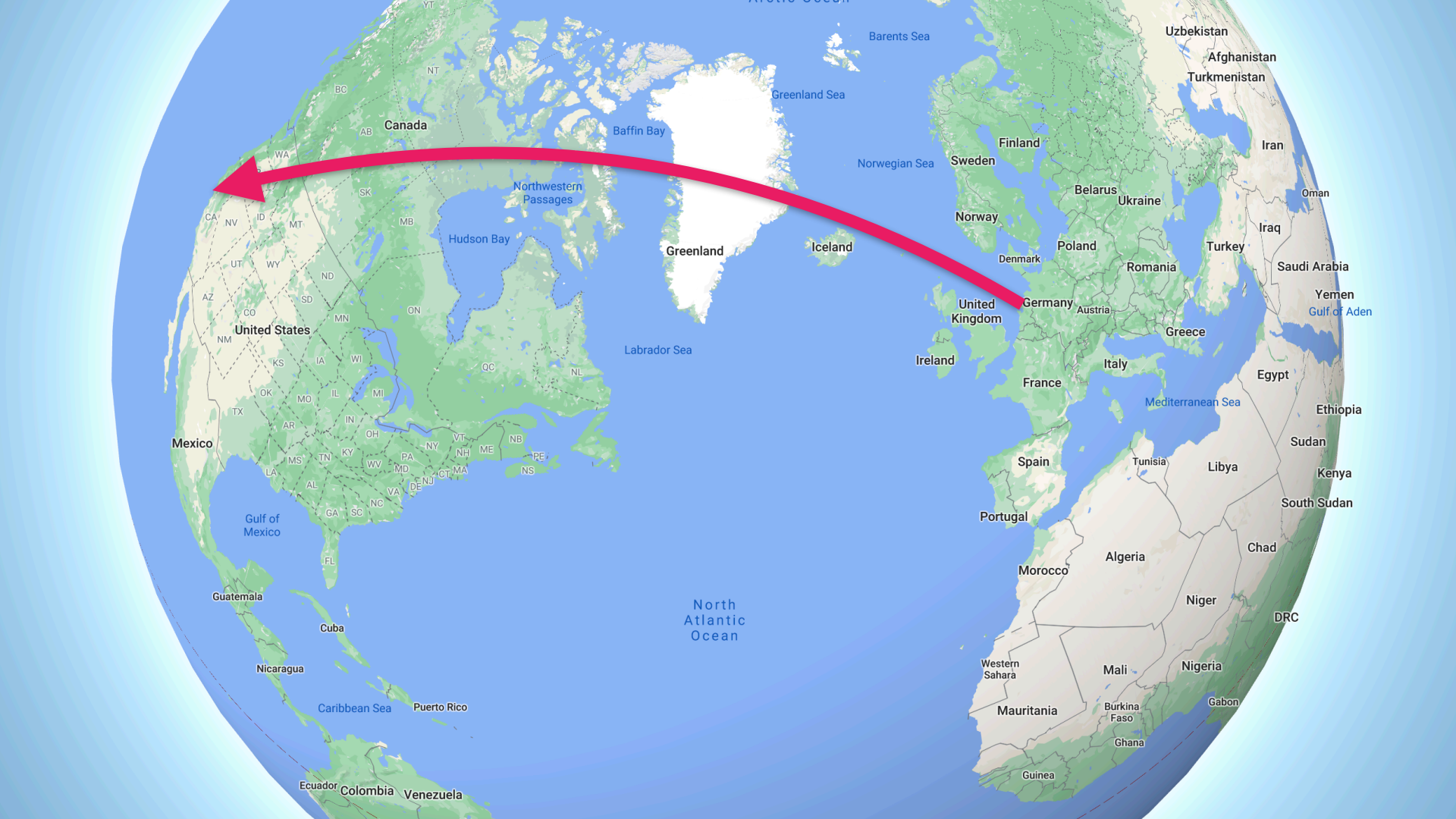
SANDER MERTENS

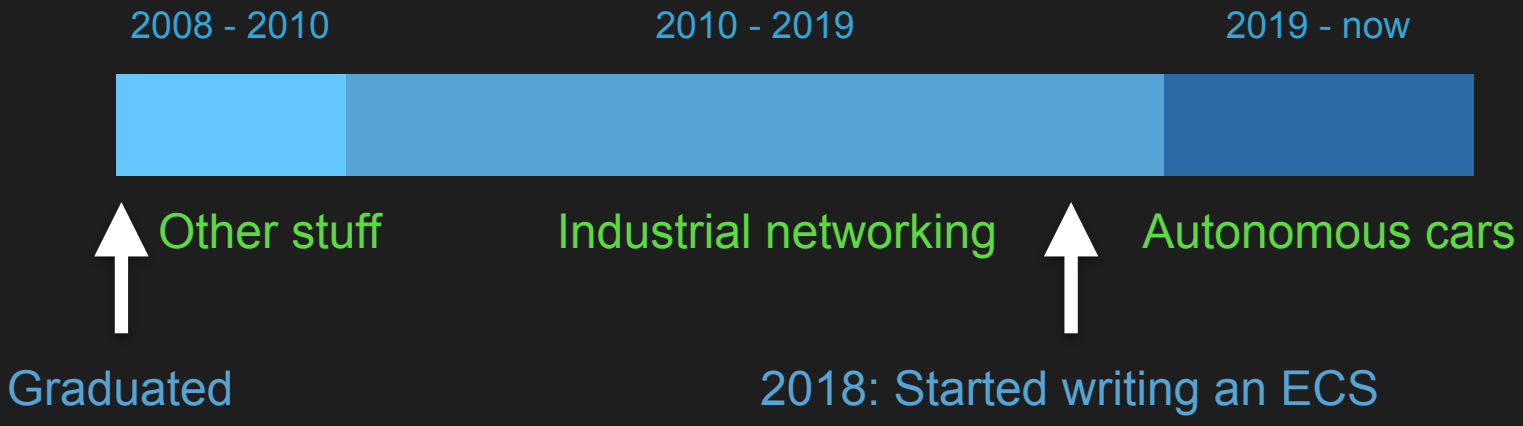
A DEEP DIVE INTO

ENTITY COMPONENT SYSTEMS

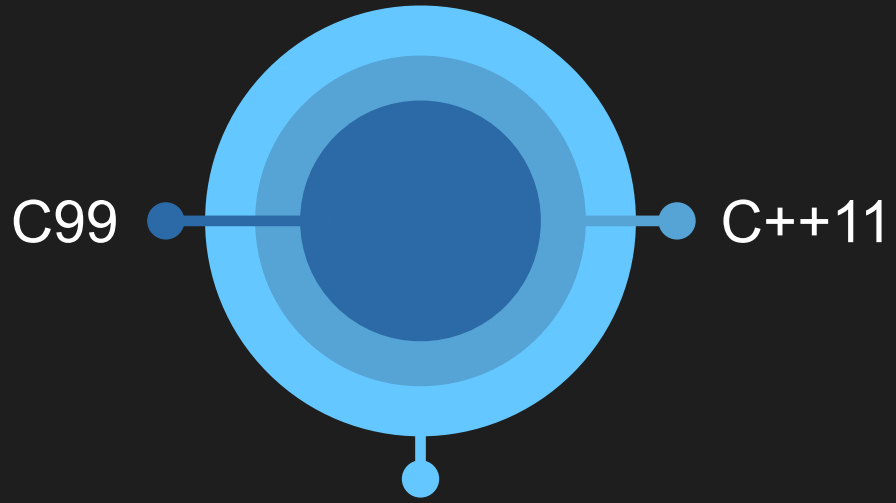

Barents Sea
Greenland Sea
Baffin Bay
Norwegian Sea
Northwestern Passages
Hudson Bay
Labrador Sea
North Atlantic Ocean
Gulf of Mexico
Caribbean Sea
Canada
United States
Mexico
Greenland
Iceland
Ireland
United Kingdom
Norway
Sweden
Finland
Belarus
Ukraine
Poland
Denmark
Germany
Austria
Romania
Greece
Italy
France
Spain
Portugal
Morocco
Algeria
Tunisia
Libya
Egypt
Mediterranean Sea
Uzbekistan
Afghanistan
Turkmenistan
Iran
Iraq
Turkey
Oman
Saudi Arabia
Yemen
Gulf of Aden
Ethiopia
Sudan
Kenya
South Sudan
Chad
Niger
DRC
Nigeria
Mali
Western Sahara
Mauritania
Burkina Faso
Ghana
Gabon
Guinea
Guatemala
Cuba
Nicaragua
Puerto Rico
Ecuador
Colombia
Venezuela


ONTARIO
NB
PE
NOVA SCOTIA
MAINE
Ottawa
WASHINGTON
MONTANA
NORTH DAKOTA
MINNESOTA
NH
OREGON
IDAHO
SOUTH DAKOTA
MICHIGAN
NEW YORK
RI
WYOMING
Chicago
PENN
Philadelphia
IOWA
ILLINOIS
OHIO
DE
NEBRASKA
NEVADA
San Francisco
UTAH
United States
WEST VIRGINIA
KANSAS
MISSOURI
KENTUCKY
CALIFORNIA
NORTH CAROLINA
Las Vegas
TENNESSEE
Los Angeles
OKLAHOMA
SOUTH CAROLINA
ARIZONA
MISSISSIPPI
NEW MEXICO
Dallas
GEORGIA
TEXAS
LOUISIANA
Houston
FLORIDA
Gulf of California
Gulf of Mexico
Dominican Republic
Puerto Rico
Cuba
Mexico
Mexico City
Caribbean Sea
Caracas

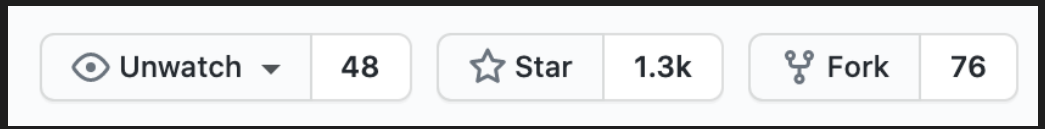

OAKLEY PARK


2008 - 2010
2010 - 2019
2019 - now
Other stuff
Industrial networking
Autonomous cars
Graduated
2018: Started writing an ECS

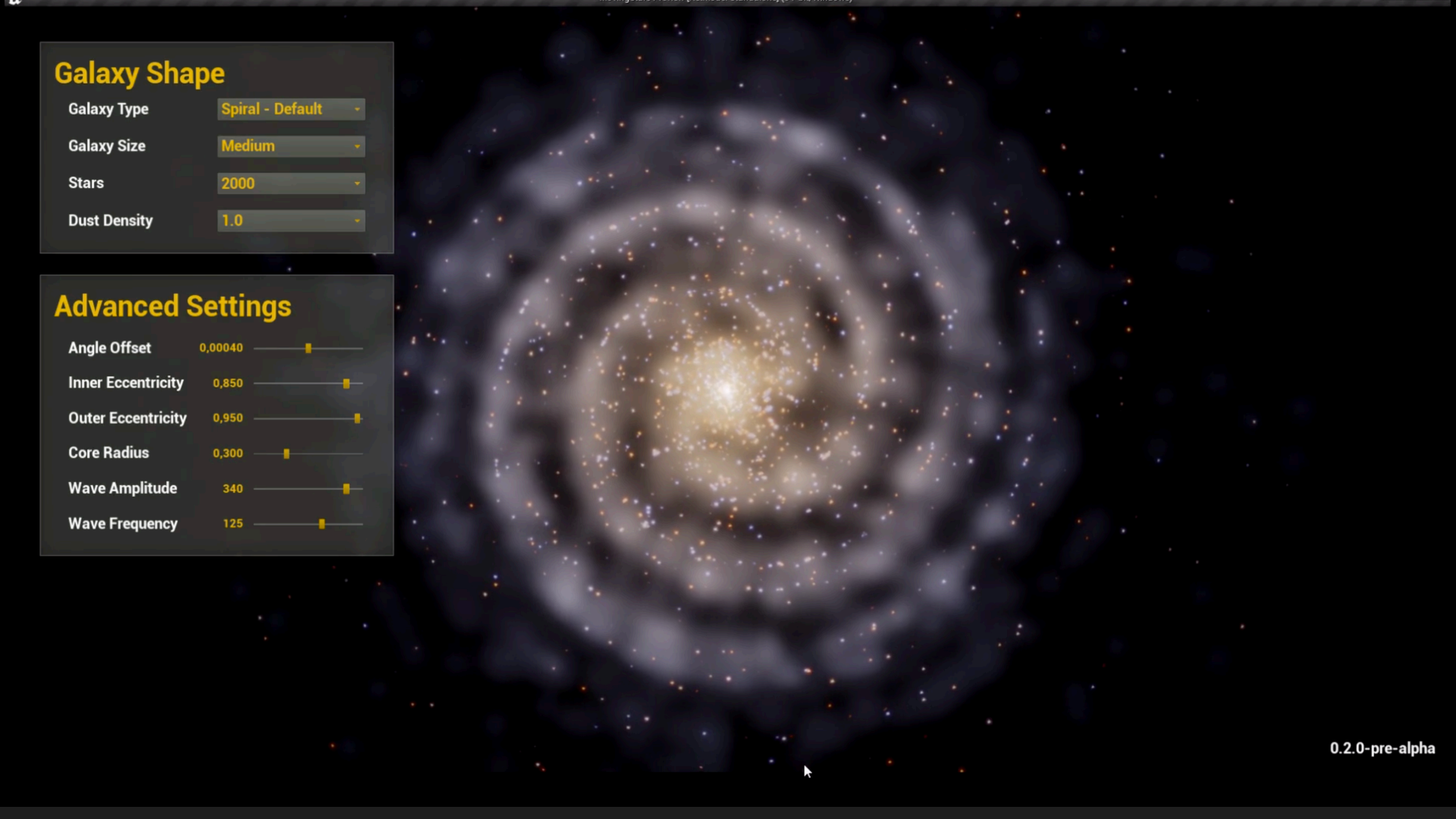
flecs



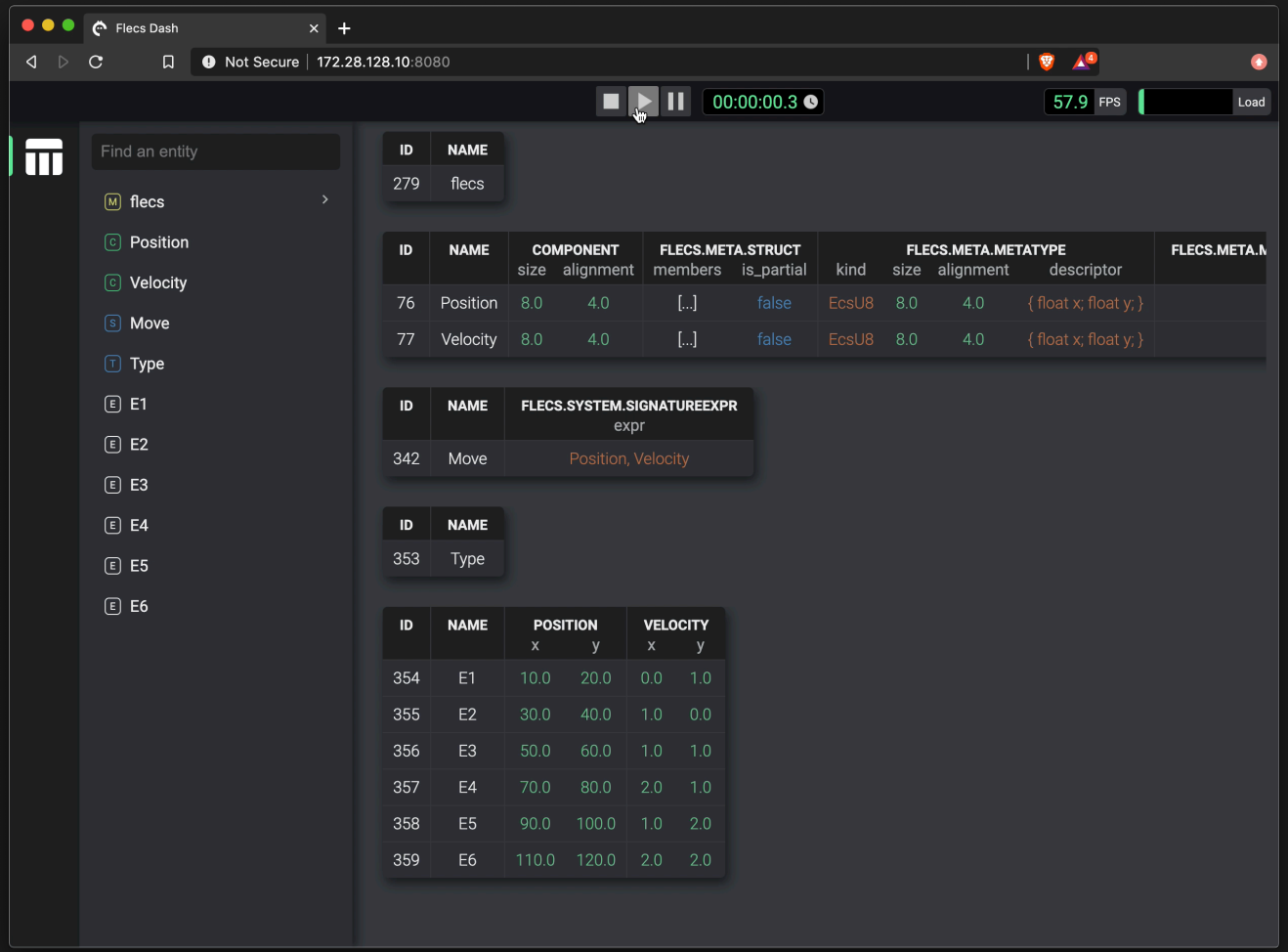
WWW.GITHUB.COM/SANDERMERTENS/FLECS

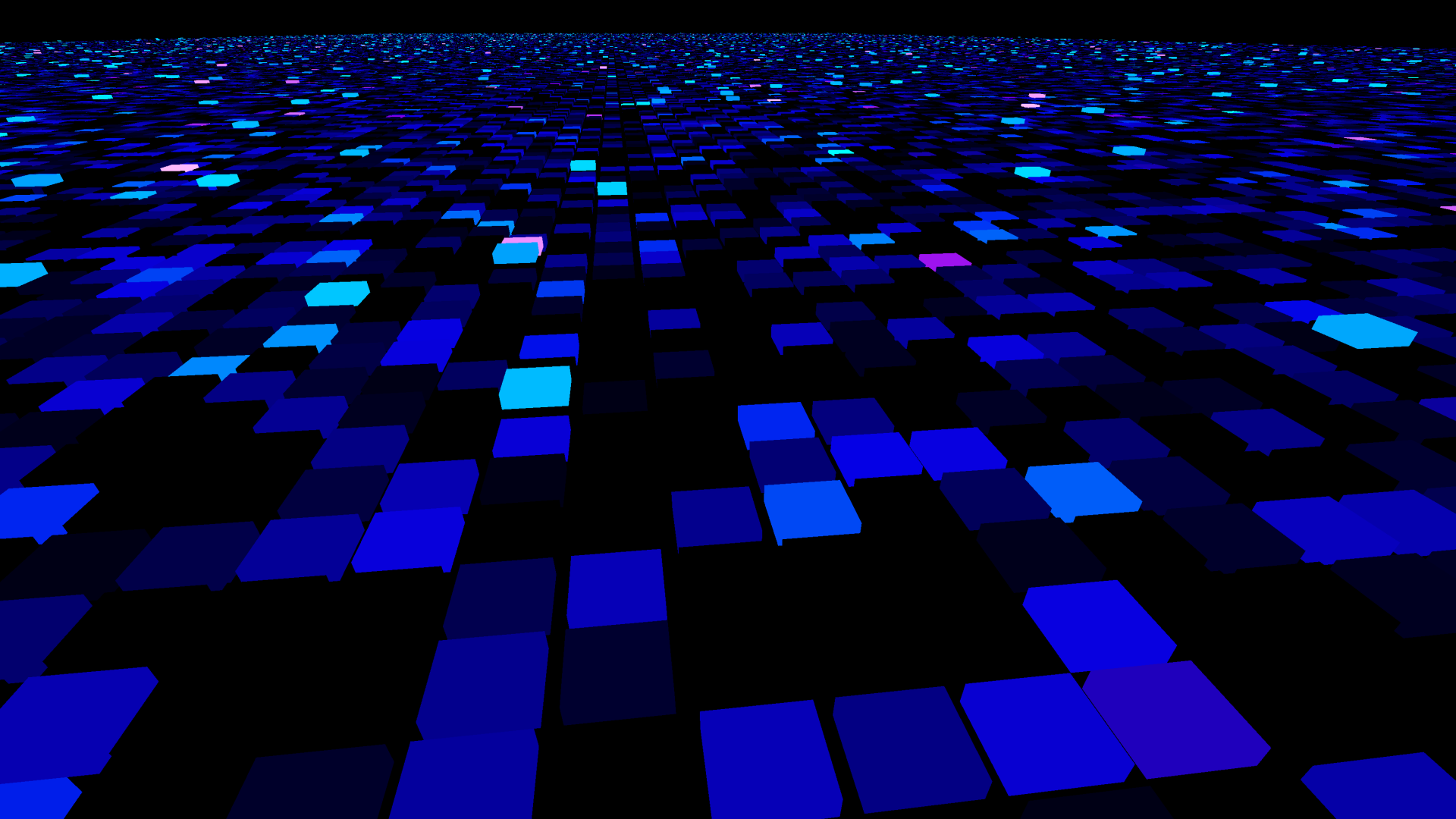



bebylon
BATTLE ROYALE


Galaxy Shape
Galaxy Type
Spiral - Default
Galaxy Size
Medium
Stars
2000
Dust Density
1.0
Advanced Settings
Angle Offset
0,00040
Inner Eccentricity
0,850
Outer Eccentricity
0,950
Core Radius
0,300
Wave Amplitude
340
Wave Frequency
125
0.2.0-pre-alpha


Flecs
Last 15 minutes
5s
FPS
60 Hz
Memory Used
81.9 kB
Entity count
1126
Component count
35
Table count
30
Thread count
1
Memory Allocated
130 kB
System count
53
Active system count
49
Allocations / s
22076
Uptime
00:19:28
FPS
70 Hz
60 Hz
50 Hz
40 Hz
30 Hz
20 Hz
10 Hz
0 Hz
01:32
01:34
01:36
01:38
01:40
01:42
01:44
fps
target
Frame performance breakdown
8.00%
7.00%
6.00%
5.00%
4.00%
3.00%
2.00%
1.00%
frame
merge
system
Memory in use
100 kB
80 kB
60 kB
40 kB
20 kB
0 B
Components
Entities
Systems
Time spent by phase
0.040
0.030
0.020
0.010
0
01:35
01:40
01:45
EcsManual
EcsOnStore
EcsOnUpdate
EcsPostLoad
Time spent by system
35 ms
30 ms
25 ms
20 ms
15 ms
10 ms
5 ms
AddDamage
Move
ProcessDamage
[hidden systems]
Allocations
25 K
20 K
15 K
10 K
5 K
0
calloc
malloc
realloc
free


Flecs Dash
Not Secure | 172.28.128.10:8080
00:00:00.3
57.9 FPS
Load
Find an entity
flecs
Position
Velocity
Move
Type
E1
E2
E3
E4
E5
E6
ID NAME
279 flecs
ID NAME COMPONENT size alignment FLECS.META.STRUCT members is_partial FLECS.META.METATYPE kind size alignment descriptor FLECS.META.M
76 Position 8.0 4.0 [...] false EcsU8 8.0 4.0 { float x; float y; }
77 Velocity 8.0 4.0 [...] false EcsU8 8.0 4.0 { float x; float y; }
ID NAME FLECS.SYSTEM.SIGNATUREEXPR expr
342 Move Position, Velocity
ID NAME
353 Type
ID NAME POSITION x y VELOCITY x y
354 E1 10.0 20.0 0.0 1.0
355 E2 30.0 40.0 1.0 0.0
356 E3 50.0 60.0 1.0 1.0
357 E4 70.0 80.0 2.0 1.0
358 E5 90.0 100.0 1.0 2.0
359 E6 110.0 120.0 2.0 2.0

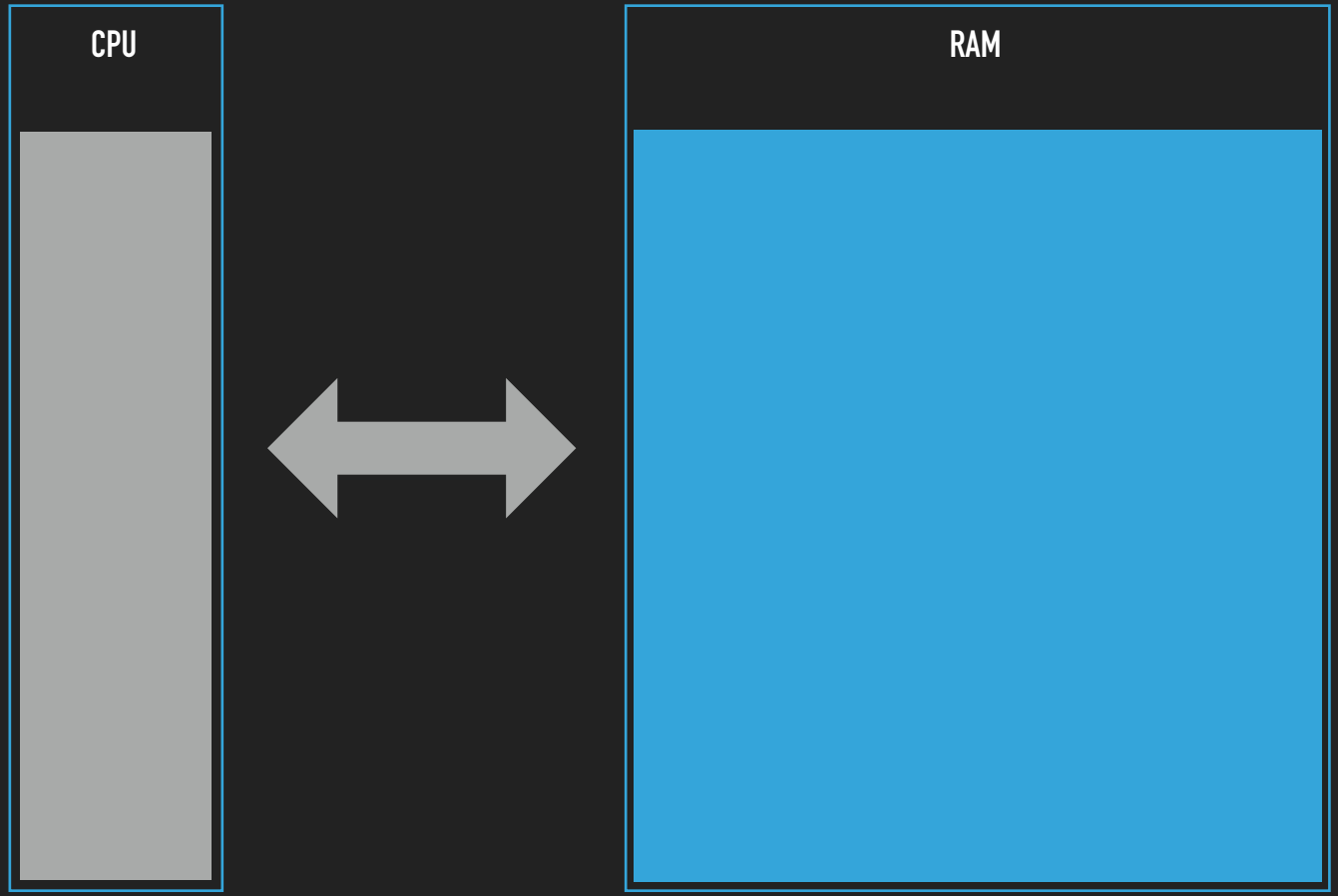


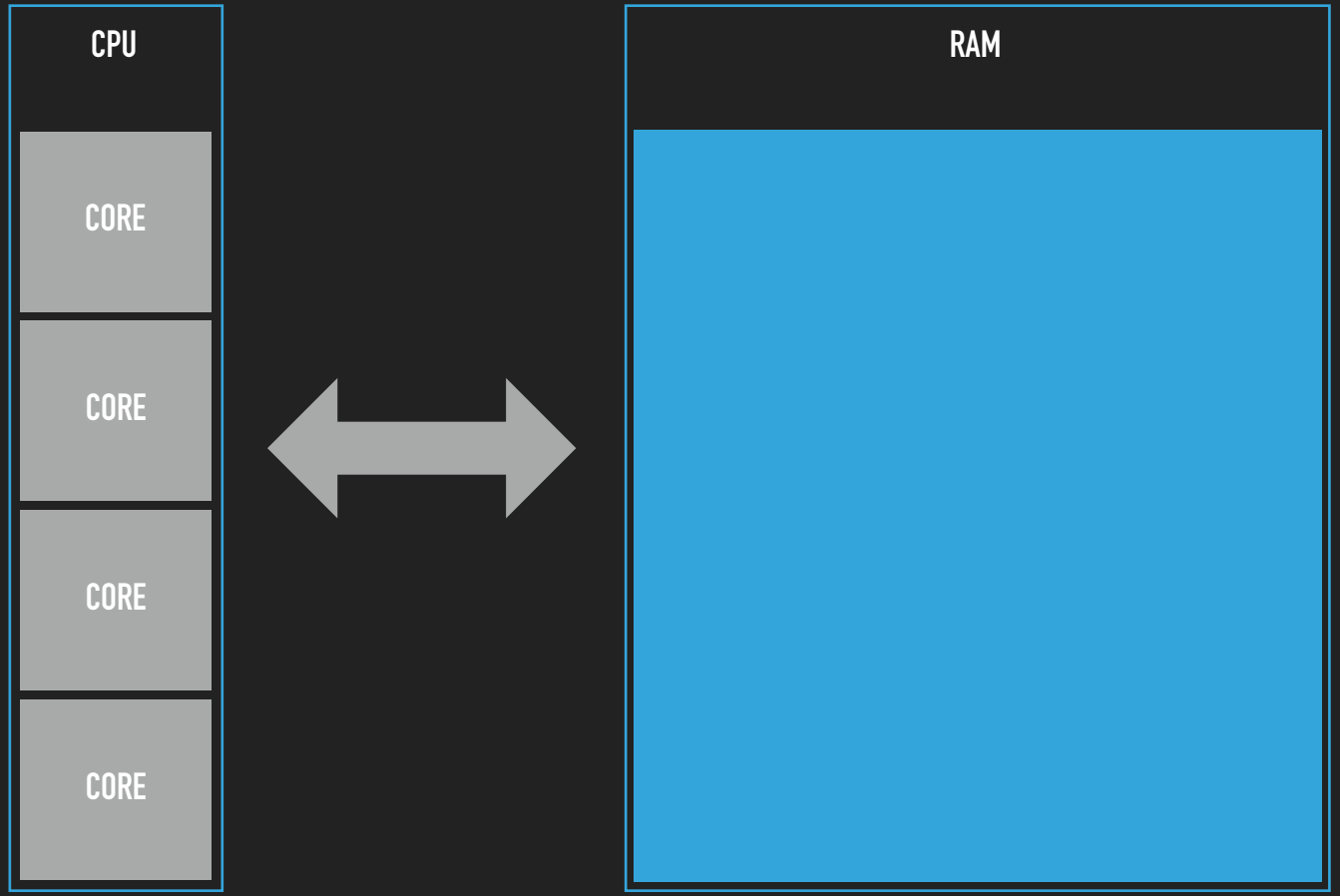
TOPICS

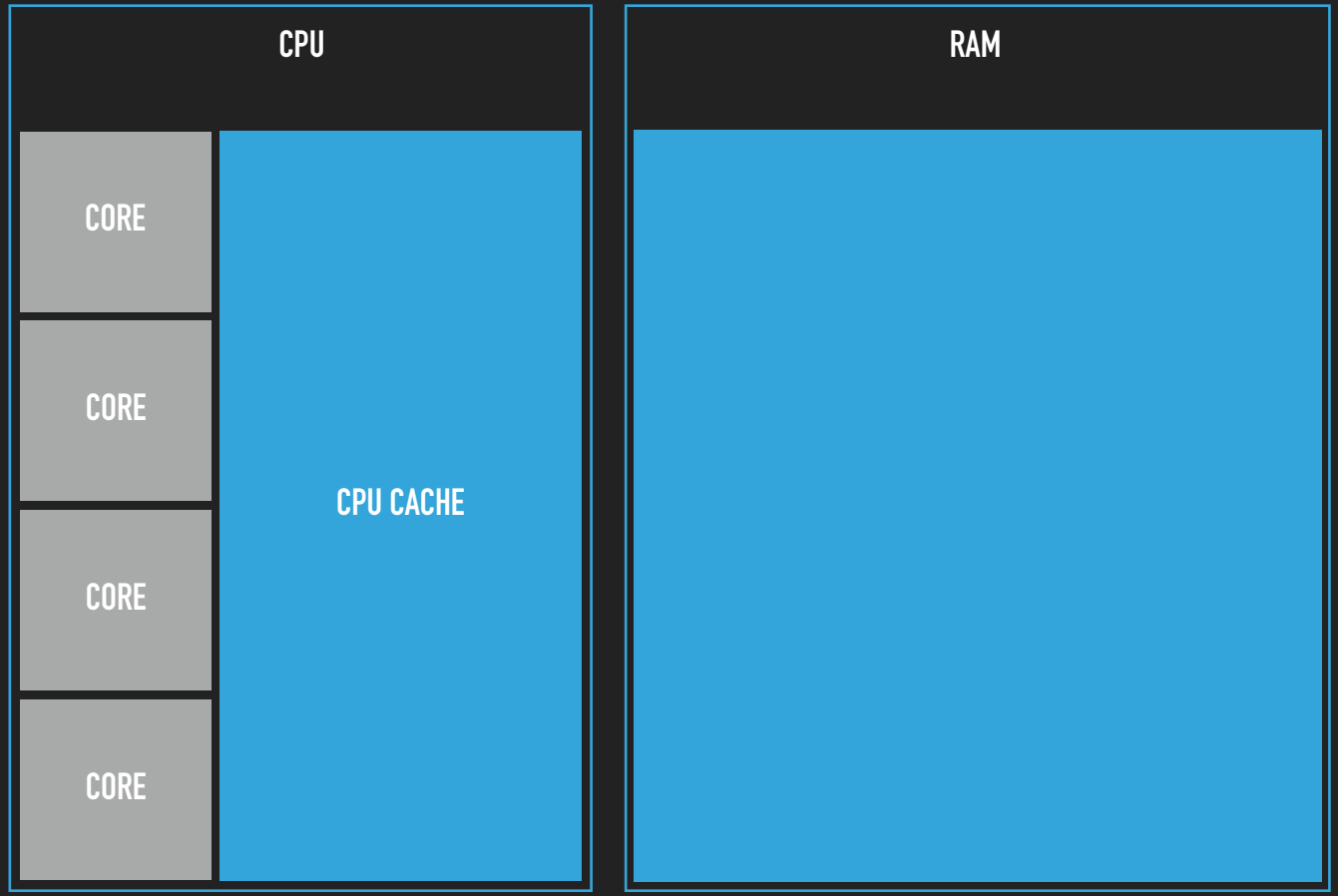
- WRITING FAST CODE
- ECS INTRODUCTION
- ECS ARCHITECTURES
- ECS IN PRACTICE

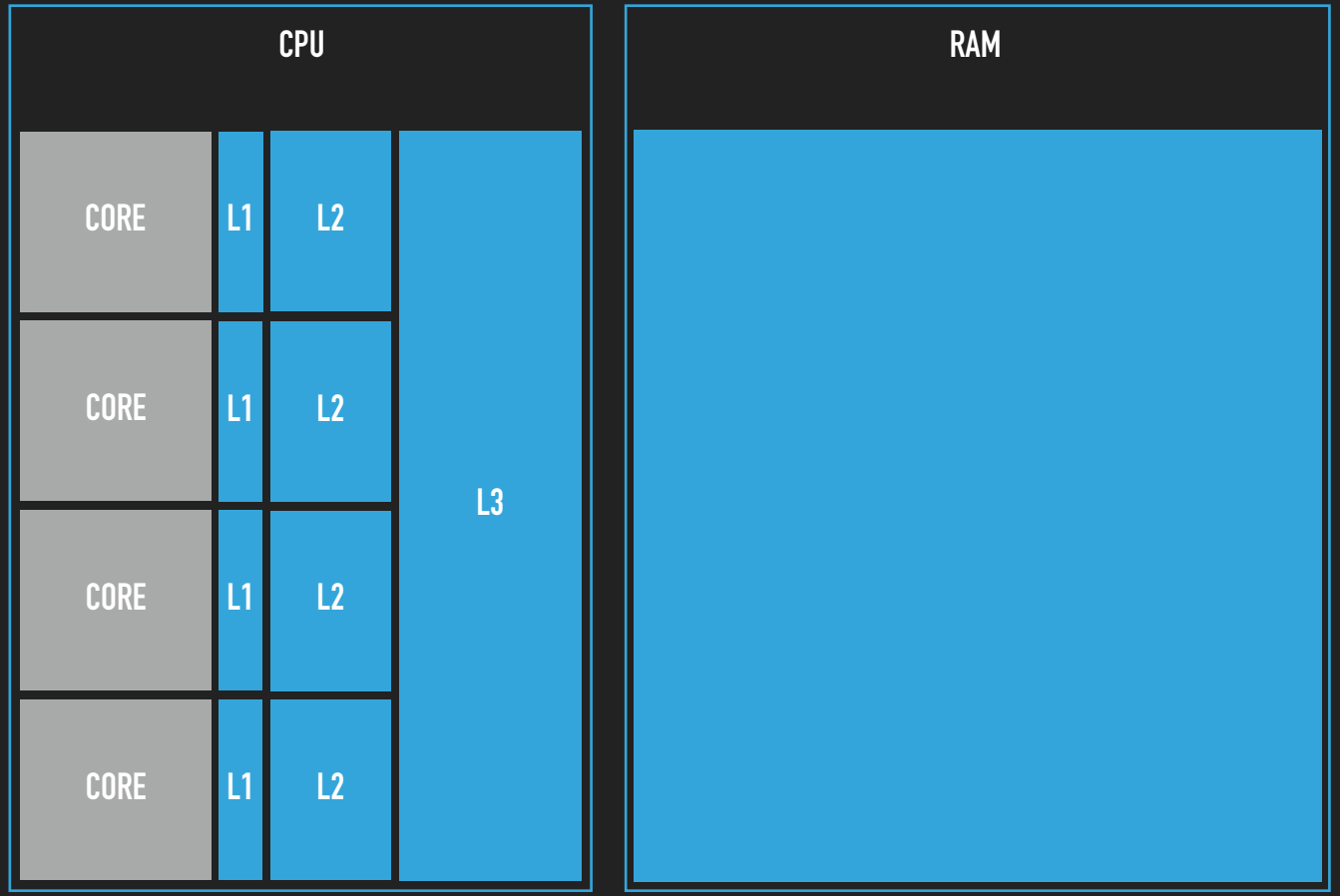
WRITING FAST CODE 1/5

THE CPU

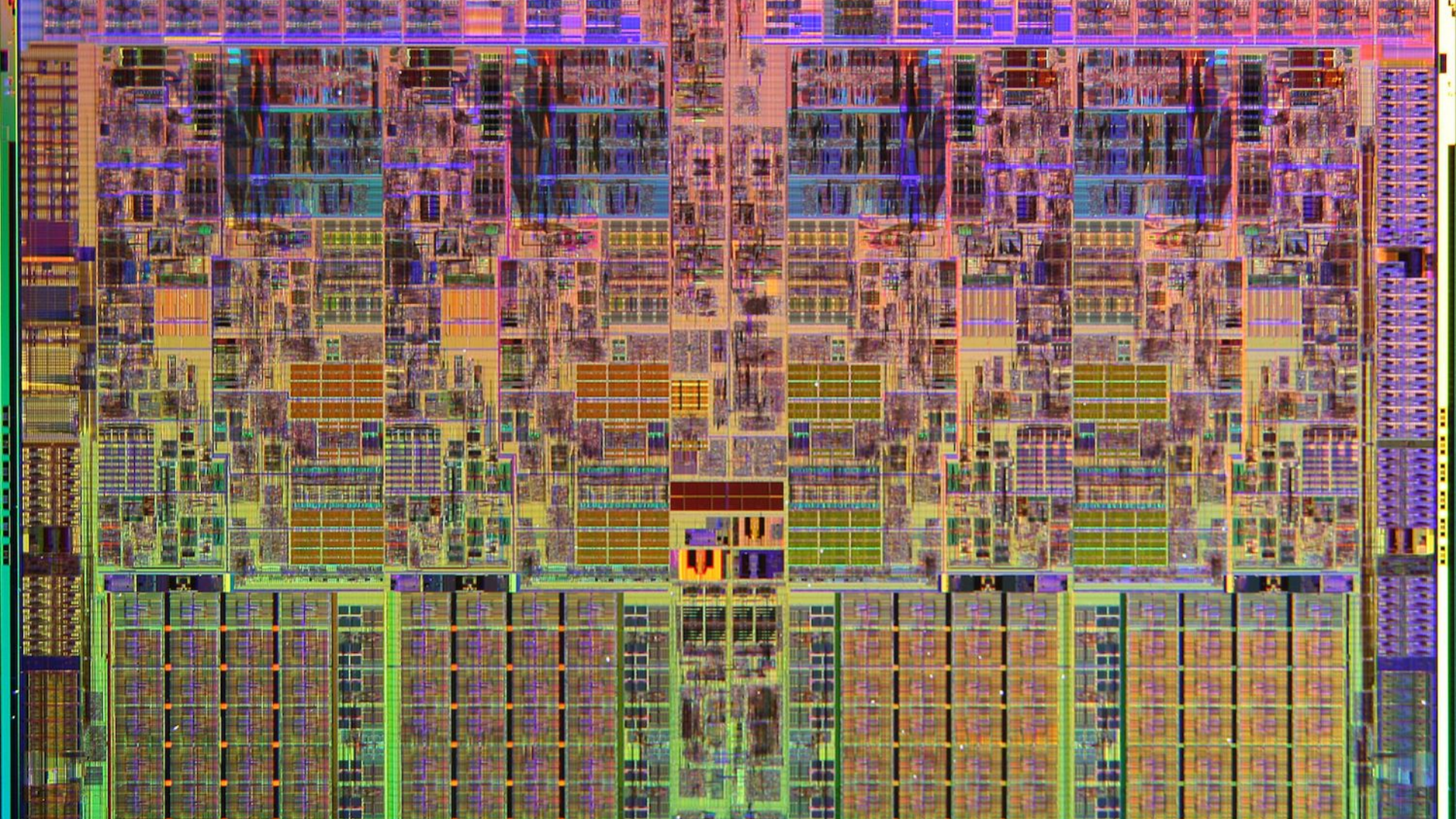

CPU
RAM


CPU
CORE
CORE
CORE
CORE
RAM

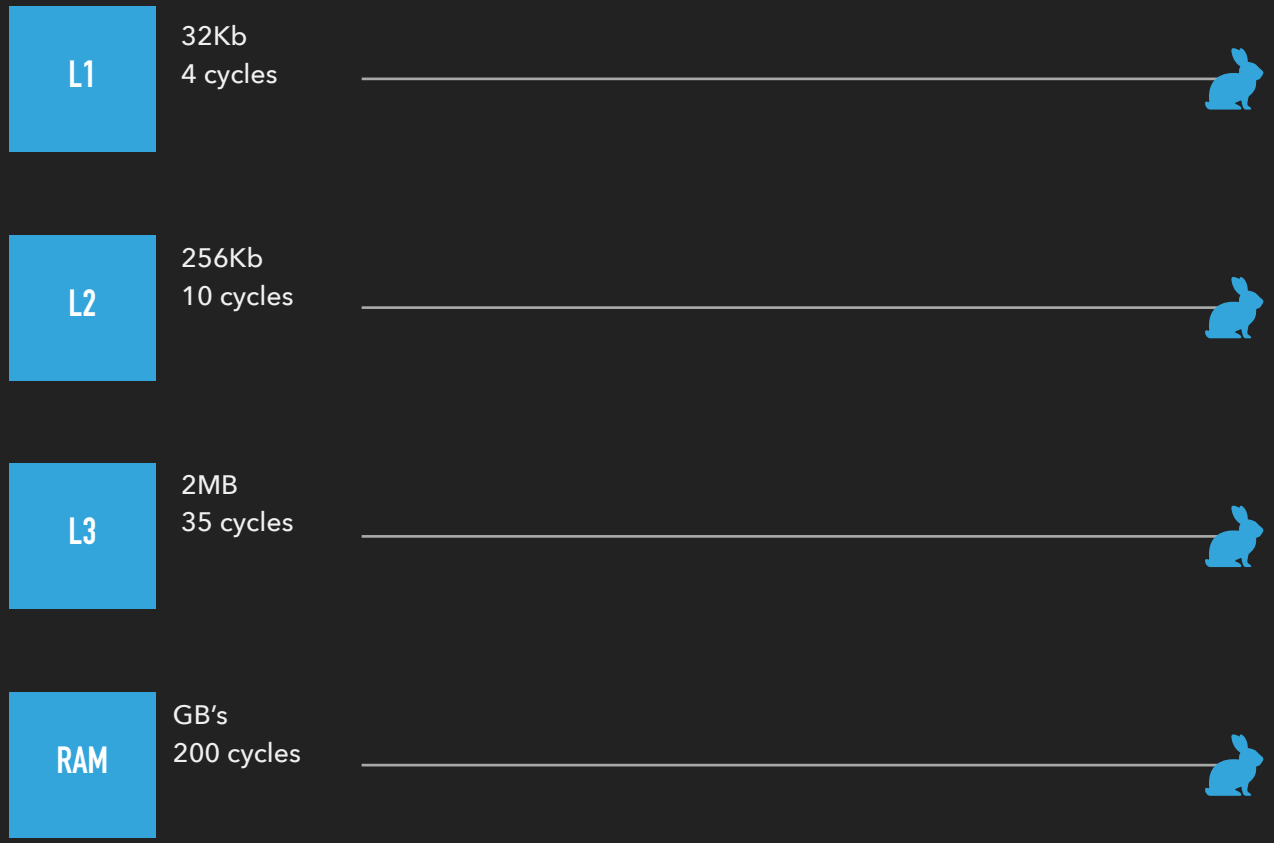

CPU
CORE
CORE
CORE
CORE
CPU CACHE
RAM


CPU
CORE
L1
L2
CORE
L1
L2
CORE
L1
L2
CORE
L1
L2
L3
RAM


RAM




L1
32Kb
L2
256Kb
L3
2MB
RAM
GB's


L1
32Kb
4 cycles
L2
256Kb
10 cycles
L3
2MB
35 cycles
RAM
GB's
200 cycles

GETTING DATA FROM RAM: CACHE LINES

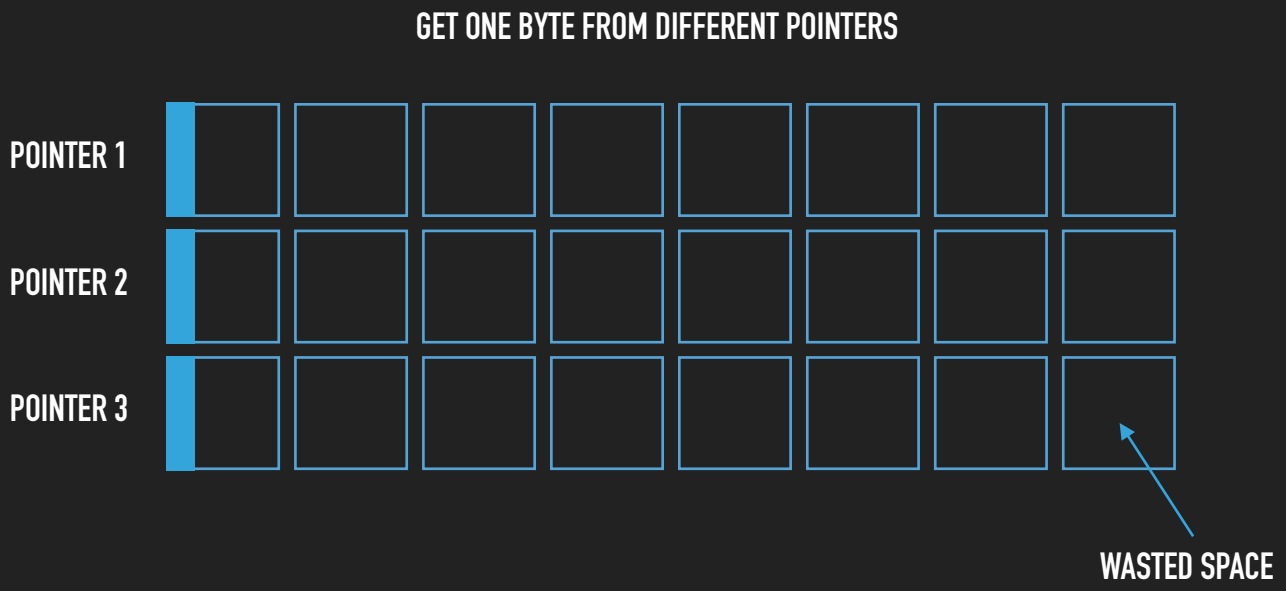
64 BYTES

GETTING DATA FROM RAM: CACHE LINES

GET ONE BYTE

64 BYTES

GETTING DATA FROM RAM: CACHE LINES



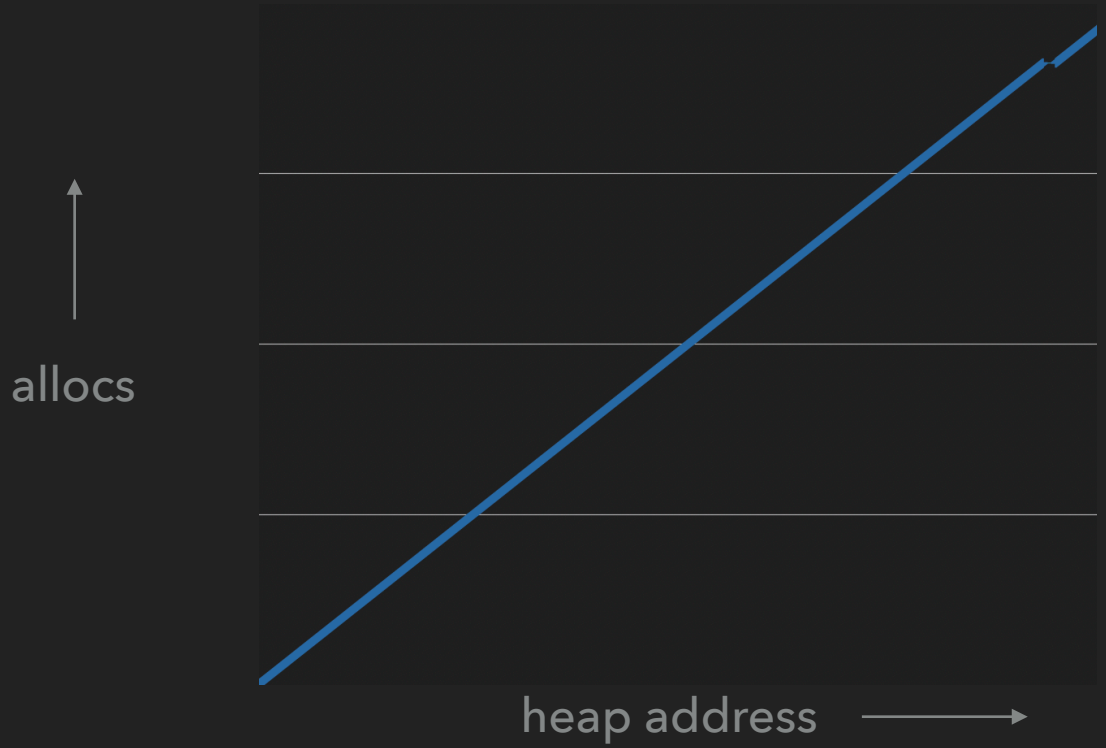
GETTING DATA FROM RAM: CACHE LINES

EACH CACHE LINE FILLS THE CACHE, UNTIL IT IS FULL, AND OLD DATA IS DROPPED

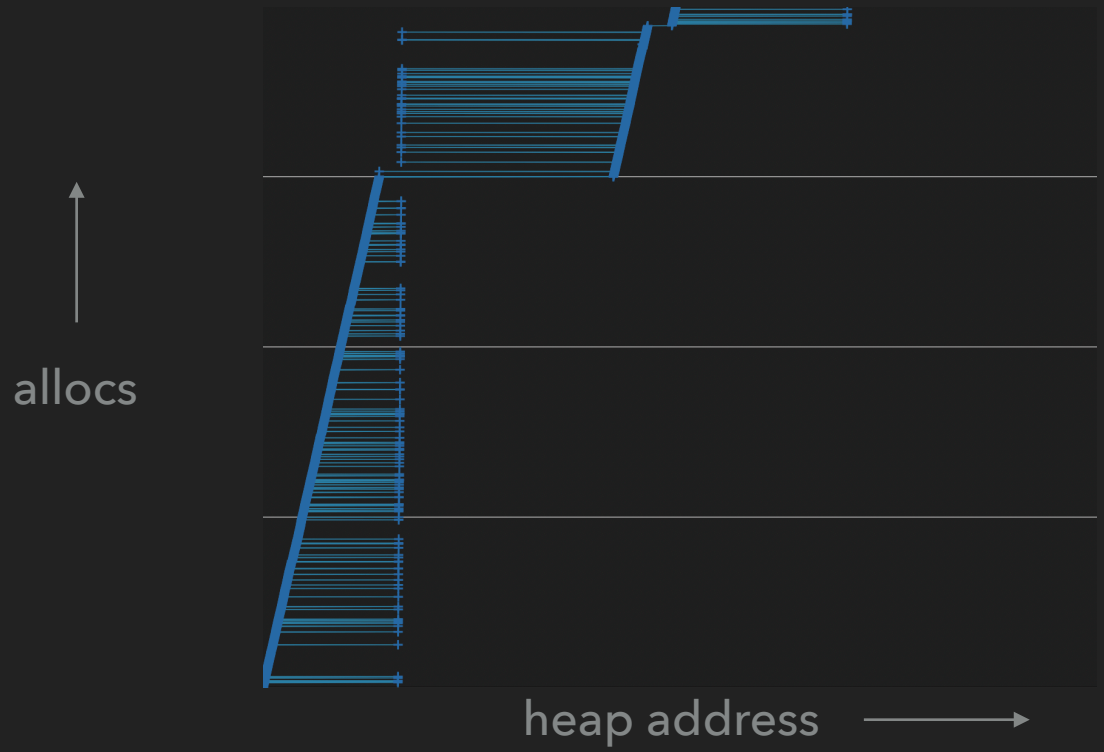
WRITING FAST CODE 2/4

THE HEAP

THE HEAP: MALLOC, SINGLE SIZE



THE HEAP: MALLOC, MULTI SIZE

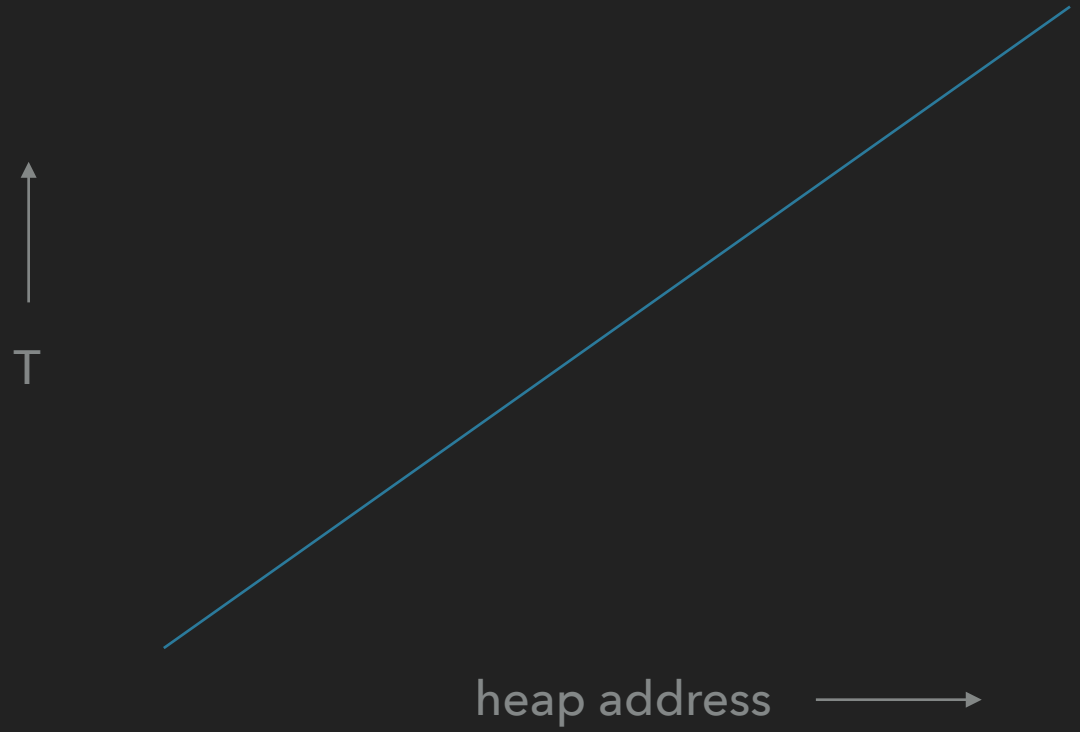


THE HEAP: MALLOC, FREE, MULTI-SIZE

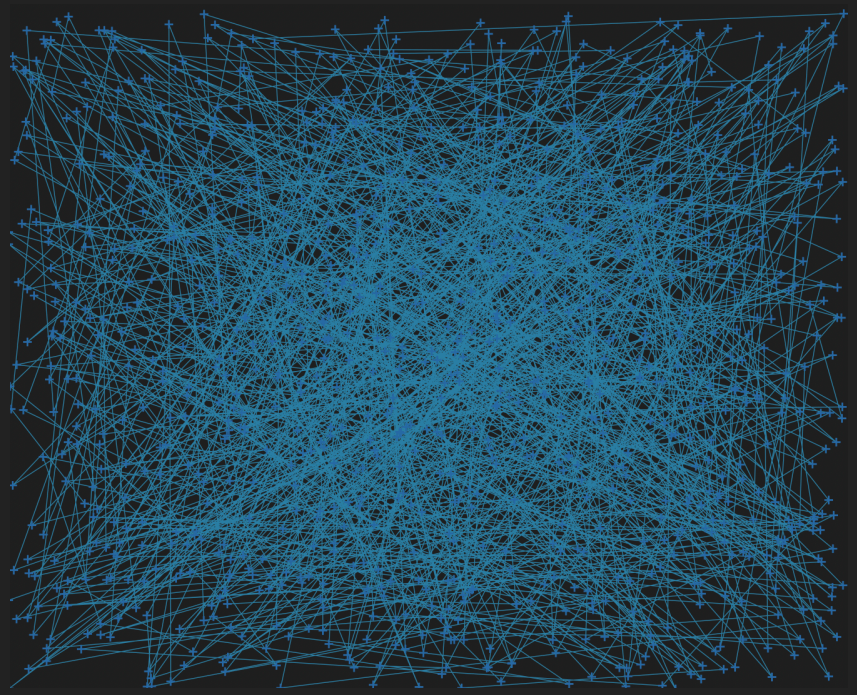
THE HEAP: IDEAL ACCESS PATTERN

100% CACHE LINE UTILIZATION

THE HEAP: IDEAL ACCESS PATTERN



THE HEAP: TYPICAL ACCESS PATTERN

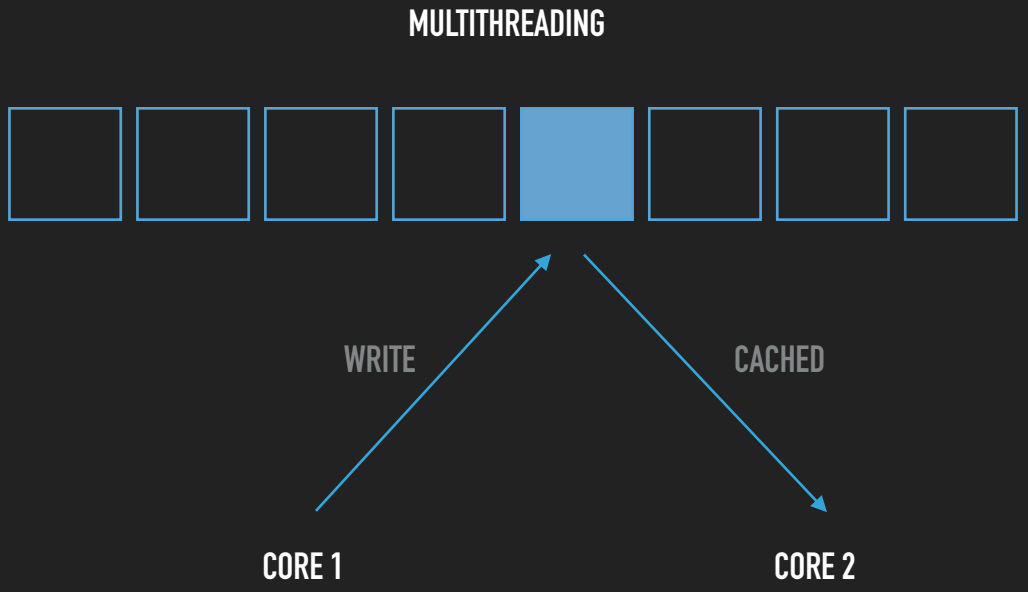



3447
3605 ATL

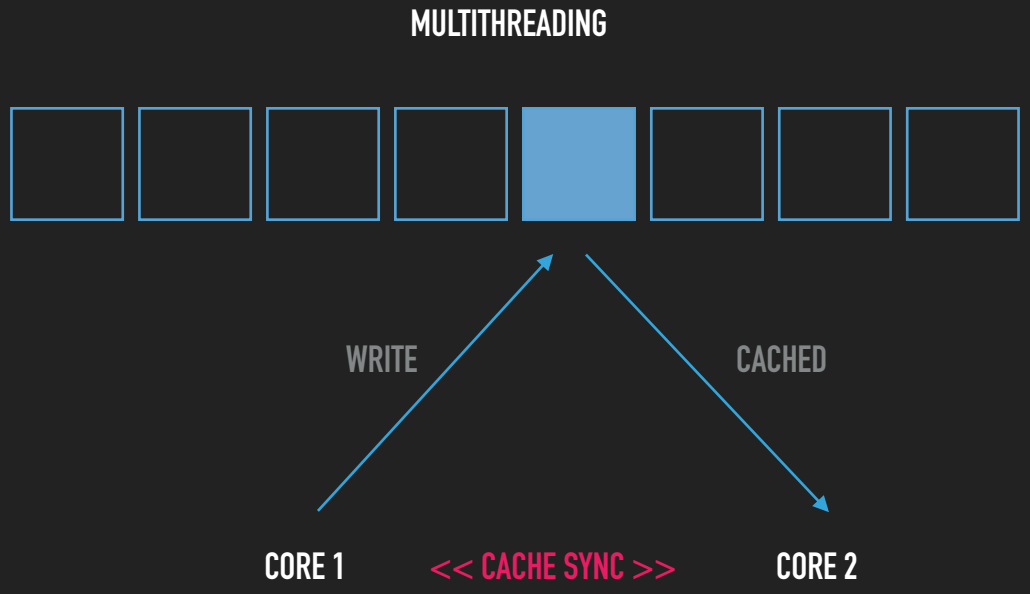
WRITING FAST CODE 3/5

MULTI THREADING

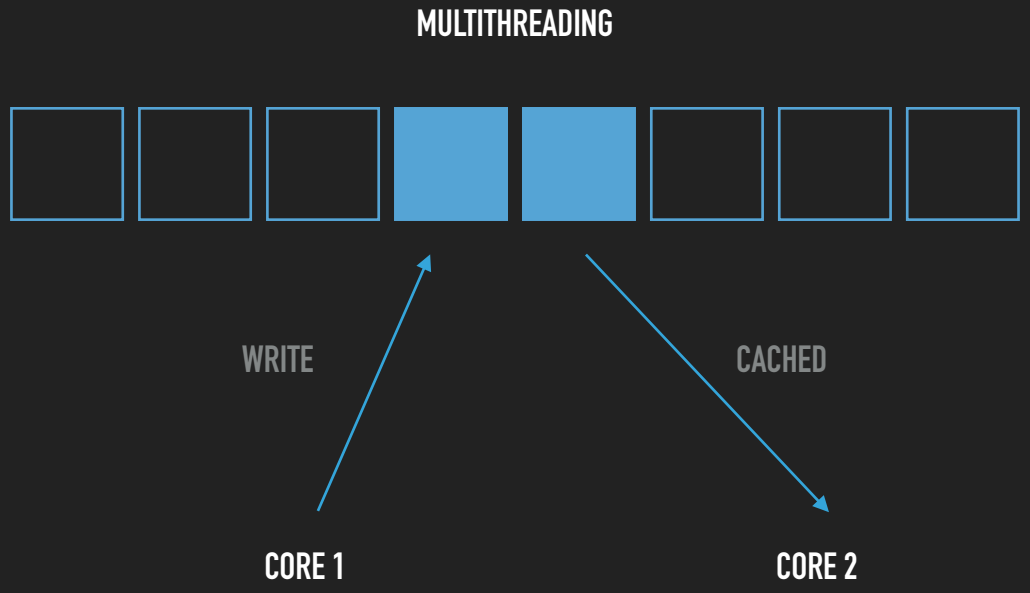
ACCESSING DATA FROM MULTIPLE CORES



ACCESSING DATA FROM MULTIPLE CORES

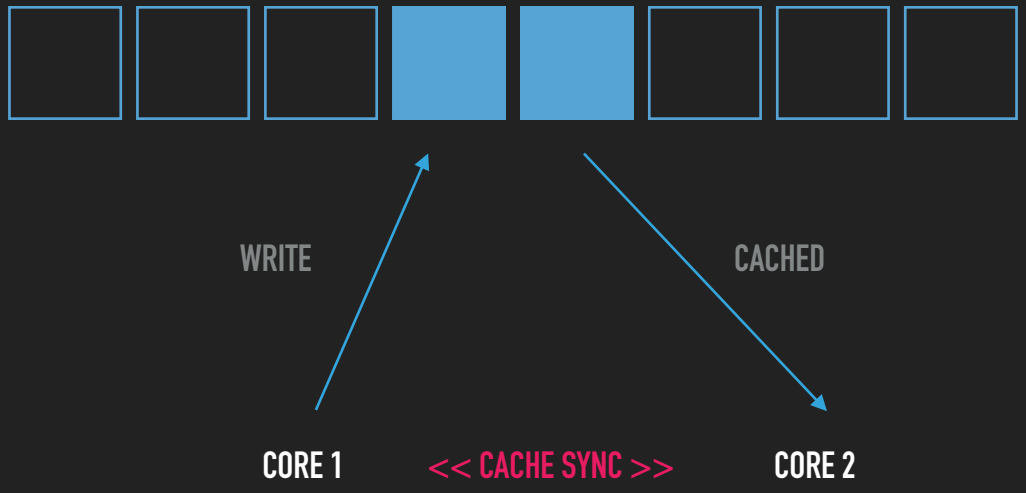


ACCESSING DATA FROM MULTIPLE CORES



ACCESSING DATA FROM MULTIPLE CORES

THIS IS CALLED FALSE SHARING



ACCESSING DATA FROM MULTIPLE CORES

```
struct Wizard {
    float x;
    float y;
    float health;
    float mana;
}
```

used by thread 1

used by thread 2

Yay, lock free ☺

ACCESSING DATA FROM MULTIPLE CORES

```
struct Wizard {
    float x;
    float y;
    float health;
    float mana;
}
```

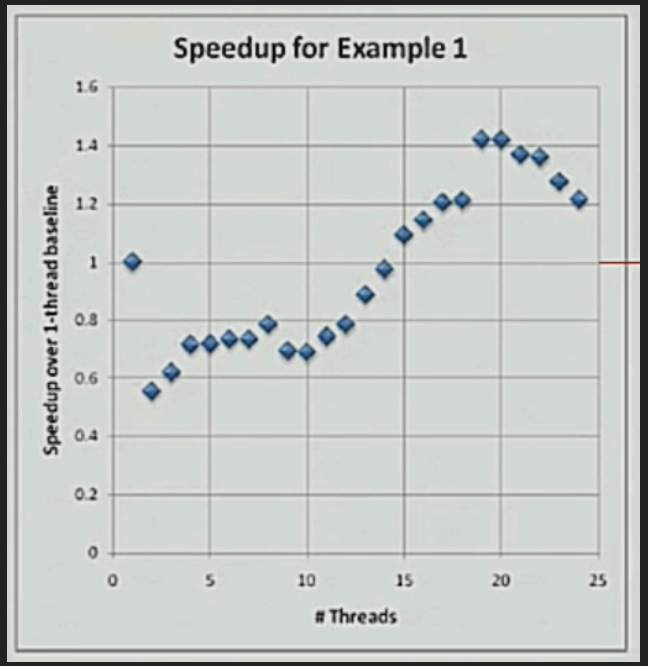
used by thread 1

used by thread 2

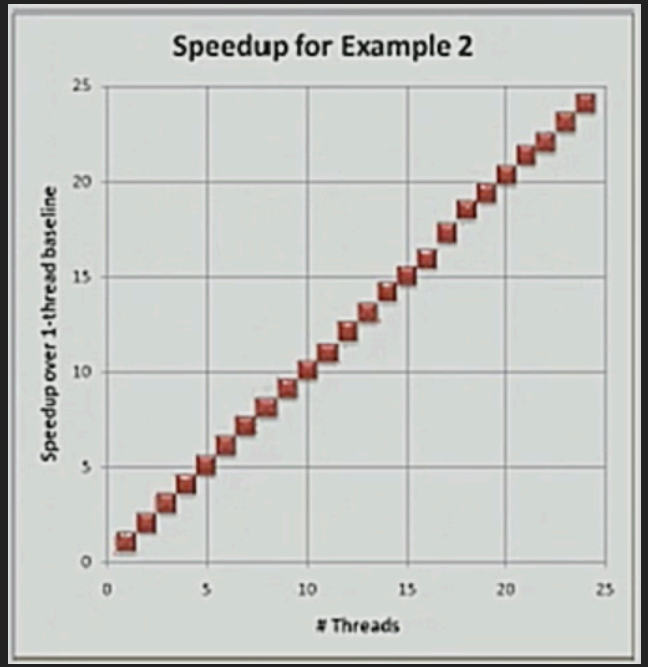
SORRY YOUR CPU IS STILL GOING TO LOCK & SYNC

CALCULATING THE SUM OF AN ARRAY WITH THREADS

FALSE SHARING



NO FALSE SHARING



Scott Meyers - CPU caches and why you care: https://vimeo.com/97337258

WRITING FAST CODE 4/5

DATA ORIENTED DESIGN

EXAMPLE 1: CREATE OBJECTS ON THE HEAP

```
struct Wizard {
    float x;
    float y;
    float health;
    float mana;
}

Wizard *wizards[1000];

for (int i = 0; i < 1000; i ++) {
    wizards[i] = new Wizard();
}
```

EXAMPLE 1: CREATE OBJECTS ON THE HEAP

```
for (int i = 0; i < 1000; i ++) {
    wizards[i]->x ++;
    wizards[i]->y ++;
}
```

EXAMPLE 1: CREATE OBJECTS ON THE HEAP

```
for (int i = 0; i < 1000; i ++) {
    wizards[i]->x ++;
    wizards[i]->y ++;
}
```

CACHE MISS

EXAMPLE 1: CREATE OBJECTS ON THE HEAP

LOTS OF WASTED CACHE LINES

EXAMPLE 1: CREATE OBJECTS ON THE HEAP

```
for (int i = 0; i < 1000; i ++) {
    wizards[i]->x ++;
    wizards[i]->y ++;
}

// Thread 2:

for (int i = 0; i < 1000; i ++) {
    wizards[i]->health ++;
}
```

FALSE SHARING

We can do better than that.

EXAMPLE 2: CREATE ARRAY OF STRUCTS (AOS)

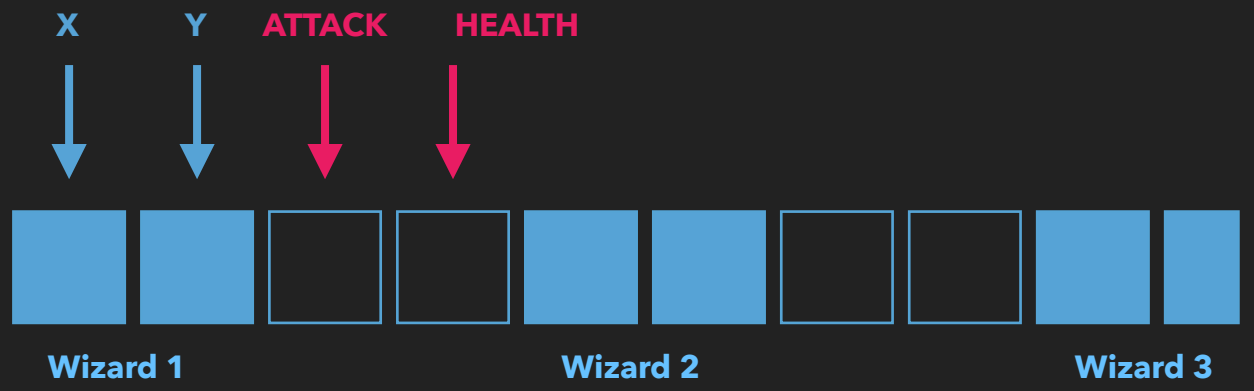
```
struct Wizard {
    float x;
    float y;
    float attack;
    float health;
}

Wizard *wizards = new Wizard[1000];
```

EXAMPLE 2: CREATE ARRAY OF STRUCTS (AOS)

```
for (int i = 0; i < 1000; i ++) {
    wizards[i].x ++;
    wizards[i].y ++;
}
```

NO CACHE MISSES, BUT...

EXAMPLE 2: CREATE ARRAY OF STRUCTS (AOS)



STILL NOT FULLY UTILIZING CACHE LINES

EXAMPLE 2: CREATE ARRAY OF STRUCTS (AOS)

```
for (int i = 0; i < 1000; i ++) {
    wizards[i].x ++;
    wizards[i].y ++;
}
```

WE STILL HAVE FALSE SHARING

```
for (int i = 0; i < 1000; i ++) {
    wizards[i].health ++;
}
```

We can do better than that.

EXAMPLE 3: CREATE STRUCT OF ARRAYS (SOA)

```
struct Wizards {
    float x[1000];
    float y[1000];
    float attack[1000];
    float health[1000];
}

Wizards *wizards = new Wizards();
```

EXAMPLE 3: CREATE STRUCT OF ARRAYS (SOA)

```
for (int i = 0; i < 1000; i ++) {
    wizards.x[i] ++;
    wizards.y[i] ++;
}
```

EXAMPLE 3: CREATE STRUCT OF ARRAYS (SOA)

```
for (int i = 0; i < 1000; i ++) {
    wizards.x[i] ++;
    wizards.y[i] ++;
}
```

NO CACHE MISSES

EXAMPLE 3: CREATE STRUCT OF ARRAYS (SOA)

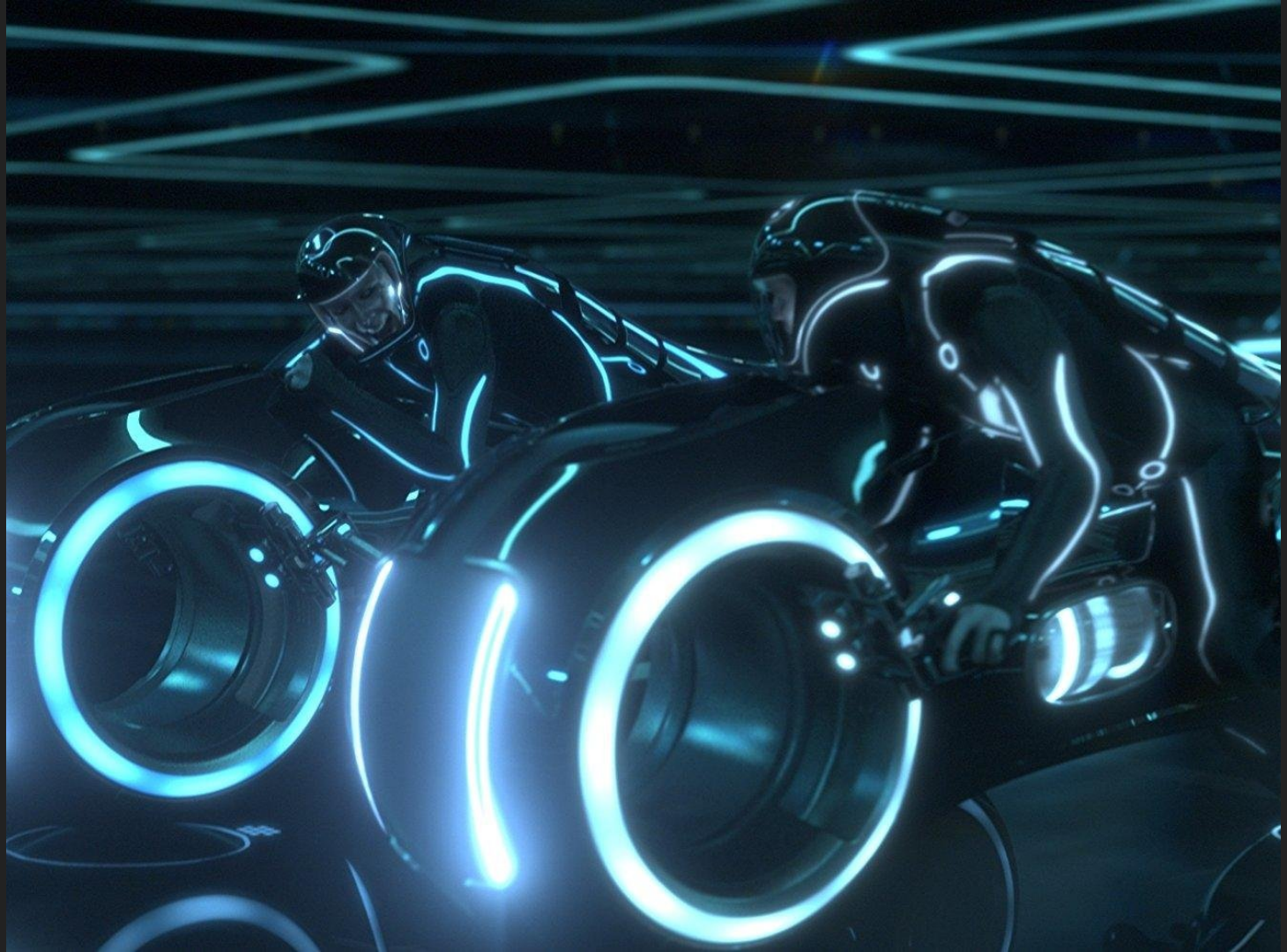
PERFECT CACHE LINE UTILIZATION

EXAMPLE 3: CREATE STRUCT OF ARRAYS (SOA)

```
for (int i = 0; i < 1000; i ++) {
    wizards.x[i] ++;
    wizards.y[i] ++;
}
```

NO FALSE SHARING

```
for (int i = 0; i < 1000; i ++) {
    wizards.health[i] ++;
}
```



WRITING FAST CODE 5/5

BRANCHES

BRANCHES

```
if (left > right_side) {
  return false;
}

if (right < left_side) {
  return false;
}

return true;
```

BRANCHES

```
if (left > right_side) {
  return false;
}

if (right < left_side) {
  return false;
}

return true;
```

Unpredictable branching make it harder for a CPU to predict which code to execute.

Code that contains jumps in the code (like if statements, function calls) cannot be vectorized.

BRANCHES

```
bool left_test = left > right_side;
bool right_test = right < left_side;
return !(left_test | right_test);
```

SAME FUNCTIONALITY, NO BRANCHES

AN INTRODUCTION INTO

ECS

AN INTRODUCTION INTO ECS

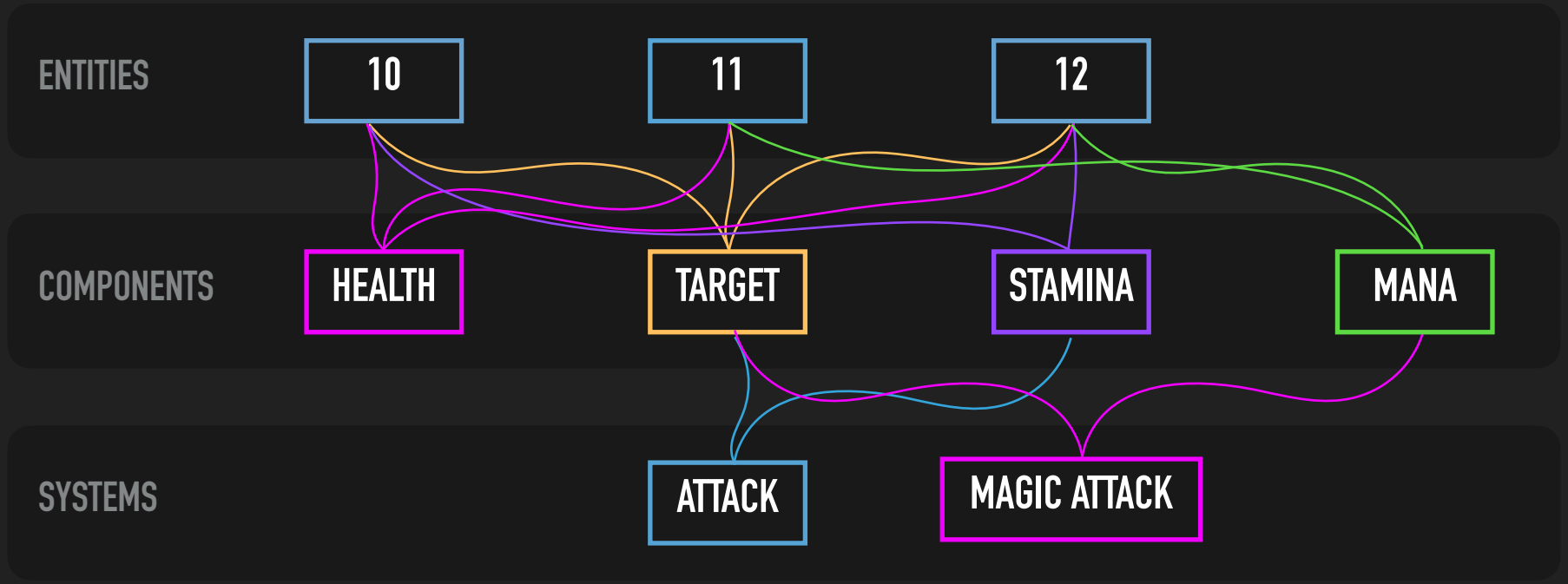
- First used in the early 2000s
- Component based development as we know it, but...
- ECS explicitly separates behavior and data
- Gaining in popularity for both performance and design benefits


GPU 2025 MHz 99 % 68 °C
MEM 6615 MB
CPU 4521 MHz 42 % 46 °C
RAM 10949 MB
D3D11 27 FPS

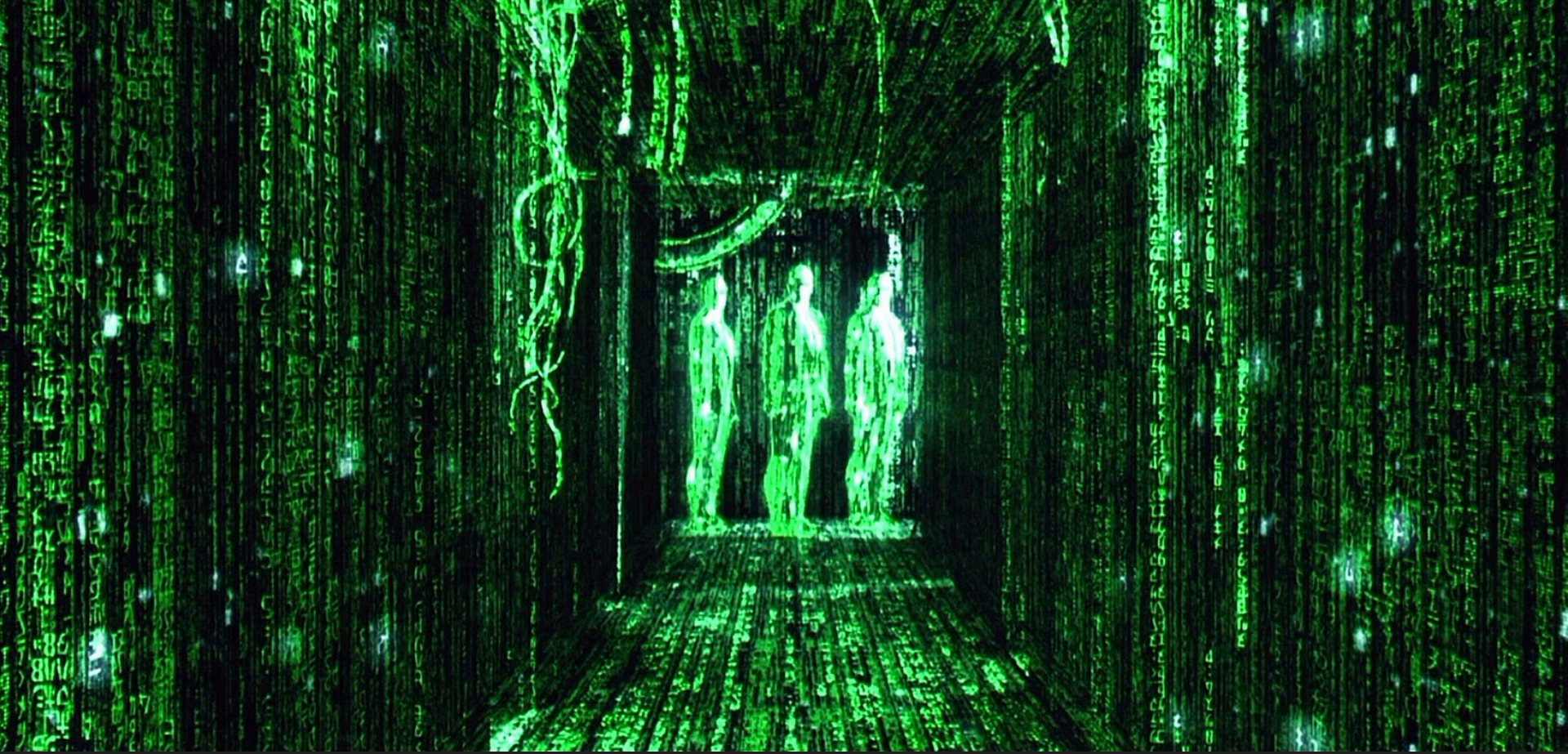
AN INTRODUCTION INTO ECS

- Entity: a unique identifier
- Component: a simple datatype
- Entities can contain 0..N components
- Systems: queries that are matched with entities based on a component list

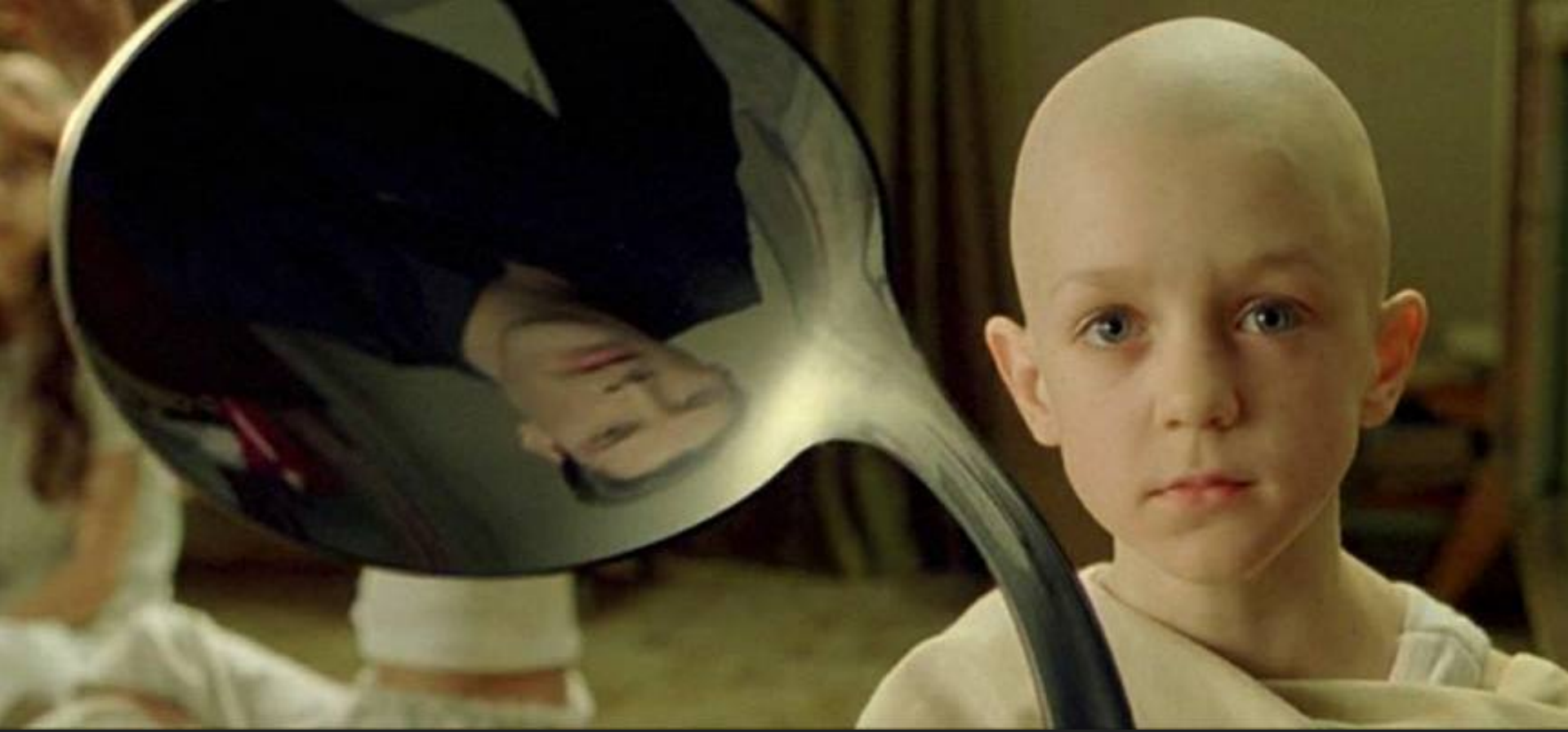
AN INTRODUCTION INTO ECS



OBJECT ORIENTED DESIGN: UNIQUELY IDENTIFIABLE OBJECTS



ENTITY COMPONENT SYSTEMS: EVERYTHING IS DATA



ENTITY COMPONENT SYSTEMS: THERE IS NO SPOON

IN OBJECT ORIENTED PROGRAMMING YOU MODEL WHAT AN OBJECT IS

IN ECS YOU MODEL WHAT AN OBJECT HAS

AN INTRODUCTION INTO ECS

```
class Warrior {
  float x, y;
  float health;
  float stamina;
  void attack(
    Warrior w);
}
```

AN INTRODUCTION INTO ECS

```
class Warrior {
  float x, y;
  float health;
  float stamina;
  void attack(
    Warrior w);
}
```

```
class Wizard {
  float x, y;
  float health;
  float mana;
  void attack(
    Warrior w);
}
```

Warriors can't attack Wizards, and Wizards cannot attack other wizards

AN INTRODUCTION INTO ECS

```
class Unit {
  float x, y;
  float health;
  virtual void
    attack(
      Unit u);
}
```

```
class Warrior: Unit {
  float stamina;
  void attack(
    Unit u);
}
```

```
class Wizard: Unit {
  float mana;
  void attack(
    Unit u);
}
```

AN INTRODUCTION INTO ECS

```
class Unit {
  float x, y;
  float health;
  virtual void
    attack(
      Unit u);
}
```

```
class Warrior: Unit {
  float stamina;
  void attack(
    Unit u);
}

class Horse: Unit {
  float stamina;
  entity mounted;
  void mount(
    Unit u);
}
```

```
class Wizard: Unit {
  float mana;
  void attack(
    Unit u);
}
```

AN INTRODUCTION INTO ECS

```
class Unit {
  float x, y;
  float health;
  virtual void
    attack(
      Unit u);
}
```

A horse doesn't attack

```
class Warrior: Unit {
  float stamina;
  void attack(
    Unit u);
}
```

```
class Wizard: Unit {
  float mana;
  void attack(
    Unit u);
}
```

```
class Horse: Unit {
  float stamina;
  entity mounted;
  void mount(
    Unit u);
}
```

Stamina is duplicated

A horse can't mount a horse

AN INTRODUCTION INTO ECS

```
Warrior {
  << has Position >>
  << has Health >>
  << has Stamina >>
  << has Target >>
}
```

```
Wizard {
  << has Position >>
  << has Health >>
  << has Mana >>
  << has Target >>
}
```

```
Horse {
  << has Position >>
  << has Health >>
  << has Stamina >>
  << has Mounted >>
}
```

ENTITY COMPONENT SYSTEM
ARCHITECTURES

ECS ARCHITECTURES

- Implementations can be categorized in two categories: DENSE and SPARSE
- There is a fierce online debate about which is the “best”
- In reality each implementation has tradeoffs
- Mature implementations often use hybrid approaches

DENSE

DATA IS STORED IN CONTIGUOUS ARRAYS, WITH COMPONENTS LINED UP
ENTITIES WITH SIMILAR COMPONENTS ARE GROUPED TOGETHER
PRO: FAST ITERATION SPEED, GOOD CACHE UTILIZATION, FAST QUERIES
CON: MORE EXPENSIVE TO ADD & REMOVE COMPONENTS, TABLES CAN FRAGMENT MEMORY

SPARSE

DATA IS STORED IN SPARSE DATA STRUCTURES, LIKE SPARSE SETS OR BITSET ARRAYS
EACH COMPONENT HAS ITS OWN SPARSE STRUCTURE, ENTITY ID IS USED AS INDEX
PRO: SIMPLE TO IMPLEMENT, ADD&REMOVE OPERATIONS ARE FAST
CON: SLOWER ITERATION SPEED, SOME OPERATIONS HAVE GLOBAL COMPLEXITY

DENSE

UNITY DOTS, FLECS, BEVY, LEGION, HECS, OUR MACHINERY, SVELTO

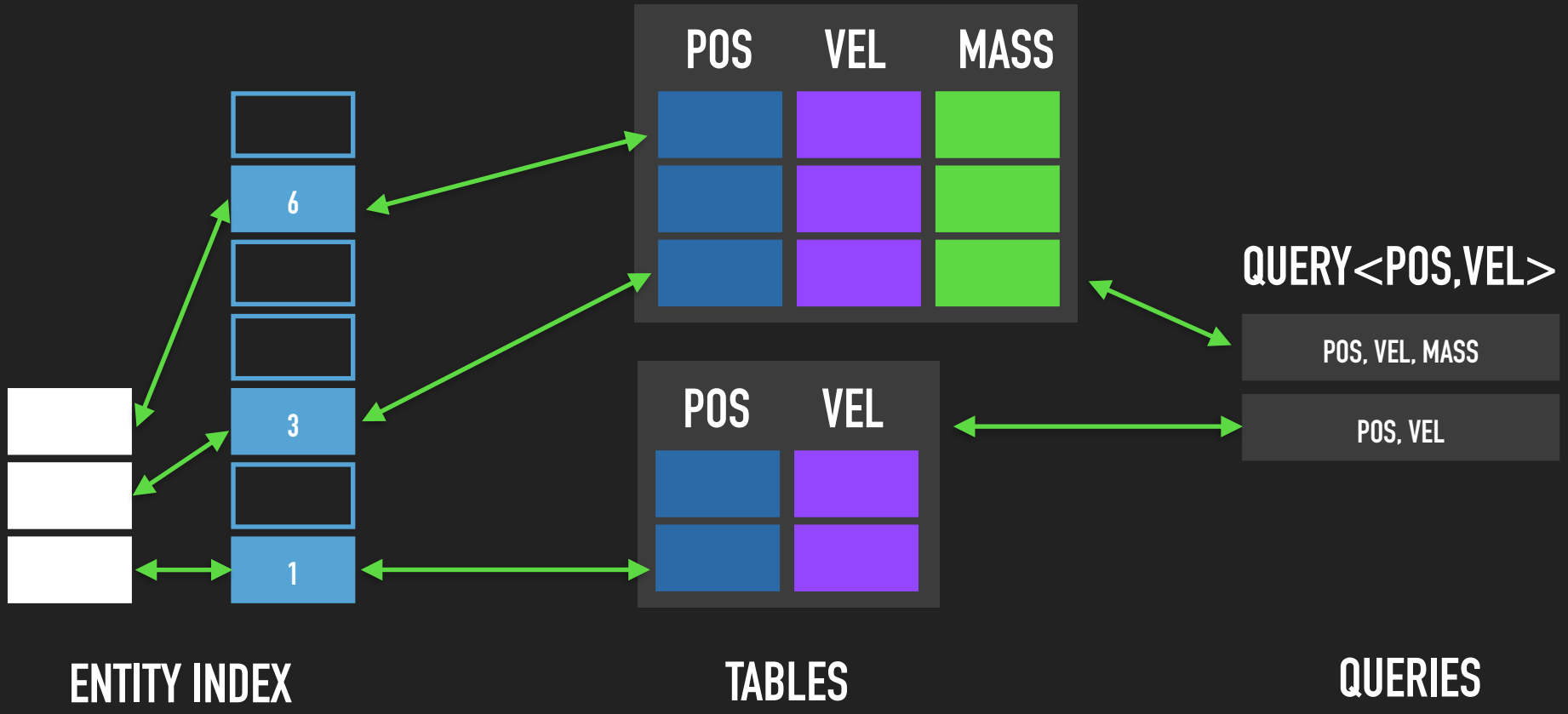
* SOMETIMES REFERRED TO AS “ARCHETYPES”

SPARSE

ENTT, SPECS, ENTITYX

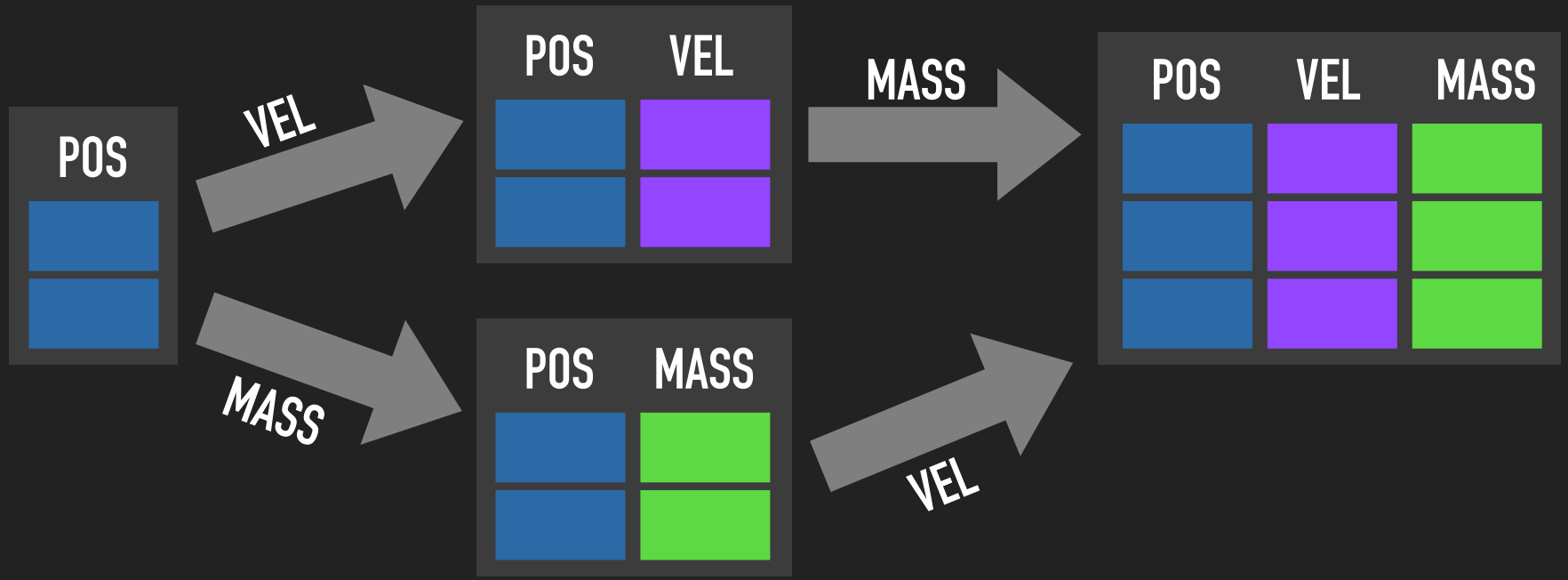
* SPARSE IS A GENERALIZATION, THERE ARE VARIATIONS WITHIN THIS CATEGORY

DENSE

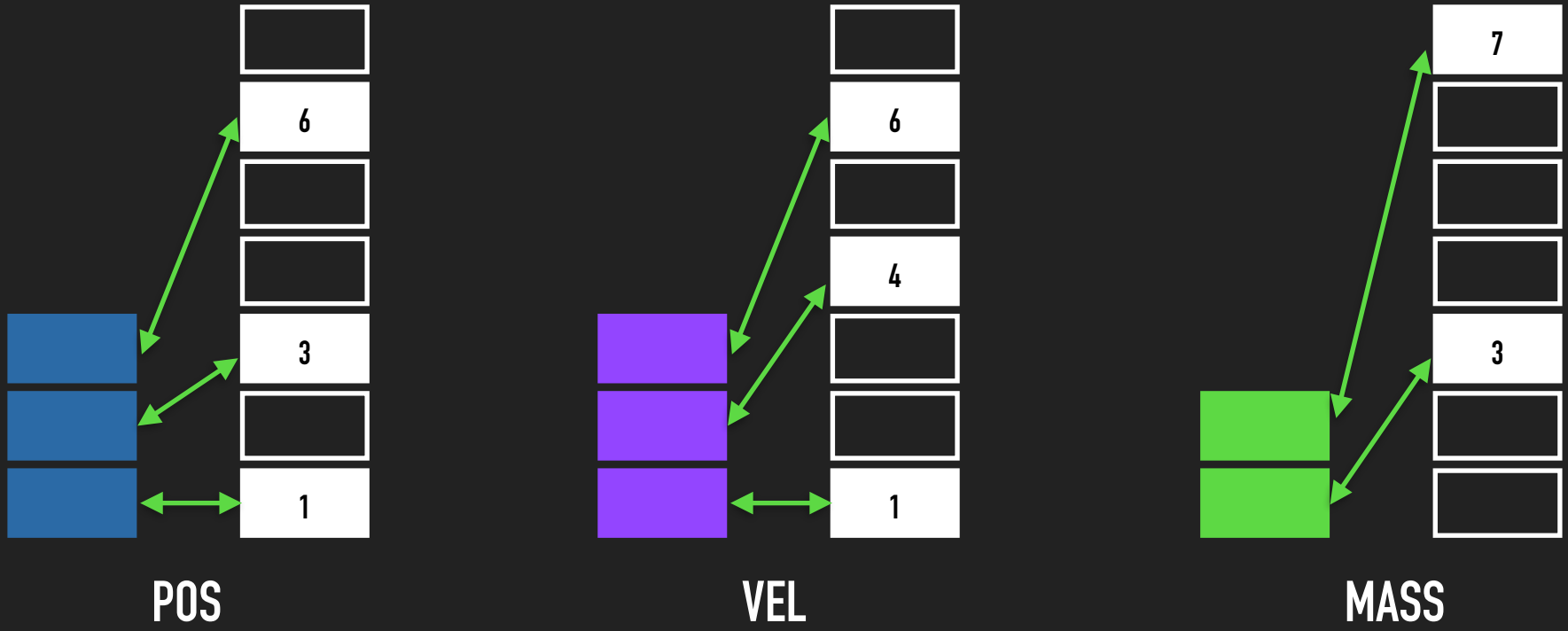

POS
VEL
MASS
6
3
1
POS
VEL
QUERY<POS,VEL>
POS, VEL, MASS
POS, VEL
ENTITY INDEX
TABLES
QUERIES

DENSE

A graph is one of the fastest ways to find a table when adding/removing components



SPARSE



DENSE

FIND ALL ENTITIES FOR {P, V}: O(1), AS QUERIES CAN CACHE TABLES
ADD/REMOVE COMPONENT: O(N), N = NUMBER OF CURRENT COMPONENTS
GET COMPONENT: O(N), N = NUMBER OF CURRENT COMPONENTS, O(1) WHEN CACHED
DELETE ENTITY: O(1)
GET ALL COMPONENTS FOR ENTITY: O(1)

SPARSE

FIND ALL ENTITIES FOR {P, V}: O(N * M), N = SMALLEST SET SIZE, M = COMPS IN QUERY
ADD/REMOVE COMPONENT: O(1)
GET COMPONENT: O(1)
DELETE ENTITY: O(N), N = NUMBER OF COMPONENTS IN WORLD
GET ALL COMPONENTS FOR ENTITY: O(N), N = NUMBER OF COMPONENTS IN WORLD

DENSE

SUPPORT SPARSE MECHANISMS:

UNITY DOTS IS IMPLEMENTING A BITSET PER COMPONENT ARRAY FOR SPARSE DATA

FLECS LETS AN APPLICATION CHANGE TAGS WITHOUT CHANGING TABLES

SPARSE

SUPPORT DENSE MECHANISMS:

ENTT HAS GROUPS, WHICH LETS YOU ALIGN COMPONENT ARRAYS AS LONG AS GROUPS DO NOT HAVE OVERLAPPING COMPONENTS

ENTITY COMPONENT SYSTEMS
IN PRACTICE

ECS IN PRACTICE

- Several projects build engine systems in ECS
- Examples are Unity DOTS, Flecs, Our Machinery, Amethyst and Bevy
- Demo: an example application written with Flecs modules
- Show how to do scene graphs, spatial structures & rendering

DISCLAIMER

The example code is written with Flecs, and takes advantage of features provided by dense ECS frameworks.

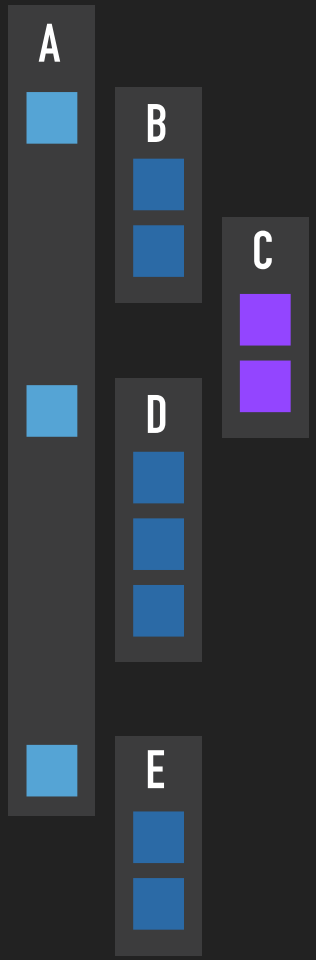
ENGINE SYSTEM #3
RENDERING

RENDERING

- When rendering, we want to pack data in large GPU buffers
- Switching pipelines should be avoided as much as possible
- We can leverage the ECS to create a query per pipeline, and copy the ECS buffers directly into an ECS buffer

ENGINE SYSTEM #1

SCENE GRAPHS



Hierarchical transforms can be complex as it requires traversing the hierarchy breadth-first. One solution is to split up tables per subtree.

QUERY

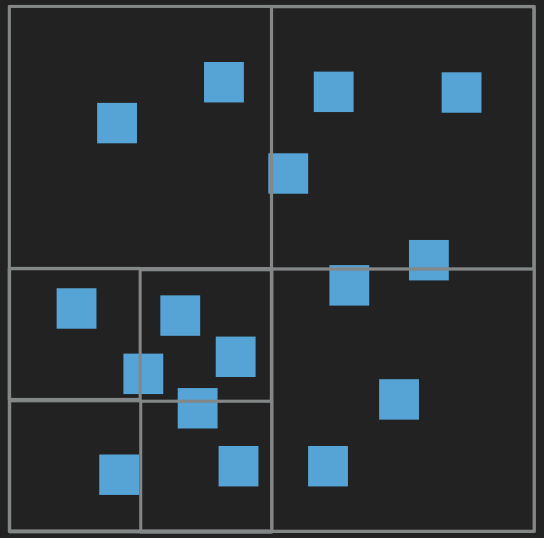
A
B
D
E
C

Tables matched with query are ordered by depth in the hierarchy.

This way a transform system can iterate entities as usual, and apply transforms top to bottom.

ENGINE SYSTEM #2

SPATIAL STRUCTURES



Spatial structures are useful for things like broad phase collision detection and culling (amongst others).

Different strategies can be used:

1. Split up tables across spatial dimensions
2. Store entities in external structure, like octree

1 works well for high-level spatial organization like world chunks, which enables fast culling.

2 works well for granular checks, like collision detection.

THAT'S ALL

THANK YOU!