

Programming Aesthetics learned from making independent games

April 1, 2011

This is a loose collection of ideas

Video going on youtube

Not interested in having nerd rage arguments
about what I say here



Braid source code

http://cloc.sourceforge.net v 1.53				
Language	files	blank	comment	code
C++	221	25736	8312	73223
C/C++ Header	239	3655	2029	9467
C	3	1046	2249	5986
HLSL	20	354	143	1287
...				
SUM:	488	30907	12746	90347

For something like Call of Duty, this is kind of small.

But as a job for one person to do, it is quite large!

Hand-written
Implementation as of final product (lots of code was written and then deleted!)

Biggest things I remember doing in college were maybe 12k-16k lines.

The Witness source code

```
http://cloc.sourceforge.net v 1.53
-----
| Language | files | blank | comment | code |
|-----|
| C++      | 329   | 37708 | 14900   | 96822 |
| C/C++ Header | 331   | 5303  | 2221    | 12889 |
| C         | 2      | 583   | 647     | 3883  |
| HLSL     | 51    | 986   | 311     | 2458  |
| ...      |        |        |          |        |
|-----|
| SUM:     | 720   | 44659 | 18095   | 116204 |
|-----|
```

Bigger. Fortunately I don't have to do the programming myself this time.

“Industry Average”: 3250 lines/year

(90000 lines) / (3250 lines/year) ≈
28 years

You really have to be productive.
(And you can't skimp out on quality to get that productivity.)



512 MB RAM (you can't use it all), no VM

3 *slow* in-order cores

Slow file access

But it can't be lousy code either. Here's the machine you have to deal with...



Certification

Program can't crash, even with antagonistic user

Loading time is capped

But it can't be lousy code either. Here's the machine you have to deal with...



Certification

“Soak Test”

$$3 \text{ days} * 86400 \text{ sec/day} * 60 \text{ frames/sec} = \\ 15,552,000 \text{ frames}$$

If you leak 4 bytes per frame, you'll fail.

So we are talking about quality code here.



game design
level design
art direction
audio direction
business development
marketing / PR
financial management

Hard enough if you are just the programmer.

But you probably have to do a lot of these jobs too:



be
extremely effective
at
getting things done

I have gotten better at this over time

I have developed an aesthetic of what's "good code" and what is not

It's very different from my ideas when I was in school!



... by the way ...

I am not the Grumpy Old Fortran Programmer

Impulses to optimize

are

usually premature.

Optimized code is:

Harder to write

Harder to understand

Buggier

Has a shorter lifetime

Imposes undesirable constraints on the rest of
the code (oh, your input has to be formed THIS
WAY)



Most code is
not
performance-sensitive.

Remember all that code we had to write! Very little of that is the inner loop.

So if you even think about performance of most of that code, even for a minute, you are wasting time and clouding your mental space!

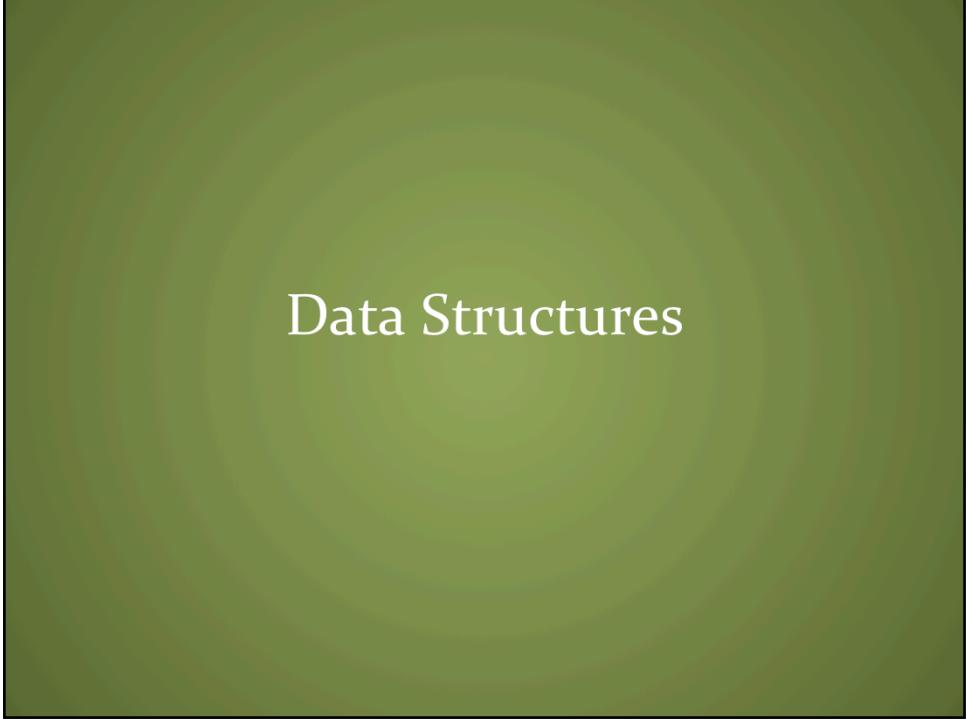
The consequence is:

Optimization is

usually bad!

(Unless practiced very carefully!)

If you are working on code, and you have the impulse to optimize, it is probably wrong.



Data Structures

Let's talk about data structures for a second.



Data structures are about
optimization.

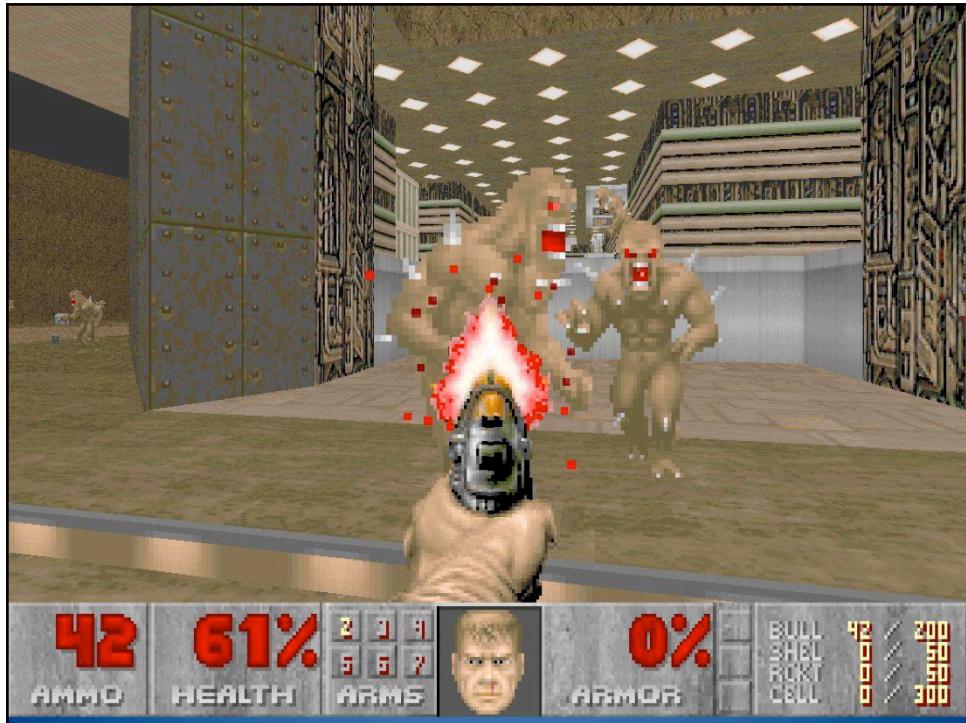
They are about answering a query that your
program has,
in an accelerated way.

If you think about it as enabling operations...
why do you want to do those operations?

“using the right data structure”

is usually bad!

(Because it is premature optimization!)



Doom asset loading anecdote

My approach would have:

Taken more time to implement

Used more memory

Fragmented the heap more

Had less-predictable performance characteristics

Made the executable bigger

Been harder to understand

Been more bug-prone

Made the source code longer

Made the source code more complicated

All for no perceivable benefit. Unequivocally



Now I use arrays of records
for almost everything.

- Array of structs
- for loop that does a compare to find the one I want
- I can type this without thinking. I know it will work. The performance almost never matters.

Even if I know I may need to change it later.
I won't change it until the need actually arises.

Things you might optimize

seconds per program execution (speed)
bytes per program execution (space)

Optimization is not ALWAYS bad.

It is good when you are optimizing things that actually matter.

It is only bad when you are optimizing for the wrong thing.

Instead, optimizing

years of my life
per program implementation
(life)

*This is a valid optimization parameter
you can consider just like those others!*

Optimization is not ALWAYS bad. It is only bad when you are optimizing for the wrong thing.

In the same way as code has speed / space-consuming behavior, it has life-consuming behavior.

Data structures are about
memory or speed optimization.

They are not about
life optimization

(unless you absolutely need that speed or memory).

So if you are using data structures often to
optimize for speed or memory, YOU ARE
PROBABLY DOING THE WRONG OPTIMIZATION.

Thus DATA STRUCTURES SHOULD BE USED
SPARINGLY.



Complicated Algorithms

are not good!

Generalizing further...

If you see a complicated algorithm somewhere,
it is probably a bad idea!

They use complicated data structures

Usually make the speed/space/life tradeoff
ignoring life

Usually they require a large life expenditure
For a marginal benefit

Has a short lifespan! (More complication ==

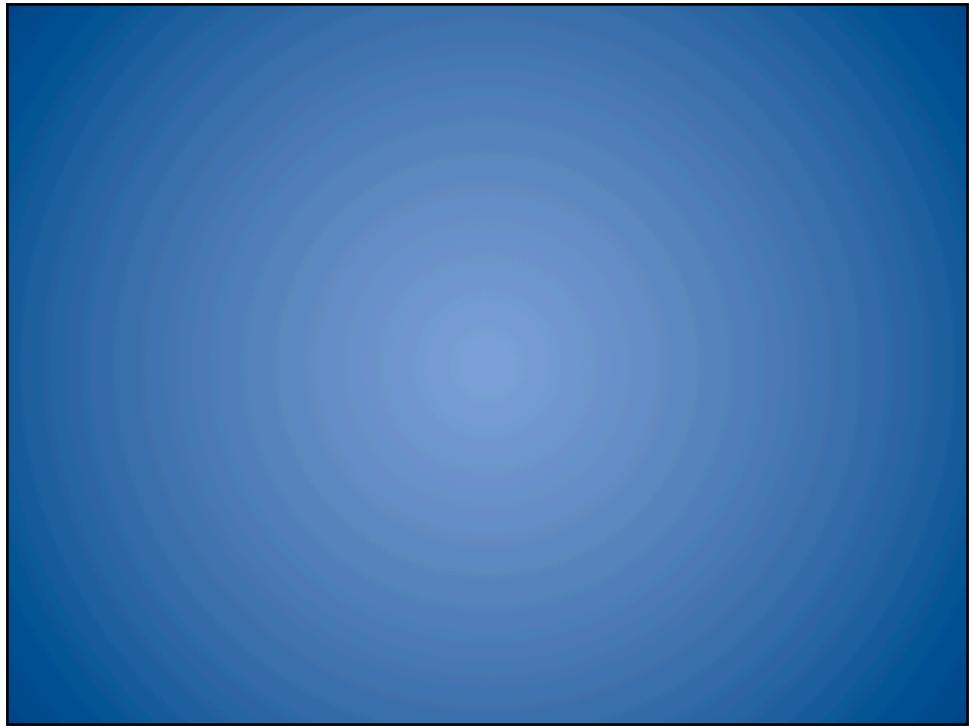
Almost all applied CS research papers are bad

propose adding a lot of complexity
for a very marginal benefit

doesn't work in all cases (limited inputs, robustness)
“supported” by bogus numbers, unfair comparisons

This isn't fooling anyone any more...

I think the applied CS community has a crisis of identity on their hands,
Or would, if they were paying attention...



Clear the air
Go on to smaller topics

A generalized system

is usually worse

than a specific / hardcoded one!

- didn't need functionality yet
- More code! (Compile, link time; harder to port)
- More general = more vague = less self-documenting

Generalized systems are harder to delete later!
Later on, how do you know if you can delete this
generalized system or not?

Adding new systems
is bad!

This should only ever be
a last resort.

deleting code >>> adding code

Nodes connected to other nodes
Complexity grows superlinearly with number of
systems

If at all possible, do something small.

Examples: fancy collision stuff, terrain LOD
system, etc

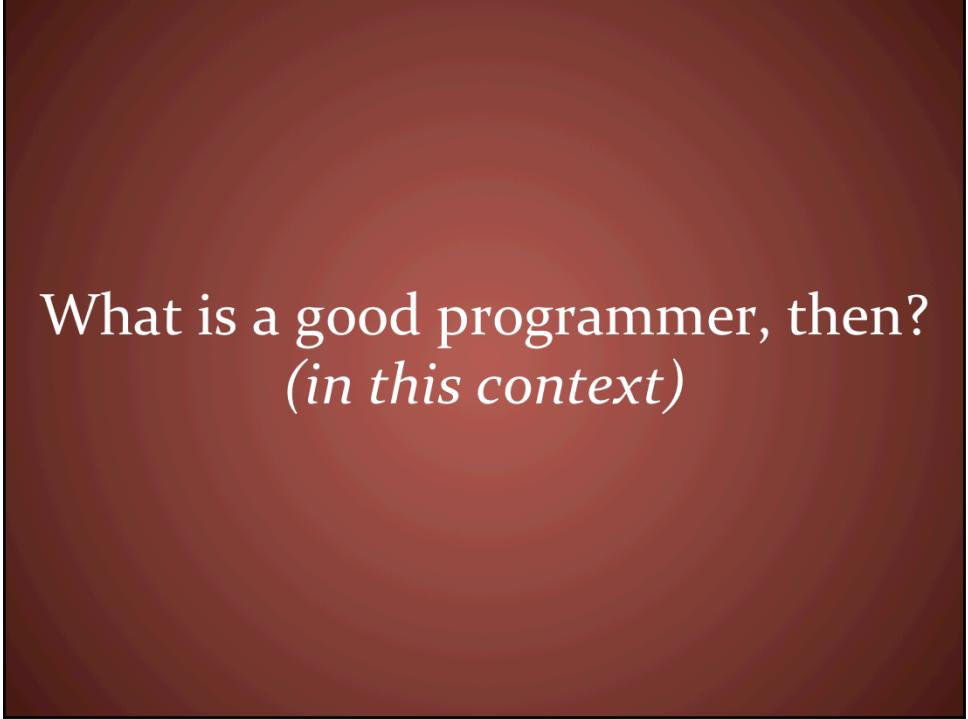
Straight-line code preferred over function calls

```
{  
    a = b + c;  
    a *= f(k);  
    a /= g(a);  
  
    print(a);  
}  
  
float  
recompute () {  
    float a = b + c;  
    a *= f(k);  
    a /= g(a);  
  
    return a;  
}  
  
{  
    recompute(a);  
    print(a);  
}
```

```
{  
    // Update a.  
    a = b + c;  
    a *= f(k);  
    a /= g(a);  
  
    print(a);  
}
```

Don't know where `recompute()` is called from!
What is the global state like? Etc, etc. Taken explicit knowledge and made it implicit – harder to see/understand.
The call graph of the program gets more complicated every time you do this.
-Overhead of changing both function and the caller... (Worse in C++ if it is a method!)

... Program complexity is super-linear with number of pieces



What is a good programmer, then?
(in this context)

Get things done quickly, robustly
Makes things that aren't buggy

Makes things that are simple

gets things done quickly

gets things done robustly

makes things simple

finishes what he writes (for real)

broad knowledge of advanced
ideas and techniques

(but only uses them when genuinely helpful)

Note that most of these things are the
ostensible goal of
All the stuff I was mentioning earlier!

School stuff: useful!
But go beyond knowing the techniques to an
operational wisdom

Programming methodologies: are supposed to
do 1-4

*it's easy to see benefits of an idea
developed for benefit's sake!*

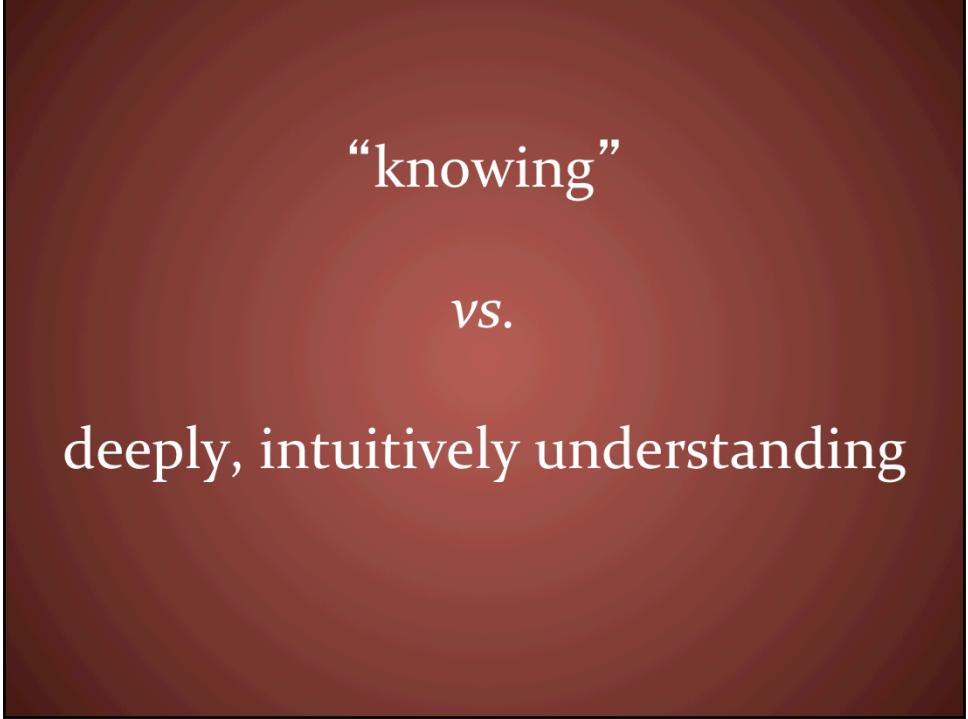
*very hard to measure subtle
negatives chained to this idea*

(which often outweigh the benefits)

Software engineering paradigms

Algorithms

whatever



“knowing”

vs.

deeply, intuitively understanding

I would have said “yeah, I know deleting code is more important than adding”

But then I go behave as though I didn’t*really* know that.

It takes a lot of practice to internalize these things.

So if you think you already know what I am talking about, consider that maybe you don’t.

