
3	2	1920
3	4	4320
3	8	12000
3	16	38880
3	32	138720

Table 3.1: Count of the pixels in tiles for 40 hyper-threads. Notice that we also have to add two to the tile size because of the splatting kernel.

can see how many pixels we are allocating in the tiles. This directly translates into the memory usage of the sampler. So for us the smaller the better in terms of memory usage.

Tile Multiplier	Tile Size	Time	Maximum Sample Size
1	2	183.82	1024
1	4	119.51	512
1	8	93.85	128
1	16	96.29	128
1	32	91.33	16
1.5	2	114.49	1024
1.5	4	101.73	512
1.5	8	88.44	256
1.5	16	91.43	32
1.5	16	89.17	64
1.5	32	85.43	16
1.5	32	91.8	128
2	2	115.66	1024
2	4	92.76	770
2	8	88.78	256
2	16	89.71	64
2	32	89.86	64
3	2	115.22	1024
3	4	102.5	512
3	8	99.88	256
3	16	95.35	64
3	32	99.85	32

Table 3.2: Results of CornellBox hyperparameter test rendering.

From the results of the tests shown in 3.2,3.3 and 3.4 we can see that the tile count multiplier greater than 1.5 does not seem to have any effect and having it at 1 is significantly slower. Therefore, we will consider only the 1.5 multiplier going forward.

As for the tile size, anything between 8 to 32 seems very reasonable. We are going to consider mainly the 16 and 32 because they provide more continuous sampling.

Note that to find a true optimum, this is not a sufficient test. We should repeat the test more times per configuration to get a reasonable estimate. We should also try to do it across multiple computers with various core and hyper-thread counts, and finally we should consider a wider range of target samples.

Due to insufficient resources, mainly in the number of available computers and time, a deeper test was not feasible to be done for this thesis.

If such a test were done it might be possible to notice more patterns in the behaviour of the sampler and from them construct a mechanism to set these parameters before each rendering according to that heuristic.

Tile Multiplier	Tile Size	Time	Maximum Sample Size
1	2	6260.46	1024
1	4	4684.7	1024
1	8	4079.09	512
1	16	3921.88	64
1	32	4032.63	64
1.5	2	4459.94	1024
1.5	4	3900.54	1024
1.5	8	3716.3	512
1.5	16	3734	128
1.5	16	3991.74	256
1.5	32	3680.09	64
1.5	32	4009.42	64
2	2	4420.49	1024
2	4	4011.12	1024
2	8	3918.5	512
2	16	3864.89	128
2	32	3917.53	32
3	2	4411.77	1024
3	4	3976.36	1024
3	8	4012.39	512
3	16	4139.95	256
3	32	4115.71	128

Table 3.3: Results of ParkedPlane hyperparameter test rendering.

Tile Multiplier	Tile Size	Time	Maximum Sample Size
1	2	4657.83	1024
1	4	3503.53	512
1	8	3252.03	256
1	16	3296.26	64
1	32	3377.89	32
1.5	2	3622.18	1024
1.5	4	3055.5	1024
1.5	8	2912.35	256
1.5	16	2922.5	128
1.5	16	3072.07	128
1.5	32	2906.18	32
1.5	32	3081.8	64
2	2	3431.8	1024
2	4	3173.48	1024
2	8	3124.45	256
2	16	3107.04	256
2	32	3090.83	32
3	2	3438.93	1024
3	4	3150.54	1024
3	8	3129.98	512
3	16	3423.14	128
3	32	3227.23	32

Table 3.4: Results of TransparentQuadricsBox hyperparameter test rendering.

4. Memory and Performance Testing

In this chapter, we are going to look at how the new tiled sampler compares to the old stochastic one. We are going to evaluate the memory consumption and the speed of the old sampler and the new sampler.

To do this, we are going to run a series of tests on the same server with 20 cores and 40 hyper threads. We are also going to render the same scenes as before. But we are going to render them at both 1024 and 2048 samples and using 20, 30 and 40 hyper-threads. Each configuration will be performed four times. This should give us an overview of how our sampler performs in comparison to the old one.

We are going to measure memory by using the GNU `time` utility, which can measure maximum Resident Set Size (RSS). RSS rougly measures how much memory is currently allocated and present in the RAM. Note that all the numbers which concern RSS will be in kB from now on.

We are also going to disallow scheduling of threads to certain hyper-threads when measuring with less than the maximum amount of hyper-threads per system. This is done because it is necessary for a fair comparison between the new tiled sampler and the old stochastic one. The tiled sampler uses one more thread actively, and this could give it an unfair advantage in the tests with fewer hyper-threads.

We are going to do this disallowing by using the Linux utility called `taskset`.

4.1 Stochastic Sampler

Unfortunately, due to how the stochastic sampler is programmed it oftentimes does slightly fewer samples than the requested amount. This happens because it divides the requested number of samples by the number of threads it should use. This is then the number of samples that each thread does. It does this by integer division and because of that if the division is not without a remainder the value is rounded down. This results in a number of total samples being different from the requested by up to the number of hyper-threads. This might slightly skew the results in favor of the stochastic sampler but it should not be a huge deviation.

Hyper-threads	Maximal RSS	Render time
20	284744	67.44
20	284536	62.11
20	284576	64.41
20	284812	62.39
30	405036	53.92
30	404764	63.68
30	404976	55.3
30	404564	57.81
40	525216	44.2
40	524748	43.62
40	525192	45.41
40	526060	43.56

Table 4.1: Result of stochastic sampler on CornellBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time
20	284928	128.08
20	284776	118.33
20	284876	119.56
20	284936	124.92
30	404752	115.35
30	404988	112.45
30	404636	105.27
30	404800	103.11
40	525320	95.06
40	525900	86.13
40	526236	88.06
40	525180	89.36

Table 4.2: Result of stochastic sampler on CornellBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time
20	3311268	2048.75
20	3310968	2159.82
20	3310972	2130.32
20	3311076	2137.4
30	4834772	1685.46
30	4834932	1695.04
30	4834848	1707.52
30	4834964	1772.71
40	6358188	1512.51
40	6358380	1446.75
40	6358424	1473.42
40	6358584	1508.27

Table 4.3: Result of stochastic sampler on TransparentQuadricsBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time
20	3311308	4475.77
20	3311340	4357.05
20	3311076	4320.3
20	3311344	4246
30	4834952	3549.74
30	4834532	3438.98
30	4834912	3451.77
30	4835064	3447.36
40	6358248	3101.94
40	6358572	3017.16
40	6358512	3102.93
40	6358484	3013.75

Table 4.4: Result of stochastic sampler on TransparentQuadricsBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time
20	5309744	2599.58
20	5309752	2551.4
20	5309764	2576.34
20	5309932	2726.53
30	7771088	2258.56
30	7771088	2125.12
30	7771072	2361.35
30	7770852	2174.73
40	10232148	1869.02
40	10232168	1806.25
40	10232108	1858.38
40	10232204	1854.3

Table 4.5: Result of stochastic sampler on ParkedPlane while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time
20	5310296	5116.11
20	5310136	5150.47
20	5309972	7906.98
20	5309908	5050.65
30	7771000	4469.84
30	7771244	4422.96
30	7771340	4245.45
30	7771052	4273.54
40	10232060	3715.47
40	10232428	3724.41
40	10232328	3746.18
40	10232400	3776.16

Table 4.6: Result of stochastic sampler on ParkedPlane while doing 2048 samples per pixel.

Scene	SPP	Hyper-threads	Avg Maximal RSS	Avg Time	STD of RSS	STD of Time
CornellBox	1024	20	284667	64.08	132.15	2.45
		30	404835	57.67	215.06	4.31
		40	525304	44.19	548.01	0.85
	2048	20	284879	122.72	73.63	4.57
		30	404794	109.04	146.51	5.79
		40	525659	89.65	495.09	3.84
TransparentQuadricsBox	1024	20	3311071	2119.07	140.52	48.53
		30	4834879	1715.18	86.49	39.4
		40	6358394	1485.23	162.92	31.06
	2048	20	3311267	4349.78	128.34	95.85
		30	4834865	3471.96	231.13	52.12
		40	6358454	3058.94	142.15	50.23
ParkedPlane	1024	20	5309798	2613.46	89.71	77.9
		30	7771025	2229.94	115.57	103.47
		40	10232157	1846.98	40.04	27.85
	2048	20	5310078	5806.05	174.18	1401.23
		30	7771159	4352.94	159.91	110.19
		40	10232304	3740.55	168.03	27.01

Table 4.7: Averaged results of the stochastic sampler in all tested configurations.

In the tables 4.1,4.3 and 4.5 we can see the results of the tests while doing only 1024 samples per pixel. And in the tables 4.2,4.4 and 4.6 we can see the results when sampling with 2048 samples.

Finally, we can see the averages and standard deviations of each kind of test in the table 4.7.

4.2 Tiled Sampler with Tile Size 16

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	67328	64.11	32
20	65012	61.86	32
20	67392	60.73	16
20	66964	61.56	32
30	74752	51.44	64
30	74780	50.99	32
30	76308	49.57	64
30	74036	52.17	32
40	84080	47.74	128
40	83880	48.73	64
40	82208	47.84	64
40	82188	48.82	128

Table 4.8: Result of tiled sampler with tile size of 16 on CornellBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	62976	117.17	32
20	72736	121.68	32
20	66932	114.77	32
20	66712	124.57	32
30	74852	97.39	32
30	77128	103.25	64
30	74748	99.88	32
30	74412	106.63	32
40	82280	88.79	128
40	84088	91.65	256
40	88172	90.34	256
40	82056	93.95	256

Table 4.9: Result of tiled sampler with tile size of 16 on CornellBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	354720	2106.1	16
20	361008	2092.73	16
20	358552	2058.82	16
20	360708	2173.4	32
30	362504	1699.99	64
30	373204	1721.07	64
30	367444	1660.29	32
30	367728	1690.1	64
40	372216	1552.51	64
40	373928	1538.55	256
40	382236	1648.33	256
40	378708	1612.67	128

Table 4.10: Result of tiled sampler with tile size of 16 on TransparentQuadricsBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	357560	4184.14	32
20	355112	3961.48	16
20	359928	4370.4	32
20	360700	4123.98	32
30	366272	3451.29	128
30	372252	3251.53	64
30	370772	3517.92	64
30	372972	3476.96	64
40	378236	3070.91	128
40	382644	2995.48	256
40	378632	3640.51	256
40	378524	3157.81	128

Table 4.11: Result of tiled sampler with tile size of 16 on TransparentQuadricsBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	527036	2709.69	16
20	525236	2628.53	16
20	526624	2649.36	32
20	528552	2641.83	32
30	537764	2238.7	64
30	535196	2237.74	32
30	535760	2196.32	32
30	536772	2219.78	64
40	546620	2024.12	256
40	546624	1976.75	256
40	543520	1996.7	386
40	545000	2059.32	256

Table 4.12: Result of tiled sampler with tile size of 16 on ParkedPlane while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	527420	5260.78	32
20	528228	5371.01	32
20	528196	5299.29	64
20	523700	5275.85	128
30	539504	4370.38	128
30	538304	4333.53	128
30	536232	4322.48	64
30	529936	4365.25	64
40	543404	3957.6	128
40	546516	3993.29	256
40	543736	3915.88	512
40	545920	4098.01	256

Table 4.13: Result of tiled sampler with tile size of 16 on ParkedPlane while doing 2048 samples per pixel.

Scene	SPP	Hyper-threads	Avg Maximal RSS	Avg Time	STD of RSS	STD of Time
CornellBox	1024	20	284667	64.08	132.15	2.45
		30	404835	57.67	215.06	4.31
		40	525304	44.19	548.01	0.85
	2048	20	284879	122.72	73.63	4.57
		30	404794	109.04	146.51	5.79
		40	525659	89.65	495.09	3.84
TransparentQuadricsBox	1024	20	3311071	2119.07	140.52	48.53
		30	4834879	1715.18	86.49	39.4
		40	6358394	1485.23	162.92	31.06
	2048	20	3311267	4349.78	128.34	95.85
		30	4834865	3471.96	231.13	52.12
		40	6358454	3058.94	142.15	50.23
ParkedPlane	1024	20	5309798	2613.46	89.71	77.9
		30	7771025	2229.94	115.57	103.47
		40	10232157	1846.98	40.04	27.85
	2048	20	5310078	5806.05	174.18	1401.23
		30	7771159	4352.94	159.91	110.19
		40	10232304	3740.55	168.03	27.01

Table 4.14: Averaged results of the tiled sampler with 16 tile size in all tested configurations.

In the tables 4.8,4.10 and 4.12 we can see the results of the tests while doing only 1024 samples per pixel. And in the tables 4.9,4.11 and 4.13 we can see the results when sampling with 2048 samples.

Finally, we can see the averages and standard deviations of each kind of test in the table 4.14.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	71184	56.86	4
20	71760	58.33	4
20	68988	54.5	4
20	71236	55.65	4
30	83664	48.35	16
30	83396	47.37	16
30	79180	46.63	8
30	83708	47.32	8
40	93396	43.16	16
40	91248	44.66	64
40	91932	44.45	32
40	94184	42.98	32

Table 4.15: Result of tiled sampler with tile size of 32 on CornellBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	71920	107.7	8
20	69404	109.89	4
20	69656	108.83	8
20	71644	109.52	8
30	79532	93.19	16
30	83348	96.88	16
30	81512	98.85	16
30	81376	92.01	8
40	93568	86.57	16
40	93448	85	16
40	93368	83.99	32
40	92168	88.46	32

Table 4.16: Result of tiled sampler with tile size of 32 on CornellBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	364672	2013.54	8
20	363396	2060.86	8
20	363716	2004.86	4
20	363420	2033.53	16
30	374288	1683.95	16
30	379608	1692.06	16
30	374224	1687.8	16
30	377464	1715.55	8
40	384340	1516.52	64
40	388792	1520.82	64
40	390100	1516.4	64
40	388784	1563.22	64

Table 4.17: Result of tiled sampler with tile size of 32 on TransparentQuadricsBox while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	365584	4065.2	8
20	364812	4108.78	8
20	361856	4208.51	8
20	367340	4017.88	16
30	375524	3400.54	32
30	375740	3329.68	16
30	377876	3321.91	16
30	377272	3401.33	32
40	391952	3070.14	32
40	387820	3033.3	64
40	392100	2993.02	64
40	387296	3200.66	64

Table 4.18: Result of tiled sampler with tile size of 32 on TransparentQuadricsBox while doing 2048 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	532692	2634.68	8
20	533968	2601.12	8
20	531336	2575.03	8
20	532868	2618.49	8
30	545036	2148.79	16
30	541764	2151.25	16
30	543160	2146.63	32
30	544084	2196.66	16
40	554804	1924.18	32
40	555632	1894.07	64
40	557344	1939.82	64
40	554308	1919.55	64

Table 4.19: Result of tiled sampler with tile size of 32 on ParkedPlane while doing 1024 samples per pixel.

Hyper-threads	Maximal RSS	Render time	Maximal Tile Size
20	533884	5177.84	16
20	533652	5183.15	16
20	534364	5216.67	16
20	533304	5528.05	16
30	546328	4398.75	16
30	543428	4279.75	32
30	545848	4375.39	16
30	543048	4285.61	16
40	555816	3781.53	256
40	556020	4014.16	64
40	553108	3782.35	64
40	555716	4033.06	128

Table 4.20: Result of tiled sampler with tile size of 32 on ParkedPlane while doing 2048 samples per pixel.

4.3 Tiled Sampler with Tile Size 32

Scene	SPP	Hyper-threads	Avg Maximal RSS	Avg Time	STD of RSS	STD of Time
CornellBox	1024	20	70792	56.33	1230.47	1.64
		30	82487	47.41	2208.97	0.7
		40	92690	43.81	1339.7	0.86
	2048	20	70656	108.98	1309.11	0.96
		30	81442	95.23	1558.86	3.18
		40	93138	86	651.86	1.95
TransparentQuadricsBox	1024	20	363801	2028.19	598.62	24.86
		30	376396	1694.84	2621.62	14.19
		40	388004	1529.24	2519.75	22.74
	2048	20	364898	4100.09	2287.29	81.25
		30	376603	3363.36	1151.39	43.49
		40	389792	3074.28	2589.16	89.94
ParkedPlane	1024	20	532716	2607.33	1079.44	25.52
		30	543511	2160.83	1393.93	23.95
		40	555522	1919.4	1331.8	18.98
	2048	20	533801	5276.42	444.62	168.62
		30	544663	4334.87	1664.32	61.06
		40	555165	3902.77	1377.15	139.74

Table 4.21: Averaged results of the tiled sampler with 32 tile size in all tested configurations.

In the tables 4.15,4.17 and 4.19 we can see the results of the tests while doing only 1024 samples per pixel. And in the tables 4.16,4.18 and 4.20 we can see the results when sampling with 2048 samples.

Finally, we can see the averages and standard deviations of each kind of test in the table 4.21.

4.4 Test result

Scene	Hyper-threads	Tiled 16	Tiled 32
CornellBox	1024	-2.02	-7.75
		-6.63	-10.26
		4.09	-0.38
	2048	-3.18	-13.74
		-7.26	-13.81
		1.53	-3.65
TransparentQuadricsBox	1024	-11.31	-90.88
		-22.32	-20.34
		102.78	44.01
	2048	-189.78	-249.69
		-47.54	-108.60
		157.23	15.34
ParkedPlane	1024	43.89	-6.13
		-6.81	-69.11
		167.24	72.42
	2048	-504.32	-529.63
		-5.03	-18.07
		250.64	162.22

Table 4.22: This table shows the average difference of the 2 new versions of the sampler from the old one. Negative numbers indicate that the tiled sampler is faster on average. Positive ones imply otherwise.

From examining the tables in the last three sections, we can clearly see that we have improved memory consumption significantly in both the tiled samplers. We can also see that the increase in memory footprint when adding hyper-threads is not as drastic in the tiled samplers as in the stochastic sampler, which means that ART could now run on machines with more hyper-threads without running out of space.

In the table 4.22 we can see that both the new samplers are on average about as fast as the old sampler. There are cases where both the new one or the old one is faster.

We can also see that the tiled sampler with tile size 32 is better on average over the one with tile size 16. This also concludes our pick for the preferred hyper-parameter.

Conclusion

We have explored two different new designs for the sampler of ART.

We chose one based on maintainability and memory usage. We have then implemented this sampler and also included multiple improvements. One improvement was the IPC between instances of `artist` and `impresario` and the second was the addition of real-time visualization of the rendered image via `tev`.

We have then performed multiple series of measurements.

The first was used to determine the hyperparameter candidates for optimal working of the new tiled sampler.

The second series was used to compare the old and new sampler in terms of speed and memory usage, and to pick the final hyperparameters which should be used going forwards.

We have successfully demonstrated that the new sampler is about as fast as the old one while lowering memory requirements drastically.

5. Possible Future Improvements

5.1 Adaptive variance sampling

When sampling an image with raytracing it is often very noisy especially in scenes with volumetric materials, refractive or diffractive materials. Even then, however, there are many pixels where the estimated spectral radiance settles fairly quickly and does not change much when being sampled after a few initial samples.

This could be exploited to increase the speed of ART. Those samples used on those pixels that are settled provide no additional information and could be omitted.

Part of the infrastructure is already present in the current implementation, but some things would have to be added.

Firstly, new semantics for sample count would have to be added. It is important that at least a minimal amount of samples is made when sampling a pixel. And of course, a maximal (or unlimited) amount should be provided too. Then, a desired amount of variance should be provided. too.

Tiles would have to be extended with a new boolean buffer which would be filled when creating a new task for the rendering thread. This array would contain the information whether that pixel should be sampled in the rendering thread.

There is already such an array present for the master image where the merging thread merges the tiles. This array is present because some cameras can have invalid pixels, which should not be sampled at all. This array could then be moved into the tile structure, and the whole master image could then be considered a very big tile. This would simplify parts of the code and make it more encapsulated and readable.

When merging the image, an estimate of the variance of the spectral samples could be made. This estimate could be done per spectral bucket or summed over the spectrum. This first approach would be more correct, but it would probably take more memory because a running estimate of the variances would, we think, have to be kept alongside the spectral radiance data.

The second one would likely take less memory because only one running estimate would have to be kept per pixel.

How exactly these estimates could be made is not clear to the author now, but we should note that we know that it is possible because the Cycles renderer in Blender implements a similar system [9].

5.2 Render queue per rendering thread

It might be a good idea to provide each rendering thread with a private rendering queue of fixed size. Each tile would then be associated with a particular rendering thread.

This could provide two benefits. Mutexes of each rendering queue would then be less contested which could reduce the increase in the adaptive sample size, which in turn would increase the continuity of the rendering image.

If this was implemented, we could then associate the rendering threads with particular hyper-thread. That would mean that these threads would only run on the associated hyper-thread. This along with each tile being associated with the rendering thread could increase the spatial locality of rendering thread which might increase the speed. This is the possible second benefit.

Bibliography

[1] Wikipedia. Ray Tracing Visualization. https://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[2] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering From Theory to Implementation*. Morgan Kaufmann.

[3] Alexander Wilkie. The Advanced Rendering Toolkit, 2018. https://cgg.mff.cuni.cz/ART/.

[4] Alexander Wilkie. NPGR025. Course materials for the lecture NPGR025, MFF, Charles University.

[5] Wikipedia. CIE colour space. https://en.wikipedia.org/wiki/CIE_1931_color_space.

[6] Wikipedia. Tone mapping. https://en.wikipedia.org/wiki/Tone_mapping.

[7] Wikipedia. Anti-aliasing. https://en.wikipedia.org/wiki/Spatial_anti-aliasing.

[8] Thomas Müller. Tev — the EXR viewer. https://github.com/Tom94/tev/.

[9] Blender Online Community. Blender - a 3D modelling and rendering package, 2018. https://docs.blender.org/manual/en/latest/render/cycles/render_settings/sampling.html#adaptive-sampling.

List of Figures

List of Tables

List of Abbreviations

- RGB: Red, green, blue format of images.
- ART: Advanced Rendering Toolkit, the sampler we are improving.
- HDR: High-dynamic range.
- 3D: Three dimensional.
- CPU: Central processing unit.
- PBRT: Physically Based Ray Tracer, a renderer described in [2].
- VFX: Visual Effects.
- CGG: Computer Graphics Group.
- RAM: Random access memory.
- IPC: Inter Process Communication.
- OOP: Object Oriented Programming
- API: Application Programming Interface
- TCP: Transmission Control Protocol.
- POSIX: Portable Operating System Interface.
- GNU: GNU is No Unix.
- RSS: Resident Set Size.
- SPP: Samples per Pixel.
- Avg: Average.
- STD: Standard Deviation.

A. Attachments

A.1 Source Code of ART

The attached zipped directory contains the source code of ART with the new tiled sampler. Also note that this directory can be downloaded via git on page `https://gitlab.com/art-development-team/ART`, and checking out the EffectiveParallelSampler.