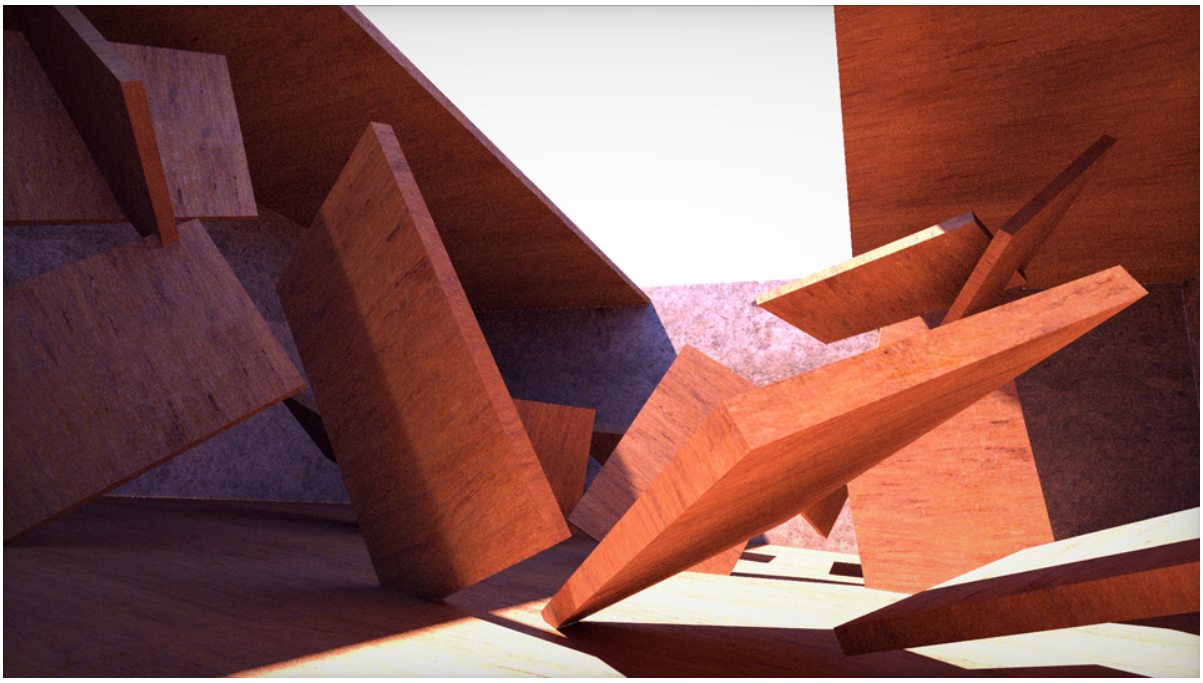
website
articles
rational rendering and floating bar

Intro

"So, what would happen if we replaced floating point numbers with rational numbers and tried to render an image?"

This was the question I posed to myself after some pondering around a tweet by Morgan McGuire, a computer graphics researcher and educator. He was reflecting on how computer science students get surprised when they first learn that the nature floating point numbers in our current computers have precision tradeoffs that makes simple things like checking whether a point belongs to a triangle difficult. The problem of course is that checking for the coplanarity of four points through a determinant or some cross products (same thing really) will never result in a value of exactly zero, which those mathematical methods require. Even if the actual coplanarity computations were exact, the same precision trade-off would guarantee with almost probability 1.0 that the four points themselves would not coplanar.

That sparkled a though on me - assuming the input data to a renderer (vertex coordinates, 3D transformations, etc) was all given as rational numbers, wouldn't all operations from ray generation, to acceleration structure traversal, to the ray-triangle intersection generate rational numbers only? If that was the case, then we would be able to resolve the coplanarity test exactly! You might be asking why would an input 3D scene expressed as rational numbers produce results in only rational numbers too...



A simple scene pathtraced with rational arithmetic, using the "floating **bar**" number system instead of floating **point** numbers

First, a rational number is one that can be expressed as the ratio of two integers, such as 1/2 or 355/113. Secondly, the "typical rendering operations" such as bounding box tests, ray-triangle intersections, ray reflections, etc, are based on cross and dot products and scalar divisions (this includes coordinate transformations and matrix inverses, quaternions, etc), which are in turn based on the four basic arithmetic operations of addition, subtraction, multiplication and division. Now, the addition, subtraction, multiplication and division of rational numbers produce rational numbers too. Formally, a mathematician would say that set of rational numbers forms a field, which implies it's closed under the four basic arithmetic operation. What this means for us is that as long as one sticks to rational numbers, indeed, one should be able to go from input 3D scene to a whole rendered image without ever leaving the world of rational numbers.

Exceptions to the rule of "operations on rational number produce rational numbers" are the square roots and the trigonometric/transcendental functions. For the later, I always express my opinion that if you find yourself doing trigonometry in the geometrical internals of your renderer, you are most probably doing things wrong (and have shown how to fix one of the most typical examples). For the former, the square root, except for conics sections (spheres, cylinders, etc) and actual shading/brdfing/coloring, one doesn't really need to normalize the rays and surface normals as often as we usually tend to. Certainly not for ray generation, traversal,

intersection, reflections, etc. Sadly, often times I find programmers normalizing things for no good reason other than "dunno, I do it to be safe". In practice, you rarely need to normalize things in the geometry tracing part of the rendering, so I had hopes one could indeed trace a whole scene never leaving rational numbers - what I'd call "rational rendering".

Now, to put this into practice, I'd need to build a number system that is based on rational numbers that the computer wouldl be able to use. Then I should be able to implement our usual pathtracing algorithms on top, compute some images without any loss of precision and have coplanarity checks that are answerable exactly, and make all students of computer graphics happy.

This article is the story of a little two nights journey in my explorations on how feasible that is, and the multiple things that I learnt, the things I invented, and the few surprises I found on the way. It's written in more or less chronological order as things unfolded. Also, it's written in my unusual informal and (proudly) very non-academic way. Now, the image above could be seen as a bit of a spoiler, but stick to the end if you can because I bring you both good and bad news.

Setup

The first thing I did was to implement a bare minimal tracer in Shadertoy for a super simple scene composed of a plane, a sphere, a box and a triangle - the building blocks of a real renderer. Then I copy+pasted the code to a C++ file and after a couple minor tweaks, I compiled it with my piLibs framework. That gave me my traced image rendered in the CPU with regular IEEE754 floating point numbers, for reference. I also removed all ray normalizations in the tracing code, since none of them were really needed as discussed earlier. As a reminder, normalization involves a square root and rational numbers are not closed under rooting (the square root of a rational number is not a rational number). And while later on we'll see that we can of course still do square roots, I wanted to keep the code as mathematically pure as possible to see how far could I go with exact, rounding-free arithmetic of rational numbers.

The last preparation step was to take all the vec3, mat4x4 and other basic algebra/math classes and modify them to use "rational" instead of "float". Since my "rational" struct overloaded all common operators (add, sub, mul, div, negation, comparisons, etc) the change went smoothly. I quickly implemented the rest of the usual operations (abs, sign, mod, fract, floor, sqrt, etc), which in theory was all I really needed to do in order to get pretty rational renders.

Test 1 - the naive thing

But let's see how this initial implementation was. I always try the simplest thing first, see how that goes. And the simplest way to implement rationals is to use two integers to represent them. As you can guess perhaps from the title of the article, that was not going to be my final approach, but for a first try it was sensible. So, any number **x** was going to be represented as a numerator **N** and a denominator **D** that produce a value **N**/**D**. The value of **x** gets approximated by the best possible pair **N**/**D** (within some bit usage budget) that gets closest to the true value of **x**. I chose to enforce both numbers to be positive and I stored the sign of the number in a separated bit, to simplify things and remove some ambiguities, although this is not really important. At this point both numerators and denominators where of unsigned type. Even with the sign extracted, the **N**/**D** had still lots of redundancy, for 1/4 and 7/28 represent the same number but have two complete different bit representations. But we'll touch on this later, let's ignore that for now and see what the four basic arithmetic operations look like in this rational form.

First, note the subtraction of **a** - **b** is just the addition of **a** and the additive inverse of **b**, ie, **a** + (**-b**), where **-b** can be computed simply by flipping the sign bit of **b**. Similarly, performing the division **a**/**b** is the same as doing the multiplication of **a** and the (multiplicative) inverse of **b**. Or in other words, **a**/**b** = **a** · (1/**b**), where (1/**b**) can be computed by simply swapping the numerator $\mathbf{b_n}$ and denominator $\mathbf{b_d}$ of **b**. So, this is the first interesting property of rational arithmetic - divisions and multiplications have the same cost, so unlike in regular floating point rendering where division are usually avoided or delayed or hidden under the latency of slow texture fetches, in rational arithmetic there's no need to fear them.

Ok, so moving on to the actual addition and multiplication, and knowing that additive inverse and multiplicative inverses are trivial to compute, we have:

The propagation of the sign for the multiplication is trivial, just an xor, since two positives do a positive and so do two negatives. The propagation of the sign for the addition is more tricky and I implemented it quickly with three branches (the addition is trivial if both signs of **a** and **b** agree, but when they don't you have to pick the smallest of the two and subtract it from the other - I won't bother with the small implementation details here anymore, I'll put my source code somewhere).

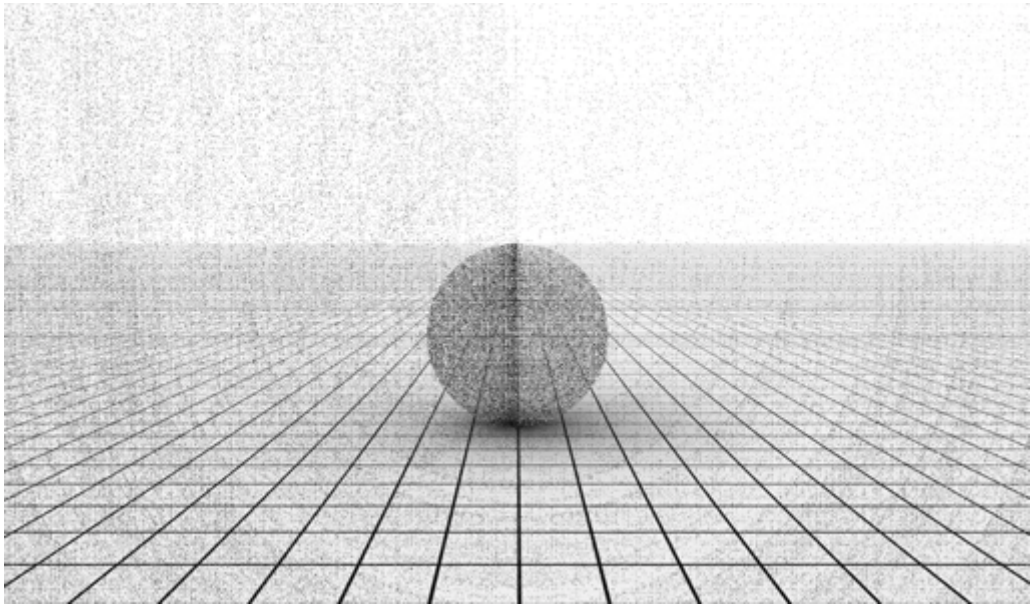
I will also skip the implementation of the fract() and floor(), which you'll find easy and beautiful if you try. The comparison operators deserve some care as well. Once signs have been taken care of and assuming **a** and **b** are positive, then

The important thing to note here is that even comparison require us to do a couple of multiplications and potentially need to promote things to the next word size, and this will become important in a bit later in this article.

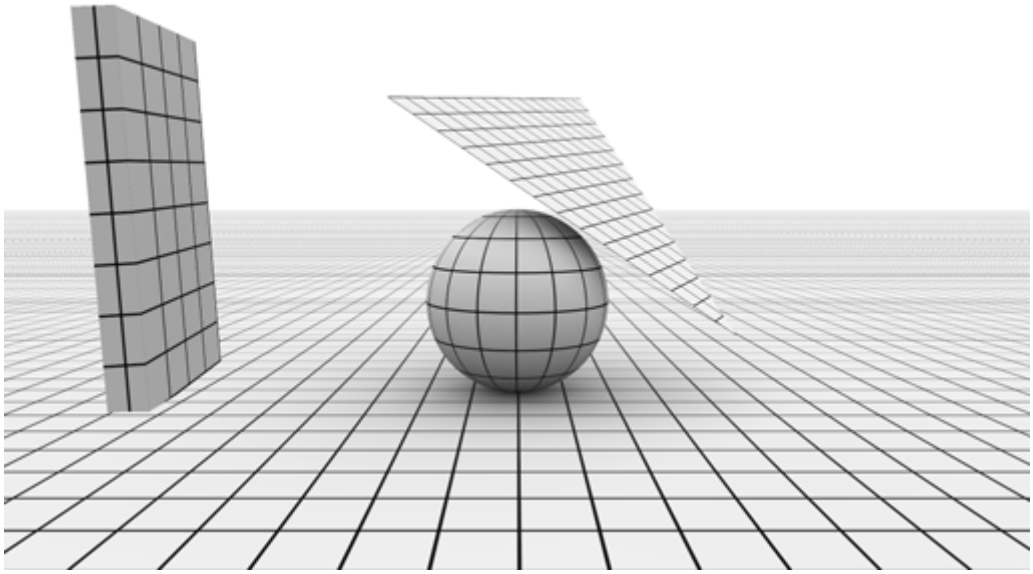
Lastly, we'll consider square roots later in its own dedicated section as well, knowing that for the most part we don't really need them (except for the sphere in this initial test).

For now that was enough to launch a first render of the plane+sphere+triangle+box test scene and tracer, and see how it would go. I also was generous for this first

test used 65 bit rationals, which is really a lot of data (similar to a "double" data type), made of 32 bit for numerator, 32 for denominator and 1 for sign. The left image is what I got with this naive implementation, and on the right the reference image:



Naive 65 bit rationals



Floating point reference

The result was pretty bad, the box and triangle didn't even show up in the render, and the sphere and floor plane were noisy. The problem was (of course) that every time my rational numbers performed any basic arithmetic operation in any of the

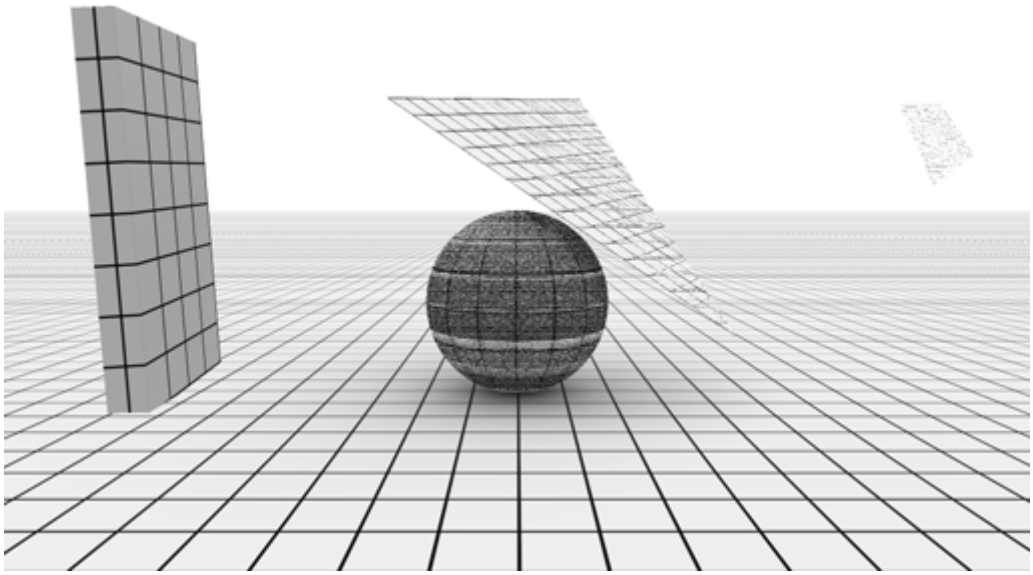
rendering algorithmic steps, the numerator and denominator were getting bigger and bigger without control due to the integer multiplications involved. Just think of the following: if our original world units were meters are we are snapping our source geometry (vertices and camera) to a millimeter accuracy, just the source data would be on the 16 bits usage already for a pretty small scene. At the same time, for a typical screen resolution of HD and 4X antialiasing, the ray direction rationals need 12 bits easily. So, as soon as the first ray and geometry interaction happend, just on the very first arithmetic instruction that combined both data inputs, the numbers would become 28 bits long - close enough to our 32 bit limit that I gave myself in this first implementation. That was even before we have completed our very first dot or cross product. By the time the cross product were completed the renderer would already be needeing rationals of hundreds of bits in order to represent the numbers. That's the worst case of course, but the average case was not far from that, and given I only had 32 bit capacity allocated for numerator and denominator, it's easy to see how things overflowed rapidly in this test - no wonder I was barely seeing anything, except for the floor plane a bit and part of the sphere.

Test 2 - GCD reduction

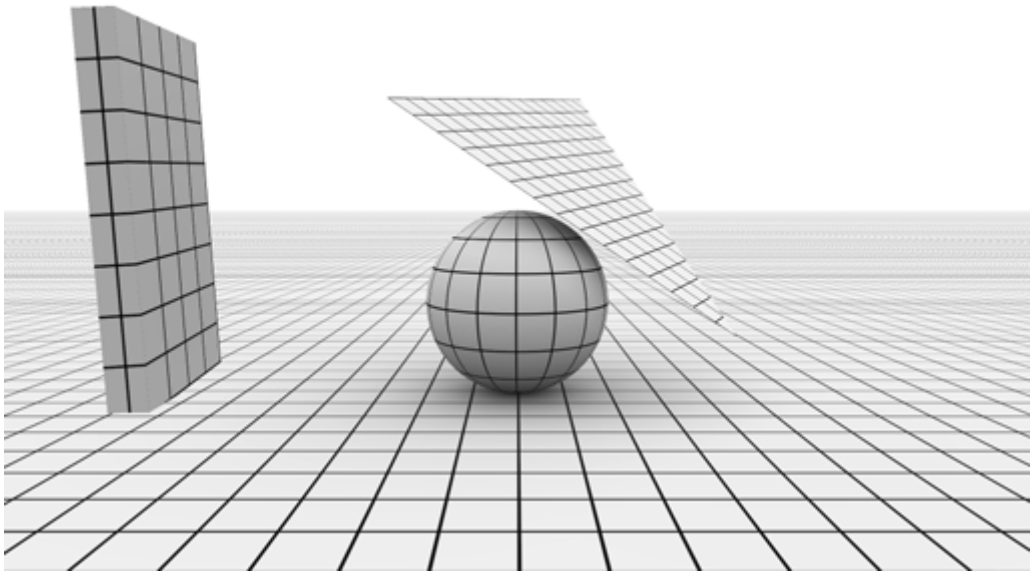
So, one thing I did is related to something I quickly mentioned earlier - different rational numbers can represent the same quantity. Indeed 6/12 is the same value as 1/2, yet it uses a lot more bits than the later. So the idea was - if after each basic arithmetic operation (or before it) I extracted all the common factors from the numerator and denominator and reduced the fraction to its simplest form, then maybe I could keep things under control and keep operating with exact arithmetic and without losing any precision for a longer time. Maybe for a long enough time to get clean rendered images? So, I'll take a small break to give one more example: 588/910 can be simplified to 42/65, since the 14 divides both 588 and 910. But 42/65 clearly needs less bits than 588/910 to be stored. Finding the biggest number possible that divides two other numbers simultaneously can be done with the Great Common Divisor (GCD) algorith, of which there are efficient forms everywhere (in my case I copied directly from the Wikipedia and sped up the thing a little by doing the bit scan step with x64 intrinsics). So, armed with the GCD, my "rational" class

should constantly keep simplifying my fractions as they were being generated during the rendering process. Now, there are two ways I could have done that:

This first one was to promote the intermediary result of the addition and multiplication operators to the next bit data type (uin64_t in the case of my current naive approach), perform the GCD in that wider data type, and then demote the result to the source bit length (32). The second approach was to analyze how **a_n**, **a_d**, **b_n** and **b_d** combine with each other in both arithmetic operators, and extract common factors across them before performing the multiplications. This second approach prevents in principle the need to go high bit lengths. Knowing that doing the later was possible if needed, I went for the former because it was easier to implement and allowed me to move faster (the night is short). With that in place, let's see what kind of render I was able to produce now:



65 bit rationals with GCD reduction



Floating point reference

Much better! Still not there of course, but this was promising. I got my box and triangle to show up and the sphere felt more solid now. There was some funny artifact on the top right of the image though, and the rationals still overflowed for plenty of the pixels producing lots of image acne. However, it's worth realizing that for some (many) of the pixels I was getting **exact** and perfect results! Meaning, the tracer found the mathematically exact intersection points and distances, which was kind of the point of trying rational numbers in the first place.

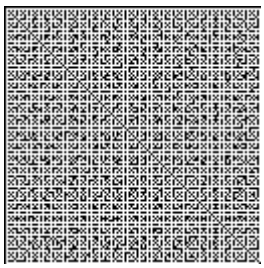
Before going on my next step in the quest of making rational numbers actually viable, I want to stop a little and share some of the findings I did regarding GCD and rational number reduction.

The first one is related to the bit usage of rational numbers. Even though I could still not render pretty images and it was more important to get that than worrying about data size optimizations, and even if this early implementation was still using a huge number of bits (1+32+32), I was already thinking about the previously mentioned wasting of bits in the form of redundant fractions. In particular, after I added the GCD step, bits combinations like 2/4 are no longer valid since they got automatically reduced to 1/2 before it got written to any register/variable. So in a way, from all the 2^{64} bit combinations that I could have with numerator and denominator, many were invalid. And that was not a good use of the bits. Or was it. How much bit space are we really wasting? I did a small detour to study this.

Detour - on coprime numbers

Below are two pictures that show the bit usage for rational numbers of 5/5 bits and 7/7 bits respectively. The horizontal and vertical axis of the graphs represent the values of the numerator and denominator of all possible rational numbers that have numerators and denominators up to 5 bits (31) and 7 bits (127) respectively. The black pixels are the unused combinations and the white pixel are valid fractions. For example, the diagonal is all black except for the 1/1 pixel, since all fractions of the form n/n will reduce to 1/1.

bit usage for 5/5 rationals



bit usage for 7/7 rationals

If you pixel count, as I did, you'll quickly realize that the proportion of useful pixels tends to 60.8% as you increase the number of bits. A bit of research online tought me that this proportion happens to be exactly $6/\pi^2$, since it's also the probability of two random numbers to be coprime (to not have common factors). What does PI have to do here, you might ask? "Six over Pi squared" happens to be the value of one over the Riemann Zeta function evalated at 2, $1/\zeta(2)$. This might be not completely surprising since the Riemann Zeta function frequencly shows up in problems where there are prime and coprime numbers involved.

In any case, it seemed I was wasting about 40% of the bit combinations in my rational representation, which while it feelt like a lot, I decided to look at it as actually being less than a bit... in which case it no longer felt super bad (show stopper bad). With that in mind, I decided to move forward with other completely different approaches instead of trying to locally optimize this one issue. I did, however, do some rapid learning on Stern-Brocot and Calkin-Wilf trees, which would

have allowed me to do full use of all the available bits, but the range of values I could get with them is really tiny, so I discarded the idea quickly and moved on. I think at this point I should formally credit Wikipedia for being a constant source of learning.

Going back to analyzing what I got so far, I was at a place where I kind of could render some broken images, but I was pretty much at the mercy of the distribution of prime numbers in the computations. Those little primes controlled when the GCD algorithm could simplify things or not - as soon as a prime number or factor made it into any of the numbers on the renderer (vectors, scalars, matrices), then it would "pollute" all numbers that followed from it in the arithmetic manipulations, and would stay there forever. So, over time, things were guaranteed to explode anyways, it was just a matter of when. While this was inevitable it was also necessary, for it's the coprime factors that carry the information for the value of a number. But at the same time, big prime numbers will screw things quickly. So there's a tension there.

One last thing to mention is that I was still using twice as many bits as a standard floating point number, for no real benefit just yet. I of course tried to use 16/16 rationals, which would have be a more fair comparison to actual requirements for floating point arithmetic, but at 16/16 precision the current numerator+denominator+GCD approach produced images that where just unrecognizable.

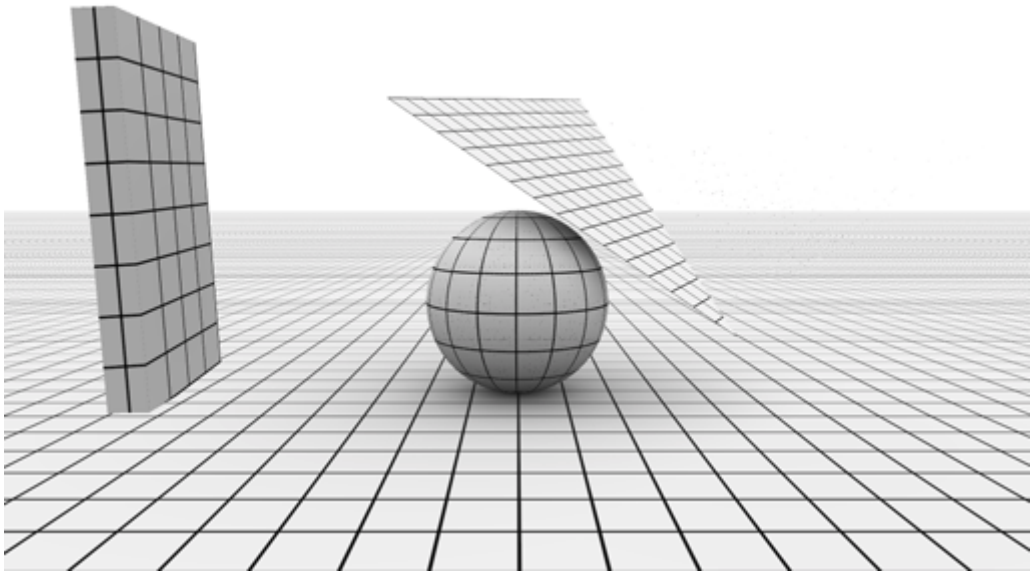
Test 3 - rational normalization

So now I was at a place where I needed to do something dramatic. It seemed like I needed to start chopping numbers and losing precision if I wanted to prevent the numbers from growing without limits. The whole exercise started from the idea of exploring **exact** rendering, but at this point I felt I had to give up on that idea and keep exploring other areas, if only for fun, and see where I would land (the original idea that kicks a research process is just that, an idea to kick a process, and often you land in totally unexpected places. Or as John Cleese once said, wrong ideas can lead you to good ideas, the creative process does not need to be a sequence or

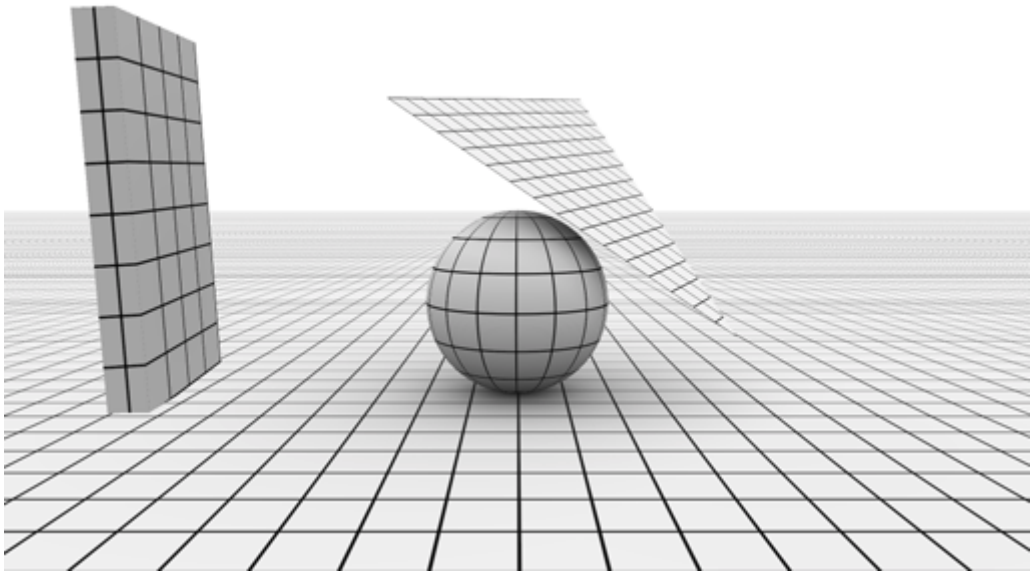
progression of logically correct steps at all time).

Anyways, I decided to see what I'd happen to the renders if I somehow managed to prevent numerator and denominator from overflowing. The simplest thing to do was to shift both numerator and denominator enough bits to the right, when necessary, until they fitted in the assigned bit space. This effectively does an integer division in both numerator and denominator by the same amount, and hence the value of the number stays **approximately** unchanged. And this is where I deviated from the original intent of the exercise.

In my first implementation I looked at the number of bits needed by the numerator and the denominator, took the maximum of both and shifted both by that amount of bits (making sure I was rounding to the nearest integer when doing so). When this was implemented both in the addition and multiplication operators, things started to look visually reasonable:



65 bit rationals with GCD reduction and normalization



Floating point reference

Since things looked pretty good, this was the point where I started to finally pay attention to the large bit usage of the current implementation. I tried 16/16 (33 bits) instead of 32/32 (65 bits), and images surprisingly came pretty good! I could still see that some of the edges in the sphere had little holes and the texture pattern in the triangle had some tiny discontinuities. But it was kind of good for something close enough to a floating point number in terms of storage. That gave me energy to keep lookin for some new ideas.

Test 4 - floating bar

At this point I decided to switch gears and stop putting excuses - if I was going to find anything interesting around using rational numbers for rendering, it better be exactly 32 bits total and no more. I rather get some good idea or stop there and call it a night (well, two nights, since this was the beginning of my second session).

My first thought was that I could stick to the GCD and renormalizations ideas, but I needed to be smarter about the storage and bit utilization. The first thing that came to mind was that, even though numerator and denominator can get big, often they

don't. Or at least not simultaneously. So, at the times the numerator is small, the denominator should be allowed to be big. And vice-versa. The unused bits of one of the two integers could be used by the other to express bigger values. Then I realized that the same way a floating point number is pretty much a fixed point format with the "fixed" point made variable, I could take my rational numbers and make the bit location of the bar of the fraction variable as well. Meaning, instead of hardcoding it to be 16/16, I could let the same 32bit variable sometimes be 16/16 but also some other times be 5/27, or 13/19, as needed.

It was worth trying. A few lines of packing/depacking code in the internal setters and getters for numerator and denominator was fast to code anyways. The most logical layout to me was 1|5|26, meaning:

1 bit : sign
5 bits: bar position (B)
26 bits: numerator and denominator data merged, numerator high 26-B bits, denominator low B bits

where the bar position (B) determines the size of the denominator. For example, the number 7/3 would in principle be

7/3 = 0 00010 00000000000000000000111 11

where the sign is 0 for "positive", the bar is "2" to indicate the the denominator (the number 3) needs 2 bits to be represented and that the rest of the bits are left for the numerator.

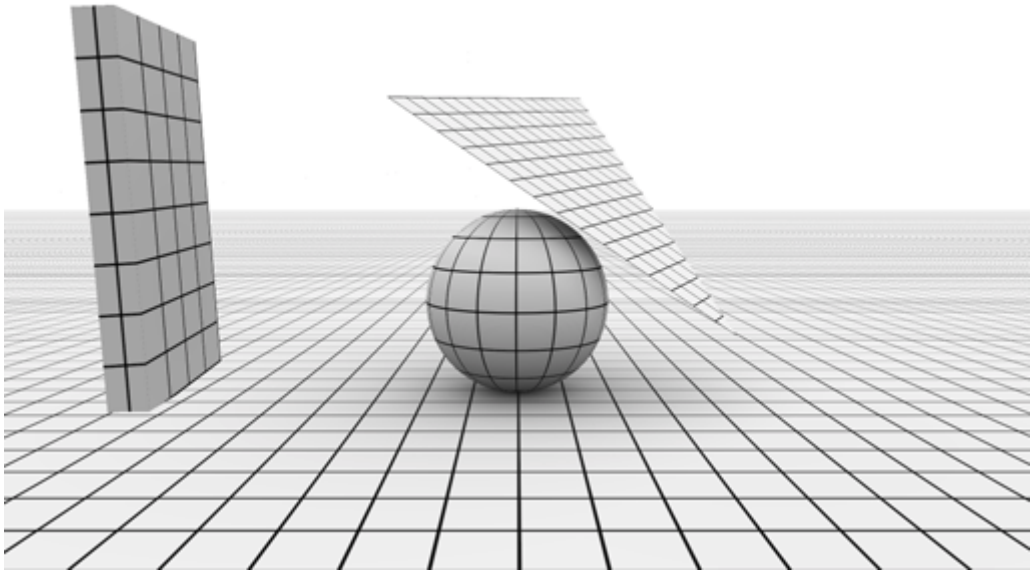
Now, the readers that have worked with the IEEE754 standard might find this observation familiar: the binary representation of the denominator will always start with "1", for the bar number will always chop it to its shortest representation. That means, that the first bit of the denominator needs not be stored. In this case, the number "3" can be represented just by the binary value "1" and a bar value of "1":

7/3 = 0 00001 00000000000000000000111 1

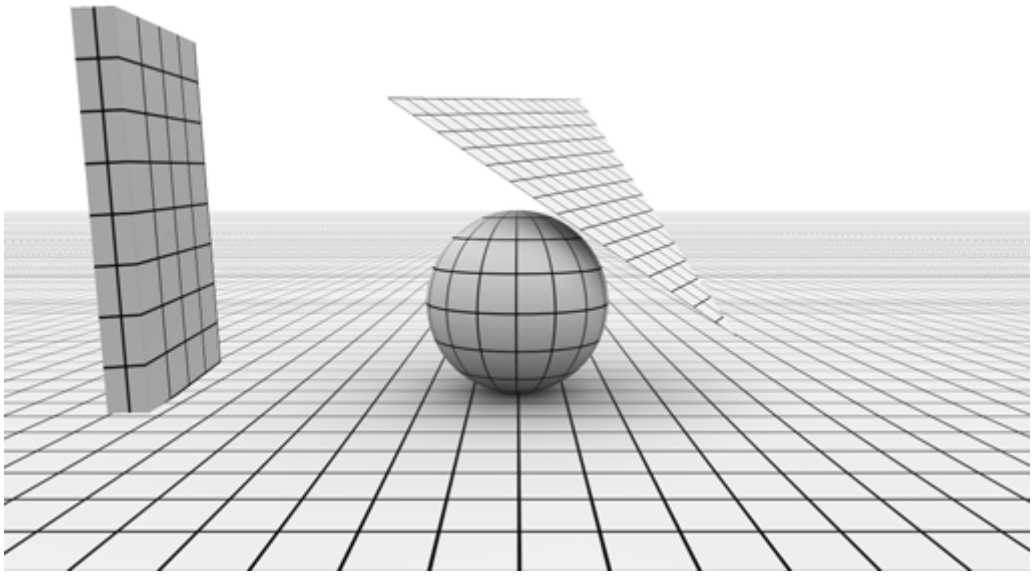
This trick not only saved me one precious bit, but it also came with a beautiful side effect: when the bar is made equal to zero, it naturally and simultaneously implies that the denominator is 1 and that no room is needed to store it. What that meant was that my rational representation of numbers was suddenty fully compatible with

regular integer representation and arithmetic, as long as the numbers stayed below 2^{26} that is, which is a pretty decent number. Such a nice surprise! So in theory I could use the exact same data type, "rational", to perform the common rendering and shading operations but also do all the logic and control flow tasks in the path tracer - I'd no longer need to have two data types like in most renderers ("int" and "float") and convert back and forth! However the clock was ticking, so I did not change all my loop indices from "int" to "rational". The night was running short and I still had multiple things to try with the quality of the renders.

Anyways, once I got the implementation it was time to actually put it to the test:



32 bit (1|5|26) floating **bar** rationals

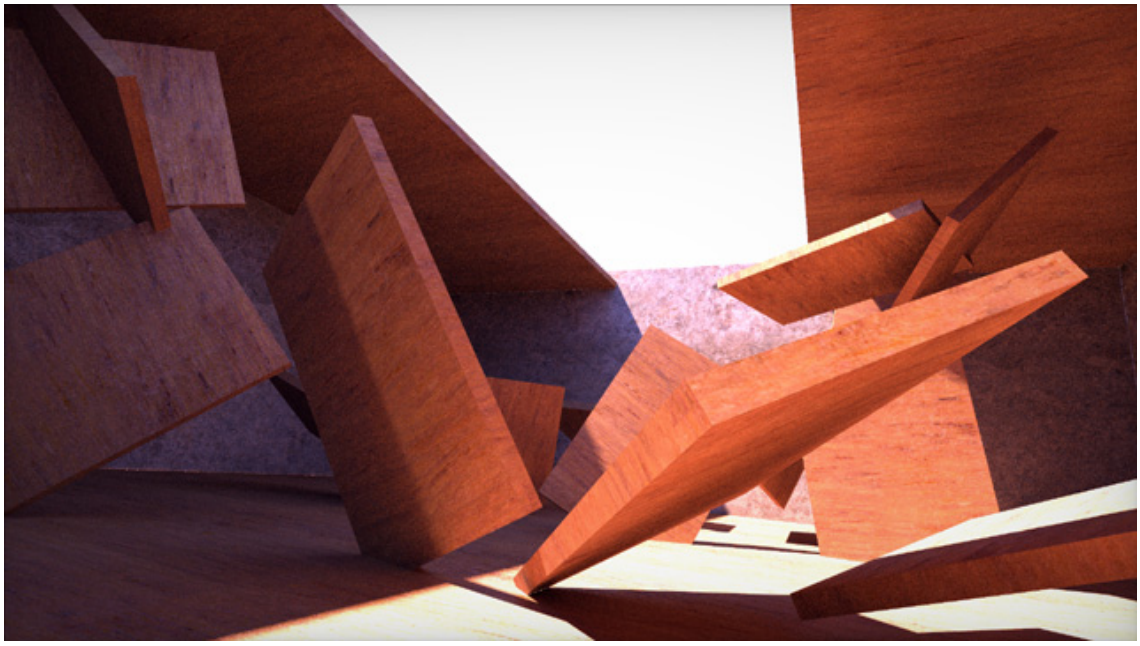


32 bit foating **point** reference

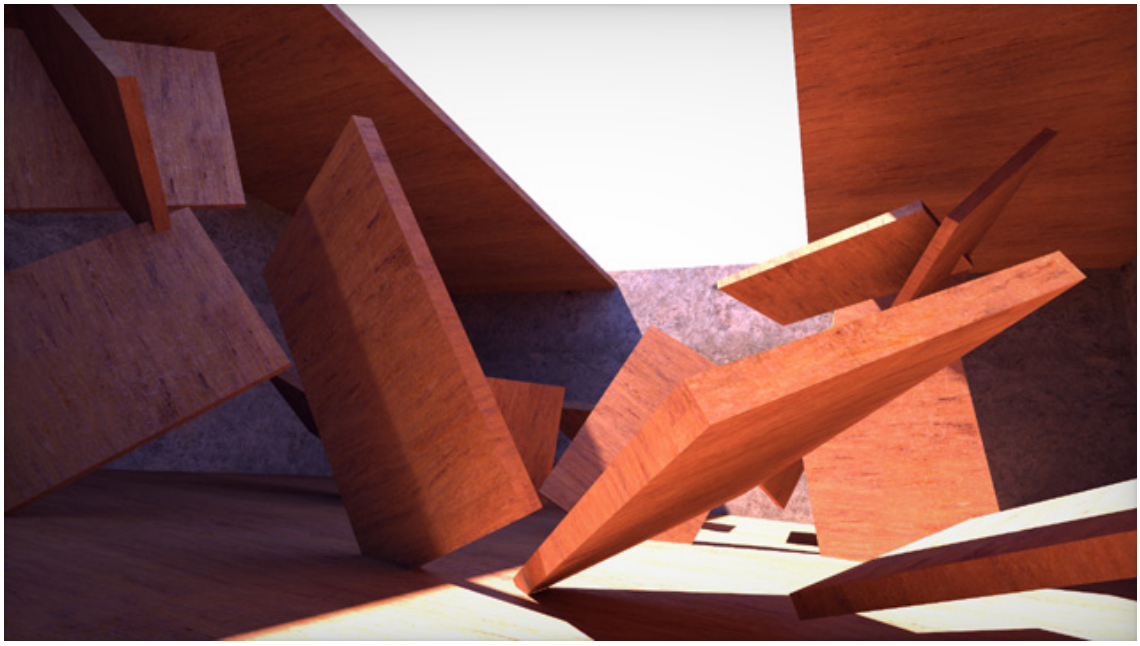
Ohhh, not bad! I still had some artifacts in the sphere which I decided I'd for now blame on my poor square root implementation, but the box and triangle were really clean. The number of exactly resolved pixels per image also went up. I think that by allowing larger numbers to exist before reaching overflows in denominator or numerator, I increased the chances of the GCD to find common factors and do reductions. So, the floating bar not only increased the range of the representable numbers and delayed overflows driven lossy normalizations, but also got an extra kick of quality improvements due to the increased chance of reductions.

At this point I was ready to try a bigger test (although still a toy - nothing close to production). I implemented a bare minimum pathtracer (not necessarily physically accurate or even physically motivated) and set a scene with some boxes and two light sources with a reference GPU implementation that you can find here: https://www.shadertoy.com/view/Xd2fzR.

I converted the scene to the C++ framework again, removed some unnecessary ray normalizations once more, and launched the render. This is what I got:

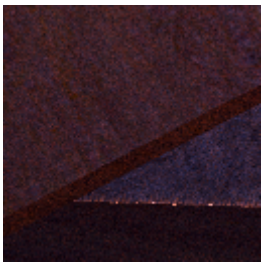
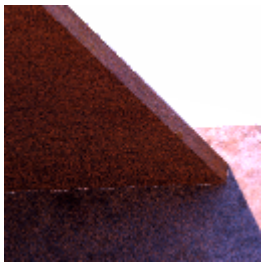
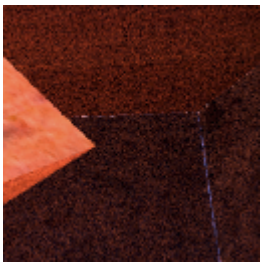


32 bit floating **bar** rationals



32 bit floating **point** reference

Okey, now this was really not bad! Although there were some clearly visible light leaks at the corners where the edges of the floor and ceiling meet. Check these close ups:

These could be due to a problem in my ray-box intersection implementation that only manifested in rationals, I wouldn't be surprised. Or maybe I was simply hitting the limits of what rationals can do. Anyway, I was pretty happy. And also I had other developments or experiments I wanted to try or think about in the short time left:

Some other experiments

Exact Arithmetic in 64 bit

The premise of exact arithmetic couldn't be fulfilled with 32 bit naive 64 bit rationals nor the 32 bit (1|5|26) floating bar rationals. But, would floating bar in 64 bits work?

I implemented 1|6|57 rationals quickly (although I had to learn some more cool x64 intrinsics for bit shifting). These 57 bits of numerator/denominator allowed for much bigger range of distances to be traced. And indeed I could trace a scene of a few triangles with all **exact** arithmetic (not the just mentioned boxes globally illuminated scene, but just a few triangles in front of the camera). This was a success! However, ironically enough, the coplanarity test that I implemented in order to test the correctness required a few dot and cross products, which made the numbers start renormalizing themselves :) So, while I knew the render was exact, I couldn't "prove it" experimentally. Heh. Well, all it means is that 64 bits was okey for a few triangles but wouldn't hold for more complex scenes anyways. However, some other thing that this made me think of was whether there is some algorithm I could have used to test coplanarity that was based on modular arithmetic instead of

absolute numbers. In modular arithmetic the rationals shouldn't explode? I had no time to explore this at all, nor I am versed in number theory.

Square roots

Square roots. This is a topic that I decided to stop a bit and learn new things this my second and last night of research. I wanted to implement the best possible square rooter for rationals. My current naive approach, which is **bad**, was take integer square root of the numerator (with proper rounding) and then do the same with the denominator. Since the square root of a fraction is the fraction of the square roots of its numerator and denominator, overall this approach returns something okeish that is not too far from the best possible answer. But it definitely does not produce the best rational approximation to the square root of a rational number. Intuitively, it is performing two approximations instead of a single one.

What I tried is this. What we are after here, really, are two integers **x** and **y** such that

Then what we can do is to rewrite this as finding the (non trivial) solution to the following diophantine equation (diophantine means that we are only interested in integer solutions):

After some online Wikipedia browsing I found this particular equation is a so called "Modified Pell's" equation or "Pell's Like" equation. There are algorithms to find the smallest values of **x** and **y** that solve the equation. Unfortunately my focus quickly shifted to other fun diophantine math and I didn't get to try implementing any of it.

More efficient reduction

In my last minutes I thought of exploring the idea of taking advantage of the multiple terms that combine together in complex geometrical operators such as the cross product. For example, the first component of a cross product was

assuming s.y=a/b, t.z=c/d, t.y=e/f, s.z=g/h

This meant that now I could try finding common factors between a and d, or e and h, for example, and use them for early reductions.

Another idea I had was that if at some point rendering speed was a concern, I could try skipping the GCD steps altogether and only implement normalization. A quick test proved that doing so still produces reasonable renders and works just fine at a much faster speed. It does come with less number of arithmetically exact results though, of course.

However, a compromise could be to not implement a generic GCD routine or circuitry, but just something mathematically simple, hardcoded and efficient that finds divisibility by 2, 3 and 5 only. While not an exaustive factorization, it should be able to catch a great amount of reductions in practice. Think that divisibility by 2 is 3 times more common than divisibility by 7, and 20 times more common than divisibility by 41!

Conclusion

After this exercise I ended up believing that there's perhaps room for a number representation that is based on rational numbers similar to what I am now calling "floating bar", a representation that is compatible with integers and can do many operations in exact arithmetic for many problems (provided the input is rational). The 64 bit version (1|6|57) can go a long way, although the 32 bit version (1|5|26)

can already produce some interesting renderings.

If this hadn't been a two nights experiment but something made in a professional context at a studio or so, next steps would have been:

* Get an histogram of number of exactly traced pixels vs not (or in other words, how often that normalization happen)
* Try the hardcoded 2, 3 and 5 factor reduction, and measure percentage of lost exact pixels
* Show pixel difference between floating point rendering and float bar rendering
* Find creative ways to use the spare values of the "bar" bit packet, probably to express inf and nan
* Implement detection of nan, inf, underflow, overflow

Overall, this was a fun exploration. There were a few surprises on the way, an small invention, and lots of learning about Pell's equation, square roots, GCD, x86_64 intrinsics, Riemann Zeta function, and some other things. I'm pretty happy with it!

inigo quilez - learning computer graphics since 1994