Data-Oriented Design

Sean Middleditch
DigiPen Game Engine Architecture Club

About Me

DigiPen CSRTIS Senior

Games: Subsonic, SONAR, Core

GEAC Founder

Engine architecture nerd

Talk Key Points

- Definition of Data-Oriented Design
- Why DOD matters
- DOD techniques
- Sensible use of DOD
- Example DOD architectures

Data-Oriented Design Definition

What does that even mean?

Approaches to Software Design

There are many different approaches to software design and programming paradigms

- Object-Oriented Programming
- Functional Programming
- Procedural Programming
- Data-Driven Design
- Data-Oriented Design

OOP Recap

Most game developers are intimately familiar with object-oriented design

OOP focused on *interfaces* (how code is used), with the key concepts being *encapsulation* and *abstraction*

Makes it easy to build easy-to-use, maintainable, flexible software

Data-Oriented Design

Data-oriented design focuses on the data and algorithms that manipulate that data as top priority

Key concepts are *data dependency*, and how that affects *memory access patterns* as well as *algorithmic simplicity*

Data-Oriented Design (ii)

Data-oriented design focuses on making tightly packed contiguous chunks of memory for data structures

Data-oriented design focuses on algorithms that process a single task (e.g. physics) in a single tight loop before moving on to the next task (e.g. graphics)

Maximizes efficiency, simplifies architecture

Plain Old Data

DOD focuses on Plain Old Data (POD)

Simple C-like structs

External methods/logic

Focus on the data rather than the behavior

Simple OOP vs DOD Example

```
// OOP style
class Foo {
private:
  int m_Data;
public:
  int GetData() { return m_Data; }
  void SetData(int v) { m_Data = v; }
  void DoSomehing();
};

// DOD style (C++, in Foo namespace)
struct FooData {
  int data;
};

void DoSomething(FooData& foo);
```

Data Dependencies

Data-Oriented Design stresses the removal of data dependencies

Each composite type (struct/class) should have all data it needs in itself, with no dependencies (pointers, or implied references) to other types

Makes it very easy to multi-thread and write efficient processing loops over data!

Understanding the Performance Issue

Using DOD to make games faster

Problems with OOP in Games

Take an average naive game object system:

Each new game object (or component) is allocated independently with **new**

Individual objects are spread all over memory

Update code is doing per-object physics, graphics, logic many times per frame

Performance Issues

Modern CPUs have deep pipelines (like an automobile factory)

Any *pipeline stall* and the *wait state* can have serious impacts on performance

Modern CPUs have lots of silicon dedicated to reducing this problem, but require good coding to circumvent the worst problems

CPU Pipelines

Pipeline stalls occur when one instruction must wait for a previous one to completely execute

Mostly dealt with by the compiler in a majority of cases

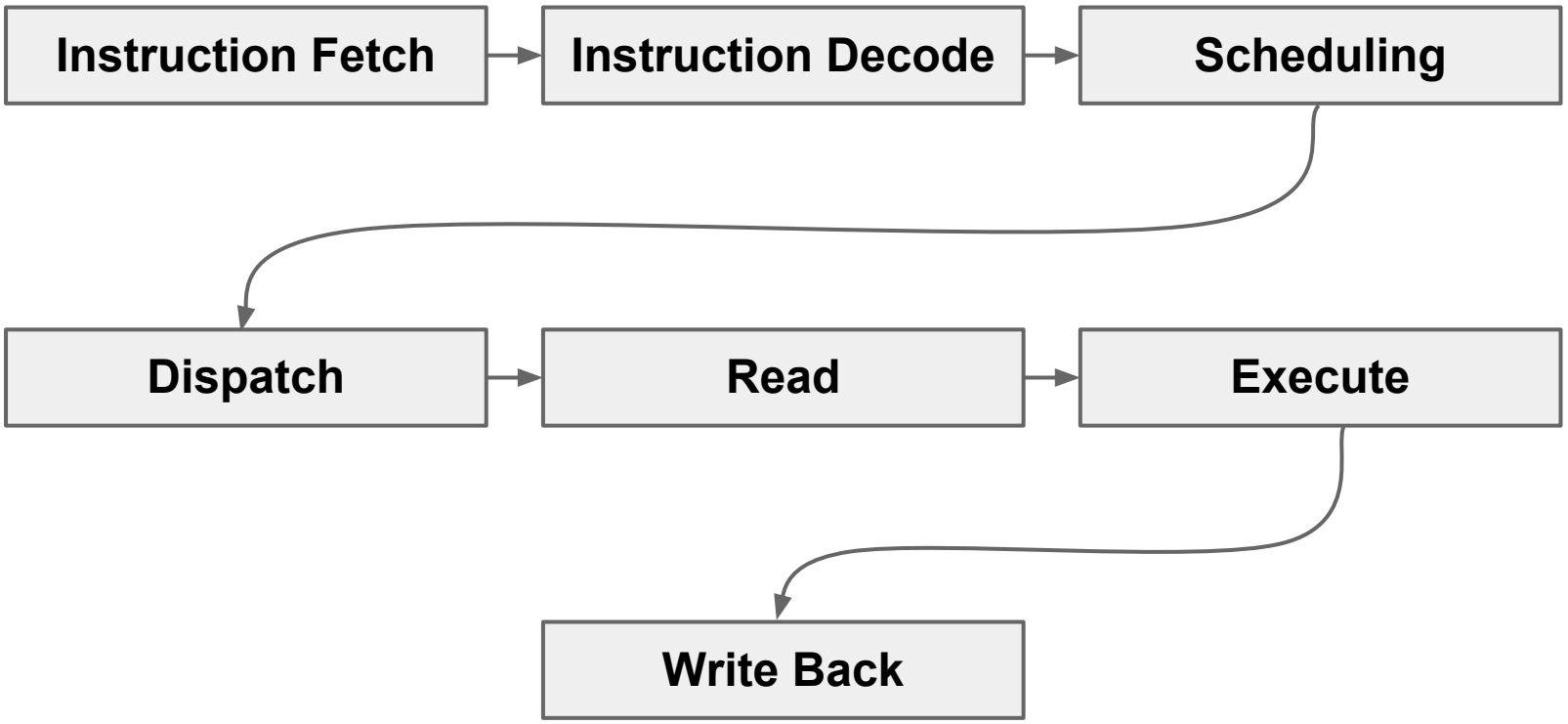
Branch-heavy code can cause stalls

Branching & Pipeline Stalls

Whenever the CPU branches (e.g. an **if** statement, indirect jump from a virtual function call, etc.) the pipeline must clear out, ignoring branch prediction

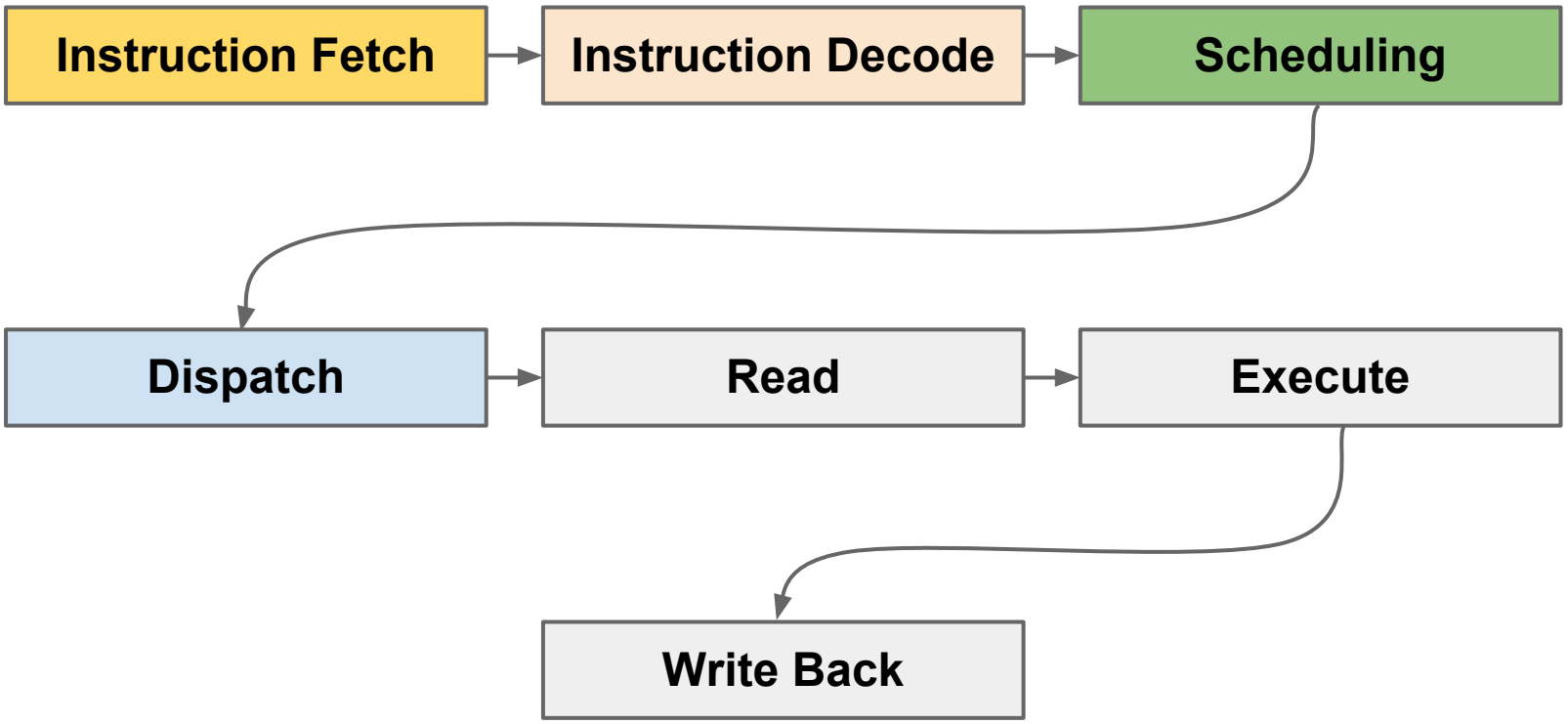
Code that iterates over a collection of arbitrary components and invokes virtual methods on it will incur a lot of branches (and branch prediction won't help here)

Simplified CPU Pipeline



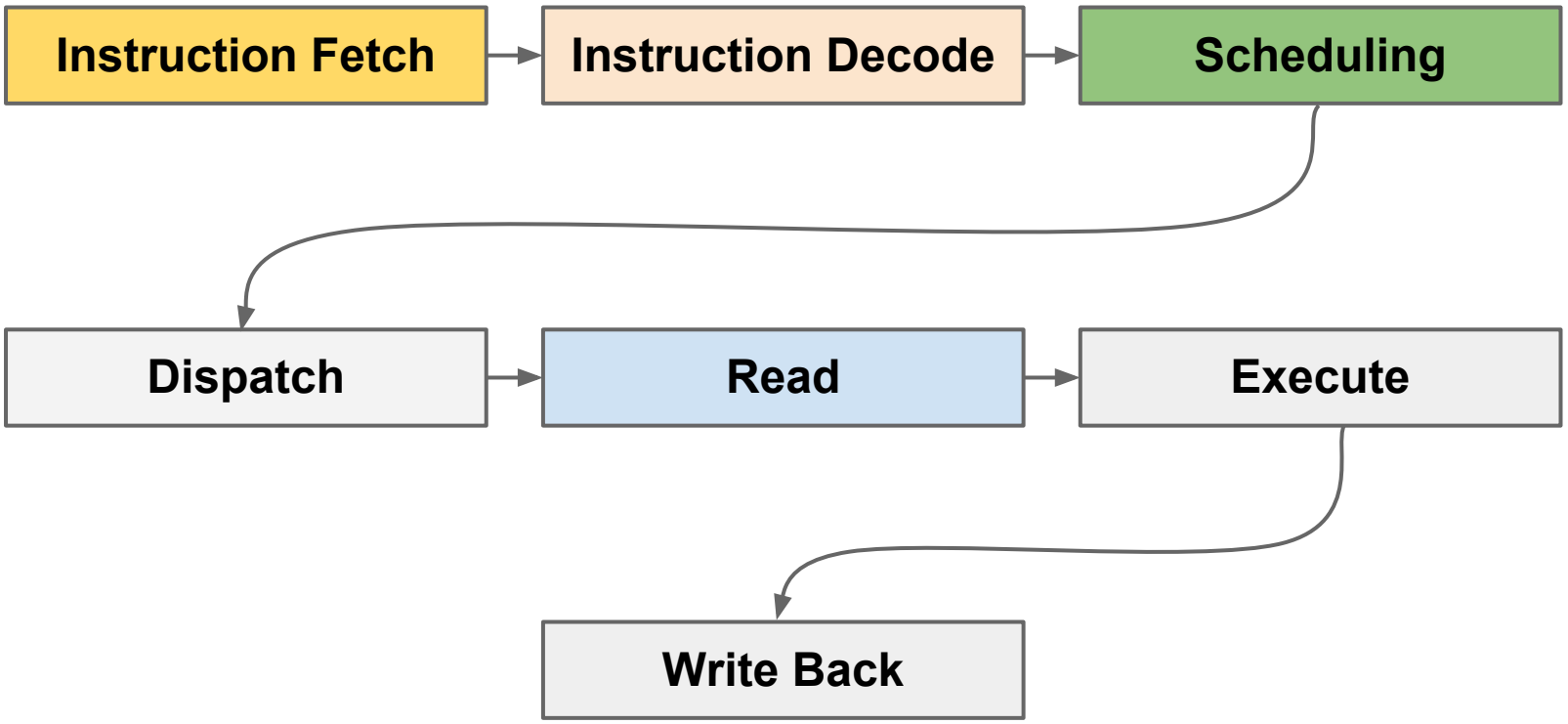
Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Instructions in Flight



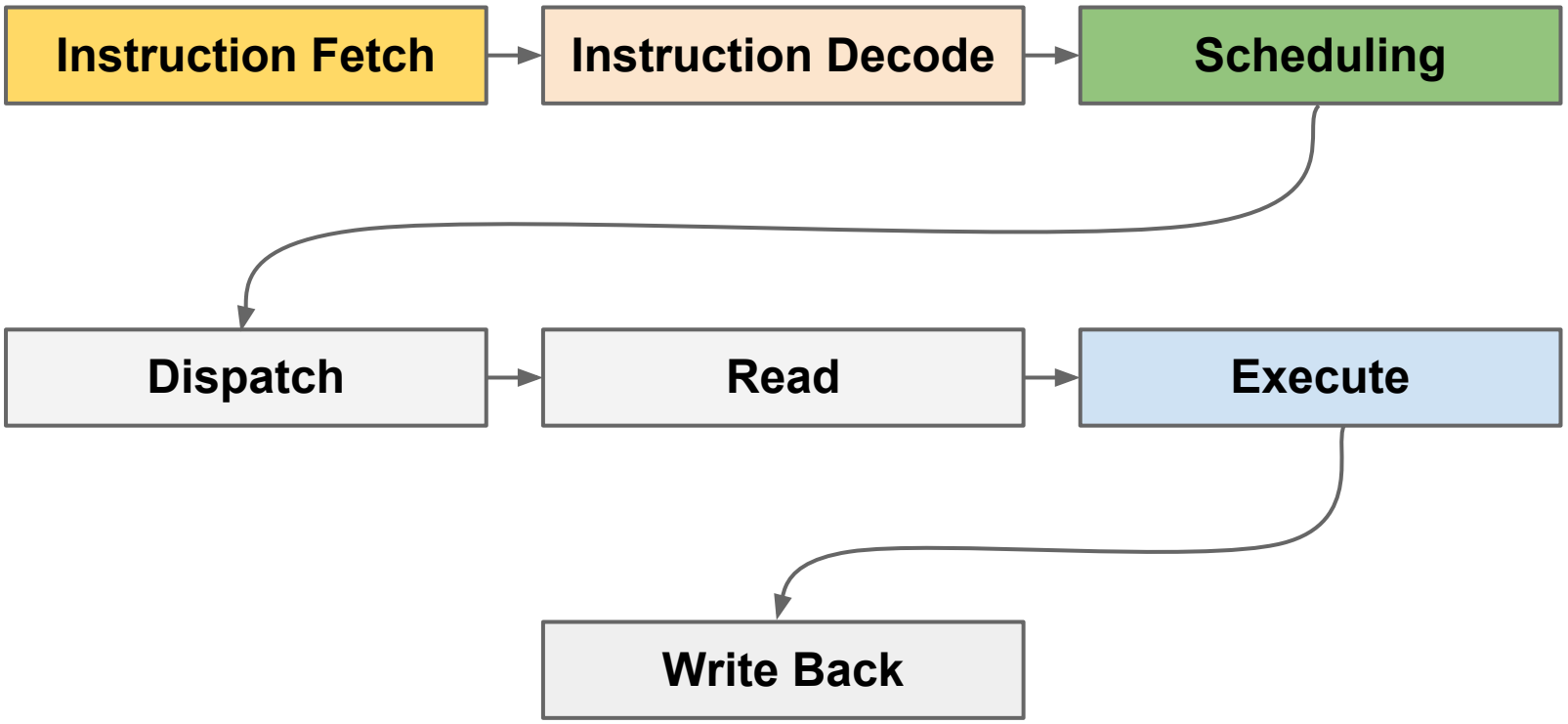
Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Green Instruction Depends on Blue Instruction



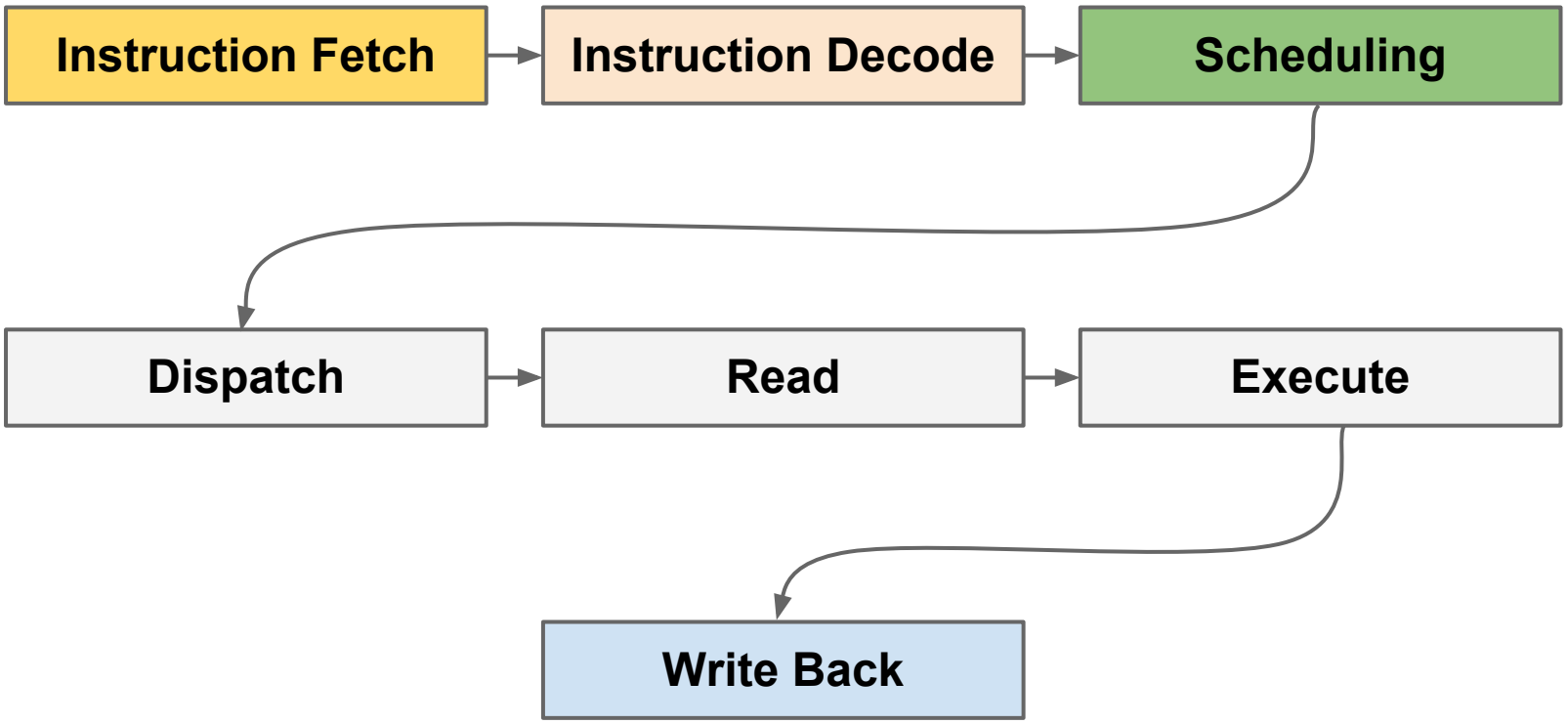
Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Green Instruction Depends on Blue Instruction (ii)



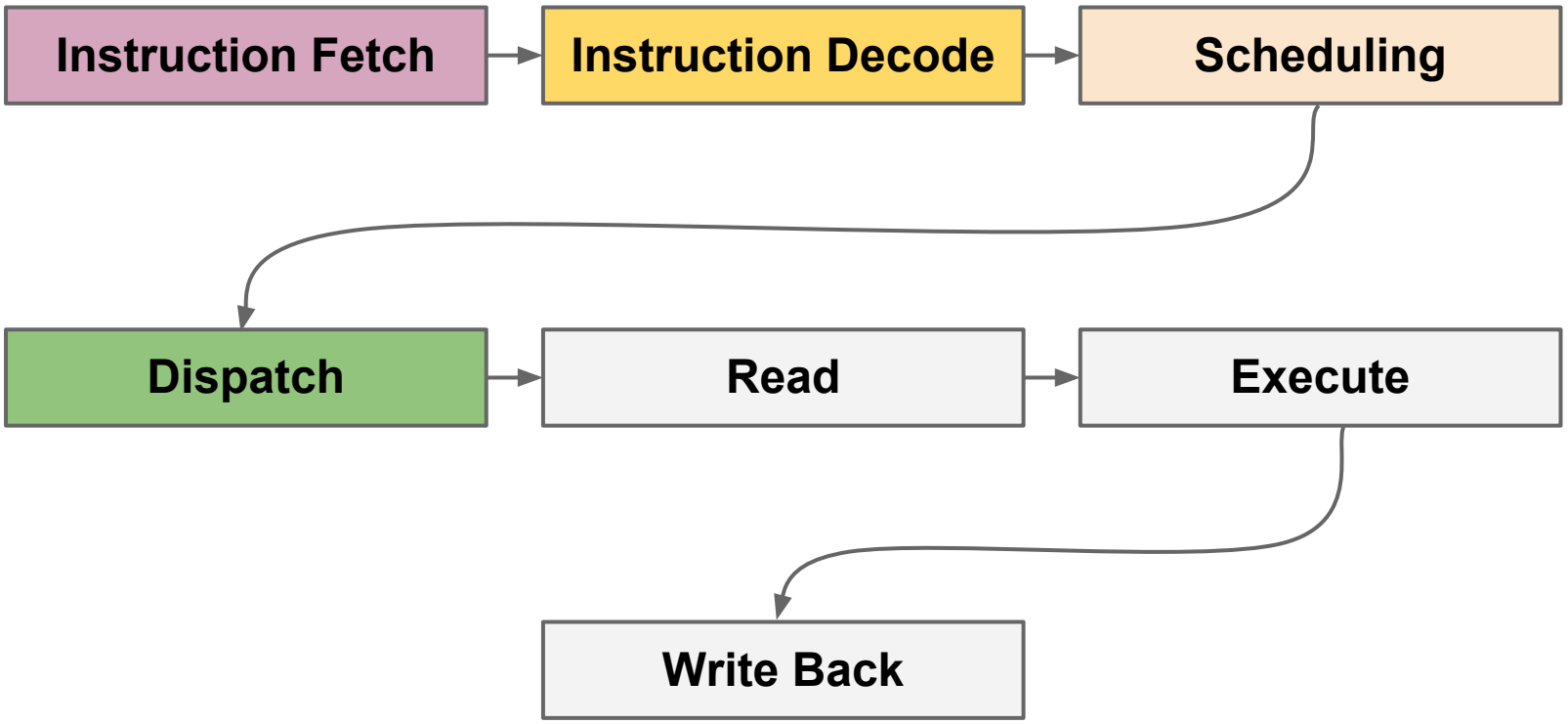
Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Green Instruction Depends on Blue Instruction (iii)



Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Instruction Stream Can Now Continue



Intel Core has 14 stages
AMD Bulldozer has 16-19 stages
Pentium IV had 20 stages

Wait State

Wait state is when the CPU is stuck waiting for a memory access

Random memory access can potentially take dozens or even hundreds of CPU cycles to service

Possibly the most serious performance problem when using modern CPUs

DRAM Memory

Dynamic Random Access Memory

Low-power through capacitors, requires periodic refreshing to maintain memory state

Organized into banks, rows, row-bank cache

Inconsistent latency due to bank access patterns, row refreshes, etc.

SRAM Memory

Static Random Access Memory

Requires constant power to maintain state

Much faster than DRAM but more complex and power hungry

Not suitable for modern multi-gigabyte memory sizes

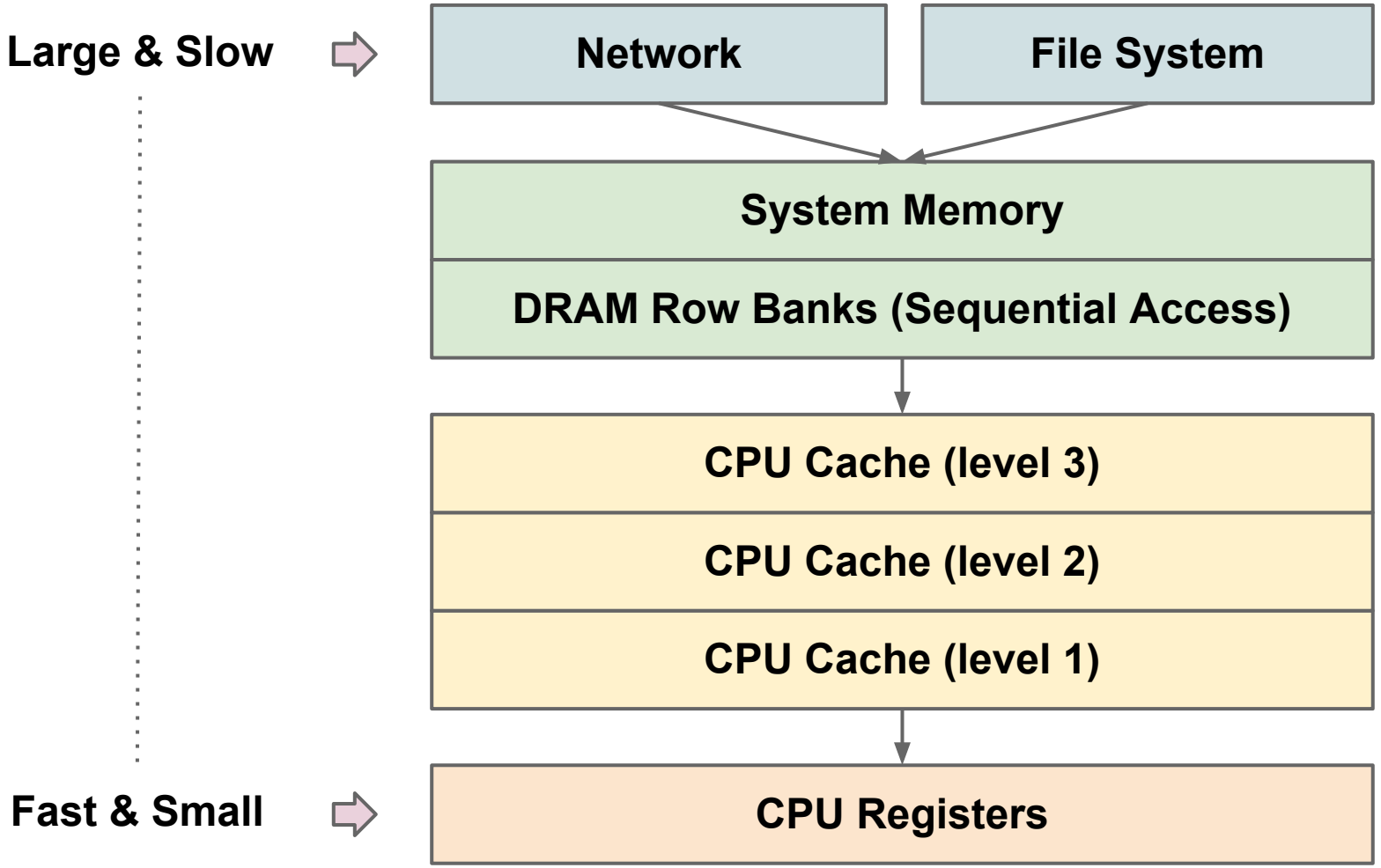
CPU Cache

CPUs have built-in cache (SRAM memory)

Data and instructions can be **prefetched** into cache, so memory access latencies are incured while the CPU is doing useful work

Requires that data be in contiguous packed memory without indirections, and not spread all over RAM

CPU Data Access Hierarchy



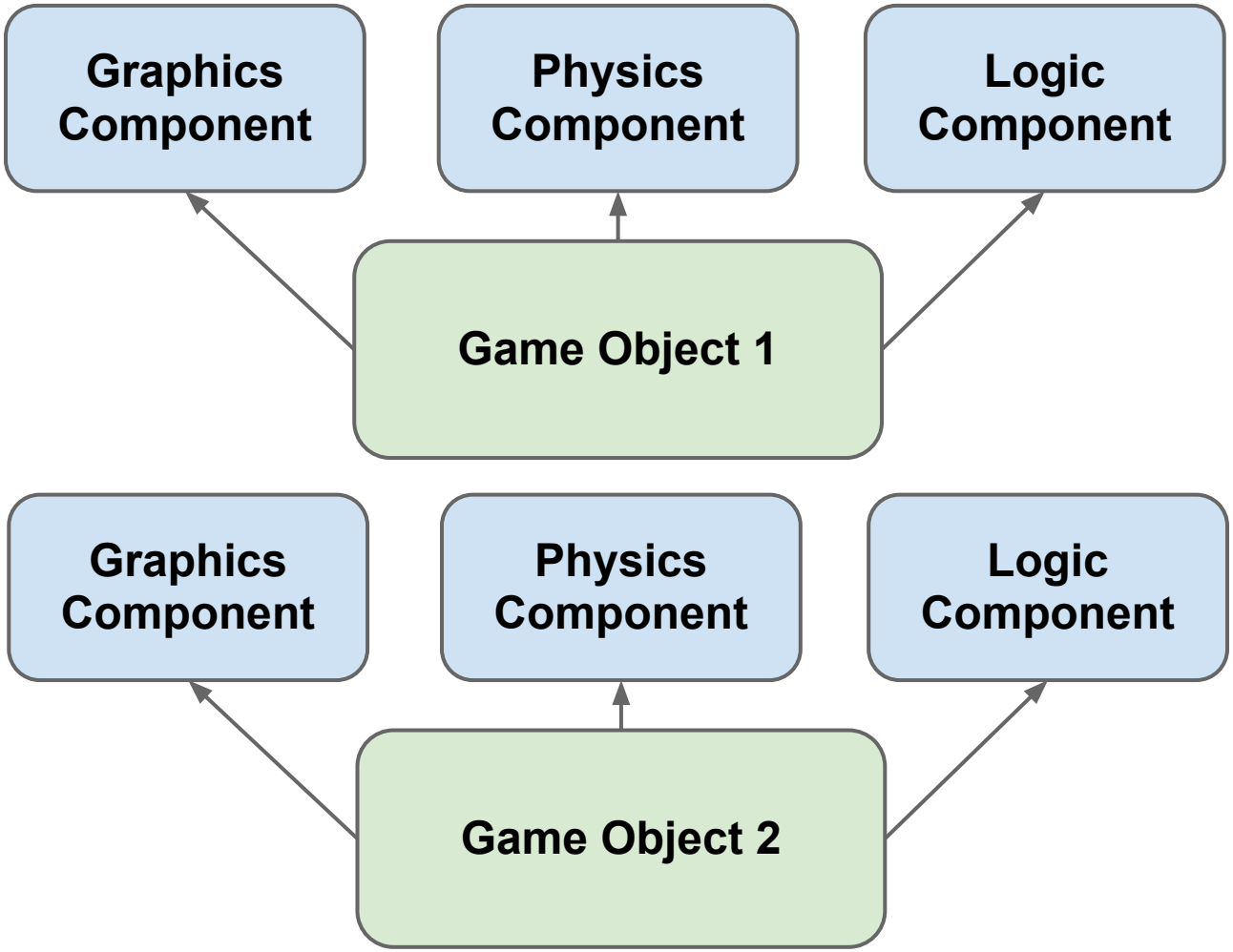
Memory Access Patterns

Allocating with **new** spreads objects all over memory

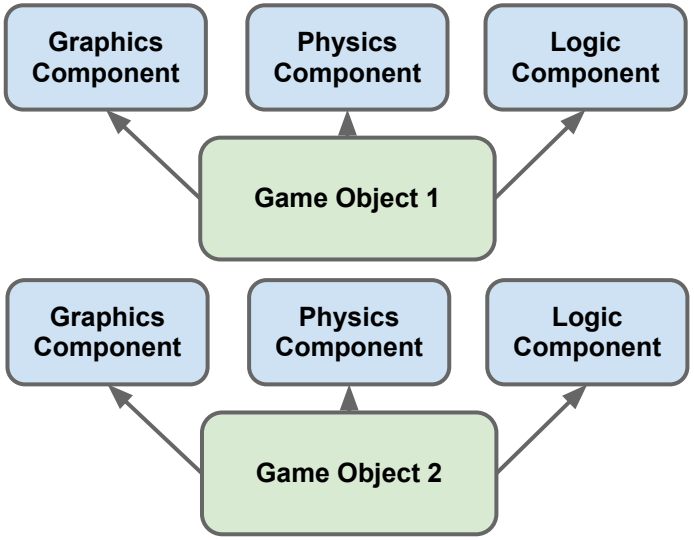
Iterating over these objects will mean they are rarely in cache, and very rarely in DRAM row bank

This can incur a significant performance penalty (often one of the largest in AAA games!)

Naive OOP Memory Layout

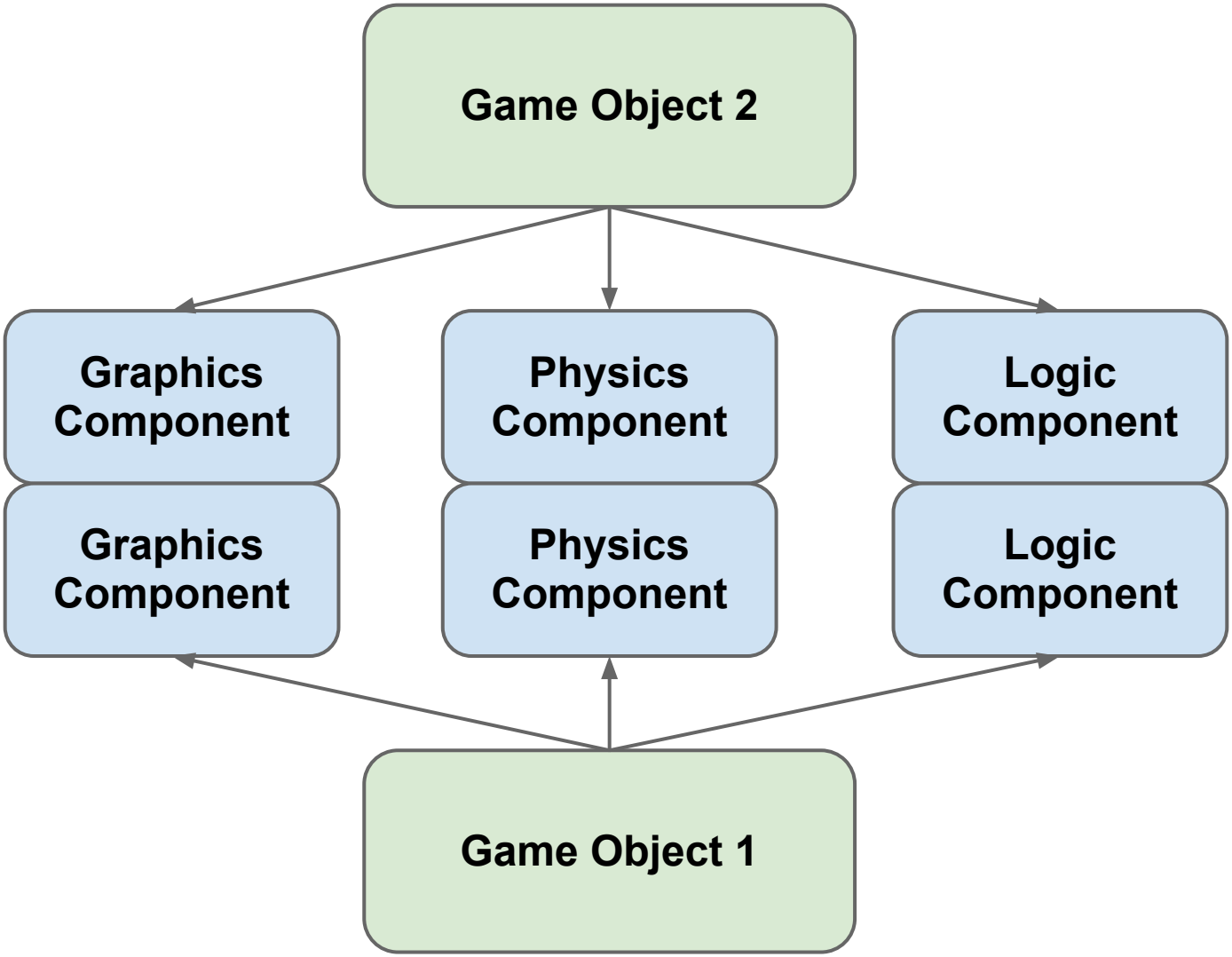


Naive OOP Memory Layout (ii)

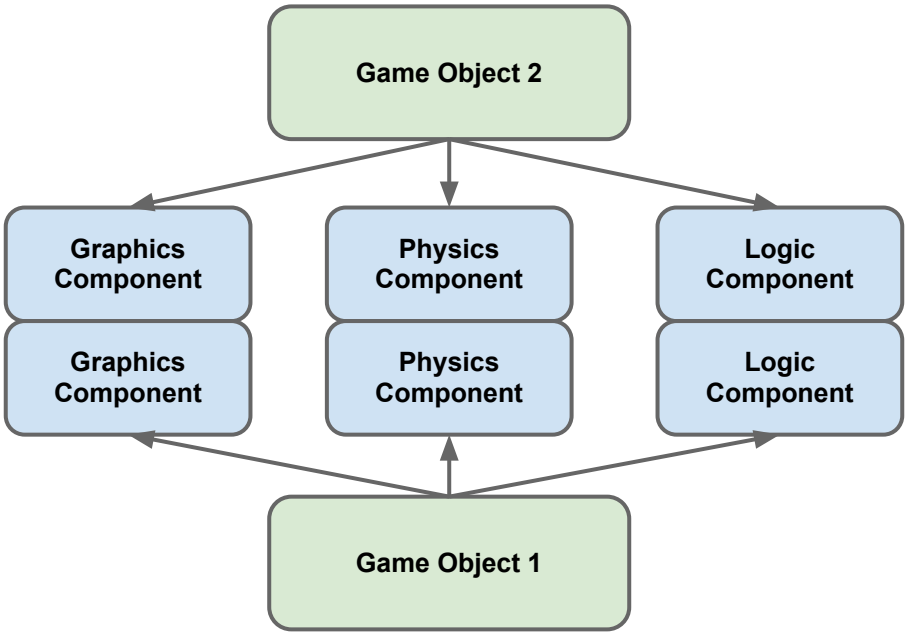


All objects are spread out in memory and require pointer indirections to find or process

DOD Memory Model



DOD Memory Model



Components are packed together and can be efficiently updated in a single loop with no indirections (GOs rarely used)

"What Every Programmer Should Know About Memory"

Read. This. Now.

(Well, after the talk, at least)

http://ftp.linux.org.ua/pub/docs/developer/general/cpumemory.pdf

by Ulrich Drepper

Architecture Improvements from Data-Oriented Design

How DOD can make code cleaner and simpler

The Truth About Computer Science

"All problems in computer science can be solved by another level of indirection."
- *David Wheeler*

"... except for the problem of too many layers of indirection."
- *Kevlin Henney*

Indirection Hell

Object-oriented programming naturally leads to lots of indirection

Encapsulation, abstract data types, and interfaces all require that developers "code to the abstraction" instead of directly manipulating data

These layers can grow unwieldy, confusing, error-prone, and lead to performance problems

Data-Oriented Programming to the Rescue

DOD's focus on Plain Old Data necessarily removes concepts like encapsulation and interface abstraction

Abstract data types (templates) can still be used!

"Clean" vs "Flexible"

DOD removes complicated abstractions and lets the programmer directly achieve a desired result, efficiently and without fuss

Abstractions and indirections do solve problems, and allow for more runtime flexibility and composability

Use each wisely

Beware Opinions

Be wary that terms like "clean" do not have a widely accepted common definition

Many folks find OOP to be horrifically messy

Many folks find DOD to be horrifically messy

Some systems certainly make more sense with OOP or DOD than others do

Finding Your Own Opinion

Compare the following array implementations

Bitsquid Foundation Library Array.h (Bitbucket)

- Data-Oriented - 162 lines
- Basic ops only, PODs only, custom API

Electronic Arts' EASTL vector.h (github)

- Object-Oriented - 1,623 lines
- Fully featured, any type, STL compliant

Using Data-Oriented Design in a Game Engine

Making use of all this stuff

The Truth About Your Game's Performance

Your game is not suffering from performance problems that will be solved by Data-Oriented Design

Most AAA games can get by just fine without making use of DOD in core logic and game object code

Your performance problems are probably all from suboptimal Direct3D/OpenGL use ☺

Critical Areas for DOD

Several parts of a game engine really need DOD to work well at all

- Physics; all parts, but especially:
  - Ray testing
- Graphics; all parts, but especially:
  - Particle engines
  - Culling
  - Batching

Non-Critical Areas for DOD

Many parts of game engines do not really need DOD in order to have adequate performance, and can benefit from extra flexibility

- Game logic
- AI
- Tools; excepting
  - Online lightmap calculation
  - Large scale batch operations

Game Objects & Components

There are engine designs that focus entirely on DOD in game objects

"Entity Systems" are such an approach

- Entity Systems are the Futureof MMORPGs

- Artemis Entity System Framework

Entity Systems

Entity systems assign each component to a **system** which manages those components

Components have *no logic* and are ideally POD

Entities (game objects) are only an identifier used to associate components in systems with each other

Entity Systems (ii)

Each system is responsible for updating all of its components in a single pass

Components should have no dependencies upon each other

Conservative Approach

Data-only components (no logic) as much as possible, but no explicit ban on logic components

Each component's associated factory has an allocator, and can use a tightly packed contiguous pool allocator when necessary

Allocate logic components together as possible

Conservative Approach (ii)

Systems like physics manage the updating of their own components

Only logic components have an Update() method, and logic components are only used for actual game logic and AI

Use DOD where it makes sense, use OOP where it makes sense, reap the benefits of both

The Conservative Approach is Already Common Practice

Physics systems and well-written graphics engines are already internally written with at least a partial DOD focus

These systems manage their own data structures, and components only bridge these internal data structures to game objects

Most Important Notes

PROFILE CODE

Writing a design that is *supposed* to be fast doesn't mean that it *is* fast

> I've seen a few entity systems written by folks who don't understand DOD and defeat the whole point of an entity system in the first place

Most Important Notes

KEEP IT SIMPLE

There are many critical problems to solve in making a fun, enjoyable, hit game

Having a perfect internal architecture *is not one of them*

Don't sweat DOD vs OOP unless you have to

Questions

I may not be a font of wisdom,
but I can satiate a thirst for knowledge