ECS: A PROGRAMMING PARADIGM

Azriel Hoh

ABOUT

- **Before:** Automated distributed system management.
- **Now:** Making a Rust game.

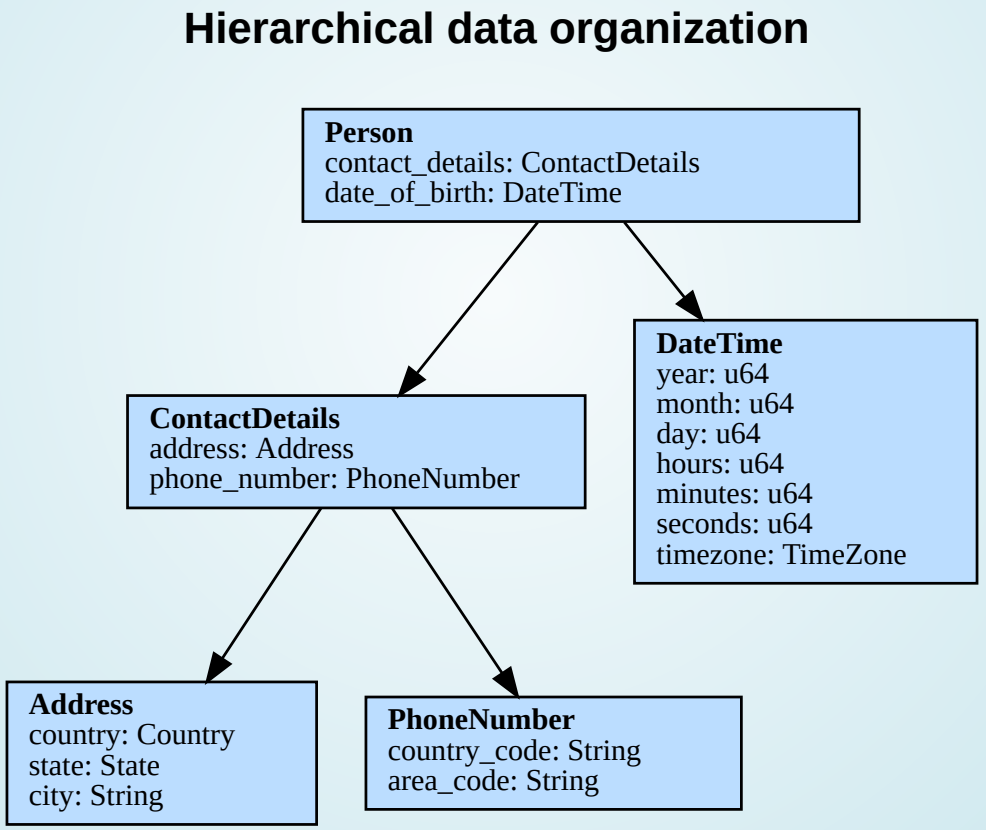
AGENDA

1. Data Organisation
    - OO: Array of Structs
    - EC: Struct of Arrays
2. Game Scenario
3. Patterns
    - Generational Arena
    - Storages
4. Logic
    - MVC
    - Systems
    - Dispatcher
5. Summary
    - Trade-offs

DATA: OO

DATA: OO

We usually define data types hierarchically:



DATA: OO

In code, that looks something like this:

```
struct Person {
    contact_details: ContactDetails,
    date_of_birth: DateTime<Utc>,
}

struct ContactDetails { /* fields */ }
struct DateTime<Tz> { /* fields */ }
```

DATA: OO

A list of people can be stored like this:

```
struct World {
    people: Vec<Person>,
}
```

This layout is:

- Hierarchical.
- Coined *arrays of structs* (AOS).

DATA: STRUCT OF ARRAYS

DATA: STRUCT OF ARRAYS

Split `Person` into parts, keep the same parts together:

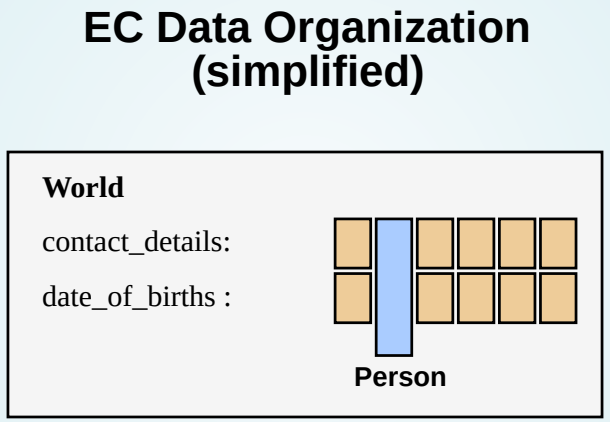
```
struct World {
    contact_details: Vec<ContactDetails>,
    date_of_births: Vec<DateTime<Utc>>,
}
```

Each index into the `Vec<_>`s represents a person.

Need to make sure `Vec<_>`s have the same length.

DATA: STRUCT OF ARRAYS

In picture form:



A `Person` is a vertical slice from both `Vec<_>`s.

`Person` no longer exists, it's now an abstract concept.

DATA: STRUCT OF ARRAYS

Why would anyone want to do that?

CALCULATE AVERAGE AGE

CALCULATE AVERAGE AGE

Formula:

```
average_age = total_age / number_of_people
```

CALCULATE AVERAGE AGE

```
struct Person {
    contact_details: ContactDetails,
    date_of_birth: DateTime<Utc>,
}

let people: Vec<Person> = /* ... */;
let now = Utc::now();

let mean_age = people
    .iter()
    // Sum everyone's age
    .fold(0., |sum, p| sum + years_since(p.date_of_birth, now))
    // Divide by number of people
    / people.len() as f32
```

MEMORY ACCESS

Here is the data that we iterate over:

P0	Contact Details	Address
		Phone Number
	Date of Birth	
P1	Contact Details	Address
		Phone Number
	Date of Birth	
P2	Contact Details	Address
		Phone Number
	Date of Birth	
P3	Contact Details	Address
		Phone Number
	Date of Birth	

MEMORY ACCESS

Here is the data that we need:

P0	Contact Details	Address
		Phone Number
	Date of Birth	
P1	Contact Details	Address
		Phone Number
	Date of Birth	
P2	Contact Details	Address
		Phone Number
	Date of Birth	
P3	Contact Details	Address
		Phone Number
	Date of Birth	

CALCULATE AVERAGE AGE: SOA

```
let contact_details: Vec<ContactDetails> = /* ... */;
let date_of_births: Vec<DateTime> = /* ...  */;
let world = World { contact_details, date_of_births };

let now = Utc::now();

// Note: we don't actually read `contact_details`.
let mean_age = world
    .date_of_births
    .iter()
    // Sum everyone's age
    .fold(0., |sum, dob| sum + years_since(*dob, now))
    // Divide by number of people
    / world.date_of_births.len() as f32
```

MEMORY ACCESS: SOA

Here is the data that we iterate over:

D0	Date of Birth
D1	Date of Birth
D2	Date of Birth
D3	Date of Birth

MEMORY ACCESS: SOA

Here is the data that we need:

D0	Date of Birth
D1	Date of Birth
D2	Date of Birth
D3	Date of Birth

MEMORY ACCESS: SOA

Should I care?

DATA LOCALITY

DATA LOCALITY

When you fetch data, the computer thinks:

Maybe I should fetch more!

DATA LOCALITY

	Hierarchical	SOA
Cache Capacity	P0	DateTime0
		DateTime1
	P1	DateTime2
		DateTime3
	P2	DateTime4
		DateTime5
	P3	DateTime6
		DateTime7
	P4	DateTime8
		DateTime9

BENCHMARK: AOS VS SOA

 azriel91/aos_vs_soa/lib.rs#L112-L117

```
git clone git@github.com:azriel91/aos_vs_soa.git
cd aos_vs_soa
cargo bench
```

BENCHMARK: AOS VS SOA

Sample output:

```
running 2 tests
test tests::array_of_structs::avg_age ... bench:   2,427,309 ns/iter (+/- 201,343
test tests::struct_of_arrays::avg_age ... bench:   2,192,988 ns/iter (+/- 136,117

test result: ok. 0 passed; 0 failed; 0 ignored; 2 measured; 0 filtered out
```

BENCHMARK: AOS VS SOA

```
name        aos ns/iter  soa ns/iter  diff ns/iter  diff %  speedup
::avg_age   2,427,309    2,192,988        -234,321  -9.65%   x 1.11
::avg_age   2,438,766    2,215,783        -222,983  -9.14%   x 1.10
::avg_age   2,444,267    2,196,445        -247,822 -10.14%   x 1.11
::avg_age   2,430,973    2,195,141        -235,832  -9.70%   x 1.11
::avg_age   2,462,707    2,185,442        -277,265 -11.26%   x 1.13
::avg_age   2,414,035    2,211,378        -202,657  -8.39%   x 1.09
::avg_age   2,431,339    2,219,773        -211,566  -8.70%   x 1.10
::avg_age   2,433,808    2,225,186        -208,622  -8.57%   x 1.09
::avg_age   2,404,514    2,175,141        -229,373  -9.54%   x 1.11
::avg_age   2,409,768    2,200,761        -209,007  -8.67%   x 1.09
```

BREATHER

SCENARIO: GAME

SCENARIO: GAME

Disclaimer

These examples are based on the `specs` crate.
There are many other ECS implementations, whether in Rust or other languages.

SCENARIO: GAME

Player
- Render
- Position
- Velocity
- Input

Monster
- Render
- Position
- Velocity

Speech Bubble
- Render
- Input

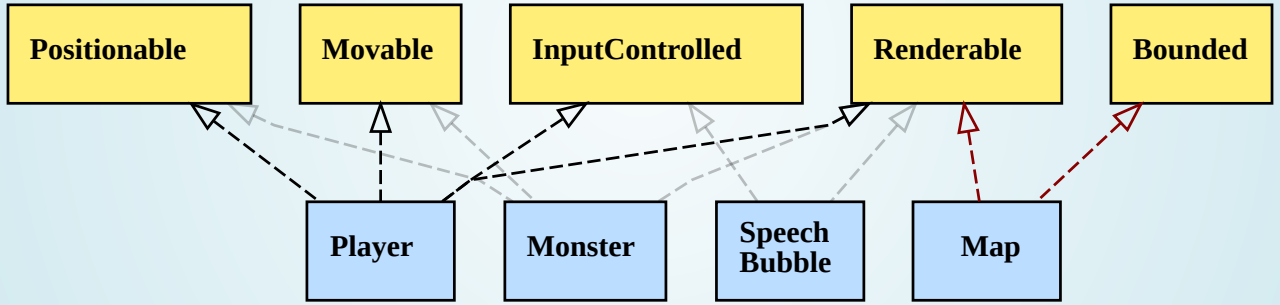
Map
- Render
- Bounds

OO MODEL

- **G**ood **O**bject **O**riented **D**esign (GOOD)
  - Interface inheritance
  - Composition
- **O**bject **O**riented **P**rogramming **S**yntax (OOPS)
  - Implementation inheritance

OO MODEL

Interface Inheritance



OO MODEL

In code:

```
trait Renderable {}
trait Positionable {}
trait Movable {}
trait InputControlled {}

struct Player;
impl Renderable for Player {}
impl Positionable for Player {}
impl Movable for Player {}
impl InputControlled for Player {}

struct Monster;
impl Renderable for Monster {}
impl Positionable for Monster {}
impl Movable for Monster {}

struct World {
    players: Vec<Player>,
    monsters: Vec<Monster>,
}
```

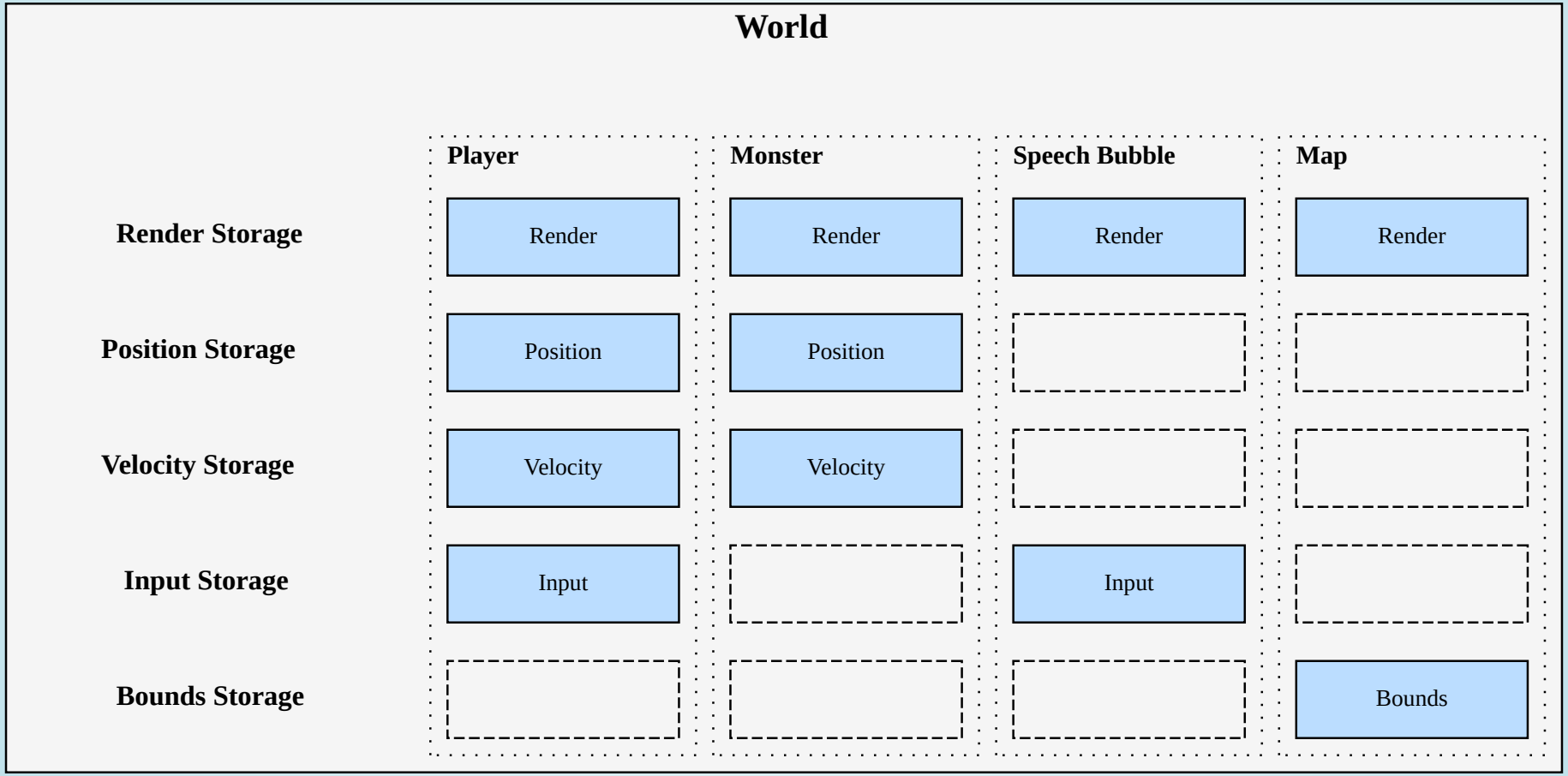
OO MODEL

Problems:

- Development overhead for common logic.
- Borrowing parts of an object.
  - `Arc<Mutex<_>>`
  - `Copy` / `Clone`
  - Deconstruct / Reconstruct
  - "View" structs

EC MODEL

Some `Components` only apply to some entities.



EC MODEL

In code, components can be represented like this:

```
struct World {
    // Note: Can't simply store traits like this,
    // In this example, components must be concrete types
    renderables: Vec<Option<Renderable>>,
    positions: Vec<Option<Position>>,
    velocities: Vec<Option<Velocity>>,
    inputs: Vec<Option<Input>>,
    bounds: Vec<Option<Bound>>,

    // How to track each type of entity? `usize`!
    players: Vec<usize>,
    monsters: Vec<usize>,
    speech_bubbles: Vec<usize>,
    maps: Vec<usize>,
}
```

EC MODEL

Problem?

EC MODEL

More like, problems!

- **Memory bloat:** Unused slots.
- **Entity creation:** Finding a free index.
  - **Overflow:** What if we run out of indices?
- **Entity deletion:**
  - **Mutable access:** Remove components from all storages.
  - **Can we delete:** Is anyone referencing this entity?

BREATHER

GENERATIONAL ARENA

GENERATIONAL ARENA

Arena? The Colosseum!

GENERATIONAL ARENA

Arena: Request a chunk of memory, and manage it yourself.

Generational: Use a *generational index* to track free space in the arena.

GENERATIONAL ARENA

Instead of referencing an entity by `usize`, use:

```rust
#[derive(Eq, PartialEq, ..)]
pub struct GenerationalIndex {
    index: usize,
    generation: u64,
}
```

If the generation does not match, it's a different entity.

GENERATIONAL ARENA

Key concepts:

- Keep a pool of memory, remember which slots are "empty"
- Track a generation number used when inserting data.
- When freeing a slot, ensure the generation is at least one more than the generation of the freed slot.

GENERATIONAL ARENA

Instead of:

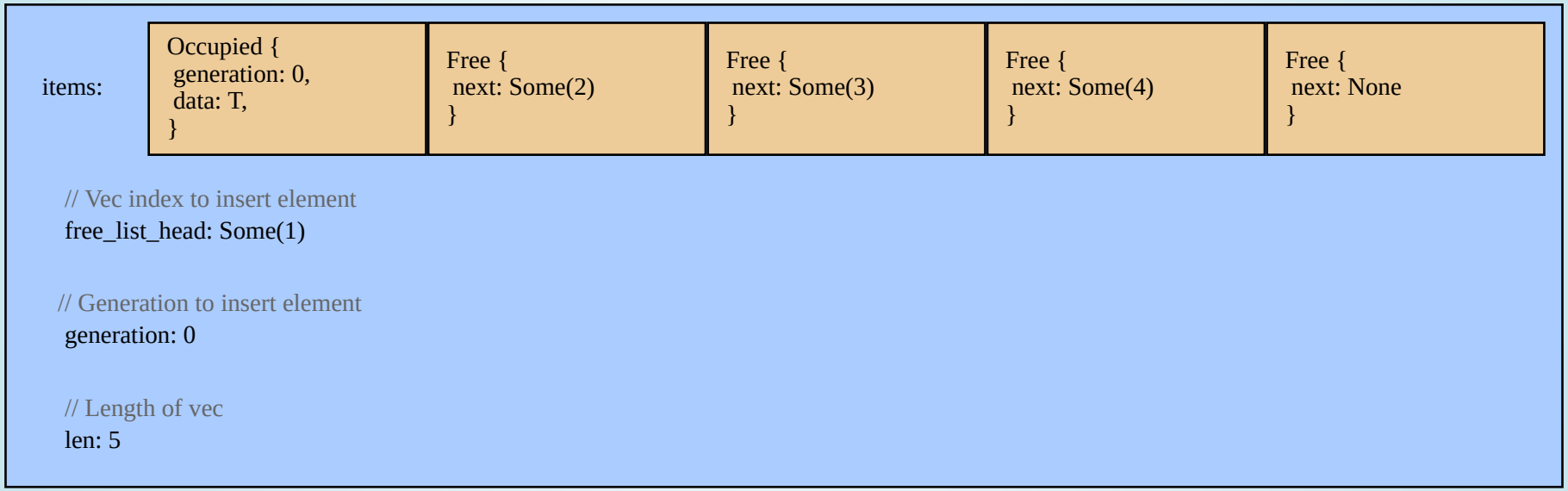
- `Vec<T>` where `T` is the stored component
- An incrementing `usize` to track next free index

Use `Vec<Entry<T>>`, where `Entry` is:

```
enum Entry<T> {
    Free { next_free: Option<usize> },
    Occupied { generation: u64, value: T },
}
```

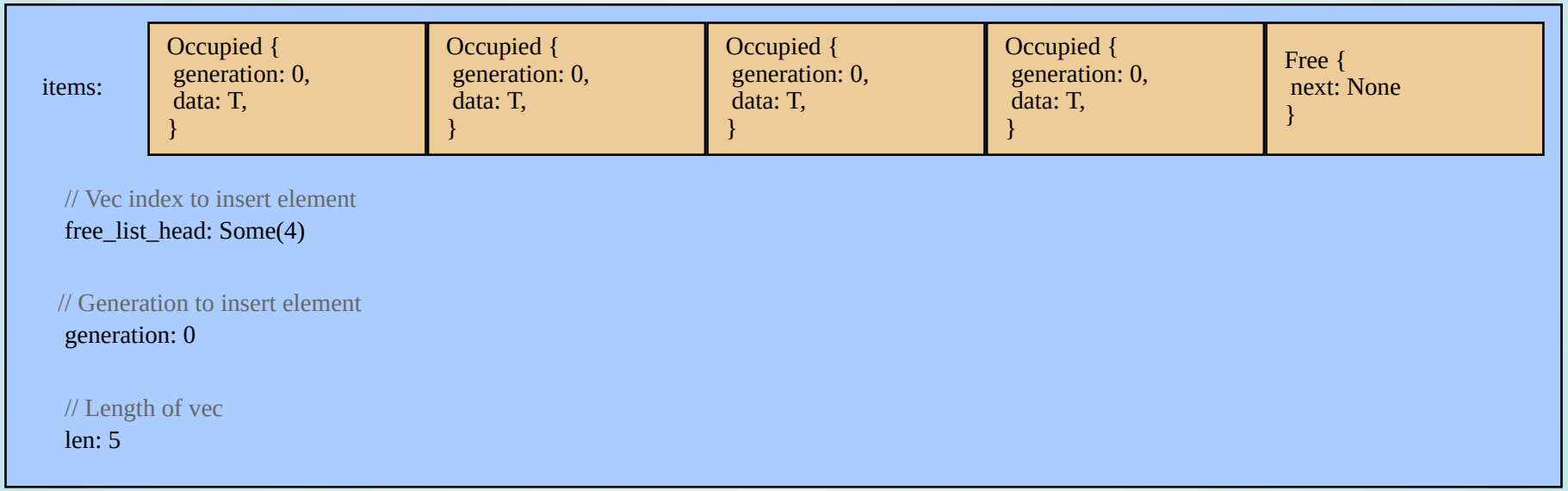
GENERATIONAL ARENA

Generational Arena



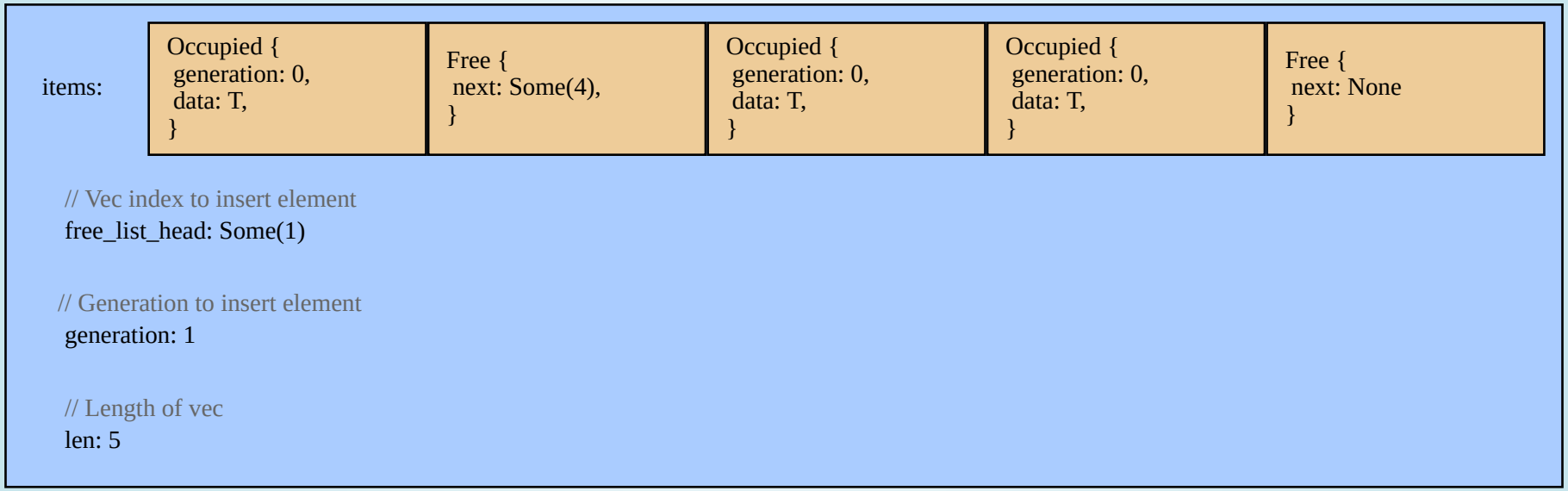
GENERATIONAL ARENA

Generational Arena



GENERATIONAL ARENA

Generational Arena



GENERATIONAL ARENA

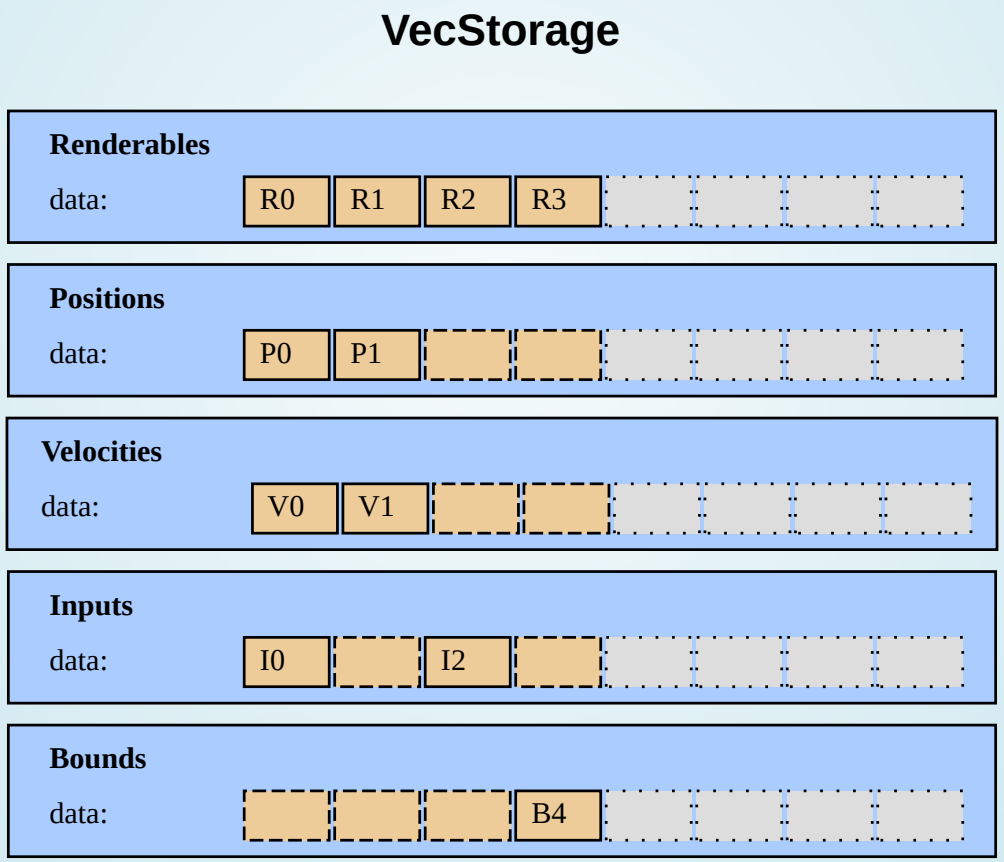
Solves:

- **Memory bloat:** Unused slots.
- **Entity creation:** Finding a free index.
  - **Overflow:** Reuse existing indicies
- **Entity deletion:**
  - **Mutable access:** Old gen signals absence.
  - **Can we delete:** Old gen signals absence.

EC STORAGES

EC STORAGES

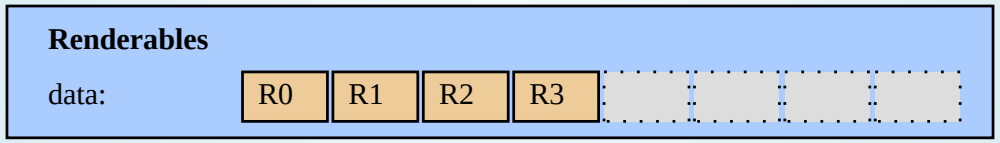
`VecStorage` can be a waste of memory:



EC STORAGES

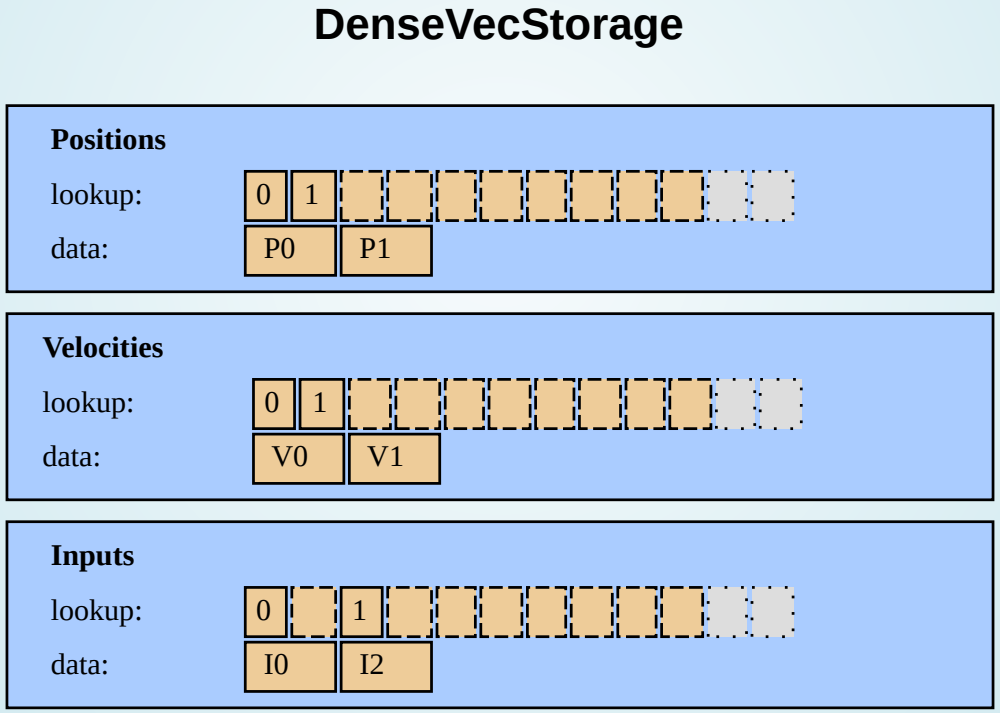
Vec is optimal for frequent components:

VecStorage



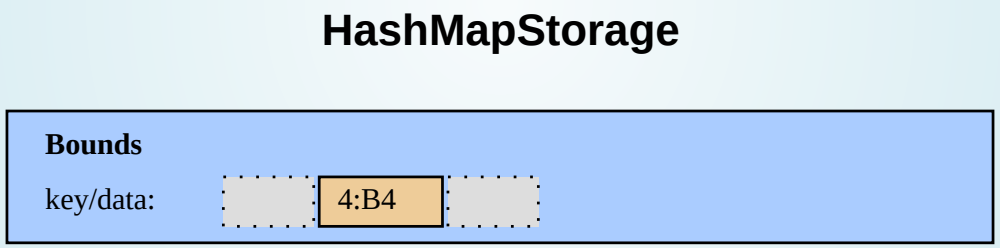
EC STORAGES

A lookup Vec saves memory for common components:



EC STORAGES

Rare components can be stored using a `HashMap`:



EC STORAGES

Solves:

- **Memory bloat:** Unused slots.
- **Entity creation:** Finding a free index.
  - **Overflow:** Reuse existing indicies
- **Entity deletion:**
  - **Mutable access:** Old gen signals absence.
  - **Can we delete:** Old gen signals absence.

BREATHER

LOGIC

TASK: UPDATE POSITIONS

`Players` and `Monsters` have a position and velocity.

Write a function to update position based on velocity.

```
position += velocity
```

LOGIC: OO

Write the `update` function inside the `class`.
Hiding the code means better encapsulation.

– every CS course (how we heard it)

LOGIC: OOPS

How to increase software maintenance costs:

```
impl PositionUpdate for Player {
    fn update(&mut self) {
        self.pos += self.vel;
    }
}

impl PositionUpdate for Monster {
    fn update(&mut self) {
        self.pos += self.vel;
    }
}
```

LOGIC: GOOD (MVC)

```
type Kinematic = i32;
impl Positionable for Player { fn pos_mut(&mut self) -> &mut Kinematic { &mut sel
impl Positionable for Monster { fn pos_mut(&mut self) -> &mut Kinematic { &mut se
impl Movable for Player { fn vel(&self) -> &Kinematic { &self.vel } }
impl Movable for Monster { fn vel(&self) -> &Kinematic { &self.vel } }

trait GameObject: Positionable + Movable {}

fn position_update(game_objects: &mut [Box<GameObject>]) {
    for i in 0..game_objects.len() {
        let vel = game_objects[i].vel();
        *game_objects[i].pos_mut() += *vel;
    }
}
```

playpen

LOGIC: GOOD (MVC)

- Logic is generally pretty clean.
- Difficult to update objects due to borrowing rules.

  Choose from:

  - `Arc<Mutex<_>>`
  - View structs – `&field`.
  - Deconstruct, reconstruct.
  - ...

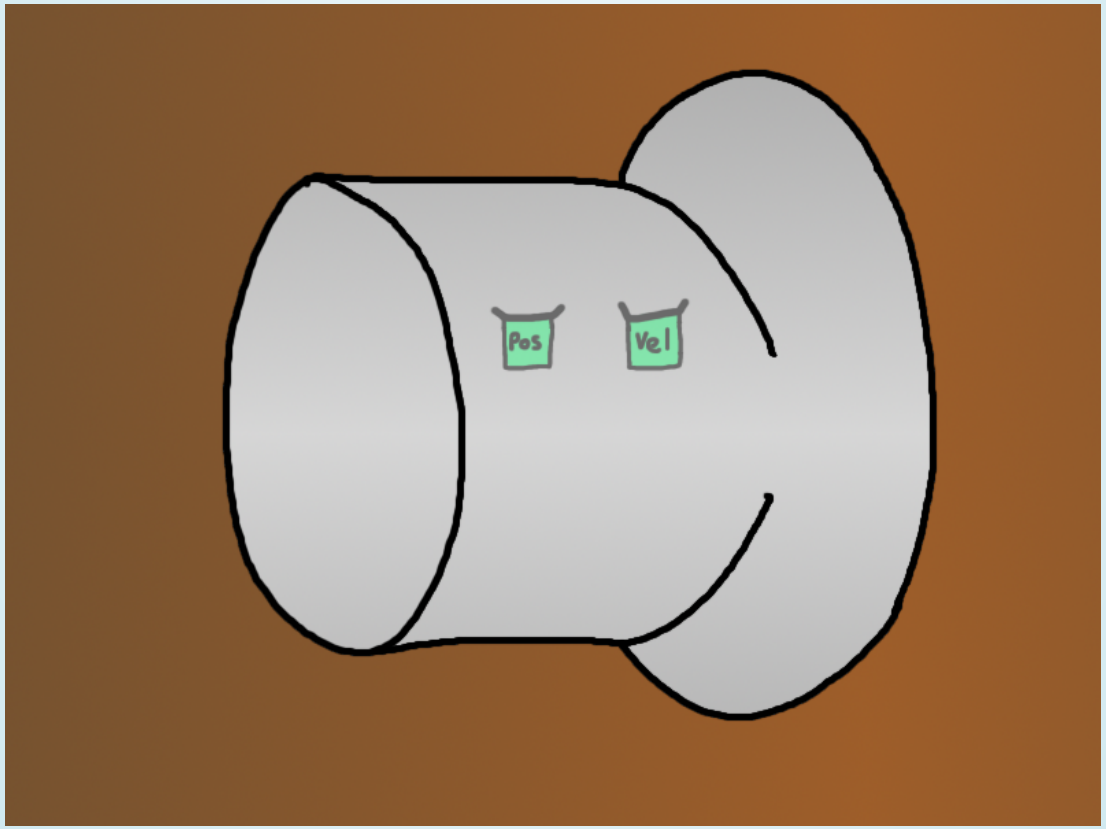
SYSTEMS

SYSTEMS

The **S** in **ECS**.

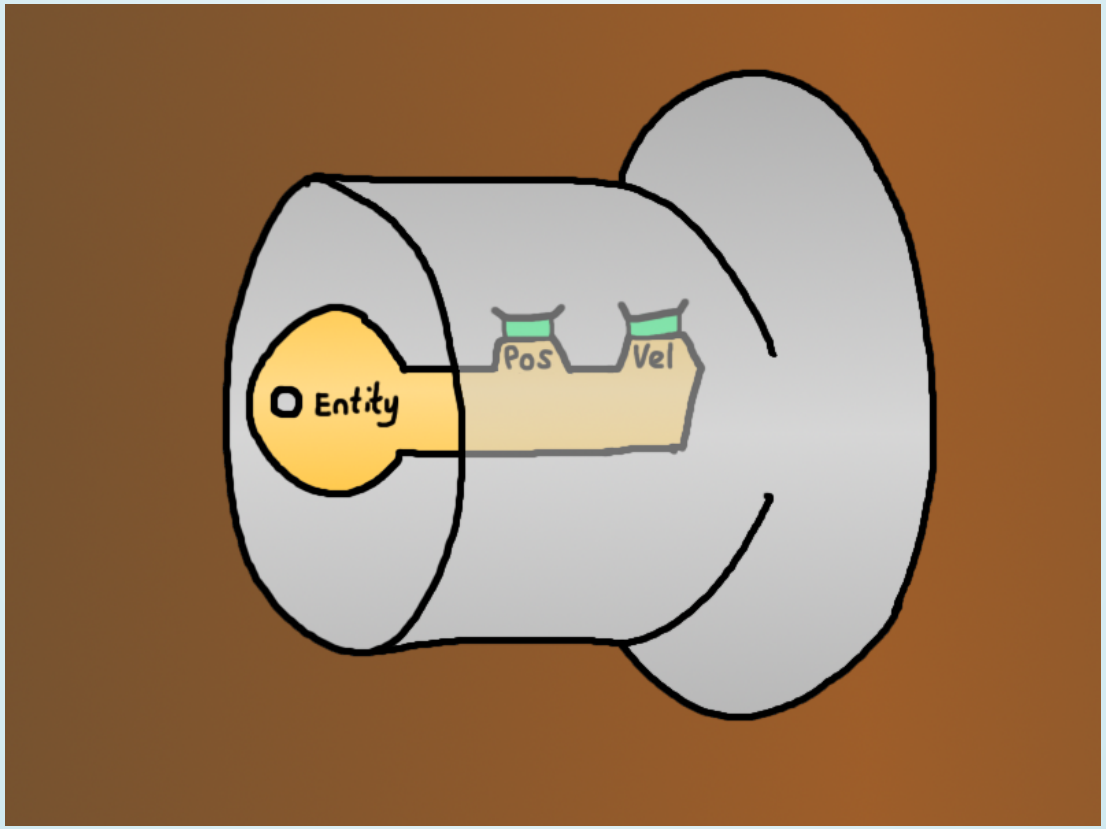
LOCK & KEY ANALOGY

Given this lock:



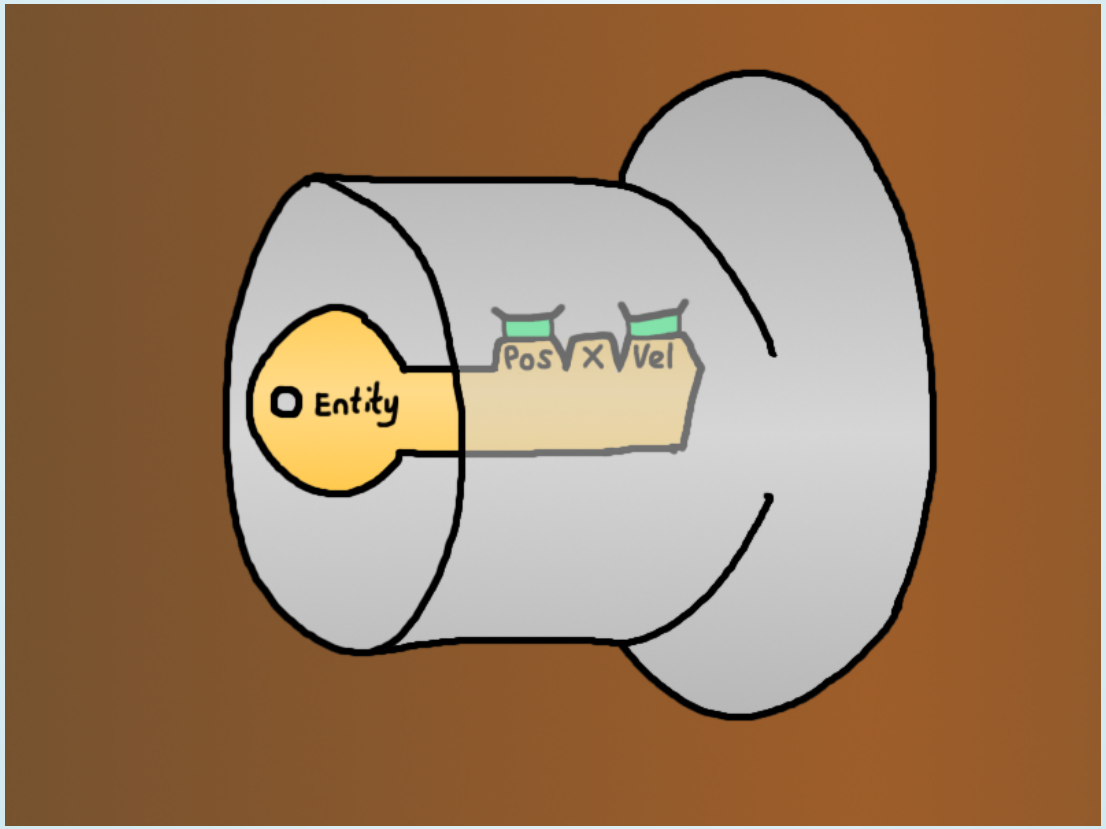
LOCK & KEY ANALOGY

This key works:



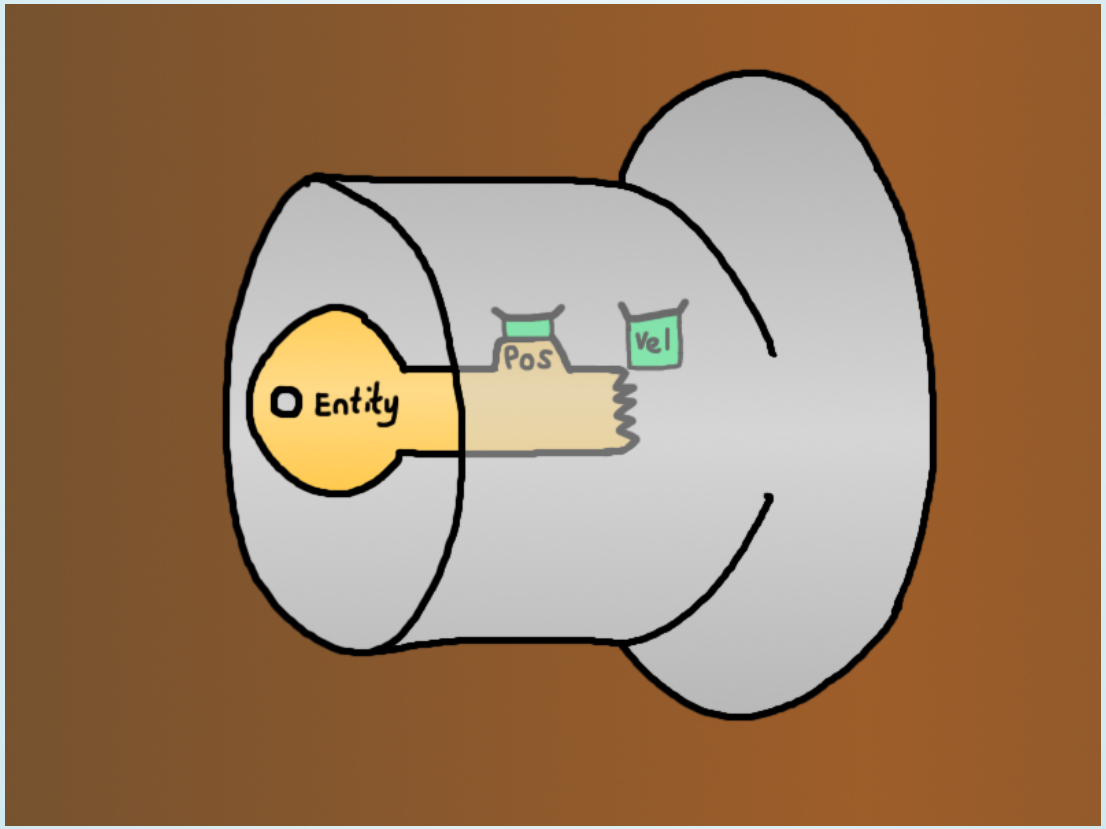
LOCK & KEY ANALOGY

So does this:



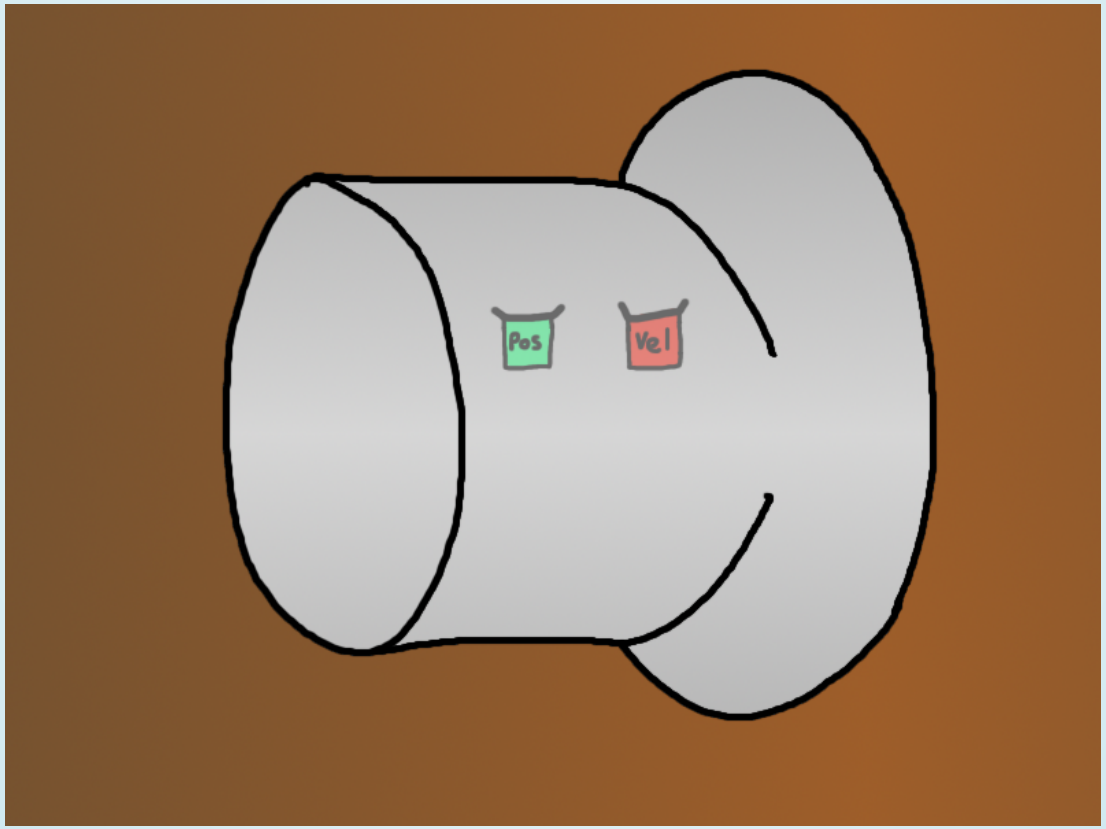
LOCK & KEY ANALOGY

But not this:



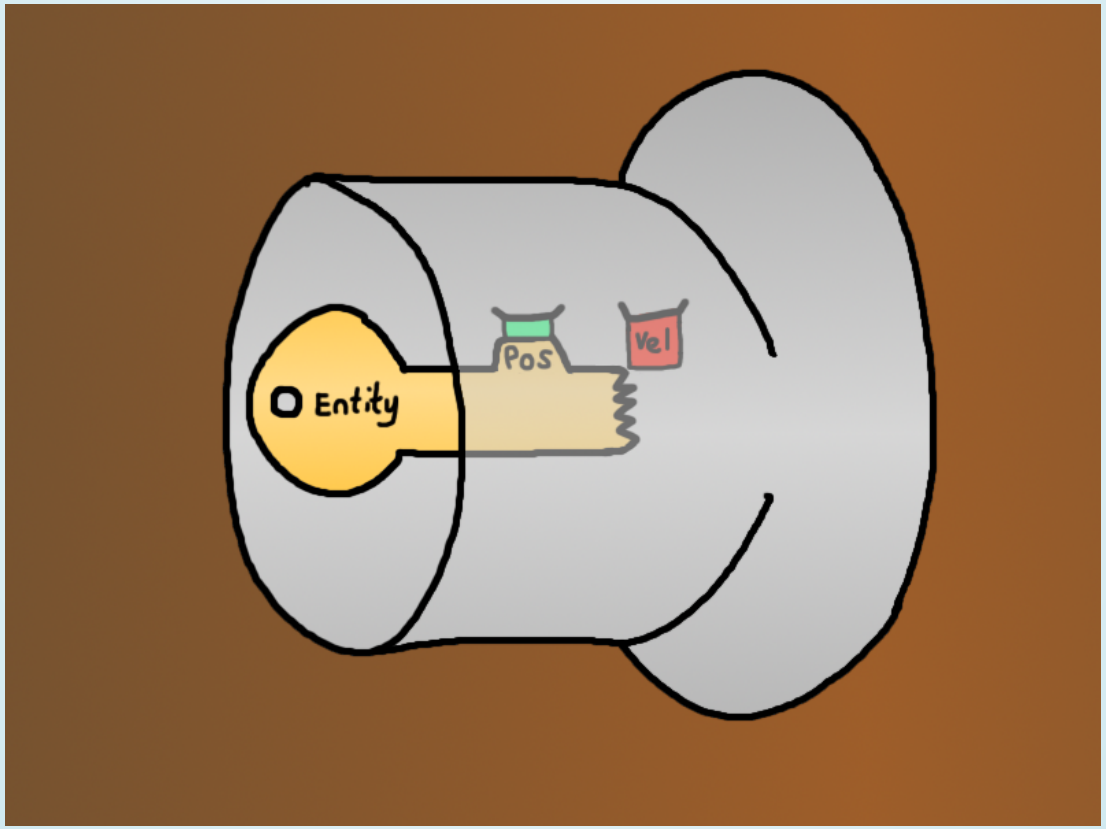
LOCK & KEY ANALOGY

For this lock:



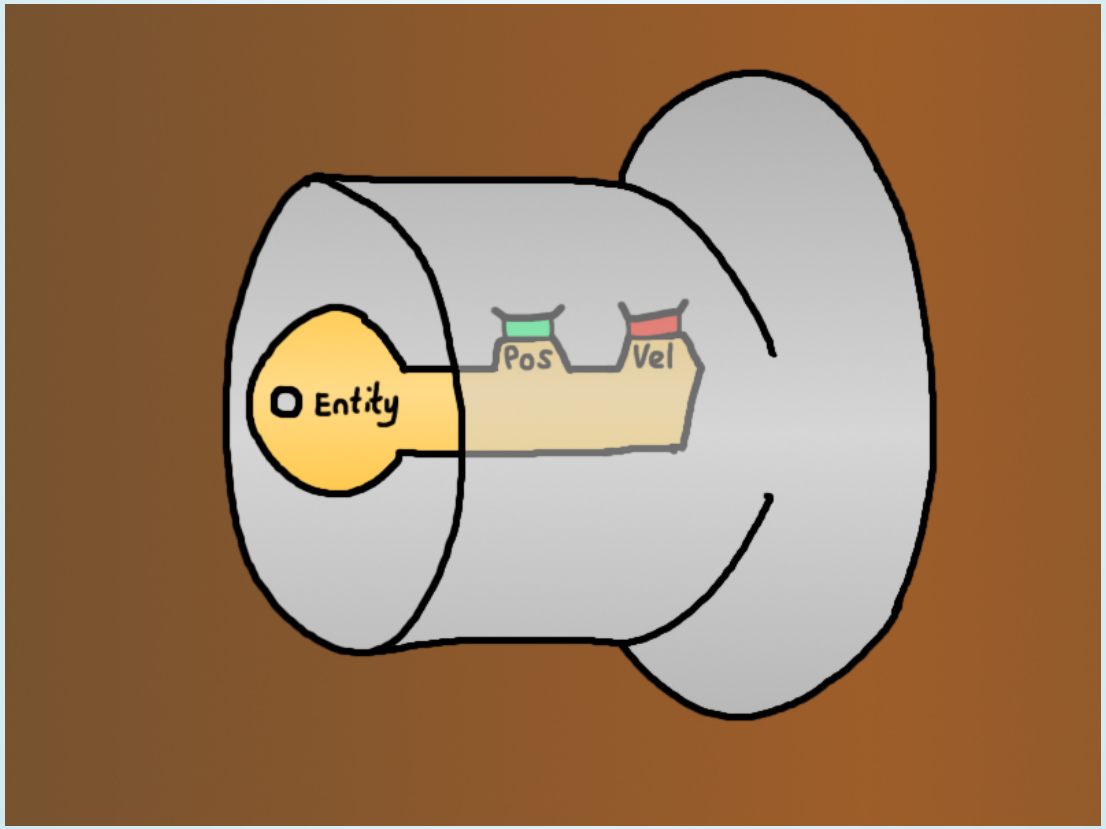
LOCK & KEY ANALOGY

This key works:



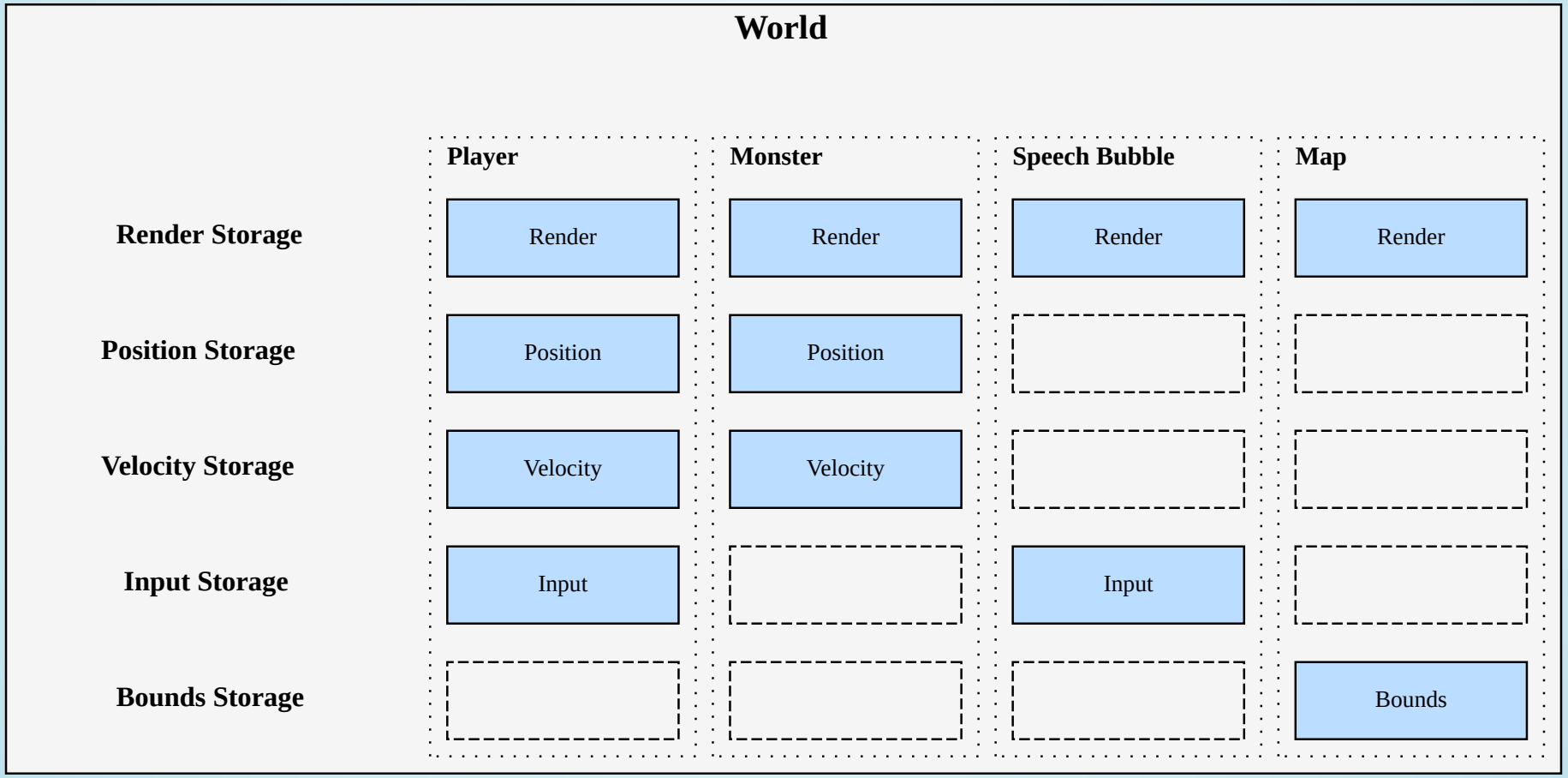
LOCK & KEY ANALOGY

But not this:



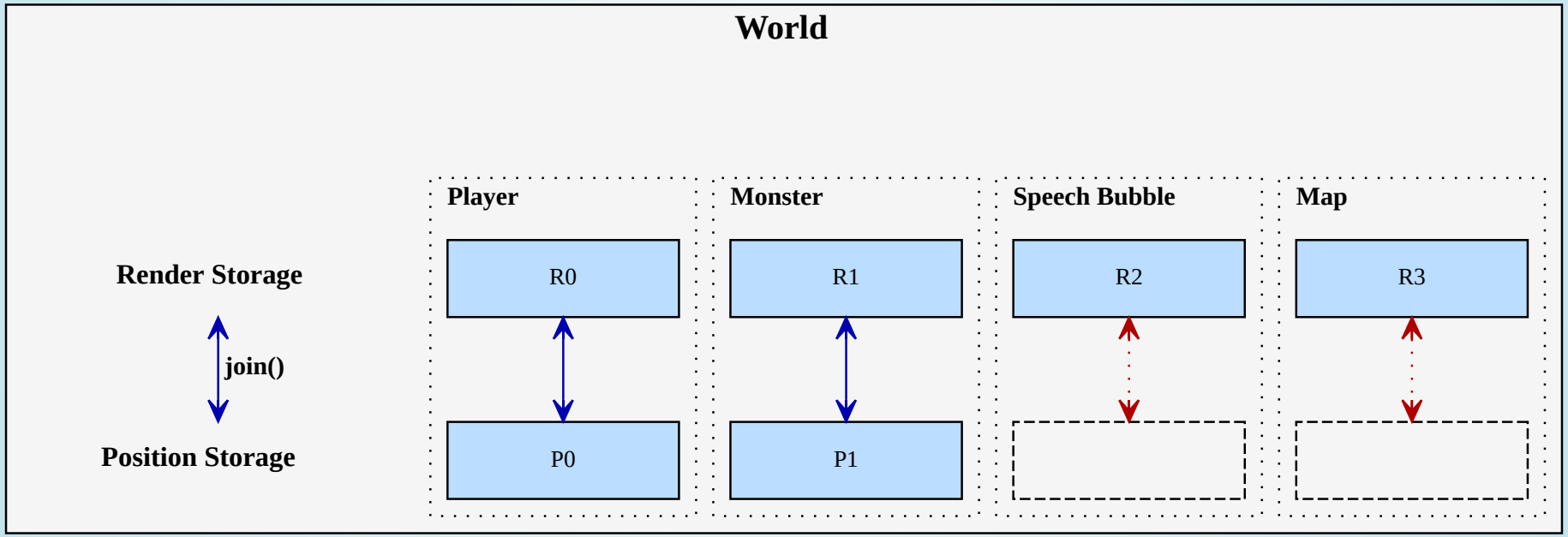
SYSTEMS

Remember this:



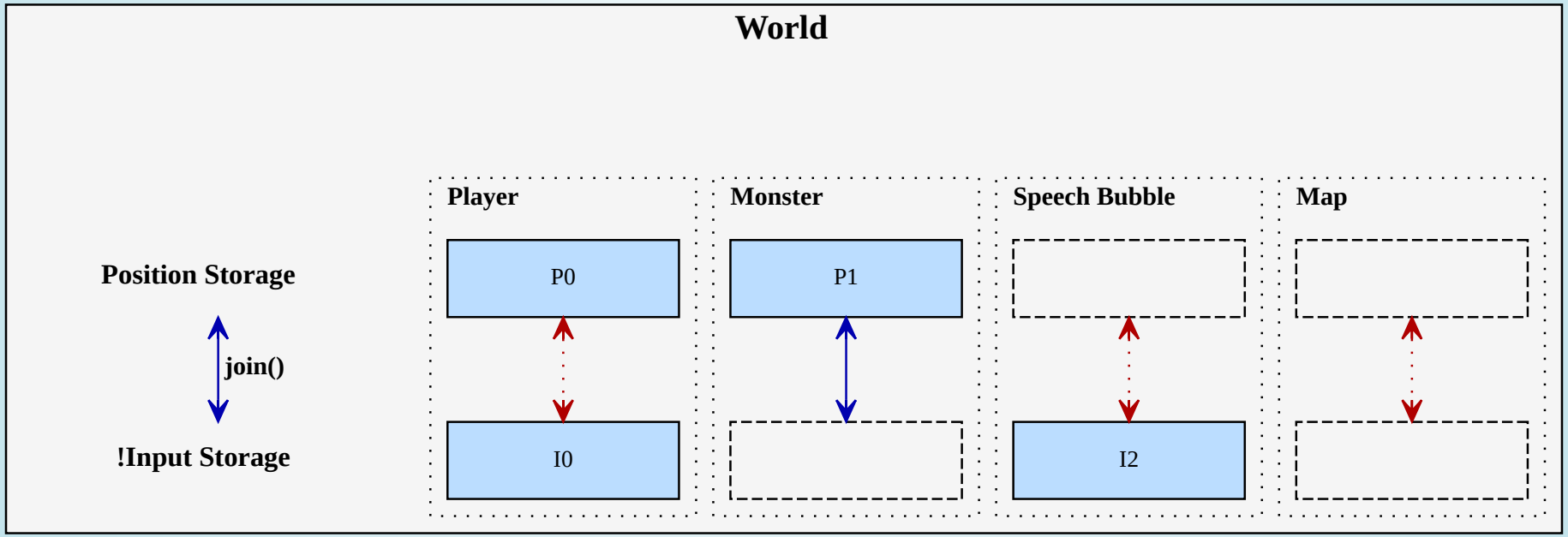
SYSTEMS

```
// Whitelist components (must haves)
(&renderables, &positions).join() -> [(R0, P0), (R1, P1)] // player, monster
```



SYSTEMS

```
// Exclude / blacklist components
(&positions, !&inputs).join() -> [(P1, ())] // monster
```



SYSTEMS

Compare:

```
/// OO position_update():
for i in 0..game_objects.len() {
    *game_objects[i].pos_mut() += *game_objects[i].vel();
}

// PositionUpdateSystem run():
for (pos, vel) in (&mut positions, &velocities).join() {
    *pos += *vel;
}
```

BREATHER

SYSTEM DATA

SYSTEM DATA

Systems operate over data.
We call that `SystemData`.

`SystemData` looks like this:

```
/// Systems operate on resources that are borrowed from the `World`.
/// This is the lifetime that those references live for.
///
/// => The resources must live for at least as long as `'s`.
type PositionUpdateSystemData<'s> = (
    WriteStorage<'s, Position>,
    ReadStorage<'s, Velocity>,
);
```

SYSTEM DATA

And is used like this:

```
struct PositionUpdateSystem;

type PositionUpdateSystemData<'s> = (
    WriteStorage<'s, Position>,
    ReadStorage<'s, Velocity>,
);

impl System<'s> for PositionUpdateSystem {
    type SystemData = PositionUpdateSystemData<'s>;

    fn run(&mut self, (mut positions, velocities): Self::SystemData) {
        // Here is the system logic!
        for (pos, vel) in (&mut positions, &velocities).join() {
            *pos += *vel;
        }
    }
}
```

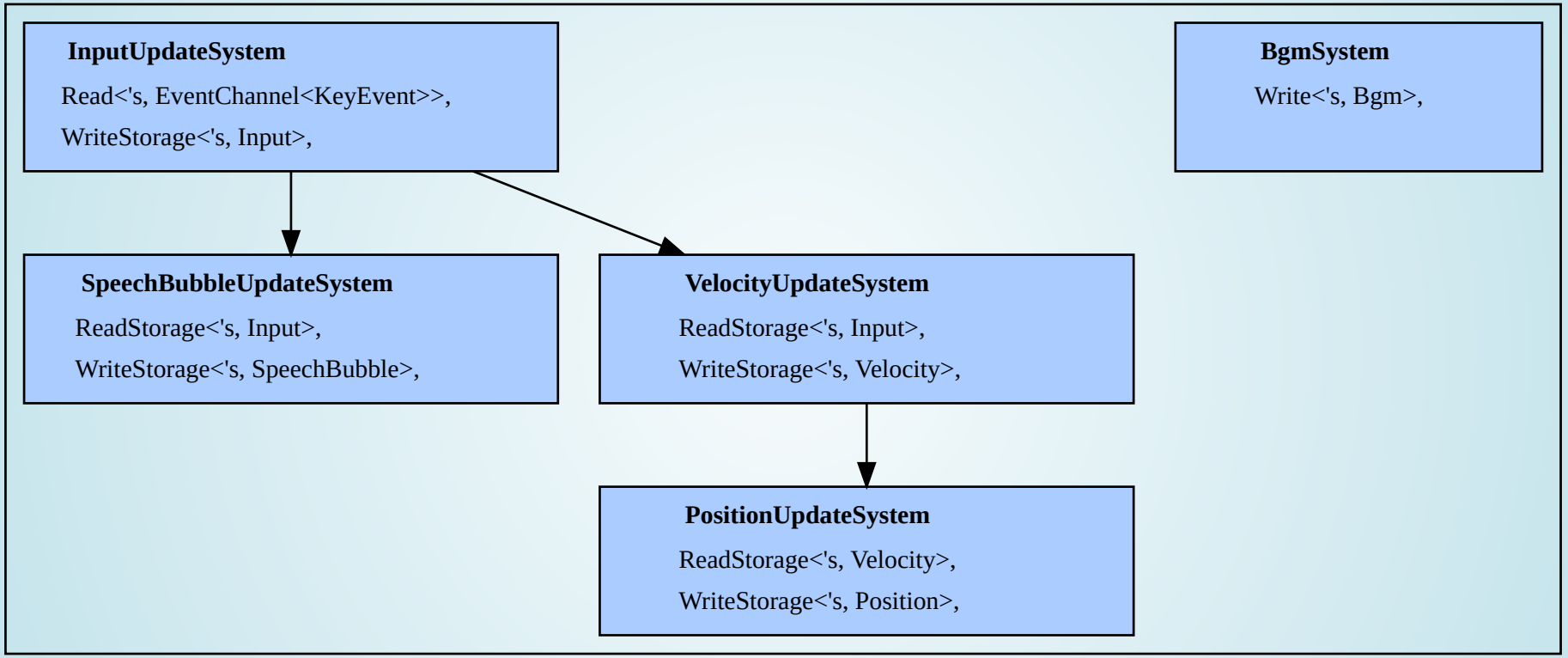
DISPATCHER

DISPATCHER

What it is:

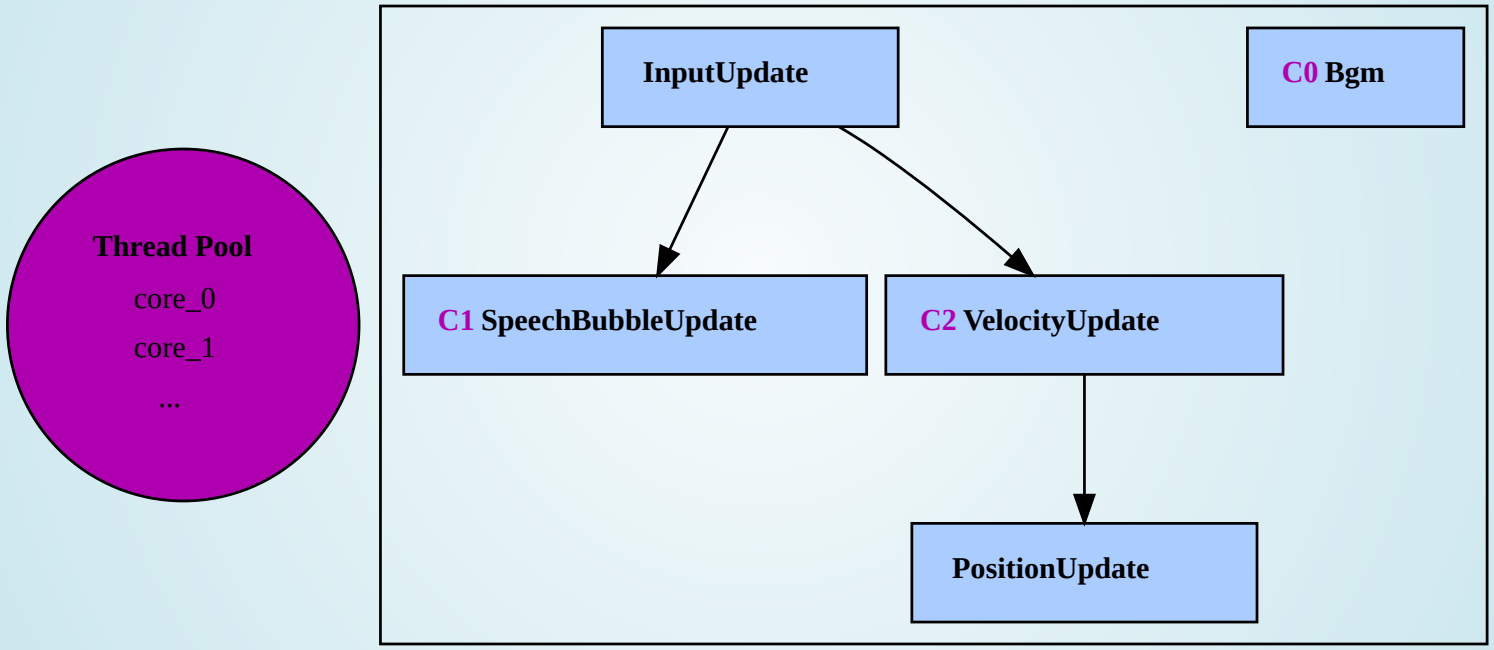
- System graph
- Thread pool
- Execution
  - Parallel
  - Safe: multiple readers / exclusive writer

DISPATCHER: SYSTEM GRAPH



specs/examples/full.rs#L221-L230

DISPATCHER: THREAD POOL



SUMMARY

- Compare GOOD with ECS, not OOPS with ECS
- It's a trade-off.

	OO	**EC**
Logical partitioning	State	Behaviour
Visible concrete model		
Optimized for cache usage[1]		
Borrow-checker management[2]		

[1] Rust doesn't auto-vectorize floats well.

[2] Ease of passing data for parallelization

QUESTIONS, ANSWERS AND COMMENTS

THANKS!

- Catherine West / `@Kyrenite`: Learnt *a lot* from her blog.
- The `specs` team: Who have written such a great library.
- Maik Klein: For expanding my knowledge on auto-vectorization.
- `@medusacle`: For much feedback on improving these slides.

LINKS

- Slides: https://github.com/azriel91/ecs_paradigm
- RustConf 2018 Closing Keynote: https://www.youtube.com/watch?v=P9u8x13W7UE
- `@Kyrenite`'s Blog: https://kyren.github.io/2018/09/14/rustconf-talk.html
- `specs`: https://crates.io/crates/specs
- `soa-derive`: https://github.com/lumol-org/soa-derive
- Benchmark comparison: https://github.com/azriel91/aos_vs_soa
- Component Graph System: https://github.com/kvark/froggy/wiki/Component-Graph-System