Data oriented Design

Point and click adventure using Table-based finite state machiens (e.g. pg 203)

Instruments and cache misses

Turn the Dungeon example into a larger game

Ideas of Experiments

Something with the table-oriented health / weapon timer from pg 216

The various C++ examples - run, measure. Try with Swift

DoD Changes to Borkle itself?

PRESO

Code/Data coupled

Objects all over address space

OOP Bad, mmmmkay?

vtables involve an extra indirection / cache miss / mispredict

Table Normalization

Binary Search

Flags in the extra bytes

Use of Cachelines

look at DoD as writing code with a pattern similar to what finally runs on the CPU. When you do that, it's much easier to reason about how it will run on actual hardware.

Other coding paradigms that don't at all resemble the procedural nature of CPU execution rely quite a bit on the promises and capabilities of the language/compiler to deliver "zero-cost abstractions". Because, in the end, whatever you write gets translated to procedural machine code. - *Random Comment of Data Oriented Design In Practice* talk

DUNGEON CHAPTER

Because in languages like, C++ you need instances before you can relate them, so have to create all rooms before creating the doors that connect them

Oop bad

"A Relational Model for Large Shared Data Banks"

"Further Nomalization of the Data Base Relational Model"

EF Codd

Insert, update, delete, query

Reduce the need to maintan a deep understanding of how the data was laid out to use it well

claimed to reduce the likeyhood of introducing internal inconsistencies

Traps

Say a gas trap that fills rooms slowly

Timed events

"hunt for keys to unlock doors to escape"

Monsters on patrol

pickups

Exit

Keys (for particular door in particular room)

things that affect player stats

Potions

Armor

Like images / audio - yes, they're big, but you can't really cut them up without them losing integrity

"migrate from a web of connected complex objects to a simpler to reason about relational model of arrays"

Relational model calls for atomicity when working with data

Converting existing data structures and relationships into something linear

because we've connected and given meaning to the structure of data

"collections of linear storable data entries"

the idea of "**meaningful boundary**"

Converting from the "network model" (hierarchical)

"games are so complex they odn't appear to have databases. They are so complex for the sake of performance they have forgotten they are merely a data transform"

The most flexible database in common use is the file system - primary key is a path, and a string for the data

extendability

performance

readability

reuse

The structure of any data is a trade-off

maintenance

future-proofing

The more complex the tables get, the less future proof and less maintanable, but higher performance and readability

When normalizing, try to stay at the level of nouns / the namable pieces

BCNF normal form is probably as far as you normally need need to go

May want to normalize your data for hot./cold access patterns

Normalizing

1. Go from object graph to messy tables, with NULLs and multiple items per cell

First Normal Form:
every cell contains one and only one atomic value.
And Every Row is Distinct

can consider tables to be more like sets where the whole row is the set value

key should be uniquie and small as possible

The key is often used to order the sorting of the table in physical media

But, it could be the entire row is also the key

2. reduce the number of elements of each cell to 1

and increase number of elements to 1 where currently NULL

Interesting

(OOP) When adding player movement logic, the "move:direction" moved from VC to Dungeon to Player before actually doing anything - like "who owns the current player location?"

Hit a (OOP) bug where only the right/below rooms were added to the room, not the top/left. A side effect of how the generator did its thing

(OOP) Moving L->R, honors lockage, but R->L moves right through

Snorgle

Characters

Flongwaffle