

fløcs

(not for Dummies)



Agenda

Introduction

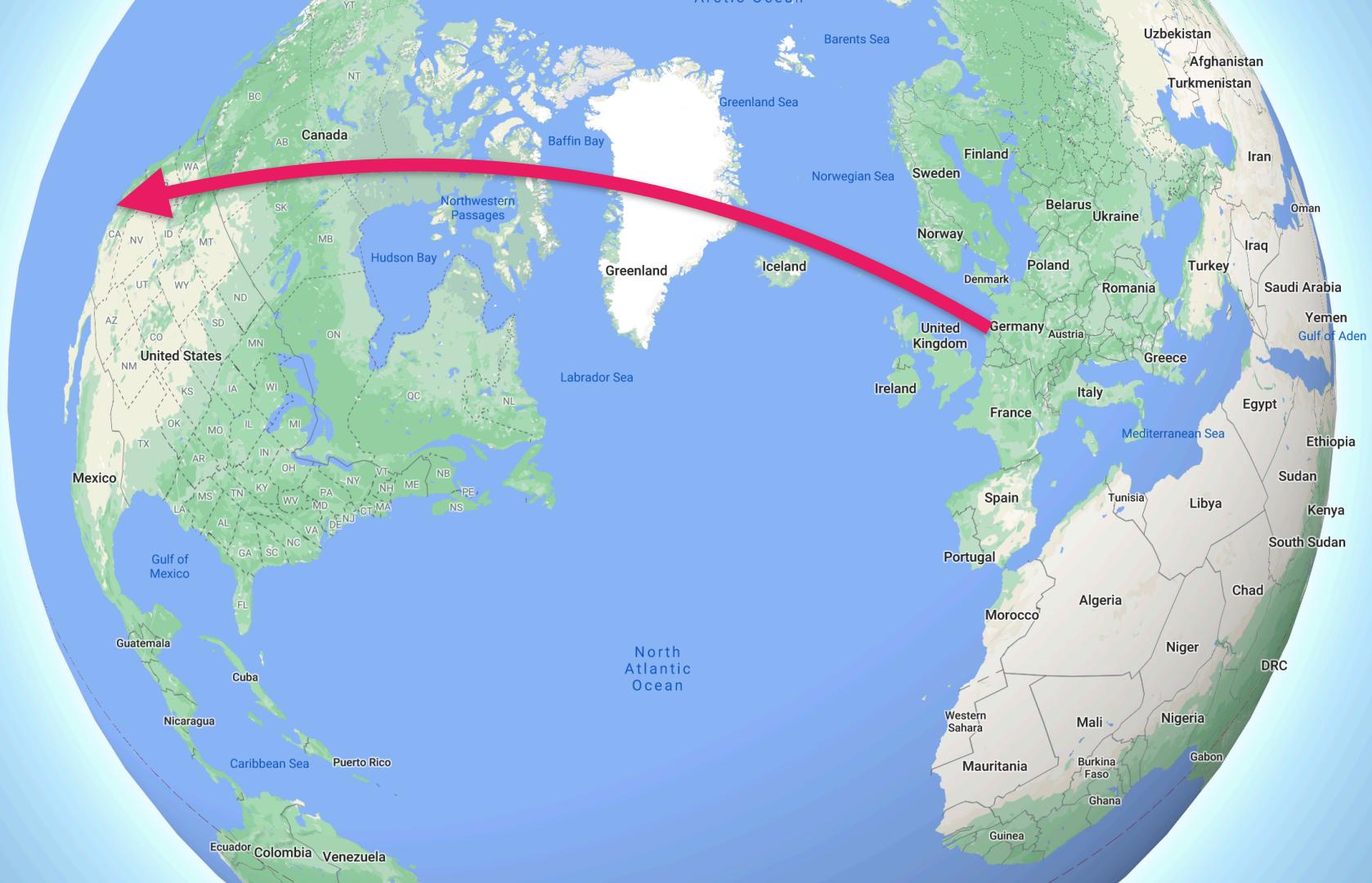
How to write fast code

Flecs Basics

Flecs Advanced

Flecs in Practice

Introduction



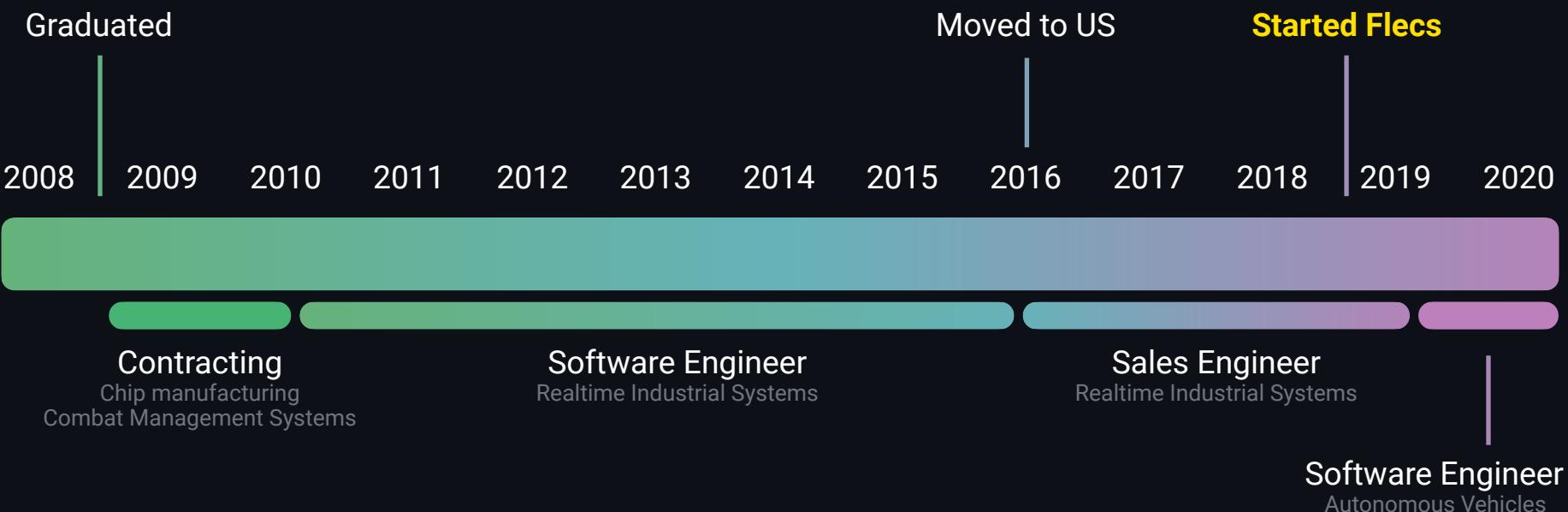






Introduction

Timeline





Introduction

Project Overview

Unwatch ▾

50

Star

1.5k

Fork

105

C99 CORE HIGH PERFORMANCE

C++11 API PORTABLE

OPTIONAL ADDONS LOW FOOTPRINT

MODULE ECOSYSTEM PROMOTES REUSABILITY

ZERO DEPENDENCIES EASY TO USE



Introduction

High Level Concepts

STORAGE Store entities, components, relationships

QUERIES Expression that matches components

SYSTEMS Which code do I run for matched data

PIPELINES When do I run systems

MODULES How are systems & components organized



Introduction

Unique Features

HIERARCHIES Store & query hierarchies natively

RELATIONSHIPS Store & query relationships natively

PREFABS Create & instantiate entity templates

THREADING Thread-safe command queue

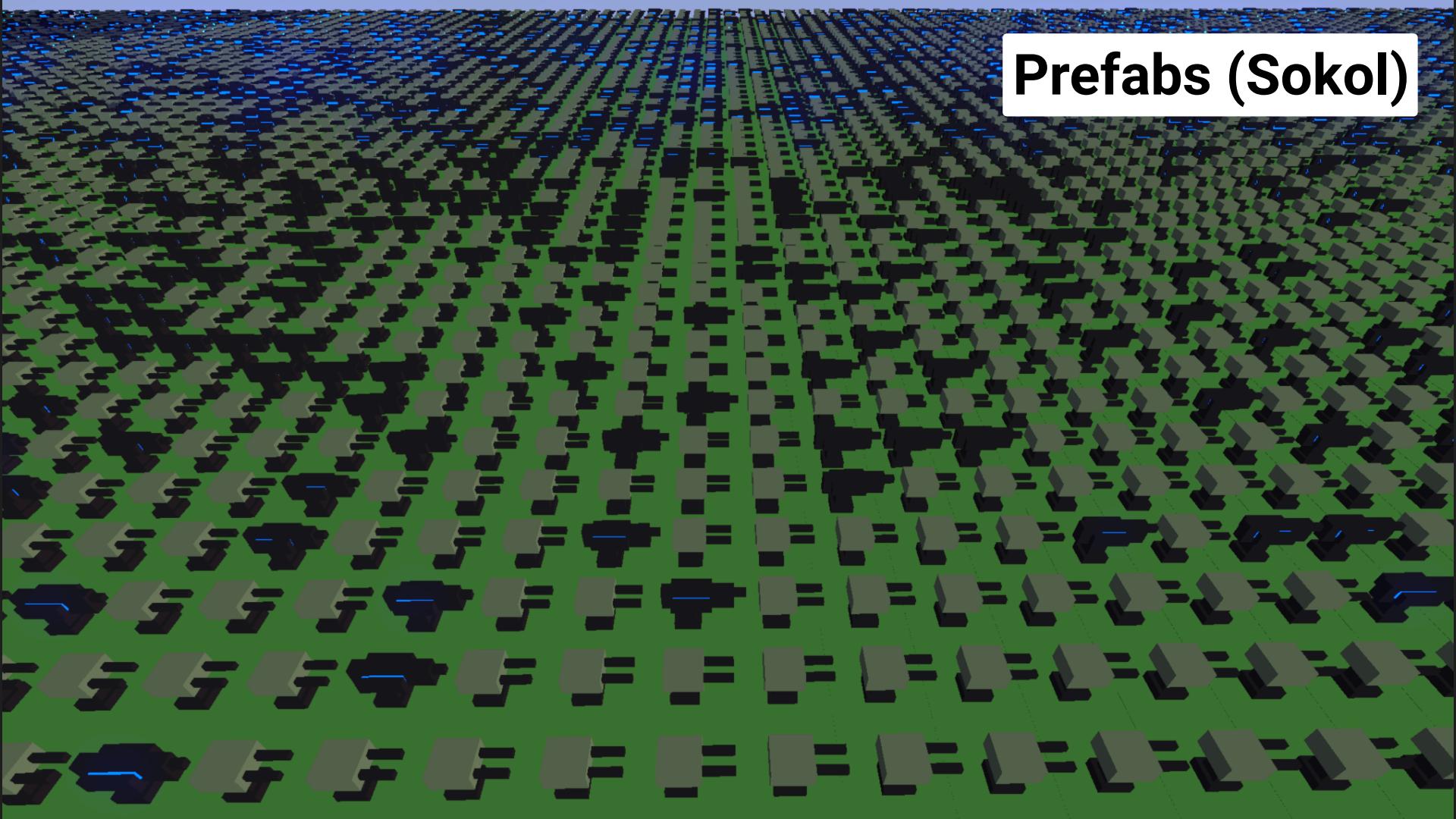
TOOLING Statistics, dashboard, reflection

Dashboard (Web)



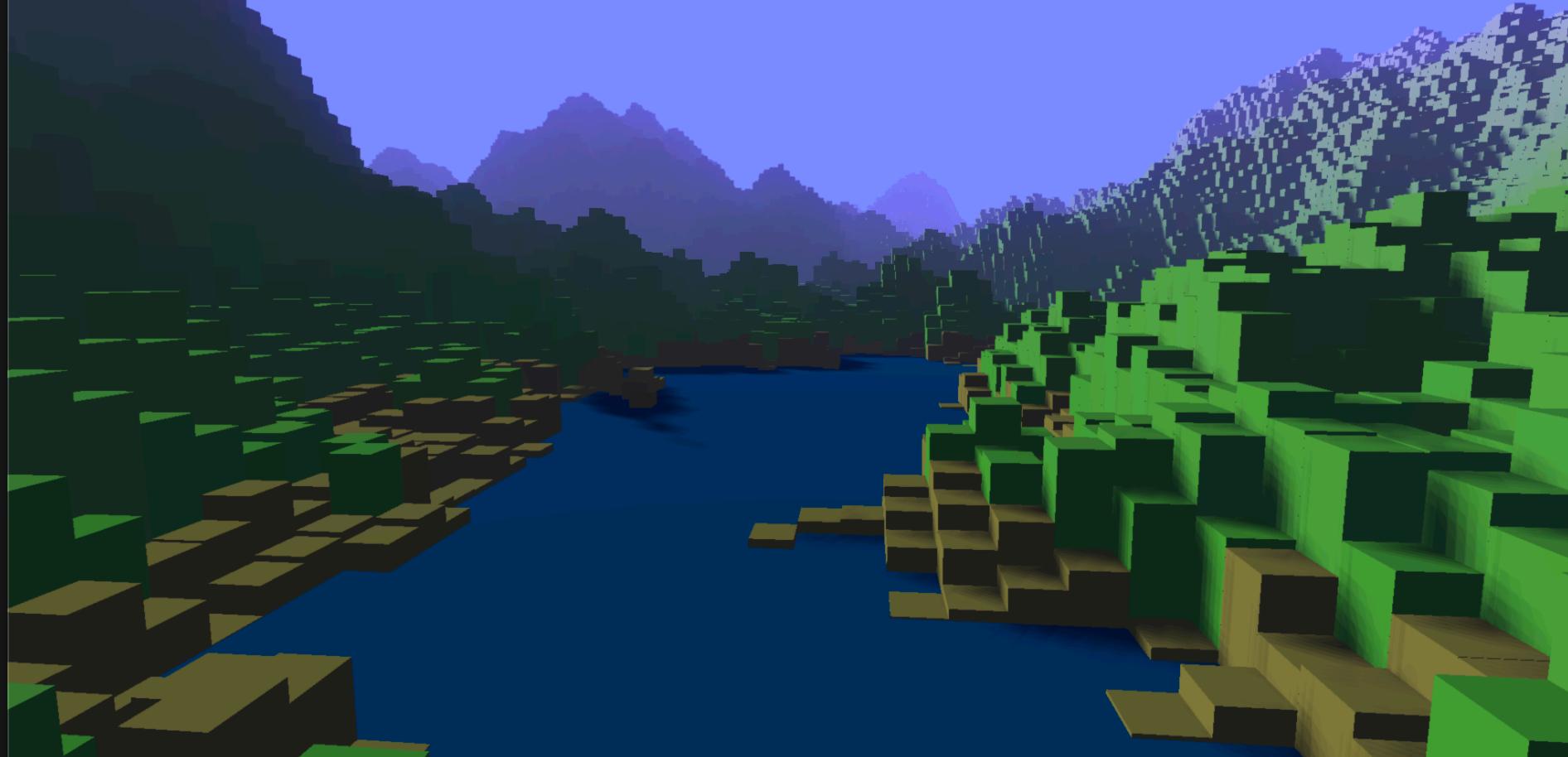
Tower Defense (Sokol)





Prefabs (Sokol)

Voxels (Sokol)





Introduction

Community

23 CONTRIBUTORS

117 PULL REQUESTS

105 FORKS

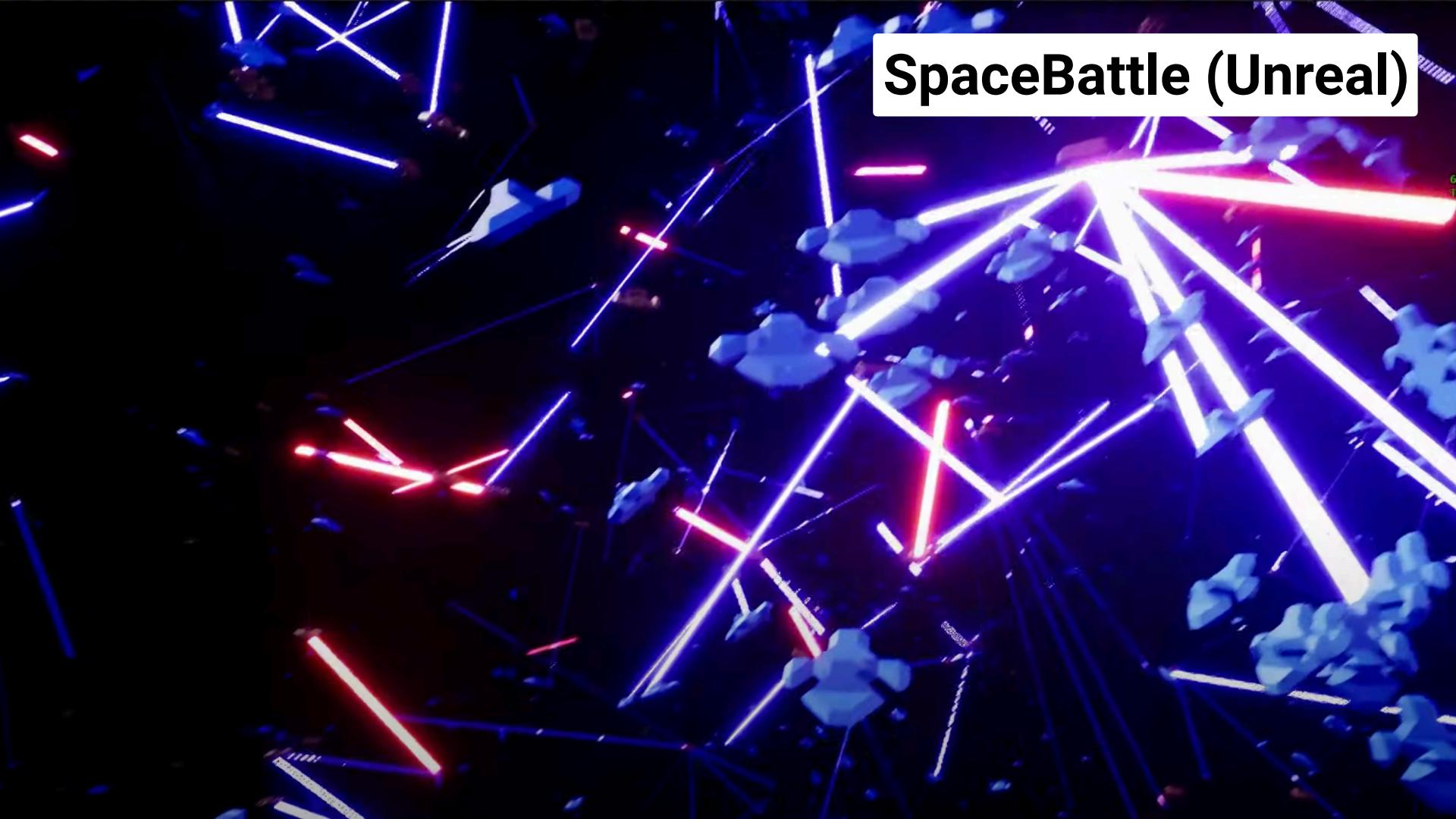
~400 DISCORD MEMBERS

COMMERCIALLY USED

2D RTS (SDL)



SpaceBattle (Unreal)



16.45 MB

500
450



1050

Test Shop

Gold Ingot

Sell

Gold Ingot

Buy

Gold Ingot

Buy

Gunpowder

Craft

Dynamite

Craft

Crate

Craft

2D RPG (Vulkan)

Coal	25.00 / 25.00
Sulphur	25.00 / 25.00
Coins	1050 / 50



2D RPG (Vulkan)

15.77 MB
1999

Universe Sim (Unreal)

Galaxy Shape

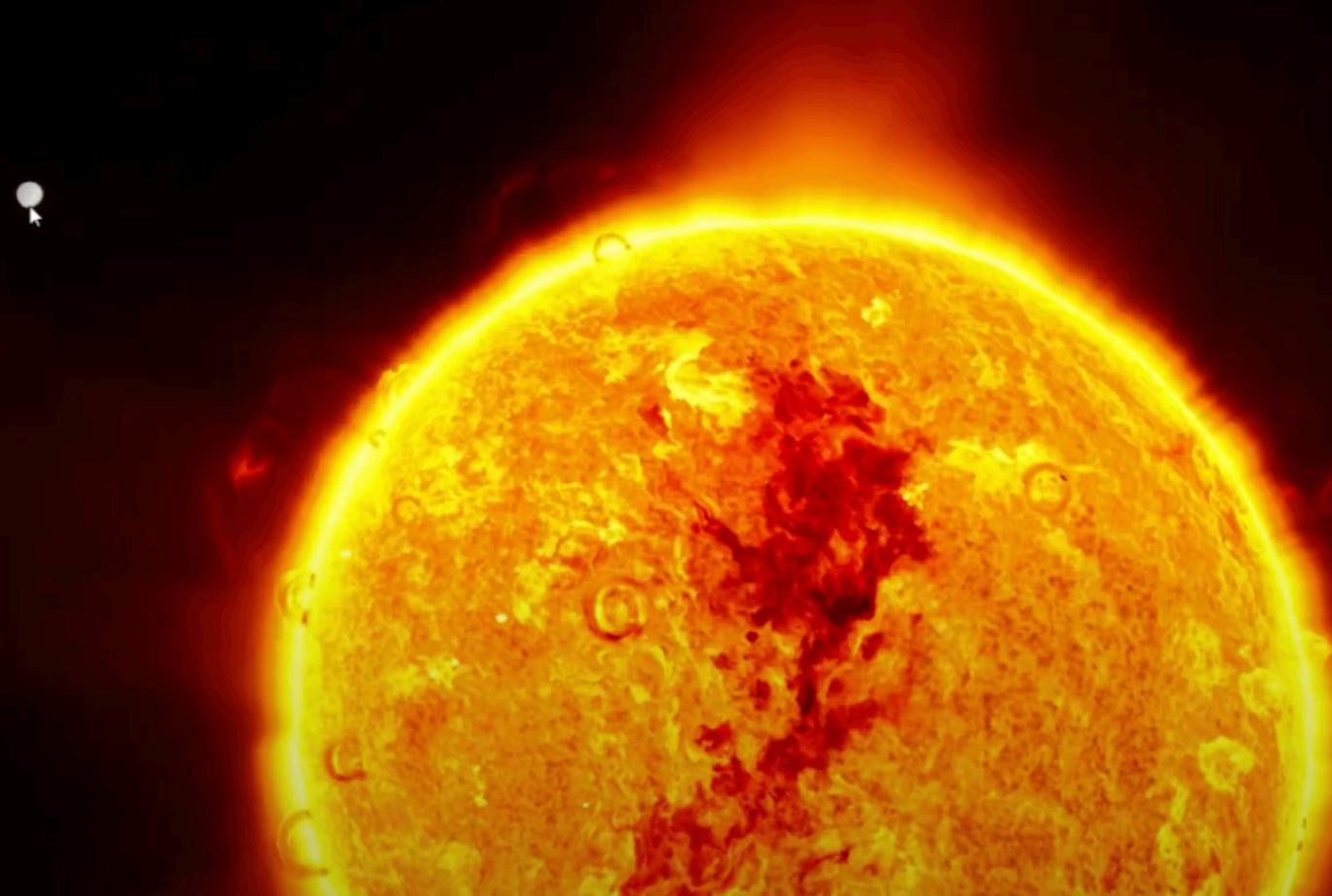
Galaxy Type	Spiral - Default
Galaxy Size	Medium
Stars	2000
Dust Density	1.0

Advanced Settings

Angle Offset	0,00040
Inner Eccentricity	0,850
Outer Eccentricity	0,950
Core Radius	0,300
Wave Amplitude	340
Wave Frequency	125



Universe Sim (Unreal)



A screenshot from the video game Universe Sim, developed with Unreal Engine. The scene is set in deep space, featuring a large, detailed planet on the left with prominent reddish-brown and white cloud formations. In the center-right, Earth is visible, showing its blue oceans and green continents. A white cursor arrow points towards the top of Earth's sphere. The background is a dark, textured void of space.

Universe Sim (Unreal)



bebylon

BATTLE ROYALE

Battle Royale (Unreal)

Writing Fast Code

Writing Fast Code

Storage

Writing Fast Code

CPU

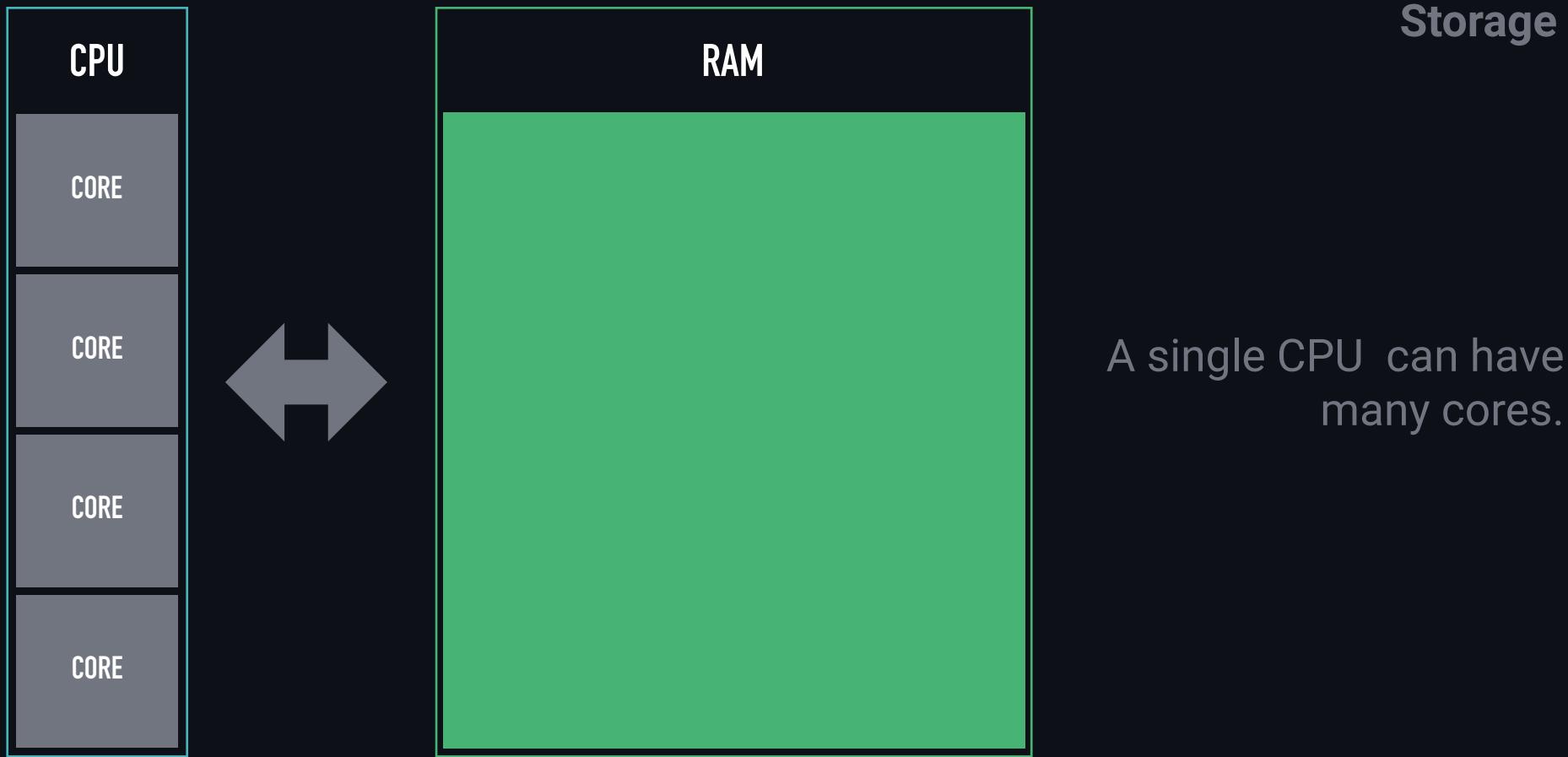
RAM

Storage



CPU fetches & stores
data from RAM when
running code.

Writing Fast Code



Writing Fast Code

Storage



Data fetched from RAM
is temporarily stored in a
local CPU cache

Writing Fast Code

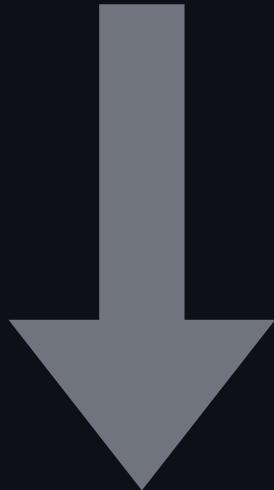


Each core has its own L1 and L2 cache, with a shared L3 cache

Writing Fast Code

Storage

RAM



CPU

There is a LOT more
RAM than there is space
in the CPU cache

Writing Fast Code

Storage

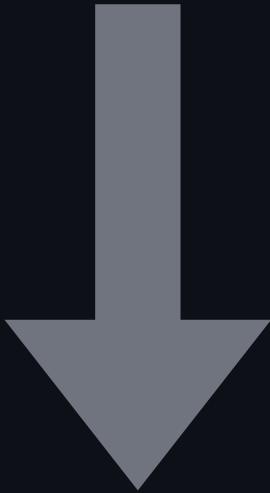
HARD DRIVE

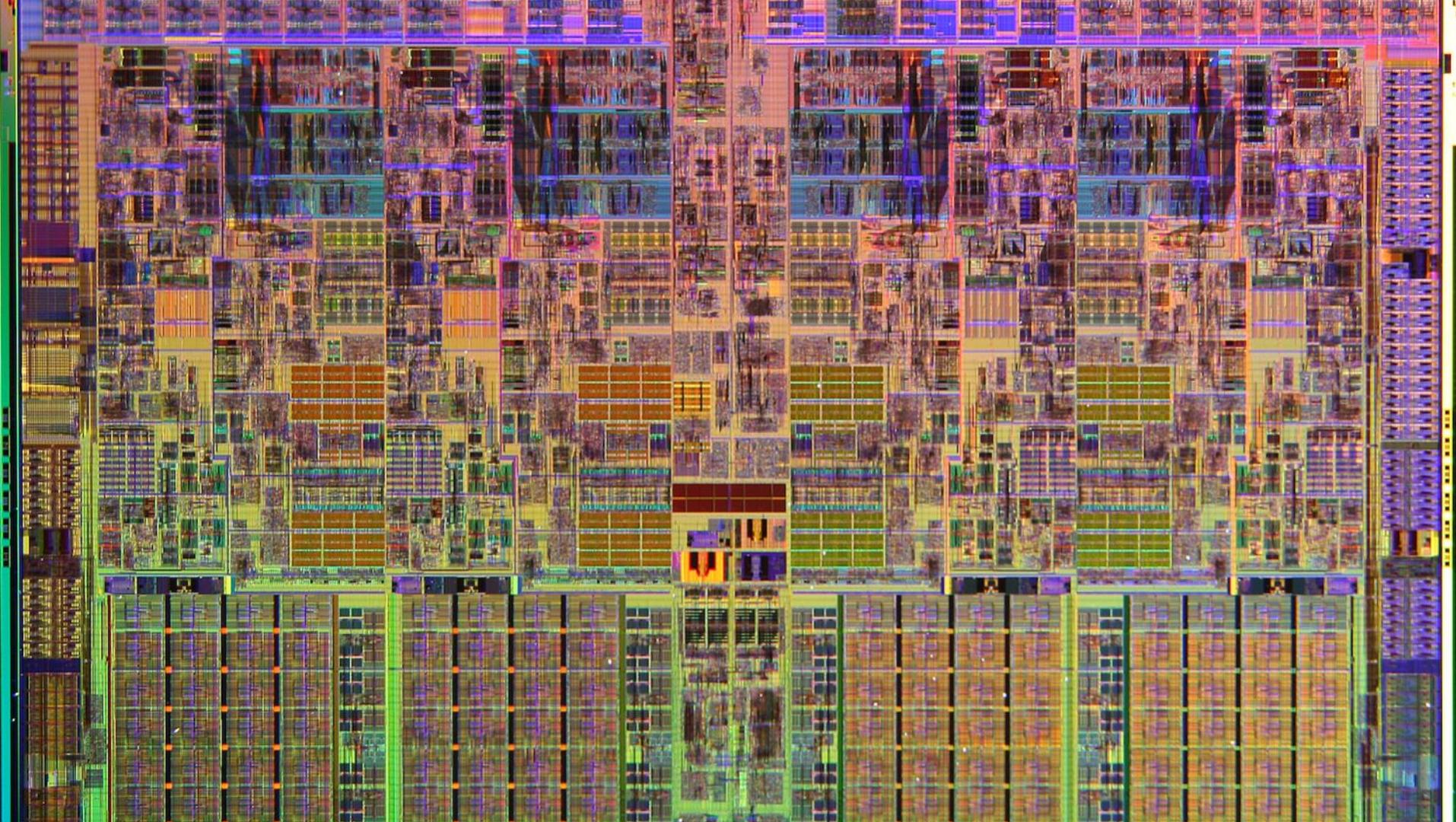
There is a LOT more
disk space than RAM

CPU



RAM





Writing Fast Code

Storage

L1

32Kb



L2

256Kb



L3

2MB

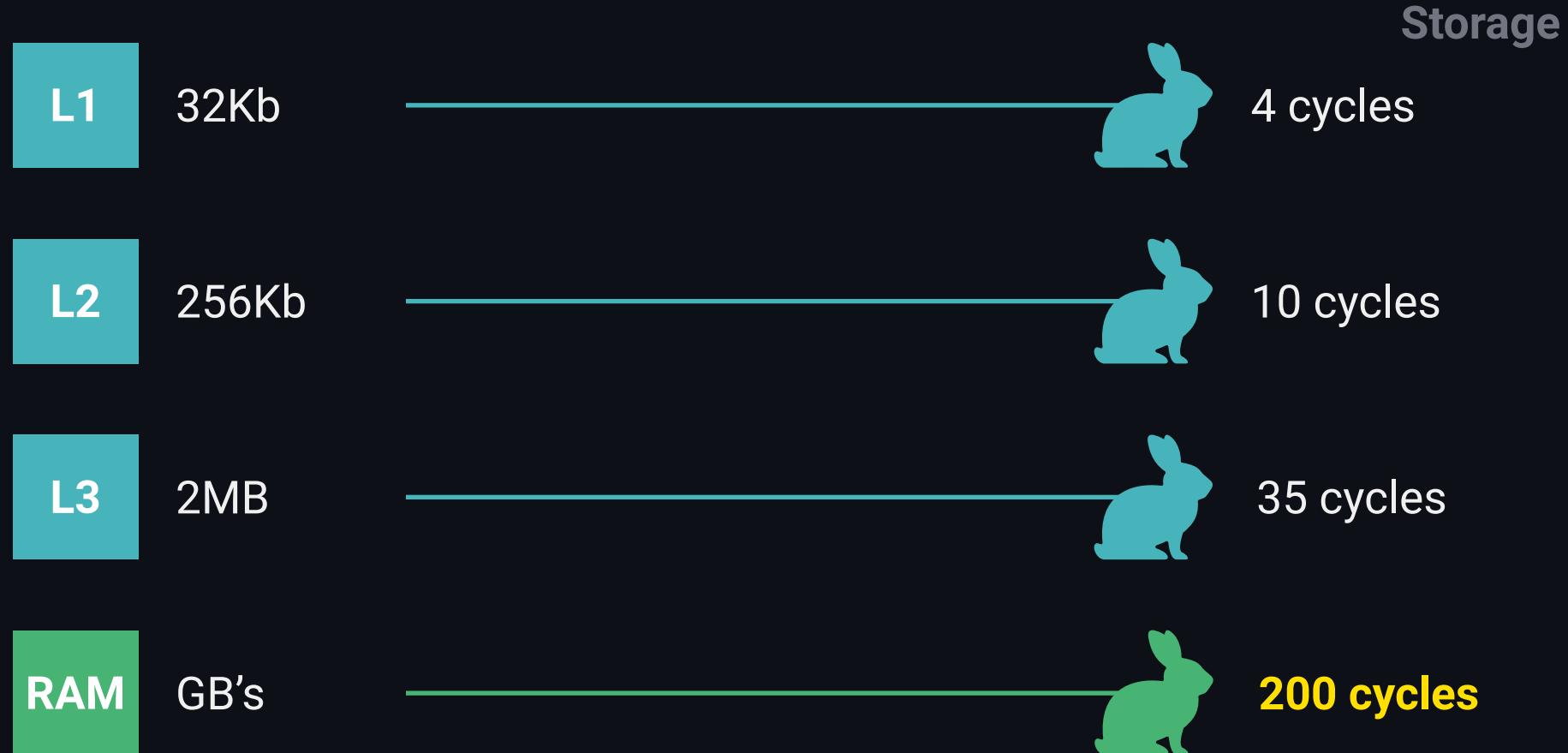


RAM

GB's

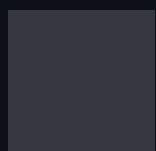


Writing Fast Code



Writing Fast Code

Storage



TB's



3400 cycles

Writing Fast Code

Storage

Why do we not just make RAM that is as fast
as a CPU cache? Three reasons:

COST

ENERGY USAGE

PHYSICS

Writing Fast Code

Cache Lines

Writing Fast Code

Cache Lines



64 Bytes

Writing Fast Code

Cache Lines

starting address



Get one byte

64 Bytes

Writing Fast Code

Cache Lines

Get some bytes from different pointers (aka OOP)



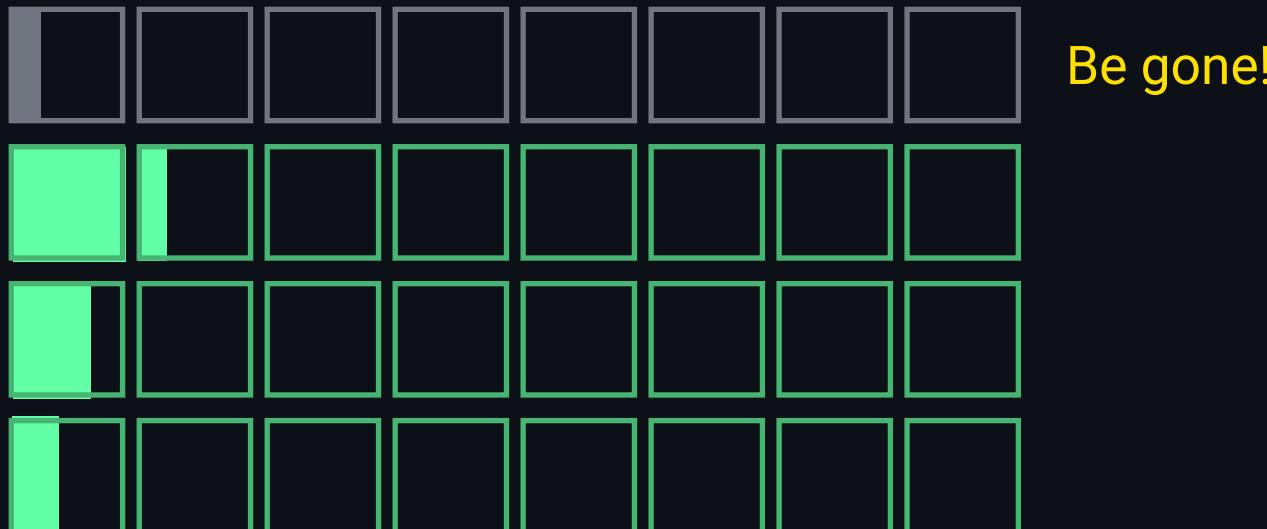
Sad cache :(

lots of wasted space

Writing Fast Code

Cache Lines

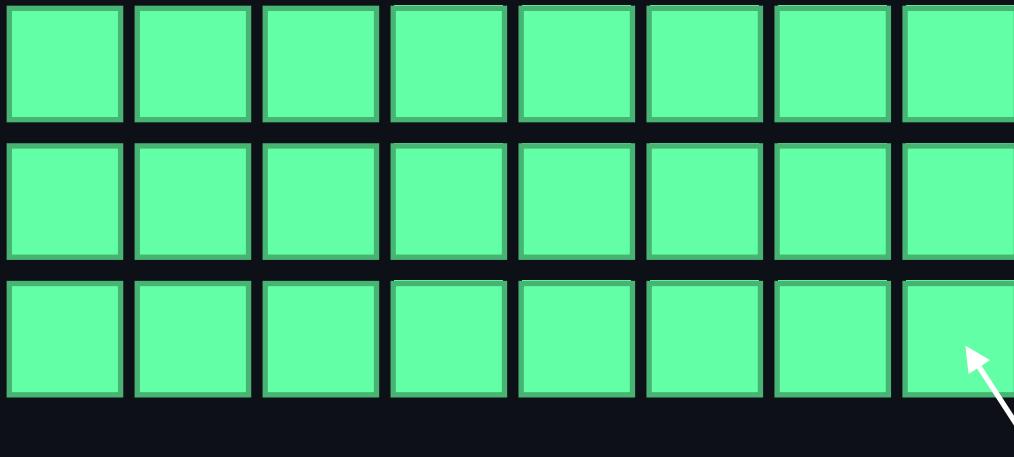
If the cache is full, old data is evicted



Writing Fast Code

Cache Lines

Fetching arrays loads more useful data, reducing RAM roundtrips



Happy cache :)

no wasted space

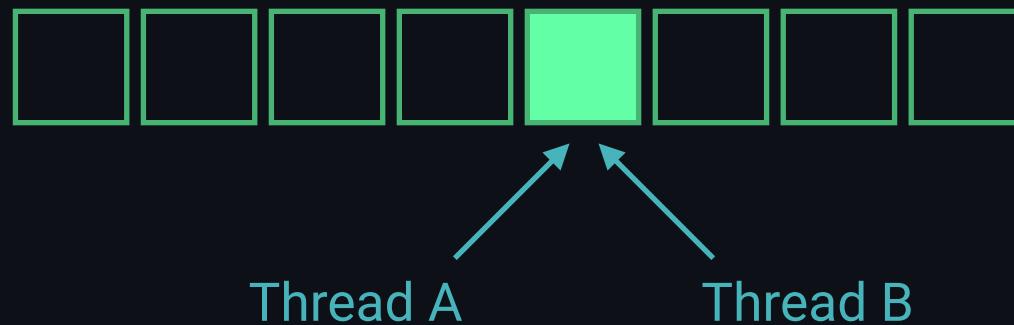
Writing Fast Code

Multithreading

Writing Fast Code

Multithreading

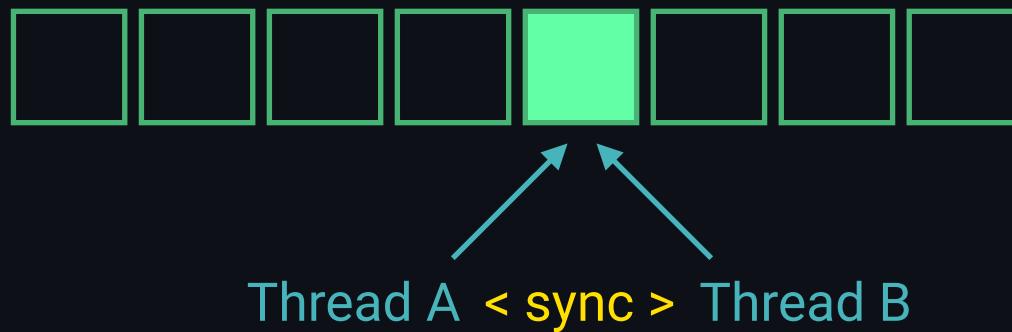
What about threads?



Writing Fast Code

Multithreading

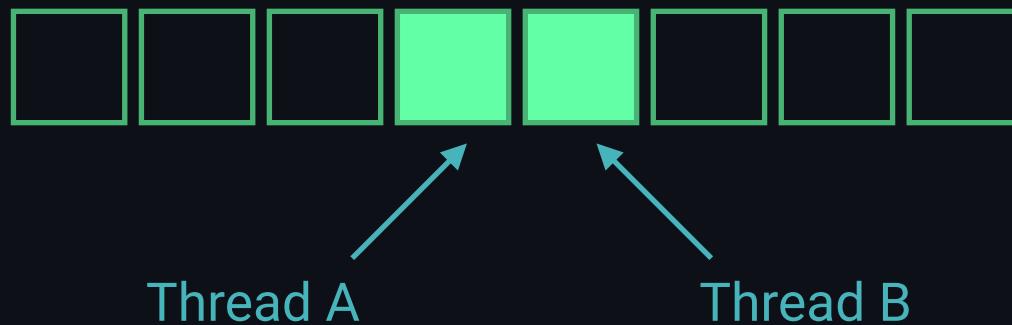
Threads are on different cores, so L1/L2 caches must be synced



Writing Fast Code

Multithreading

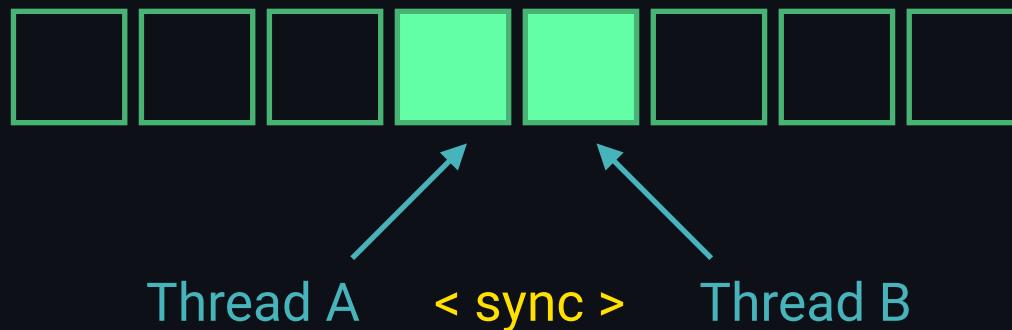
What if the data is on 2 separate addresses?



Writing Fast Code

Multithreading

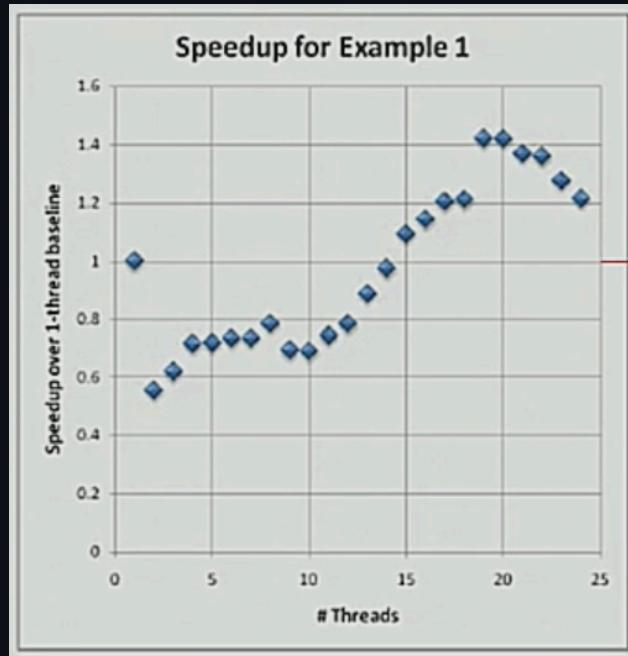
Sync still happens inside the same cache line. This is called **False Sharing**



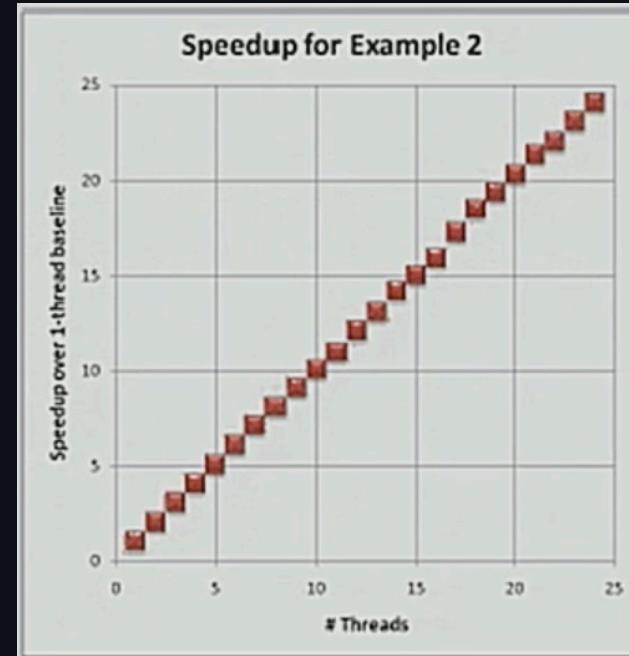
Writing Fast Code

Multithreading

False Sharing



No False Sharing



Writing Fast Code

Data Oriented Design

Writing Fast Code

Data Oriented Design

Data Oriented Design is about processing data in a way that takes advantage of capabilities of the underlying hardware, such as:

CACHING

VECTORIZATION

THREADING

Writing Fast Code

Data Oriented Design

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

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

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

Bad data oriented design™

Writing Fast Code

Data Oriented Design

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

Writing Fast Code

Data Oriented Design

```
// Thread 1
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

Writing Fast Code

Data Oriented Design

Wizard 1



Wizard 2



Wizard 3



lots of wasted space

Writing Fast Code

Data Oriented Design

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

Better data oriented design™

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

This approach is called
Arrays of Structs (AoS)

Writing Fast Code

Data Oriented Design

```
for (int i = 0; i < 1000; i++) {  
    wizards[i].x++;  
    wizards[i].y++;    ← No cache miss  
}  
}
```

Writing Fast Code

Data Oriented Design

```
// Thread 1
for (int i = 0; i < 1000; i++) {
    wizards[i].x++;
    wizards[i].y++;
}
```

← Still got false Sharing

```
// Thread 2
for (int i = 0; i < 1000; i++) {
    wizards[i].health++;
}
```

← Still got false Sharing

Writing Fast Code

Data Oriented Design

Wizard 1 Wizard 2 Wizard 3 ...



less wasted space

Writing Fast Code

Data Oriented Design

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

Good data oriented design™

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

This approach is called
Struct of Arrays (SoA)

Writing Fast Code

Data Oriented Design

```
for (int i = 0; i < 1000; i++) {  
    wizards.x[i]++;  
    wizards.y[i]++; ← No cache miss  
}
```

Writing Fast Code

Data Oriented Design

```
// Thread 1
for (int i = 0; i < 1000; i++) {
    wizards.x[i]++;
    wizards.y[i]++;
}
```

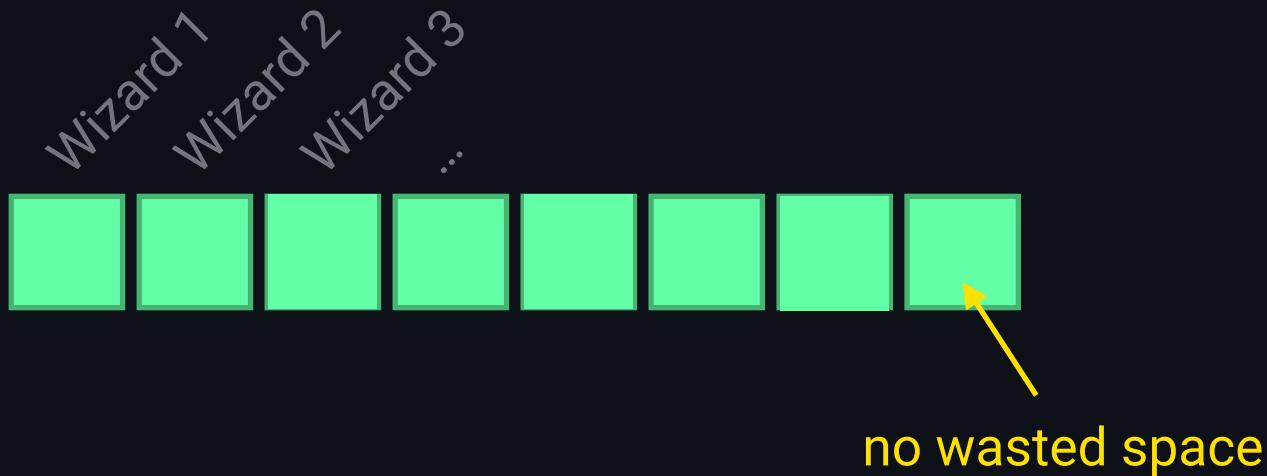
← No false Sharing

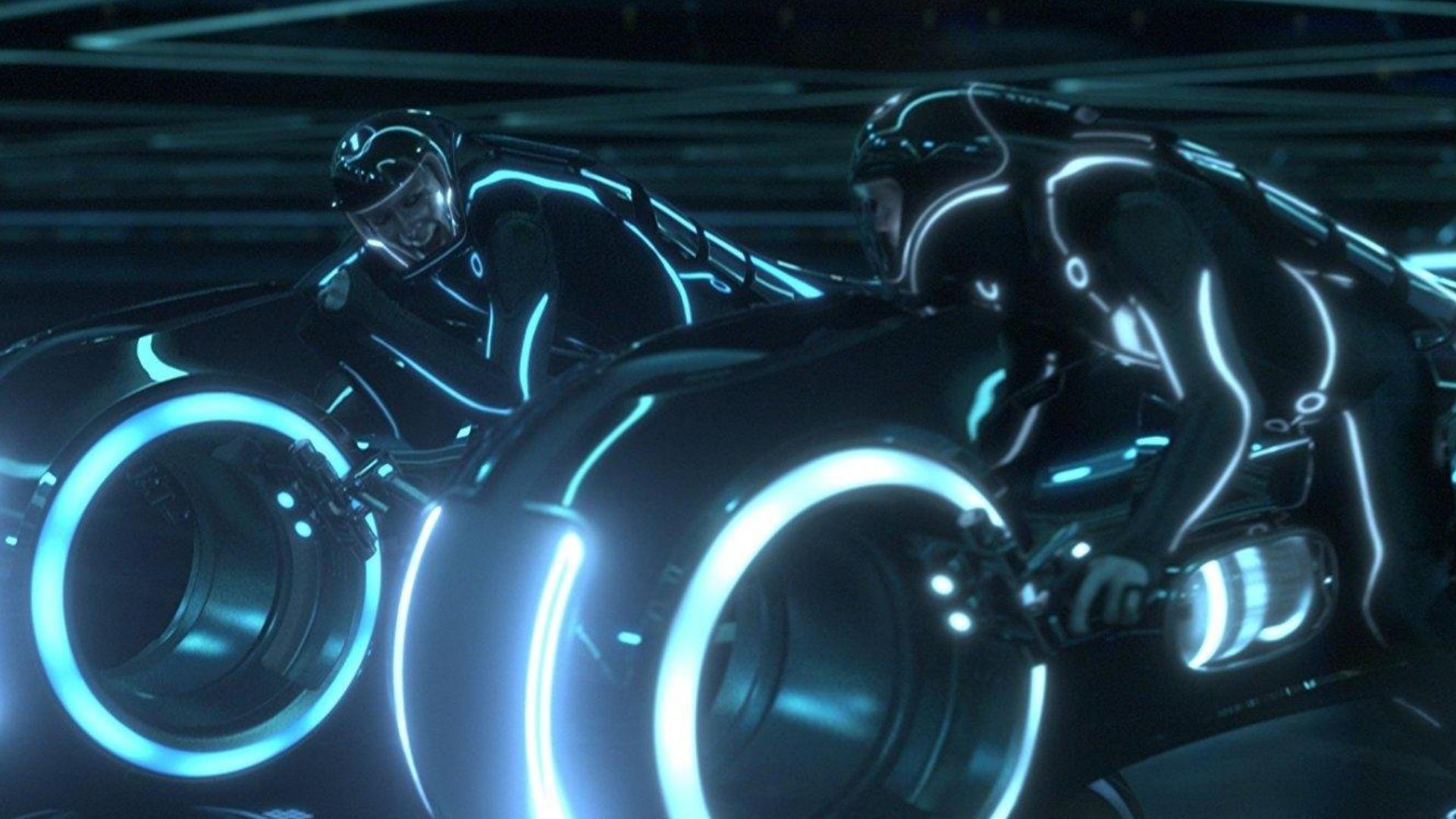
```
// Thread 2
for (int i = 0; i < 1000; i++) {
    wizards.health[i]++;
}
```

← No false Sharing

Writing Fast Code

Data Oriented Design





Writing Fast Code

ECS

Writing Fast Code

ECS

ECS is a tool that (amongst others) helps us write
data oriented code.

Writing Fast Code

ECS

ECS is a tool that (amongst others) helps us write
data oriented code.

ECS is NOT performance by default!

Writing Fast Code

ECS

ECS is a tool that (amongst others) helps us write data oriented code.

ECS is NOT performance by default!

To realize the benefits of ECS, code has to be structured in a way that is very different from OOP.

Flecs Basics

Flecs Basics

Entities, Types, Components

Flecs Basics

Entities, Types, Components

Flecs is an Entity Component System

Flecs Basics

Entities, Types, Components

Flecs is an Entity Component System

Flecs Basics

Entities, Types, Components

Flecs is an Entity Relationship System

(but behaves like a regular ECS most of the time)

Flecs Basics

Entities, Types, Components

Flecs is an Entity **Relationship** System

(but behaves like a regular ECS most of the time)

Let's take a look at how that works

Flecs Basics

Entities, Types, Components

E

An entity in Flecs is a unique integer

Flecs Basics

Entities, Types, Components

E

→ Each entity is associated with a type

T

Flecs Basics

Entities, Types, Components



A type is a set (stored as an array) of identifiers

Flecs Basics

Entities, Types, Components



These identifiers indicate which components an entity has

Flecs Basics

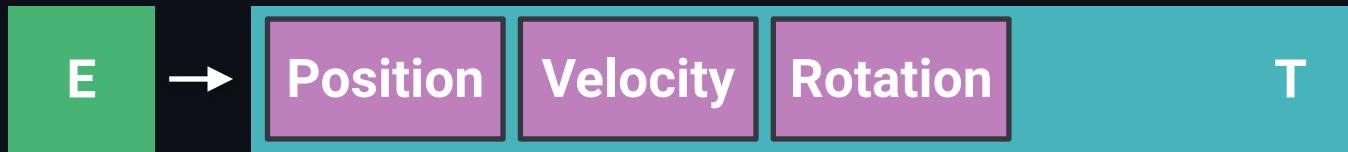
Entities, Types, Components



Components are associated with datatypes

Flecs Basics

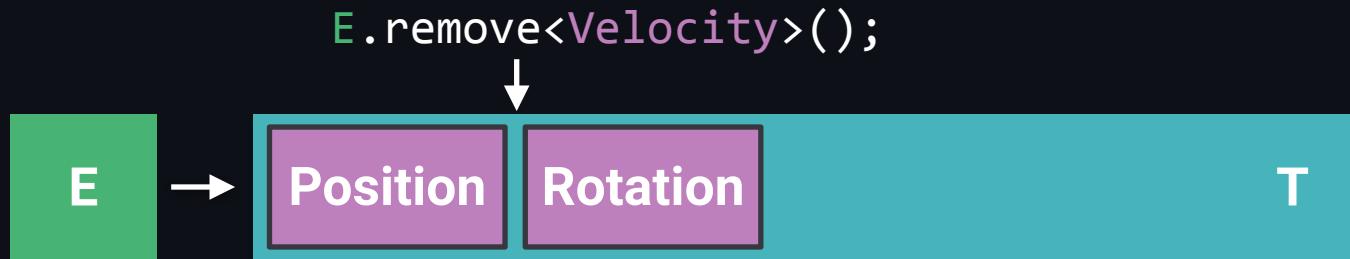
Entities, Types, Components



Flecs has “add” and “remove” operations that add & remove identifiers to a type

Flecs Basics

Entities, Types, Components



Flecs has “add” and “remove” operations that add & remove identifiers to a type

Flecs Basics

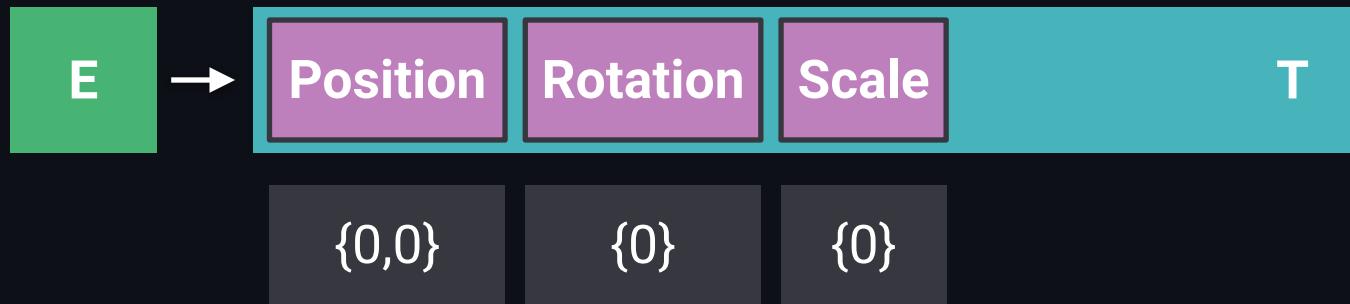
Entities, Types, Components



Flecs has “add” and “remove” operations that add & remove identifiers to a type

Flecs Basics

Entities, Types, Components

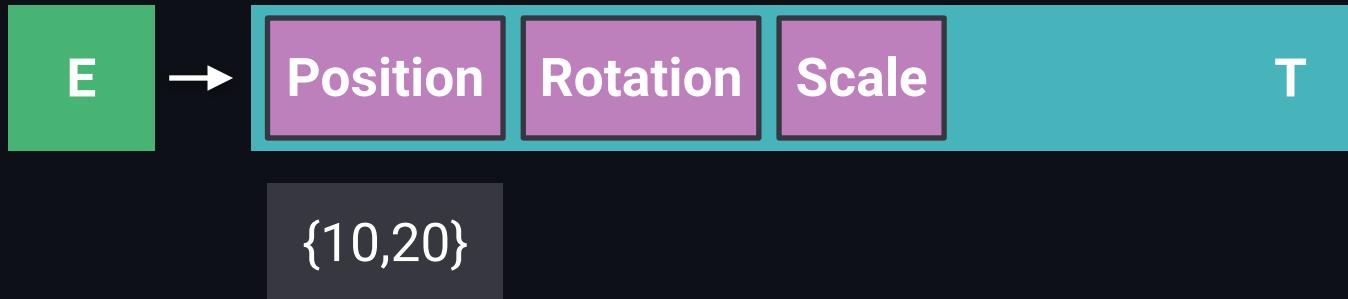


Each component on an entity is associated with a value, which is an instance of the datatype

Flecs Basics

Entities, Types, Components

```
E.set<Position>({10, 20});
```

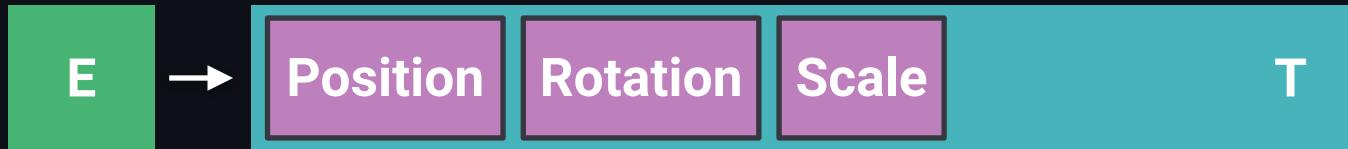


The “set” operation changes a value

Flecs Basics

Entities, Types, Components

```
const Position *p = E.get<Position>();
```



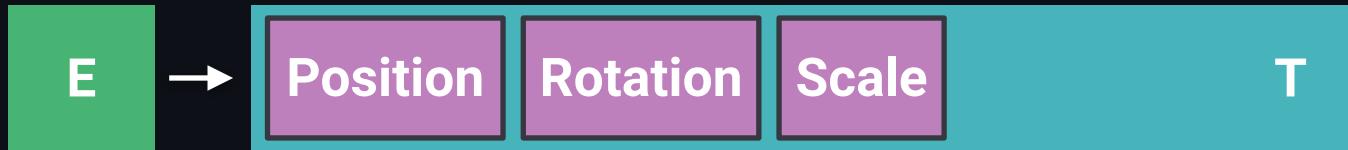
{10,20}

The “get” operation returns a value

Flecs Basics

Entities, Types, Components

```
// Outputs "Position,Rotation,Scale"  
std::cout << E.type().str() << std::endl;
```



Types can be inspected and converted to a string.

Flecs Basics

Entities, Types, Components

E

So far so good?

Flecs Basics

Entities, Types, Components

E

Let's get to the fun part

Flecs Basics

Entities, Types, Components

E

An entity in Flecs is a unique integer (recap)

Flecs Basics

Entities, Types, Components

E

An entity in Flecs is a unique integer

C

An *component* in Flecs is also a unique integer

Flecs Basics

Entities, Types, Components

C = E

Internally, components are stored as entities

Flecs Basics

Entities, Types, Components

C = E

Internally, components are stored as entities

Why is this important?

Flecs Basics

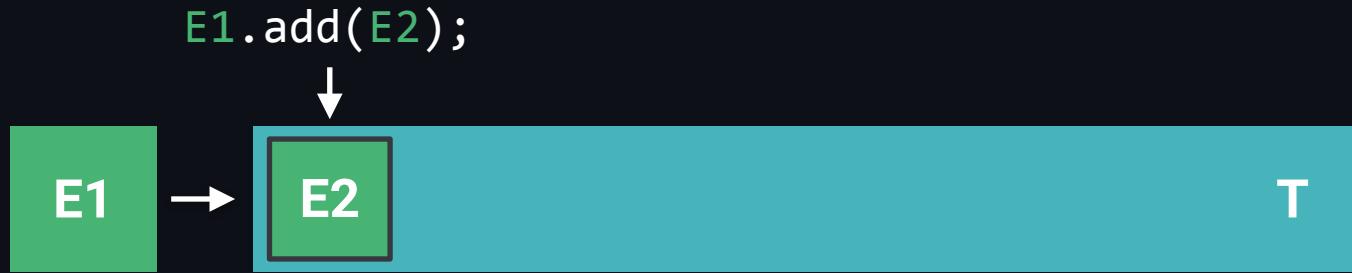
Entities, Types, Components



It means that a *Type* is actually a list of entities

Flecs Basics

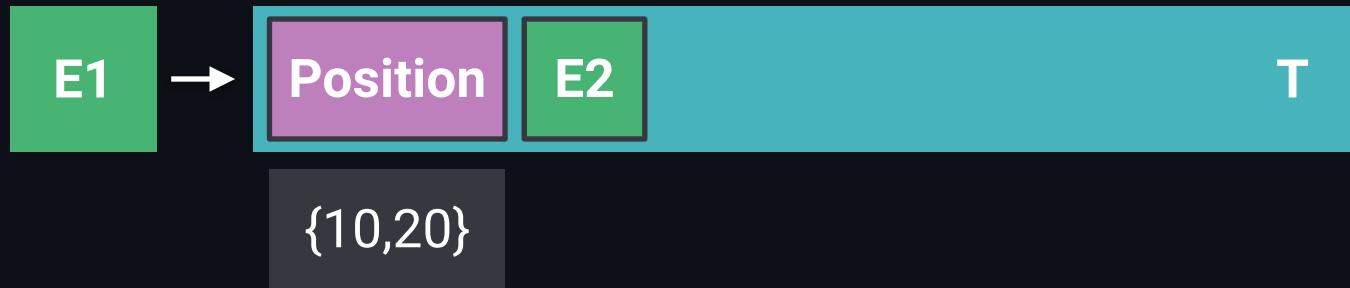
Entities, Types, Components



This means that you can add Entities to Entities

Flecs Basics

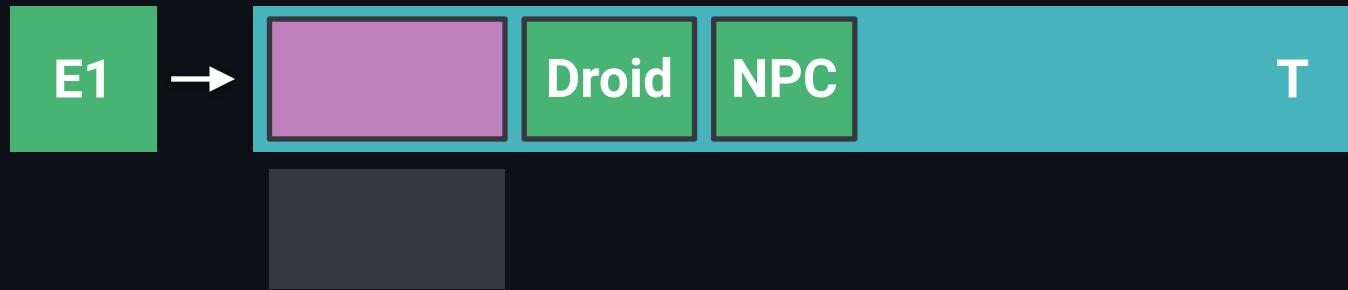
Entities, Types, Components



Not all Entities are Components. A component is associated with a value, a “regular” entity is not.

Flecs Basics

Entities, Types, Components

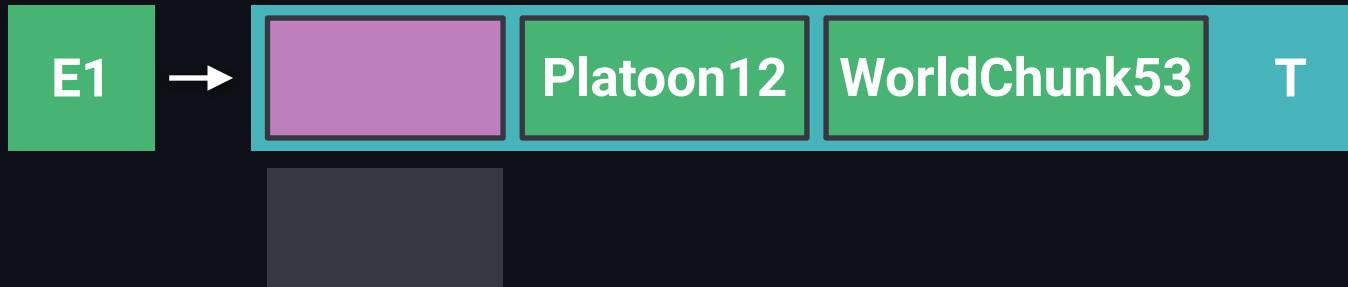


Adding entities is useful for tagging. Tags are useful for subdividing entities, which enables more granular queries.

Flecs Basics

Entities, Types, Components

```
auto Platoon12 = world.entity();  
E1.add(Platoon12);
```



Because we use regular entities for tagging, we can create tags at runtime. This is useful when we must filter on runtime generated game “stuff”.

Flecs Basics

Entities, Types, Components

```
struct EmptyType {};  
E1.add<EmptyType>();
```



One note: Flecs recognizes empty types, and does not treat them as components but as tags.

Flecs Basics

Entities, Types, Components

Show me code!

Flecs Basics

Entities, Types, Components

So far so good?

Flecs Basics

Relationships

Flecs Basics

Relationships

Relationships are prevalent in games:

A Unit belongs to a Platoon

A Platoon belongs to a Faction

Player1 is an ally of Player2

Flecs Basics

Relationships

Relationships in Flecs are a 1st class concept.

This means you can create relationships using
the Flecs API.

It also means that you can use ECS queries for
relationships, which is much faster than
manually iterating & testing a list of entities!

Flecs Basics

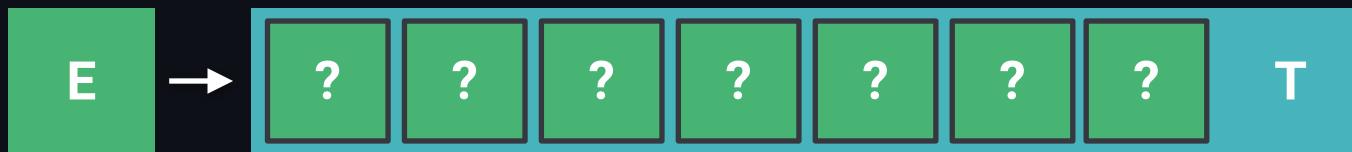
Relationships



A Type is a list of entity identifiers (recap)

Flecs Basics

Relationships



A Type is a list of entity identifiers (recap)

Ok this is not 100% correct

Flecs Basics

Relationships



A Type is a list of entity identifiers (recap)

Each element in a type is a 64 bit integer

Flecs Basics

Relationships



A Type is a list of entity identifiers (recap)

Components and tags only take up 32 bits

Flecs Basics

Relationships



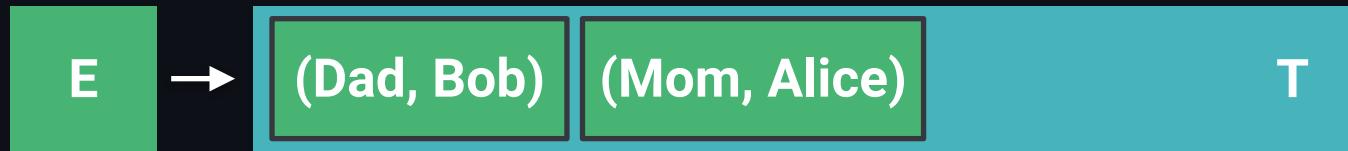
This lets us store 2 values per element. This second value can be used to store the relationship. This is called a “trait” in Flecs.

Flecs Basics

Relationships

```
auto Dad = world.entity();
auto Mom = world.entity();
auto Bob = world.entity();
auto Alice = world.entity();
```

```
E.add_trait(Dad, Bob)
    .add_trait(Mom, Alice);
```



Here's a small example of how this can be used to construct a small family tree

Flecs Basics

Relationships

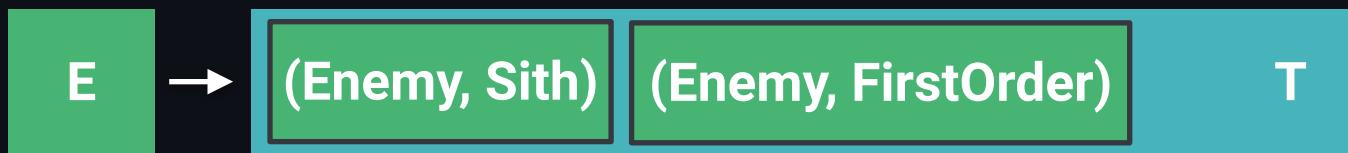
```
E.has_trait(Dad, Bob);
```



To test if an entity has a relationship, use the “has_trait” operation.

Flecs Basics

Relationships



Traits can occur multiple times in a Type

Flecs Basics

Relationships

Show me code!

Break! (10 min)

Tip: make sure to **join the Flecs Discord**. It contains a ton of information and users that have built all sorts of games & are excited about ECS :)

Flecs Basics

Queries

Flecs Basics

Queries

Queries

Queries let an application find all entities that match
a certain set of components

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();
```

When a query is created, you specify the components it subscribes for.

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();  
q.iter()
```

```
[](flecs::iter it, Position *p, const Velocity *v) {  
})
```

An application can then invoke the “iter” function as many times as needed, to get the entities & components that match.

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();  
q.iter()  
  
[](flecs::iter it, Position *p, const Velocity *v) {  
}  
})
```

ProTip: queries are expensive to create, cheap to evaluate. Cache your queries, don't re-create them each time you need one!

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();  
q.iter()  
  
// Outer loop, invoked for each matched Type  
[](flecs::iter it, Position *p, const Velocity *v) {  
  
    // Inner loop, invoked for each entity in the Type  
    for (auto i : it) {  
        p[i].x += v[i].x; // Contiguous arrays: Happy cache!  
        p[i].y += v[i].y;  
    }  
})
```

Flecs groups entities by **Type**. Queries return entities in batches: one batch per matched **Type**.

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>("!Speed");
q.iter(
    // Outer loop, invoked for each matched Type
    [](flecs::iter it, Position *p, const Velocity *v) {
        // ...
    })
)
```

Components can either be specified as types, or in a query string. The query string has support for more complex features, like operators.

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();  
q.each(  
    [](flecs::entity e, Position& p, const Velocity& v) {  
        p.x += v.x;  
        p.y += v.y;  
    })
```

The “each” function lets you iterate a query with less code: it automatically combines the outer and the inner loop.

Flecs Basics

Queries

```
q = world.query<Position, const Velocity>();  
q.each(  
    [](flecs::entity e, Position& p, const Velocity& v) {  
        p.x += v.x;  
        p.y += v.y;  
    })
```

ProTip: Flecs queries are **super fast** to evaluate, regardless of the number of components. Use this to your advantage by creating smaller systems that match more specific sets of entities (using tags).

Flecs Basics

Queries

Show me code!

Flecs Basics

Systems

Flecs Basics

Systems

Systems

A system is a combination of a query and a function.

Systems can be automatically ran by Flecs.

Systems are an optional feature. Applications can be built entirely with just queries. This is especially common when integrating Flecs with existing engines.

Flecs Basics

Systems

```
world.system<Position, const Velocity>("Move")
```

Just like a query, you specify a list of components for a system. You can optionally provide a system name, which helps debugging

Flecs Basics

Systems

```
world.system<Position, const Velocity>("Move")  
.kind(flecs::OnUpdate)
```

In addition, you can specify when the system should be ran. By default this is the OnUpdate “phase”.

Flecs Basics

Systems

```
world.system<Position, const Velocity>("Move")  
.kind(flecs::OnUpdate)
```

```
// Executed after Move  
world.system<const Position, Target>("FindTarget")  
.kind(flecs::OnUpdate)
```

Systems assigned to the same phase are guaranteed to be executed in order of declaration.

Flecs Basics

Systems

```
world.system<Position, const Velocity>("Move")
    .kind(flecs::OnUpdate)
    .each([](flecs::entity e, Position& p, const Velocity& v) {
        p->x += v->x;
        p->y += v->y;
    });
}
```

The implementation for a system is set with the
“each” or “iter” functions.

Flecs Basics

Systems

```
world.system<Position, const Velocity>("Move")
    .kind(flecs::OnUpdate)
    .each([](flecs::entity e, Position& p, const Velocity& v) {
        p->x += v->x;
        p->y += v->y;
    });
world.progress();
```

To run all systems registered with a world, run the `world.progress()` function.

Flecs Basics

Modules

Flecs Basics

Systems

Modules

Modules are an optional feature that let you organize and modularize your components and systems.

With proper usage of modules, you can build features in a way that can be easily reused across applications.

Flecs Basics

Modules

```
class MyModule {
    MyModule(flecs::world& world) {
        world.module<MyModule>();

        world.component<Position>();
        world.component<Velocity>();

        world.system<Position, const Velocity>("Move")
            .iter( ... );
    }
};
```

A module is defined as a class (in C++) that creates all systems and components it contains in the constructor.

Flecs Basics

Modules

```
world.import<MyModule>();
```

To use a module, an application can simply import it. This will register all systems and components with the world.

Flecs Basics

Modules

```
class MyModule {  
    MyModule(flecs::world& world) {  
        world.module<MyModule>();  
  
        world.import<YetAnotherModule>();  
  
        // ...  
    }  
};
```

Modules can (and often do) depend on other modules.

Flecs Basics

Modules

```
namespace systems {
    class Physics {
        Physics(flecs::world& world) {
            world.module<Physics>();
            world.import<components::Physics>();
        }
    };
}
```

ProTip: split modules up in `components.*` and `systems.*` modules. This lets you easily swap different implementations of different features, as the components stay the same!

Flecs Basics

Modules

```
world.import<flecs::components::transform>();  
world.import<flecs::components::physics>();  
world.import<flecs::systems::transform>();  
world.import<flecs::systems::sokol>();  
world.import<flecs::json>();  
world.import<flecs::dash>();
```

Check out github.com/flecs-hub! It contains lots of example modules that show how to create various game systems.

Flecs Basics

Modules

Show me code!



Flecs Basics

That's all for the Flecs basics.

Congratulations, you now know the fundamentals of Flecs! 

The next part of the presentation is about
quality of life features, such as

Hierarchies, Prefabs and State Machines.

Short Break! (5 min)

Tip: Check the [flecs-hub organization](#) on GitHub! It has lots of (working) example code that shows how to build all sorts of systems.

Flecs Advanced

Flecs Advanced

Hierarchies

Flecs Advanced

Hierarchies

Hierarchies

Probably the #1 question from ECS users is, “*howto hierarchies*”.

DIY ECS hierarchies are a minefield of a design space, with more things you can do wrong than right :(

Flecs has a builtin, efficient hierarchy implementation that is an especially good fit for scene graphs :)

You can still DIY ofc, at your own risk (or pleasure)

Flecs Advanced

Hierarchies

```
auto SpaceShip = world.entity();
```

```
auto Railgun = world.entity()  
.add_childof(SpaceShip);
```

```
auto TurretLeft = world.entity()  
.add_childof(SpaceShip);
```

```
auto TurretRight = world.entity()  
.add_childof(SpaceShip);
```

Creating hierarchies is simple with the builtin “add_childof” function.

Flecs Advanced Hierarchies

```
Railgun.remove_childof(SpaceShip);
```

Hierarchies are dynamic. A parent can be removed with the remove_childof function.

Flecs Advanced Hierarchies

```
SpaceShip.destruct();
```

Deleting a parent deletes all of its children.

Flecs Advanced

```
q = world.query<Position>("PARENT:Position");
q.iter(
    [](flecs::iter it, Position *p) {
        // Request parent component from iterator. "2" is the 2nd column
        // of the query signature.
        auto parent_p = it.column<Position>(2);

        for (auto i : it) {
            p[i].x += parent_p->x; // Notice parent_p is a pointer
        }
    });
});
```

Hierarchies

Queries & systems are hierarchy aware. Applications can explicitly request components from parent entities.

Flecs Advanced

```
q = world.query<Position>("CASCADE:Position");  
q.iter(
```

Hierarchies

```
[](flecs::iter it, Position *p) {  
    auto parent_p = it.column<Position>(2);  
    for (auto i : it) {  
        if (parent_p.is_set()) {  
            p[i].x += parent_p->x;  
            // If m_parent is not set, this is a root entity  
        } else {  
            p[i].x += parent_p->x;  
        }  
    }  
});
```

Scene graphs need to transform top-down. The **CASCADE** modifier enforces BFS order, without modifying the storage.

Flecs Advanced

```
q = world.query<Position>("CASCADE:Position");  
q.iter(
```

Hierarchies

```
[](flecs::iter it, Position *p) {  
    auto parent_p = it.column<Position>(2);  
    for (auto i : it) {  
        if (parent_p.is_set()) {  
            p[i].x += parent_p->x;  
            // If m_parent is not set, this is a root entity  
        } else {  
            p[i].x += parent_p->x;  
        }  
    }  
});
```

Note: Entities are not sorted, the query's private table cache is sorted. Once sorting is only applied when new tables are created.

Flecs Advanced

```
q = world.query<Position>("CASCADE:Position");
q.iter(
    [](flecs::iter it, Position *p) {
        auto parent_p = it.column<Position>(2);
        if (parent_p.is_set()) {
            for (auto i : it) {
                p[i].x += parent_p->x; // No branching in core loop!
            }
        } else {
            for (auto i : it) {
                p[i].x += parent_p->x;
            }
        }
    });
});
```

Hierarchies

ProTip: put the parent check around the loops. It's a bit more typing, but greatly reduces branching in your code!

Flecs Advanced

Hierarchies

```
auto Rocinante = world.entity("Rocinante");
```

```
auto Railgun = world.entity("Railgun")
    .add_childof(Rocinante);
```

```
auto r1 = world.lookup("Rocinante::Railgun");
```

```
auto r2 = Rocinante.lookup("Railgun");
```

Entities can be given names. Hierarchies allow you to lookup entities with scoped identifiers.

Flecs Advanced Hierarchies

Show me code!

Flecs Advanced

Prefabs

Flecs Advanced

Prefabs

Prefabs

When creating entities, it comes in handy if you can create them from a template. Flecs has a builtin prefab feature that makes this easy (and efficient!) to do.

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

Flecs Advanced Prefabs

Prefabs are like regular entities, but are created with the “prefab” method. This ensures that prefabs are not matched with systems.

Flecs Advanced

Prefsabs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

To instantiate a prefab, use the “add_instanceof” function.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

```
// This pointer points to the component of Frigate!
const MaxSpeed *ms = Rocinante.get<MaxSpeed>();
```

add_instanceof causes components to be shared between the “base” and “instance”. Shared components are stored once in memory.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

```
// This pointer points to the component of Frigate!
const MaxSpeed *ms = Rocinante.get<MaxSpeed>();
```

ProTip: if you have lots of redundant data across entities, use instancing instead. This reduces memory footprint & improves cache perf.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate)
.add<MaxSpeed>(); // Overrides MaxSpeed, copy value from Frigate
```

When instantiated, an “instance” shares components with the “base”. Components can be overridden, which adds them to the instance.

Flecs Advanced Prefabs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate)
.add<MaxSpeed>(); // Overrides MaxSpeed, copy value from Frigate
```

```
Rocinante.remove<MaxSpeed>(); // Re-exposes MaxSpeed from the base
```

When an override is removed, the base component will be re-exposed.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2})
.add_owned<MaxSpeed>();
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

```
// Rocinante now has an override of MaxSpeed with value {100}
```

Prefab components can be auto-overridden, by using the add_owned() function. This is a simple way to ensure instances have a component, while also ensuring it is a private ("owned") copy.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2})
.add_owned<MaxSpeed>()
.add_owned<Attack>()
.add_owned<Defense>();
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

ProTip: using prefabs in combination with automatic overriding is an easy and fast way to initialize component data for new entities!

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
.add_owned<MaxSpeed>();
```

```
auto FastFrigate = world.prefab("FastFrigate")
.add_instanceof(Frigate)
.set<MaxSpeed>({200});
```

Prefs can instantiate each other. This enables specialization of prefs, and makes it easier to write DRY templates.

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2})
.add_owned<MaxSpeed>();
```

```
auto FastFrigate = world.prefab("FastFrigate")
.add_instanceof(Frigate)
.add_instanceof(SportsCarMaterial)
.set<MaxSpeed>({200});
```

Multiple prefabs can be added with add_instanceof. If components overlap, the first found one is returned (so don't do that)

Flecs Advanced

Prefs

```
auto Frigate = world.prefab("Frigate")
.set<MaxSpeed>({100})
.set<Attack>({5})
.set<Defense>({2});
```

```
auto RailGun = world.prefab("RailGun")
.add_childof(Frigate);
```

```
auto Rocinante = world.entity()
.add_instanceof(Frigate);
```

```
auto r = Rocinante.lookup("RailGun");
```

Children of a prefab are automatically copied to the instance.

Flecs Advanced

```
q = world.query<Velocity>("SHARED:MaxSpeed");
q.iter(
    [](flecs::iter it, Velocity *v) {
        auto max_speed = it.column<MaxSpeed>(2);

        for (auto i : it) {
            // Notice max_speed is a pointer
            if (v[i].x > max_speed->value) v.x = max_speed->value;
        }
    });
});
```

Prefs

By default shared components are not matched by systems. To match a shared component, use the SHARED modifier.

Flecs Advanced

```
q = world.query<Velocity>("ANY:MaxSpeed");
q.iter(
    [](flecs::iter it, Velocity *v) {
        auto max_speed = it.column<MaxSpeed>(2);
        for (auto i : it) {
            if (!max_speed.is_shared()) {
                if (v[i].x > max_speed[i].value) v.x = max_speed[i].value;
            } else {
                // Notice max_speed is a pointer
                if (v[i].x > max_speed->value) v.x = max_speed->value;
            }
        }
    });

```

With the ANY modifier a system matches both SHARED and OWNED components.

Prefs

Flecs Advanced

Prefabs

Show me code!

Flecs Advanced

State Machines

State Machines

State machines are common, but can be difficult to model in ECS.

Transitioning to a new state (tag) requires knowing the previous state (tag) to remove it. Plain tags don't make this easy.

Flecs has a builtin feature that lets you model mutually exclusive tags, which come in handy when storing state machines.

Flecs Advanced

State Machines

```
auto Walking = world.entity("Walking");
auto Running = ecs.entity("Running");
auto Movement = ecs.type("Movement",
    "Walking, Running");

auto e = world.entity()
    .add_switch(Movement)
    .add_case(Walking);

e.add_case(Running); // Removes Walking
```

A “switch” models a set of mutually exclusive tags (“cases”). Adding one case will remove the previous case that belongs to that switch.

Flecs Advanced

State Machines

```
q = world.query<>("CASE|Walking");
q.iter([](flecs::iter it) {
    for (auto i : it) {
        cout << it.entities(i).name() << " is Walking!" << std::endl;
    }
});
```

Queries can request all entities with a specific state.

Flecs Advanced

State Machines

```
q = world.query<>("SWITCH|Movement");
q.iter([](flecs::iter it) {
    // States are stored in an array with raw entity ids
    auto states = it.column<flecs::entity_t>(1);

    for (auto i : it) {
        // Wrap state in entity class
        auto state = it.world.entity(states[i]);
        cout << it.entities(i).name()
            << " is " << state.name() << std::endl;
    }
});
```

Queries can also request all states for a specific switch

Flecs Advanced

State Machines

Show me code!

Flecs Advanced

Singletons

Flecs Advanced

Singletons

Singletons

Games often have components of which there is only a single instance. Flecs has an API for storing and retrieving singletons components.

Flecs Advanced

Singlenton

```
// Set Game singleton  
world.set<Game>({ 10, 20, 30 });  
  
// Get Game singleton  
const Game *g = world.get<Game>();  
  
// Return singleton entity for Game  
auto s_game = world.singleton<Game>();  
  
// true!  
s_game == world.component<Game>();
```

Singlenton are set & get with methods on the world object. A singlenton value is stored on the component entity.

Flecs Advanced

Singletons

```
q = world.query<Position>("$Game");
q.iter([](flecs::iter it, Position *p) {
    auto game = it.column<Game>(2);
    for (auto i : it) {
        if (p[i].x > game->max_x) p[i].x = game->max_x;
    }
});
```

Queries can retrieve singletons with the \$ character.

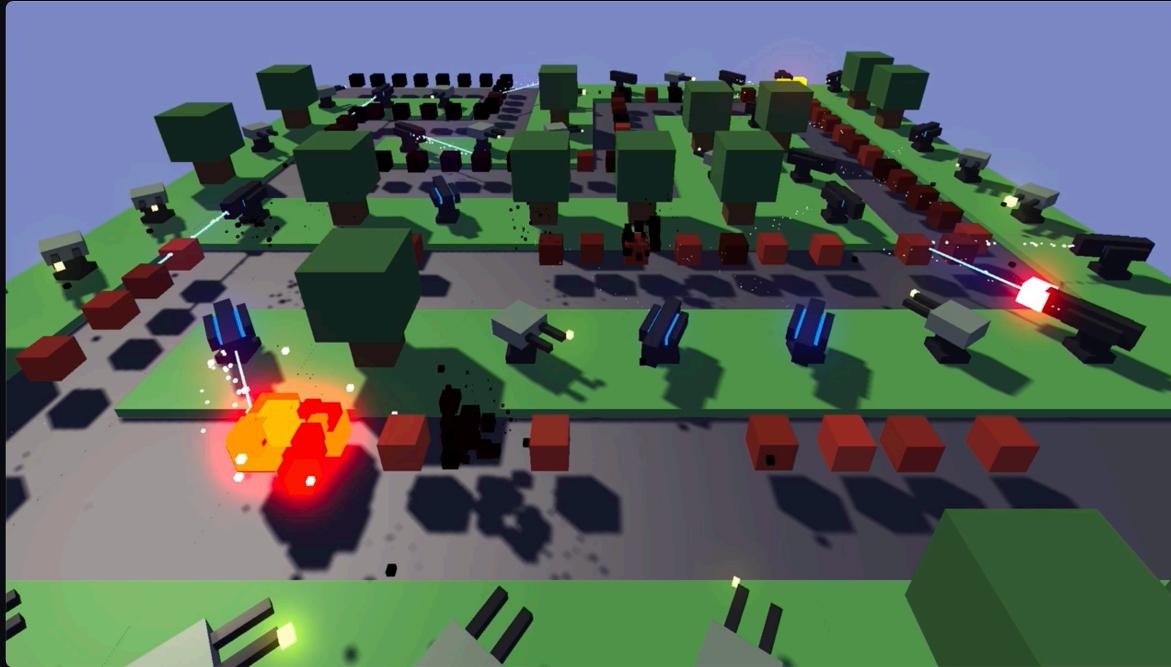
Flecs Advanced

Singletons

Show me code!

Flecs Advanced

Tower Defense Example



Small “pure” ECS demo that shows how to use modules, prefabs, systems and a lot more.

Still under construction!

https://github.com/SanderMertens/tower_defense

That's it!

Thanks for attending 

Github: <https://github.com/SanderMertens/flecs>

Discord: <https://discord.gg/BEzP5Rgrrp>