**Clean Code, Horrible performance Rust edition**

[[Amine Dirhoussi](https://medium.com/@aminedirhoussi1?source=post_page-----abf794a30e95--------------------------------)](https://medium.com/@aminedirhoussi1?source=post_page-----abf794a30e95--------------------------------)

[Amine Dirhoussi](https://medium.com/@aminedirhoussi1?source=post_page-----abf794a30e95--------------------------------)

·

Follow

4 min read

·

Oct 7

259

4

**Clean Code, Horrible performance Rust style.**

In his infamous video ["Clean" Code, Horrible Performance](https://www.youtube.com/watch?v=tD5NrevFtbU), the legendary Casey Muratori showed how trying to be cute with your code and introducing *unnecessary* indirection can hurt performance. He compared the “clean” code way of structuring your classes in an *"OOP" style, using* class hierarchy, virtual functions, and all the hoopla. He then showed how writing a straightforward version using union struct can improve by more than 10x the “clean” code version.

I had the idea in my mind to reimplement his example in Rust some time ago, and here I am stuck on a train trip, what better time to do so? The goal here is to see what a Rust port of this infamous video would look like from an idiomatic-rust style feel and of course performance.



**The “cute” dynamic implementation :**

This is the *"clean" version*, which defines a trait Shape. All shapes implement this trait and we use dynamic dispatch to compute each shape's area. Each shape is then heap-allocated and the total\_area function takes a &[Box<dyn Shape>] then calls the area function on each shape.

**The “rusty” enum implementation :**

Fortunately, Rust provides an amazing, idiomatic solution for this problem: Enum. This version defines a enum Shape which contains the various shapes. The area of each shape can be computed by a straightforward match on the Shape. Shapes here are allocated contiguously on the heap as a Vec<Shape>.

**The “chad” data-oriented implementation :**

This is the data-oriented way, chad-level, ECS style implementation. Basically, co-locating shape attributes in contiguous arrays. We have a single struct Shapes. This struct has a vector of enum Type, widths and heights. The area can be computed by matching on the type (just the tag here) and computing the area.

As a bonus, I was curious about the match statement in the **data** code. I wrote a fixed-sized table with the shape coefficients and indexed into it using the type. I wanted to know if removing it would reduce branch misprediction but I was wrong. I guess the compiler reduces the match to something much more efficient than if else statements.

I also tried to use idiomatic rust as much as possible. I used iterators and avoided using loop unrolling or any other magic …

**Benchmark**

Here is the current benchmark, generating 1\_000\_000 shapes and computing the area 100 times to warm up CPU caches, etc... Of course the tests only benchmark area computing and not shape creation :

Dyn took 16.5883ms.  
Enum took 11.50848ms. (1.4x)  
Data oriented took 11.64823ms.(x1.4)  
Struct-of-arrays took 2.838549ms. (x7)  
Data\_oriented + Table lookup took 2.832952ms. (x7)

The enum version is **1.4x** faster and the data oriented version is 1.**x** faster than the cute dynamic dispatch-y one.

**EDIT 1:** I actually had a bug in this implementation where the data oriented one was 4.1x faster but @SylphStarcraft watched it on reddit !

**EDIT 2:**After the tumultuous comments this thread received, I posted about it on Twitter and received a [great observation](https://x.com/cmuratori/status/1710711336760705390?s=20) from the man himself @cmuratori. There was an issue with the testing method, not randomizing the array of shapes led to falsifying the result. The CPU branch predictor will just predict the pattern and have nothing but hits on the match. I also added a version SoA as suggested by some comments :

Of course, you should take microbenchmarks with a grain of salt, but there is a clear winner here!

**Analyzing results**

The enum version is not that different from the data-oriented one. My hypothesis is that all enum variants have the same memory layout which helps the CPU from prefetching and accessing their data either way so there is no obvious difference in the two implementations from a performance standpoint. One really great comment on Reddit [SirClueless](https://www.reddit.com/user/SirClueless/) breaks down why :

*It shouldn’t be surprising that these are very similar. The inner loop in each case is almost identical at an opcode level: Read an enum discriminant, and based on its value compute a simple expression using one or two floating point values.*

*This means that essentially all of the variation is going to be in memory layout and cache coherency. This in general can be quite significant, but in this case both patterns of memory access are exceedingly predictable and will access essentially all of the memory of whatever data structure you use. So you can just compare total sizes. The struct-of-arrays version should be slightly less total memory because enum Type from the data-oriented implementation can be implemented in one byte, whereas enum Shape from the enum-based implementation needs to align its f32 values to 4 bytes so will have 3 bytes of padding after its enum discriminant. So I would expect the struct-of-arrays implementation to be very slightly faster, and it is.*

Another interesting result is that the table lookup and struct-of-arrays approach is a lot faster. This is due to completely eliminating the branch misprediction.

Full disclosure here, I really don't like dynamic dispatch and I lean more toward Muratori's ethos about code style than the clean code way but I don’t see the need to spray thedyn keyword in your codebase just for the f of it. In my opinion, dynamic dispatch should be used only when absolutely necessary. I also feel that the enum version is clearly the idiomatic Rust way of coding and feel that it would generalize to a lot of use cases.