WARNING

This book is not for everyone! The authors use black humor, sarcasm, and exaggerations to present a very polarizing view of modern development methods that some might find offensive. It is strongly recommended to read the summary (this document) to find out whether this is entertaining or not. The code can be read for free without being exposed to the authors' acerbic writing style. You have been warned.

Real Programming Condensed

Nils Liaaen Corneliusen
Sjur Julin (editor)
9 February 2023

Introduction

Ekte Programmering, the initial Norwegian version of *Real Programming*, was published on 9 February 2021, exactly 2 years ago today. The focus of the book is programming, but not in a regular manner with boring examples, boring theory, and boring exercises. It's a different kind of book with weird illustrations and graphs and all kinds of things going to hell while attempting to solve programming problems, with a running commentary about semi-obscure music, movies, and games. On a more serious note, it attempts to explain what kind of thinking is behind doing the opposite of everybody else. Spoiler alert: Because sometimes it gives better results, and sometimes it's just fun. It's not always about the hex values, it's about the order of those hex values.

What has happened since then? A whole lot of nothing. Were we not thorough enough? Were the planets incorrectly aligned? Might it be attributed to the fact that nobody read the book? Signs point to the latter.

There are some completely new solutions to old problems in the book that we think deserve more attention. It also debunks several programming myths that still seem to stick around in 2023. We released a set of example chapters in 2021, including all source code. This short article tries to sum up all the important points from the book: A *CliffsNotes* version for the general hodge-podge of developers used to importing code instead of writing it.

The book is written in two distinct styles: Chapter 2 went through thorough fact, grammar, style, and spell checking. The rest of the book is mainly based on articles I wrote in the period 2009-2021. Sjur had the unenvious job of ironing out all the wrinkles, testing the code, removing (the worst) insults, and translating English to Norwegian to English. That's about as wise as it sounds. Feel free to call it all hogwash, except chapter 2 and the code.

Bear in mind that the views expressed in the book are based on a polar world view: A piece of code is either the greatest thing since *The Matrix*, or it's the worst crap since *The Matrix 2*. Let's not mention the third one. It has been brought to my attention that they made a fourth movie which won't be mentioned at all, for obvious reasons. They also made 10 *Hellraiser* movies. Three of them were watchable. Both Sjur and I agree on the undeniable fact that *Hellraiser 8: Hellworld* is the worst of the lot. Even Lance Henriksen couldn't save that one.

As hinted at above, a recurring theme in the book is semi-obscure pop-culture references. A few are found in the example chapters. I'll try to refrain from referring to them here. Another recurring theme is the incompetence of managers. It's amazing how realistic the *Dilbert* comic seems to be when serving time in a large corporation.

Modern Brutalism

The first two chapters present a full-frontal assault on modern development methods. The short chapter 1: *Introduction* is available here:

https://www.ignorantus.com/books/real/Real_Programming_ch01.pdf

The modern programmer is referred to as a career programmer: Somebody who sees programming as a job and not a fascinating lifestyle. The career programmers start using computers at the University and don't

write code in their spare time. This leads to modern development methods suited to that kind of programmers. So-called "programming patterns" are introduced to fit problems into standard solutions to avoid original thinking. The result is that everything ends up being average: The code, the speed, the programmers, the managers, and the salary. The upside: The bad programmers turn average. Unfortunately, this also makes the good programmers average.

Chapter 2: *A Rant About CPUs, Languages, and Development Methods* starts off with a history of CPUs. In short: What has happened since the Z80 and 6502 revolutionized the world in the 70s? Not much. The modern CPUs are bit faster and wider, but that's about it. Unfortunately, modern programming languages seem to move further and further away from how CPUs work.

Notable CPU features are presented: The 68000 and 32-bit architectures, Tilera's TILE-Gx with a mesh network between the cores, hyper-threading, SIMD architectures, VLIW, and conditionals. There's also a look at *The Mill*[1], the first breath of fresh air in the CPU world in a long time. Too bad it's not available yet. AMD's race to the bottom and recovery is covered: Now eclipsed by Intel's race to the bottom. I knew it was coming, it was just a matter of time after a certain earlier CEO started firing engineering staff. Let's not get into that: It's water under the bridge (or corpses in the lake).

Modern programming languages receive a thorough review, though none are mentioned by name. For once, the following statement is true: What is true for one is true for all. The irony of language naming is totally missed: Why call a language *Swift* when it's not, or *Python* when *Sloth* would have been more appropriate? Is it Opposite Day already?

The popular custom of blaming what's referred to as "memory errors" on programming languages is dismantled. The truth is that the modern programmers are to blame. Examples attempt to show this in a simple manner. Modern development methods receive the same treatment.

Meanwhile, the only language well suited to the CPUs of the day, C, is still going strong. Some guidelines for writing good C are defined. They are just that: Guidelines. Enforced programming standards receive flak multiple places. The introduction of intrinsics drastically reduces the need to write inline Assembler. C is still as well-defined as it always was, except for a set of blunders from the standardization ~~morons~~ committee. That is a topic for a different article.

Contrary to widespread belief, there is a growing need for C and Assembler programmers. Every little device in your home made in the last 10 years uses a microcontroller of some kind. Obviously, the microcontroller in a lightbulb can't be programmed in a language that needs 28 bytes to store an integer. They must be programmed in C and Assembler due to extreme resource constraints. And, in the case of smoke and fire detectors to mention a few, extreme response times. The number of kids who start out by writing Z80 assembly language before the age of 10 is dwindling. The universities teaching C and Assembler programming are few and far between.

Super Happy Fun Club

Chapter 3: *A Look Back on the Age of Amiga* (1985-1991), chapter 20: *Restoration* (2019), and chapter 21: *A Jilted Generation* (2019) cover the period of the Commodore Amiga and attempts to recover some old demos and intros we made. I meddled in raytracers and MPEG-encoded video back then, too. Chapter 21 is available here:

https://www.ignorantus.com/books/real/Real_Programming_ch21.pdf

[1] https://millcomputing.com/

The period was a like a second hacker revolution, after the first one in the seventies. I still have a hope for a third one, but it doesn't seem like it's gonna happen any time soon. Chapters 20 and 21 uses code presented in the other chapters to upscale and denoise old Amiga video files and recover video from an old VHS tape. We also describe the fun of getting an old Amiga 4000 to boot again.

Encryptionists

Chapter 4: *Let's Encrypt!* (2010) covers how to optimize AES encryption in counter mode (CTR): Since only one byte changes each block, parts of the XOR-grid can be precalculated every 4096 bytes to reduce the execution time by 30%. This is quite interesting for early implementations of SRTP based on RFC 3711. It's also a gigantic security hole, which is why nobody should use CTR mode anymore.

2010 was the year that people started noticing the major blunders in *OpenSSL*. It's covered. The importance of *builtins* in gcc is discussed, for those lucky enough to use a CPU that doesn't have a branch predictor.

Chapter 6: *Let's Encrypt Again!* (2011) discusses how to do fast AES encryption on the Tilera TILE-Gx, using new instructions that does table lookups quickly. Scheduling of instructions and instruction classes on the TILE-Gx are reviewed.

Chapter 12: *Let's Encrypt Again! The Last Round* (2015) wraps up the encryption theme with a look at how the last AES round is done in *OpenSSL*. That part of the code has a completely unique style. Why? I have no idea. By observing that what it does is always pick out values from equal pairs in the encryption tables, it can be done much quicker given that the CPU has interleave and/or masked merge instructions.

Quaker State

Chapter 5: *A Quake! Doom is coming!* (2010) presents my first foray into multi-CPU work. The Tilera TILE-Gx with 36 cores is the target for running one instance per core of *Doom* or *Quake*, referred to as *MultiDoom* and *MultiQuake*. The supplied code uses the Tilera TMC library for parallelization. A resounding success, I sent a picture of 30 Dooms running on a monitor to John Carmack. He liked it. Unfortunately, I lost the mail. Should have printed it out and framed it. Damn.

The chapter also looks at *EPX* scaling. The example code from Wikipedia[2] is dismantled and reordered so it is possible to figure out what it does directly from the code. I suspect the mantra was "it was hard to write, so it should be hard to read".

Chapter 10: *Parallel Lines and Libraries* (2014) expands on this theme. The parallelization theme, that is. While MultiDoom and MultiQuake used processes and shared memory, this chapter uses threads and the Tilera mesh network for communication. It's blisteringly fast.

I looked around on the internet for a simple parallelization library. To cut a long story short: I present a new parallelization library for the TILE-Gx that's 160 lines of C code including comments. For testing, I made a simple bilinear scaler that divides the picture among the cores by splitting it into blocks of lines. This turns out to be very inefficient since no cache is shared and L2 cache is limited. Graphs show that it sucks. More about that later.

Fatty Owls

Chapter 7: *Research is Hard* (2012) is a more serious one. In 2012, Beastie Boys released their final album (hmm, sorry, I was supposed to skip that), and a herd (or flock?) of researchers from MIT et al presented a new programming language thingy called *Halide*[3]. The paper claimed:

[2] https://en.wikipedia.org/wiki/Pixel-art_scaling_algorithms
[3] https://people.csail.mit.edu/jrk/halide12/halide12.pdf

Using existing programming tools, writing high-performance image processing code requires sacrificing readability, portability, and modularity.

We prove that the statement is pure baloney. In fact, the entire paper is torn to pieces. They deliberately formatted their C code badly and used old hardware to make their numbers look good. I call that cheating. Read it here:

https://www.ignorantus.com/books/real/Real_Programming_ch07.pdf

This is a recurring theme in the book: Readable code is always fast code. The converse might not always be true. All code presented in the book is fast, readable, and succinct.

Roomy

Chapter 8: *Spaces and Colors and Layout* (2012-2013) presents a method for color space conversion on Intel SSE2. The new idea at the time was to use more multipliers than needed to avoid extra interleave and pack instructions. SSE2 multipliers are plentiful and cheap.

A similar thing is done on the TILE-Gx with a different color conversion. Like I say time and time again: Always look at the generated code. The first implementation is compiled, and the code produced by *gcc* is saved, converted to readable form with a provided *awk* script and analyzed. The observation made is that the compiler refuses to use the fast *ld_add* and *st_add* instructions, instead opting for separate ones. This makes sense in normal code, but not in intrinsics heavy code. A faster version that uses the correct intrinsics is the result. Cycle timings are provided.

Modern programmers should take note of awk. It's a scripting language that looks like C with a collection of useful string extensions. This makes it extremely simple to write text parsing code if you already know C. Awk saw the light of day in 1977. The real version of awk (not *gawk*, jeez) got an update last year from Brian Kernighan. If you don't know who that is, look it up.

Chapter 11: *Snappy Names are Important* (2014) takes a stab at two image processing operations. Consider how to apply a Scharr filter, a variation of a Sobel filter. Look that up on the internet. Possibly find an implementation in *OpenCV* or similar libraries. It's gonna look horrible. Here's the news that everybody seems to have missed: If the CPU has an 8-bit 4-way dot product instruction, or possibly an 8-bit 4-way dual dot product instruction, all the calculations can be done extremely efficiently by just zeroing 1 factor and sliding the data along. The TILE-Gx has dual dots and an instruction to slide bytes across 2 64-bit words. Neat.

Next up was calculation of the average value of an area of bytes. Normally, you end up with two *for* loops fetching and adding bytes as you go along. That is hard to optimize, since the coords may be anything. Areas may range from 1x1 to huge. One compiler tried to make multiple versions of the loops. Close, but no cigar. The point is: Do not optimize for worst case if the worst case is *small*. Make sure the inner loop is as fast as possible and cover the special cases outside that loop. While it may look like a regular wrap-up problem, the method presented is unique: On ARM NEON, mask out what's outside the area on the left and right edges and do everything fast in-between. Fold the left and right edges into one if the area is small. It's gonna be ridiculously fast on larger areas. And there are no *if* statements in there.

Some tidbits: The concept of *drive-by optimization* is defined, and *OpenCL* gets some shots across the bow. They had it coming. ARM gets flak for their hopelessly named *vreinterpret* crap. Occasionally they screw up things, too.

Separation

Chapter 9: *Separate Ways and Separate Filters* (2013) presents a new method to evaluate separable image filters. Instead of making each pass separate with the horizontal suffering a big performance hit from unaligned loads, they're both similar. How? By rotating the data before the horizontal pass. The astute reader should notice that this would literally suck, since rotating data is expensive.

Consider a 4-tap filter where 8 rows are processed at a time. 8 rows were optimal back then, since CPUs usually had 16KB L1 cache, fitting 8 rows of 1920 8-bit pixels. The task can be simplified by interleaving the data while writing them:

row 0, 0...7	row 1, 0...7	row 2, 0...7	row 3, 0...7	row 4, 0...7	row 5, 0...7	row 6, 0...7	row 7, 0...7
row 0, 8...15	row 1, 8...15	row 2, 8...15	row 3, 8...15	row 4, 8...15	row 5, 8...15	row 6, 8...15	row 7, 8...15

Then the process of rotating it is a transpose, which is much cheaper. ARM NEON has some tricks up its sleeve for that purpose since a full transpose is not needed. It's enough to have the column data on the correct row. The necessary ARM instructions are:

```
    vld4.8  {d0-d3},   [%[src]:64]!
    vld4.8  {d4-d7},   [%[src]:64]!
    vtrn.8  q0, q2
    vtrn.8  q1, q3
    vst1.64 {d0-d3},   [%[dst]:64]!
    vst1.64 {d4-d7},   [%[dst]:64]!
    // ...
```

```
Input                       Output
00 01 02 03 04 05 06 07     00 40 10 50 20 60 30 70
10 11 12 13 14 15 16 17     01 41 11 51 21 61 31 71
20 21 22 23 24 25 26 27     02 42 12 52 22 62 32 72
30 31 32 33 34 35 36 37     03 43 13 53 23 63 33 73
40 41 42 43 44 45 46 47     04 44 14 54 24 64 34 74
50 51 52 53 54 55 56 57     05 45 15 55 25 65 35 75
60 61 62 63 64 65 66 67     06 46 16 56 26 66 36 76
70 71 72 73 74 75 76 77     07 47 17 57 27 67 37 77
```

Obviously, that means the coefficients need reordering, but that's not particularly hard. Solutions for doing this on both ARM NEON and Intel SSE are provided in the book. The NEON solution is 100% inline assembler. I had the less than stellar idea that only parts needed to be written in Assembler. I was wrong. The version in book has been reformatted for readability.

It should be noted that ARM has moved away from multiple load/store instructions and vzip/vtrn in newer revisions. A wise decision. Scheduling those instructions is not for the faint of heart.

The chapter presents a method for calculating Lanczos-2 filter coefficients and a method for distributing the error that will never fail. This is where modern programmers would demand unit tests, whereas real programmers would wisely ask why. So, unit tests get some flak. There are also nifty tricks for fooling the optimizer into making slightly less bad code for Intel SSE2: There are so few SIMD registers that it's needed. Crap. This is much simpler on the TILE-Gx where there are 64 64-bit registers. It should be noted that modern NEON implementations have 32 registers. Thanks. If Intel could shape up, that would be nice. AMD must do it, I guess.

General scalers are fun, but specific ones are faster. Just for the heck of it, a method is presented for doing 8/15 scaling, also known as 1920x1080 to 1024x576 on the TILE-Gx, complete with drawings of what the heck is going on. Everything is simple when it's broken down to its essence.

Another nugget from the pile of code is an alternative way to calculate vertical dot products on the TILE-Gx. There are only 4 8-bit multipliers for normal use, so reshuffling the data so dual dot products can be used is significantly quicker. This triggered a rewrite of a set of central math libraries.

The Integrated Tegra

Chapter 13: *A GPU and Some Fractals* (2016) presents my first attempt at GPU programming. Contrary to widespread belief, GPUs are exceedingly simple. Units of cores execute the same instruction, they share a fast memory block, and registers are allocated from a register block. GPUs are not bogged down by MMUs or other crap like that, so if memory isn't touched, calculating how fast the code will run is easy. From a scheduling perspective, a (modern) GPU will never have an instruction stall unless the code is totally useless. Functions and such constructs are just for show. There is no stack either. It's always cute when C++ die-hards attempt to write GLSL code for the first time. And second. The list goes on.

Another caveat, or feature as I like to call it, is the absence of standardized floating point. IEEE 754 fans should look elsewhere. On a GPU, division by zero is ok and the square root of negative numbers is too. The answers do not matter if they are well-defined: It should always give the same answer. Whether it's negative or positive infinity or NaN or even zero I don't give a rabbit's lucky foot about. GPU code is fast and brutal. If a single pixel is an image is wrong for a fraction of a second, nobody cares. This sets the stage for creative optimizations.

The Nvidia Tegra line had a rough start. The early Tegras were underperforming. Then along came the X1 and all was well: A 256-core GPU and 4 useful ARM cores.

The chapter presents a new kind of brute-force fractal shader: Instead of wasting time checking if each step has passed the threshold, evaluate blocks of 32 depths at a time. Store the threshold comparison results (which reduce to the sign bits from subtractions) as a bitmask and use the *findLSB* function (which, obviously, uses the *BTFL* instruction) to determine the cut-off. Simple. We're not just banging rocks together here. If the entire screen is at max depth, the frame rate should not drop below 60.

The paper *Adaptive Parallel Computation with CUDA Dynamic Parallelism*[4] is discussed. It presents a neat fractal computation method that subdivides the image and checks the areas for equal border values. Interesting, but the CUDA implementation presented in the paper went too far. I'd guess just splitting the image into 16x16 blocks would yield better results. Anyway, interesting stuff.

The chapter also presents two new methods for converting HSV to RGB, and, by reduction, a value in the range 0 to 1 to the edge of a HSV hexagonal (it's not a circle, you dimwits). That's particularly useful for rendering fractals in color. Do a search for it and be amazed at the junk you will find[5]. Arrays of *if* statements, gigantic *switch* statements, *if* statements replaced by *mix* and *fmod*, the list goes on. My method is better: It puts a table in shared memory and uses 5 instructions and 2 shared loads at a cost of 2 cycles each. I made another one based on work by Trey Reynolds that uses saturated sines. It's nifty. Saves some memory and is almost correct most of the time.

Some other fractals like *Burning Ship* are looked at. Always burning, never producing anything interesting beyond the start picture. Thorn fractals are fun but needs to be rendered in higher resolution and downscaled to avoid artifacts. Interleaving the subpixel calculations made that significantly quicker.

[4] https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

[5] https://stackoverflow.com/questions/3018313/algorithm-to-convert-rgb-to-hsv-and-hsv-to-rgb-in-range-0-255-for-both

The 3-Dimensional World is Not Enough

Chapter 14: *Julia Quaternions. Err, What?* (2016) discusses Julia quaternion shaders (duh). Management summary: 3-dimensional slices are extracted from a 4-dimensional Julia fractal and rendered on a 2-dimensional screen. Close enough.

The brilliant Keenan Crane published such a (Cg) shader back in 2012[6]. I recirculated that for assorted X1 real-time videos in 2016. Obviously, I had to tinker with it since it didn't perform as well as needed, and because programming is fun (repeat after me: Programming is f-u-n…)

Looking at the generated Assembler code, it was obvious that something was going terribly wrong. Bear in mind that this is still going wrong in 2023. Math functions called from a GLSL shader won't get reordered with respect to surrounding code! That opened a can of worms. Any math function call would possibly lead to unnecessary stalls. That means reimplementing all the math functions that the shader uses. Crap. Eliminating all the stalls made the code significantly quicker.

There is a minor bug in Dr. Crane's code that I fixed. The iterations count is the same for normalization and interpolation, and that's just wrong. It's not an issue at low detail levels, which is why I guess it slipped by.

With some help from the wise Paul Bourke's musings about quaternions[7], I got it to run in 60 fps on the X1. That's fun, but it doesn't stop there. Obviously, the damn thing must rotate too. While Bourke describes how to rotate a Julia quaternion in 4 dimensions, the math is… computationally advanced. Instead, I devised a way to do it in 3 dimensions so the things can rotate without a GTX 4090 card (or 1080 back then). Like all such devices that I hack together with no clue about advanced math, it relies on dubious constants that are almost, but not quite, close to a factor of pi.

Given that the renderer is by definition a raytracer (and it's not really a 3D slice, but a sphere), it's trivial to project the snakes on a plane… sorry, quaternions on a plane. Who proofreads this stuff? Tried that, but it looks like a flat lump of clay most of time. It was fun, though.

Chapter 19: *Back for the Quaternion Attack* (2019) goes even further. A CPU-based version is presented that uses the multiple threads to render 16x16 blocks of pixels. Detail level was increased a lot without too many artifacts, skilfully avoiding Bourke's theory that would it be hard to do it without increased noise. Also, rendering in 4k and downscaling helps.

The code presented is for, hold your breath, Windows. Windows has done multiple things right. The Win32 API is as good as it gets. Spawning threads is simple and it will by default distribute them on physical cores first, then logical cores. Less work, better results. And the real Visual Studio is a great editor with an amazing debugger. I'm learning to despise VSCode on Linux a bit more each day.

A Ray of Sunshine

Chapter 15: *The GPU Raytracer* (2017) returns to the GPU roots. The chapter is available here:

https://www.ignorantus.com/books/real/Real_Programming_ch15.pdf

To sum up: After the discovery of the math library reordering problem, it was time to tackle GLSL unrolling. GLSL had, and still has, a second (or was it third?) catch: It will unroll all loops as far as it can… except if there's *break* statement in there. Then it will not unroll at all. That is quite terrible for a raytracer.

Raytracers are inherently simple. Just think about how you see with your eyes and reverse it. Job done. To do it fast on an X1 is an interesting prospect: Rendering 128 spheres required nifty optimizations. So: Keep

[6] https://www.cs.cmu.edu/~kmcrane/Projects/QuaternionJulia/
[7] http://paulbourke.net/fractals/quatjulia/

all spheres in front of the camera. That reduces the second-degree sphere hit formula to just half and kills off an unnecessary *if*. Just brute forcing it like that gives 80 spheres at 1080p60. Not enough.

A GPU starts jobs in symmetrical blocks, so rays usually go in the same direction. By defining a minimum sphere size, the idle CPU cores can be used to evaluate the corners of each block for hits. This generates a map the GPU can use to eliminate large swathes of code if an entire block is either just a background hit, a sphere hit, a floor hit, or a total miss.

Another unique piece of code not seen anywhere else: By evaluating 4 spheres at once, we need to decide which are hits. A quadruple *if* will be expensive. Use subtractions and *greaterThan* to eliminate the negative answers. Order them using *min*, but that's not possible since the compiler is running out of registers. Instead, invert the values and tests. Looks expensive but turns out to be cheaper. There's a fully loaded bag of such tricks in there.

Obviously, it's not a raytracer without shadows. The shadow loop is pretty much the same as the raytracing loop, except it's simpler. Both loops must be unrolled manually, meaning that the entire shader ended up as 472 lines of very repetitive code.

Another observation discussed in the book: What does a GPU do when it runs out of registers and there's no stack to spill registers to? It spills to… registers! Registers are allocated in quads of 32-bit registers. Unused single 32-bit slots are reused, meaning swizzles and the like will make no sense anymore. If that starts happening, reconsider your life choices. Again: Always look at the generated code!

Chapter 16: *The Binary Fixed-Point Raytracer* (2017) discusses how to do raytracing on the Tilera TILE-Gx. It has limited floating-point support, so binary fixed-point is the way to go. Having already written a parallelization library, I used that to distribute 16x16 rendering block jobs and evaluating corners and such. Fixed point numbers receive wide coverage over several pages, and why you should do it yourself. All the implementations I found on the net suffered from code-itis and/or cared about precision. The kicker was that nobody had any idea about how to quickly calculate inverse square roots[8]. In 2017? Come on! John Carmack popularized the method back in the Quake days.

For those that don't know how floating-point works, I have a thick book about it on the bookshelf. I also present a 4-line crash course in that chapter that covers everything the book does.

The mesh network is very efficient: I calculated that just 3.7% of the time was wasted on waiting/receiving messages and frame synchronization.

A couple of useful findings:

> *Corneliusen's fixed-point square root theorem: If floating point numbers can be converted to and from fixed-point quickly, square roots can be calculated using normal integer operations on the floats.*

That is based on Chris Hecker's work[9]:

> *Chris Hecker's float conversion theorem: If the range of the floating-point number is known, float add enough so the exponent is fixed, extract the mantissa and integer subtract. Shift the result as needed.*

[8] https://en.wikipedia.org/wiki/Fast_inverse_square_root
[9] https://chrishecker.com/Miscellaneous_Technical_Articles

The code controlling the raytracer is documented. It's mainly sines. There's also a discussion about some early attempts to make a raytracer for the new AArch64 NEON. A couple of intrinsics and Assembler examples are provided. This was in the early AArch64 days, so I made a *fetchadd* using the new load/store exclusive instructions. I really like that architecture.

We got a prototype board of the TILE-Gx72 with 72 cores in early 2017. It could easily trace twice the number of spheres with 16.67% lower clock speed. Cool.

It's a Bayesian World

Chapter 18: *Bayer, Bayer & Bayer* (2018) presents a fast method for converting raw Bayer to YUV420 using Intel AVX2. I can think of a reason for doing just that: Shovelling raw sensor images into a hardware encoder. The code is fast, but not precise. There's also a rant about the wiseness of implementing 256-bit vectors as 2x128 bits. Conclusion: It sucks. And some thoughts about why there are only 16 registers in AVX2 mode, while AVX2 in AVX-512 mode has 32. Intel occasionally does some things right: The instruction called *Maddubs* (I'm not joking) was designed for this kind of conversion.

On the other hand, I predicted the death of AVX-512 in the Norwegian version, and retracted that in chapter 23 of the English version available here:

https://www.ignorantus.com/books/real/Real_Programming_ch23.pdf

Now it's looking dead again, with Intel pushing crap "efficiency" cores without AVX-512, so the crap "performance" cores have it disabled. The mind boggles. Can I have SSE2 with 32 registers instead, please? ARM's new NEON has 32.

A Daylight Robbery

Chapter 17: *The Rip-Off Artist* (2017) discusses the origin of the legendary Xmas Demo. The 2017 version was made in the sweatshop that employed me back then. They kicked me out early the year after. I cannot fathom why.

The job was simple: Try to get some complex shaders from ShaderToy to run on the Tegra X2 platform for demo purposes and try to figure out what the hell was wrong with 2 of the ARM cores in there.

Last things first: 2 of the 6 ARM cores uses some *Transmeta* technology to translate the ARM code to whatever those cores use. The problem is that the translator runs as a native hypervisor: Everything locks up while it is executing, sometimes for milliseconds. Certain kinds of code make it suck even more: Combining floating-point and NEON isn't a big hit. It also has trouble executing tight loops of *fma* and the like. Graphs were made and are presented in the book. However, this investigation was never completed due to unforeseen managerial issues.

Back to the Xmas Demo. The X2 GPU has 256 cores. A long and tedious calculation gives just 2674 cycles per pixel per 1080p frame. That means using (almost) no textures since memory access time cannot be predicted precisely. The raytracer and Julia quaternions were recycled, and I found 8 cool shaders on ShaderToy that probably could be made to fit into the puny GPU. The ShaderToy reference board at the time seemed to be a GTX 1060 with 4.3 TFLOPS. I had 332 GFLOPS on the X2. Crap. Anyway, here are the central discoveries:

A set of them use nested hash functions to generate semi-random data. That's expensive. Replace with simple sines. One required recurring random data, so just precalculate a long enough table on the CPU and put it in shared shader memory.

A method call *ray marching* is used to render several of them. Look it up. It's like raytracing, only you step along the line in increments and estimate the distance to the cool stuff. Adjusting the increment related to

distance made those shaders much faster. Granted, there would be a loss of detail either further in or out, but nobody minds that at 60 fps.

Simulating a sea is hard. Extremely hard. I lucked upon a combination of sines that gave almost the same result as the original. 2 sines and 5 ops per pixel instead of an array of hash functions with *floors* and *fracts* and whatnot.

One of the shaders uses *atan2* extensively. Atan is an expensive operation, made worse by the fact that it won't get reordered, as mentioned earlier. Fine. An article called *Full Quadrant Approximations for the Arctangent Function*[10] published by IEEE has the necessary code. Convert that to shader code, and then reduce the number of calls. Turns out nobody uses atan to calculate the angle of a point, but to find out whether things are in the front or the back (or left-right, up-down, the list goes on). Which is why you shouldn't let mathematicians write shader code. They tend to have no concept of the cost.

This triggered an interesting turn of events. The IEEE guys claimed they wrote a 4-way SSE version but didn't present any code and their performance numbers weren't stellar. Hmm, probably because the thing would stall just as much as a floating-point version. The only fix is to go wider: Calculate 8 in pairs of 4 for less stalling. I made such a thing for ARM NEON that performed quite well and was named *satan2_8*. How often do you get a chance to call a function satan? It was epic.

Meanwhile, at the sweatshop, a bright colleague named Knut Inge Hvidsten produced a quicker but slightly less precise method using so-called taxicab angles[11]. It's as cool as it sounds.

I also pulled numerous tricks in the 2000 lines of control code. I still have no idea what *glBindTexture* does, but it is very important to have it in the right places. While the demo might seem to show two different shaders at once, it never does. There are only 256 cores in blocks of 64, so having multiple shaders active at the same time isn't a great idea.

Sjur and I have made newer iterations of the Xmas Demo that uses his excellent V73D library. The control code has shrunk to well below 1000 lines. The latest version from 2022 runs on the Nvidia Xavier with 2048 GPU cores and uses newer and better shaders. And I recirculated the Julia quaternion shader for the last time, this time with more details and a better coloring scheme. For once, we got almost all the timing correct. That doesn't happen very often. I predict a crap Xmas Demo this year.

https://www.ignorantus.com/pages/xmasdemo2022/

The Future Bites

That sums up all the chapters. I look forward to the next binary jubilee of Real Programming in 2025.

When was the last time you wrote a piece of software that didn't depend on other people's code? Try writing some new code from scratch. Maybe you'll have some fun, and maybe it'll be better than what would normally be imported.

[10] https://ieeexplore.ieee.org/document/6375931
[11] https://www.linkedin.com/pulse/angles-atan2-taxicabs-knut-inge-hvidsten

Article Licensing Information

About the Author

Corneliusen grew up with Z80s and 6502s. He learned Assembler programming on Amiga computers in the 80s. A self-proclaimed hacker, he attended the University of Oslo in the 90s, ending up with a frameable piece of paper. After serving 25 years in some large (and some small) corporations, he started his own company in 2021. His coding style is described as succinct, while his elusive lectures are known to be both dry and terse. Co-author (with Sjur Julin) of the ~~wildly popular~~ book *Real Programming* which has been described as the polar opposite of all other books about programming.

Author's website: https://www.ignorantus.com/

More information about *Real Programming*: https://www.ignorantus.com/books/real/

Revision History

9 February 2023: Initial release.
13 February 2023: Footnotes added. Minor corrections.
9 March 2023: Fixed broken footnote.
18 March 2023: Add warning.