My RustConf 2018 Closing Keynote

Sep 14, 2018

I promised two things when I did the closing keynote at RustConf 2018:

1. I will put the slides for the talk online
2. I will post the long-form version of the talk on a my blog

    2.a. I will actually make a blog for this purpose

…and I have taken my damn sweet time doing so. Finally, I have done what I promised I would do!

If you haven't seen it and would like to, you can see the live talk that I did here.

The slides for the talk (with a small errata) are now hosted here.

So, I said in my talk that I would post a potentially more interesting long-form version of the talk that includes things that I couldn't fit into the 30-ish minute time slot. What I've included below is my original long-form version of this talk, but my original intention was not to post this as-is. Originally I wanted to clean this up a bit more and make this into something that wasn't so much a giant wall of text, but after I started doing this I realized that I was just rewriting it entirely, and at that rate I would never get around to releasing it, which I *promised* I would do in a timely manner.

I'm not super happy with this, and in retrospect I think many of the things I cut for the 30-ish minute talk were probably wise to cut. I think the parts about console development are wildly speculative and not that helpful, and the part at the end about language boundaries being particularly difficult is, while true, kind of beside the point. However, people seemed very interested and are (very rightly) reminding me that I haven't posted this yet, so rather than delay any further I'm going to give you the very lightly edited text version of the talk that I made. Keep in mind this was before I realized I would NEVER fit this into 30 minutes :). This is written in a very informal, conversational style because this was intended as the source material for my talk.

Without further ado…

Rough draft of RustConf 2018 talk

This is a very rough outline of the things I'd like to talk about in my RustConf 2018 keynote.

This talk assumes that you know a bit of C++ in addition to a bit of Rust, and there are quite a lot of C++ code samples, but I hope they're obvious enough not to be distracting to the people who might not be used to C++.

Basic thesis of the talk:

Rust, by design, makes certain programming patterns more painful than others. This is a GOOD thing! It turns out that, for game development, the patterns that end up being the easiest to deal with in Rust closely mirror the patterns that are easiest for game development in *any* language.

I unfortunately had to learn this the HARD way!

Rust highly rewards data-oriented design with simple, understandable ownership semantics, and this is great news because this is a really good fit for game development. I suspect this is also true generally, not just for game development! (but what do I know?)

Why should you listen to me, who am I?

I was the lead programmer for Starbound, I've I was one of the first members of Chucklefish back when we were all working over IRC. I was the technical lead at Chucklefish when we first decided that we should probably have somebody be called the "technical lead". I still work with Chucklefish peripherally, but I've since moved back to the US and I work on things more independently now.

But, honestly not sure you SHOULD listen to me! I've helped release ONE (1) whole commercial game, and some of this talk is a cautionary tale about what NOT to do based on the ONE (1) commercial game I've helped make.

I definitely feel like you shouldn't make any… real *grown up* business decisions based on my advice. I can't say for sure that Rust is a good fit for you, or whether you should or shouldn't make your game in Rust, certainly there are probably larger concerns depending on, if you're an indie, what your personal skill level and tolerances and preferences are, and if you're a company what tooling you need and what engines, middleware, etc you already use.

What I can say is that personally I will be using Rust for game development (and other things!) for the foreseeable future, and if you'd like to hear why, or if you've already decided that you'd like to use Rust and would like to know what you're in for, then this talk may be useful.

This talk is going to be more… vague than I'd prefer. I'm generally more comfortable giving people specific very technical advice where I can prove my point almost from first principles, but things like architecture advice or general advice are probably never going to be so clear cut. I definitely think it's still *useful*, which is why I'm doing it (and people seem interested), but remember: this is all just like, my opinion, man.

How do you EVEN make a game in Rust?

I've gotten a lot of questions over the past year or so that more or less ask:

> How do you make a game from scratch in Rust? No, seriously… HOW. I mean, I can see how you can do it *in theory*, but for some reason when I try to apply patterns that I'm used to from other languages I just hit lots of problems? I've heard that this is called "fighting the borrow checker", but… that doesn't really seem to *help* me much. What am I doing wrong?

Or, maybe it's something like:

> I can see how rust is great if you like very strict control, I can see it being used for small utilities or things where security is paramount, but it seems very restrictive! I don't really see how you could make something large like a game without running into these limitations. How do you structure something like a game without needing Rc<RefCell> and Arc<Mutex> everywhere?

These (straw man) questions are of course about games, but they also mirror sentiments I've seen about Rust in general. In case it's not obvious, I disagree with this sentiment (I am at RustConf after all), and I think the best way for me to talk about WHY I disagree with it is to talk about what I know, so I’m going to come at this from the perspective of games.

The talk could just be me coming up to a podium and saying “Jeez, it’s not that difficult! Data oriented design! If you’re making a game, use an ECS! … Thank you.” and then walking off stage, and from a certain perspective I do feel that it *can* be that simple, but I think there’s a more important fundamental point here and that’s what this talk is about. The very very short answer to this question is: if you forget OO design and instead just concentrate on the data representation of your game state (or whatever you’re making), and you try hard not to make anything more complicated than it really has to be, things can actually be pretty simple! This talk will be working up slowly to a simple ECS implementation in Rust and trying to give justifications for each step, but not *all* the steps are necessary. I don’t even think using some specific design pattern like an ECS is essential, I think the more important larger point is to let go of some habits that, at least *I* had that make things more difficult than they need to be, and in Rust more so.

For some people the things that I say in this talk might be *blindingly obvious*. If this is all obvious to you, that’s wonderful, but this hasn’t always been obvious to me. In fact, part of the reason that I’m such a fan of Rust is that I didn’t actually learn these lessons from trying to build a game in Rust, I mostly learned them before I ever even tried Rust for the first time, from mistakes that I made early in Starbound. I need somebody to pick on to demonstrate what NOT to do, so it’s really convenient that I have such a great example in my past self.

I’m definitely going to repeat some advice that other people have given. In fact, some of these things are definitely widely known and have almost become kind of truisms: Inheritance is bad, OO is mostly bad, ECS design is good, etc. Hopefully to the extent that those *are* true I provide more evidence for them and describe why these are especially important in the presence of the borrow checker, but hopefully I can also add some nuance to these and this will be helpful to other people who want to use Rust for game development.

How do you EVEN make a game *at all*

Certainly a lot of these questions only appear when you’re trying to come up with a game architecture from scratch, and certainly many people aren’t going to actually do this. In fact, generally the advice for indie developers is that you should NEVER make your own engine, and if you use something like Unity or Unreal to make your game, a lot of these decisions are going to be made for you.

I definitely did NOT follow this advice. In fact, I think Starbound ran directly against every piece of advice you commonly give a new indie dev:

- Don’t make your own engine (Starbound is written basically in bare C++ on top of SDL / libogg / libvorbis / libfreetype etc)
- Always make a prototype and plan to throw it away (The code that I used to demo two weeks in is the same lineage as the 1.0 release)
- Seriously, don’t make your own engine, just use Unity / Unreal / Godot etc (We didn’t even use boost! In fact, at some point we added our own versions of c++17 classes like std::optional and std::variant inside starbound, because at the time it was easier than starting to depend on boost. We also came up with our own system for doing 2d texture atlasing because we couldn’t find a good way to do offline texture atlasing with so many small textures that can’t be easily grouped)
- Your first game should be simple, you should make something simple that you can release in a short time. (In starbound, you can have an item that the player holds that runs its own Lua script which affects player physics, which can cooperate with other items and abilities that the player has that have their own scripts that also can affect the player’s physics (at the same time), while also giving the player scripted status effects which can add their own custom stats to a generic stat database that most entities have, which can have stacks of generic stat modifiers also controllable from scripts. There is also an entire crappy sort of DSL for drawing material blocks that visually connect in arbitrarily complicated ways.)

I’m definitely not trying to tell anybody to make ANYTHING like what I made as my first commercial game, don’t follow in my footsteps, I don’t think I would recommend that for anybody. However, there are actually quite a lot of indie devs who don’t use Unreal / Unity style game engines and instead of using these “framework” style game engines instead go for more “library” style where they simply use individual libraries for specific features, or use small “frameworks” that provide mostly rendering, audio, and input, and create the basic structure of the games they make themselves. (I once had a porting house call this “old school” development style, MUCH to my dismay)

- Any game made with XNA or MonoGame. XNA / MonoGame definitely provide a lot in a single package (rendering, sound, input) but they don’t dictate the architecture of the game itself. This includes Terraria and Stardew Valley, just to name two.
- Any game made by Jonathan Blow (Braid, The Witness)
- Any game made by Zach Barth (Zachtronics). He has stated that when he starts developing a game, he starts with a bare event loop (input / update / render) and goes from there. (NOTE: I couldn’t find a source for this, so I hope I’m not misrepresenting him, I think he said it in an interview that I now can’t find)
- Any game made Supergiant (Bastion, Transistor, Pyre)
- Chucklefish games that are made 100% in house: Starbound obviously, also Wargroove (made with the generic C++ engine halley, but the main dev for the engine is the main dev of Wargroove).
- LOTS more that I’m not listing, many many many 2d games are made this way, some 3d games as well. There are *dozens* of us, I tell you.

Obviously the advice I give is not as applicable if you decide to use something like Unity or Unreal, which if you do and that’s great and you’re happy, then I’m not trying to convince you to leave that. Obviously if you use one of those, Rust is probably out of the question except for very very peripheral things, but honestly I still think eventually Rust WILL find its way into these sorts commercial mainstream game engine products, it’s just a matter of time. For the moment though, the most likely people to use Rust and find this talk useful are other start-from-scratch indie types, and maybe a few AAA studios that make a lot of their own technology in house (Ready at Dawn!, EA SEED!). Also, I’m convinced there’s going to be a Unity or UE competitor that uses Rust sooner or later!

So okay, suppose you’ve decided that you’re going to make a game, you’re NOT going to be using one of the opinionated game engines that decide much of the game structure for you like Unreal or Unity, and you’ve decided to use Rust, what would such a thing look like?

How games were made in the past

So, in the past, games were mostly engineered in a “data-oriented” way out of sheer necessity. There is not much room for abstraction when your target console has 128KB of RAM (SNES). I’m going to call this the “Action Replay” era of video game development. If you aren’t aware, these are cheating devices kind of like “Game Genies” or “Game Sharks” if you know those, where you can patch a game’s RAM state every frame. The “Game Genie” and similar devices allowed you to patch the ROM of the game cartridge, but the “Action Replay” devices actually let you insert hooks into say, the VBlank handler, that overwrote actual memory values at constant locations every frame to say give you infinite lives etc.

The reason this worked is because obviously with 128KB of RAM, you’re not exactly going to have an implementation of malloc!. Every byte of storage is precious, so mostly games from this era are designed with very predictable manually managed layouts for their entire game state in memory. In the NES / SNES era, there was so little memory that generally the graphical representation of your game (tiles, sprites) and the logical representation of your game are the same, and if you kind of squint you see “entity” structures, but generally the maximum number of “things” that the game can keep track of are so few that there’s not enough room for any generality. This is sometimes also known as the “healing power of off-screen” era of game development, because the moment you go off screen, everything about that part of the game level is often forgotten (and enemies are magically healed).

Imagine if you were to try to write a game for the NES in Rust (probably REALLY hard, but not impossible? No harder than C at least, which is equally really hard. I’m pretty sure basically all games from the SNES era are written directly in assembler.), you might define a single static data structure something like this completely made up example:

```rust
pub type ProjectileType = u8;
pub type EnemyType = u8;
pub type EnemyBehavior = u8;
pub type Tile = u8;

pub struct PlayerProjectile {
    pub pos: Vector2<i16>,
    pub proj_type: ProjectileType,
    // etc...
}

pub struct Enemy {
    pub pos: Vector2<i16>,
    pub enemy_type: EnemyType,
    pub behavior: EnemyBehavior,
    // etc...
}

pub struct GameState {
    pub player_pos: Vector2<i16>,
    pub player_vel: Vector2<i8>,
    pub player_health: i8,

    // You want more than 4 projectiles on screen?  Nope, if the player fires
    // more, clear the oldest ones to make room, or wait until they hit
    // something or move off screen (like megaman).
    pub player_projectiles: [Option<PlayerProjectile>; 4],

    // All level data is stored in this constant block of memory.
    pub level_tiles: [[Tile; 64]; 64],

    // 8 enemies on screen, after this, you have to wait for enemies to
    // disappear before spawning more (kirby's adventure, this is highly abused
    // in tool-assisted speed runs)
    pub enemies: [Enemy; 8],

    // etc...
}
```

You can learn an enormous amount about how software works just by looking at its data structures in this way, even if this is a pretend example.

One thing to note here is that since every SNES game was overwhelmingly likely to be made in assembler, there almost certainly was no such thing as “data hiding”, the entire state of the game is available at any point to the update loop.

This era where every memory location was precious and objects were carefully laid out in it actually lasted a huge amount of time, at least up to the N64. Let’s look at another example which I’m moderately familiar with, and is still solidly in this era, but whose design is imo much closer to modern game engines. One of my all-time favorite games in the world: Mario 64. Mario 64 is interesting because it’s 3D, and except for the low-poly simple graphics, is it really THAT much different from a modern 3D platformer? Yet, it’s still solidly in the “Action Replay” era of gaming, when you get into a “game world” (you can run and jump around as Mario) you can reliably tell (more or less) where all the level data is going to live in the N64’s expansive, luxurious 4 entire MB of ram. There’s a predictable block of memory for the level geometry, for lots of various level flags, etc, but most of the dynamic content of the game is in the form of generic “entities” (or “objects”), much like modern game engines. In Mario 64, the entity structures are all exactly 608 bytes long, and there is a hard limit to 240 of them (sometimes this is a bit less, like in Bowser in the Fire Sea, there is a limit of 232 objects).

We don’t actually know what language Mario 64 was programmed in, but it was *likely* C, so this is probably in the era where, if you had a time machine and a burning desire for a better systems language from the future, you could just as easily have used Rust to make a commercial Nintendo 64 game. Let’s take a lot of artistic liberty and imagine what the structure of such a game would look like translated as directly as possible to Rust:

```
pub struct EntityAnimation = u8;
pub struct EntityBehavior = u8;

pub struct Entity {
    // Is this entity initialized and active or is it dead and can be
    // overwritten.
    pub initialized: bool,

    // Okay, this is really interesting.  In Mario 64, all entities are divided
    // into "important" and "unimportant" entities.  Important entities are
    // things like mario himself, coins, enemies etc.  Unimportant entities are
    // things like wind effects or the stars that come out of mario's butt
    // stomp.  When the game runs out of entity slots, it will remove
    // unimportant entities to make room for important entities.  If the game
    // tries to create more than 240 important entities (or 232 in BitFS), it
    // will hang.
    pub important: bool,

    pub position: Vector3<f32>,
    pub rotation: Vector3<f32>,

    // I think most things like this are pointers in real Mario 64, but they
    // could just as easily be indexes.  This turns out to be kind of important.
    pub animation: EntityAnimation,
    pub behavior: EntityBehavior,

    pub visible: bool,
    pub damage_mario_on_touch: bool,
    pub home_in_on_mario: bool,

    // Generic timer, used for lots of animation and gameplay purposes
    pub timer: u16,
    // Generic action, used differently for different entity types
    pub action: u8,
    // Lots more stuff...
}

pub type EntityIndex = u8;
pub struct WorldRenderGeometry;
pub struct WorldCollisions;

pub struct GameState {
    pub world_render_geometry: WorldRenderGeometry,
    pub world_collisions: WorldCollisions,

    // Seriously, not ever more than 240
    pub entities: [Entity; 240],

    // Instead of a pointer, we store an index into the entities array.
    pub mario_entity: EntityIndex,

    // Lots more stuff...
}
```

So, the specific details of these representations aren't really super important, the important takeaways here are that:

- The representation of the game state (though it's dramatically over-simplified here) is still very *simple* even in the real game. It's very predictable and amenable to RAM value poking, and there isn't really much in the way of "abstraction" here, everything is quite literal and built for purpose.
- We can't know for sure about the languages used for all the games in the N64 era, but we know at least for some titles that it was definitely C (Shadows of the Empire), and probably it was C in the vast majority of cases. The game structures all tended to be simple and predictable like Mario 64, you can imagine the game state being represented mostly as global C structs or arrays of structs.
- I'm speculating here, but *probably* there was not much "data hiding", you can imagine the structure of these games more or less as a giant static global struct containing "all of the game state", visible to all of the game's code.
- There are occasional pointers to functions, but there doesn't seem to be a whole lot of OO going on. There don't seem to be vtables, or even many things *like* vtables, or anything like that. This is potentially wild speculation and is definitely just a non-expert opinion, but from my "extensive" research into the memory layout and design of SNES - N64 era titles, it usually feels like

> you can kind of imagine the very basic, boring, made-for-purpose C code that the engine is made of just from looking at the data format.

Okay, so imagine you were to write Mario 64 very similarly to how it appears to originally be designed, but in Rust. You would effectively be writing a giant, procedural, single purpose game engine in a much much nicer C, but still at its heart, pretty much like C. Effectively your game engine is something like (DRAMATIC OVERSIMPLIFICATION INCOMING):

```rust
pub struct EntityAnimation = u8;
pub struct EntityBehavior = u8;

pub struct Entity {
    // ... see above
}

pub type EntityIndex = u8;
pub struct WorldRenderGeometry;
pub struct WorldCollisions;

pub struct GameState {
    // ... see above
}

fn main() {
    // ALL of the game state, morally a global.
    let mut game_state = GameState { ... };

    loop {
        // Every time around this loop is 1 frame, which is 16ms for a 60fps game.

        // We capture the entire input in one go, once a frame.  There is not
        // really even a need for anything complex like input events, because in
        // something like Mario 64, all that's really happening to read the state
        // of controllers is reading a specific memory region for the controller
        // state.  Here, we're just doing this at the beginning of the frame, in
        // real Mario 64 I believe happens once per frame for different systems
        // but distributed over several places.
        let input_state = capture_input_state();

        // Let's have a series of functions change the state of the game_state
        // based on the previous game state and the input.  We'll be very very
        // fancy, and we'll give this very SIMPLE pattern an overly fancy name,
        // and we'll call each of these functions a "system".

        // Set state flags inside mario to start jumping, or maybe set a state
        // flag for whether you're paused.
        input_system(&mut game_state, &input_state);
        // pos += vel * dt;  Apply gravity.  Do collision detection and response
        // for all entities.
        physics_system(&mut game_state);

        // Run entity logic for every entity, this might farm out into
        // update_mario, update_baddies, update_platforms, etc.
        entity_logic_system(&mut game_state);

        // ... lots more systems

        // Render the current game state.  In the N64 era this was in some ways
        // vastly simpler than now, but even then it was probably still pretty
        // messy and stateful, so maybe our game state includes the state of
        // loaded graphics resources as well.
        render_system(&mut game);

        // Read state flags of entities and trigger new audio, this is messy
        // and stateful just like rendering.
        audio_system(&mut game);

        // Wait on VBlank
        wait_vblank();

        // repeat.
    }
}
```

So I'm NOT recommending that you make games this way! However, even if you think this is 100% a ridiculous over-simplification, or you think that even if you could make a large complex game this way, it would turn into a procedural ball of mud, let's look at this for a while and see if there are any advantages before we totally write this off. Clearly there is a huge, gaping, obvious disadvantage in that the entire game state is visible and mutable to all systems and if something changes unexpectedly, you might have no idea where to look to find what is changing the game state improperly. Nobody is *really* suggesting in the modern era that you write software with all state as effectively global. HOWEVER there is one advantage here, which is that with a little bit of care, you could write a game this way in 100% safe rust and almost definitely not run into issues with the borrow checker. You might not be able to use *pointers* safely since *certainly* if you used pointers your entire game state would have internal pointers to itself, so the one modification from Mario 64 style that we need to make is that everywhere where there would have been a pointer, instead store an index into some array. The canonical example of this is the "mario_entity" field on our GameState. This may seem overly limiting, but with a bit of thought it becomes clear especially if you had constant limits on things (you don't even have malloc, remember!), you can always add something like `all_the_textures: [TextureDescriptor; 256]`, and use integer indexes into this static array to describe textures instead of things like pointers. The other reason this is helpful w.r.t. the borrow checker is that Rust is rather good at letting you split borrows for different fields of a public struct, so since everything in our pretend-engine is just a giant nested tree of entirely public structs, you should always be able to read the state of the game and change exactly as much of the game as necessary in each system. Remember, most of the games in the "Action Replay" era are written like this, there is not usually any malloc-ing, there is at most something very limited like "allocating" only from a fixed array, and most everything is more or less carefully laid out in a static struct somewhere.

One more criticism of this style is that you might say that using indexes instead of pointers is "safe" in the strict sense, but possibly only technically, you're trading UB and potential crashes with pointers for "random but unspecified" behavior if you access the wrong or outdated index, and potentially panics. You'd be right, btw! We'll visit this some more later, but just accept for now that it's possible and safe and this design doesn't run directly against the borrow checker.

Okay, so now I've hopefully described an admittedly over-simplified way that you COULD write a game in Rust that morally sort of closely mirrors the oldest game architectures. I'm not advocating this for your next game or anything, but there ARE games written now that are little more than this, written in direct, single purpose, procedural style. I'm not going to call out specific games, but I've seen a bunch of games written in a style that is little more than this, often wrapped in a paper thin veneer of OO. I've seen game source code that has a single **12k** line world generation function. This is not an insult to such games, even marginally, the people who do this are often absolute geniuses who are just very good at knowing what to care about and what not to care about. There is real wisdom here!

But still, I'm genuinely not advocating this. Let's refer to this as the "UR-game-architecture", and we'll come back to it. This is approaching the "data-driven! use an ECS!" answer from the bottom up, let's digress and approach this now from the top down.

Wayyyyyyy too much object orientation

Now that we've covered sort of the simplest possible messy procedural C-ish game engine design, let's try to apply the principles of OO design and see if this is an improvement. I'm going to use Starbound or some simplification of Starbound as an example here, because I'm lucky enough to know a lot about it and have its source on hand, so I'm not going to be lying *too* much when I describe it.

What are the principles of OO? Not everybody entirely agrees with what OO is, but I'm including some basic hopefully non-controversial points:

- Single responsibility principle - Objects should have a single logical set of responsibilities, and methods should perform one operation inside that set of responsibilities.
- Encapsulation - You should bind data together with the functions that operate on it, keeping both safe from outside interference and misuse. This allows for refactoring by changing the internal representation of a class without changing its behavior.
- Abstraction, or "Liskov Substitution Principle", or similar - You should be able to substitute one derived class for another, as long as they share the same base and are used through that base class (or interface, or whatever).
- Interface segregation, or principle of least coupling, etc - The dependency on one class to another should use the smallest possible interface.

On a practical level, OO languages generally have a few important features to support OO design, namely object methods, private object data, inheritance, virtual methods, etc. I'm mostly going to be talking about C++ because it has a lot of OO footguns, it's what I know, and it's a super popular language for the "I'm gonna make my own engine" gamedev crowd.

We're going to try these principles out and see how they can be misused for game development, and then talk about how even though they often fail for gamedev (and this is now a "well known" state of affairs), they fail *way more spectacularly* with Rust. It *is* possible to make a game with at least some of these principles, though, and they're not *all* actually universally terrible ideas. I'm not really a proponent of OO *at all*, but there are at least a few good ideas that are sort of related to OO or kind of came out of OO (all of which Rust has and can perform admirably).

- The dot operator or "postfix functions". If you're a haskeller this is "type directed name resolution" as opposed to the normal state of affairs which is

  “name directed type resolution”. A “powerful” way to avoid having to scope 100 different functions with very short common names, or have C style prefixes for everything. Great for IDEs too!
- Interfaces with laws (contracts) - Rust traits! No inheritance in sight, and they’re best when they’re small and come with meaningful rules around them, but if you squint they’re kind of C++ pure virtual classes. They’re *awesome* in the same way Haskell typeclasses are awesome.
- Data hiding - Being able to hide data to maintain invariants is invaluable, making a safe interface to unsafe code would not be possible without it. Very useful in the small, and with “library” code.

So just for the record I’m not trying to pick on the above good parts, these are fine, great even. With that out of the way, lets look at ways the rest of OO design can fail in gamedev.

So, superficially games seem well suited to OO because when trying to come up with OO designs, there are obvious “objects” that jump out at you. Using starbound as our example, things like “Player”, “NPC”, “Monster” are pretty easily understood concepts and obvious candidates for objects in our game, so lets start with these. We’ll also include a “World” class which, like the Mario 64 example, is the basic structure of some live play arena (we’re purposefully skipping things like interfaces, menus, etc and focusing on the core part of a game engine). I’ll also have to switch to C++ because some of this is going to be difficult to express in Rust.

```
typedef uint32_t EntityId;

enum class HumanoidAnimationState { ... };
class HumanoidItem { ... };

struct Player {
    Vec2F position;
    Vec2F velocity;
    float mass;

    HumanoidAnimationState animation_state;

    HumanoidItem left_hand_item;
    HumanoidItem right_hand_item;

    Vec2F aim_position;

    float health;
    EntityId focused_entity;
    float food_level;
    bool admin;

    // So, so much more...
};

enum class MonsterAnimationState { ... };
struct DamageRegion { ... };

struct Monster {
    Vec2F position;
    Vec2F velocity;
    float mass;

    MonsterAnimationState animation_state;

    float health;
    EntityId current_target;
    DamageRegion damage_region;

    ...
};

struct Npc {
    Vec2F position;
    Vec2F velocity;
    float mass;

    HumanoidAnimationState animation_state;

    HumanoidItem left_hand_item;
    HumanoidItem right_hand_item;

    float health;
    Vec2F aim_position;

    ...
};

struct WorldTile { ... };

struct World {
    List<EntityId> player_ids;
    // Hmm, we're probably going to need an interface and downcasting here?
    HashMap<EntityId, void*> entities;

    MultiArray2D<WorldTile> tiles;

    // So, so much more...
};
```

Right away, we see repeated structure in our data types, and probably these should be sub-objects with their own methods, so let’s sketch that out a bit:

```
typedef uint32_t EntityId;

enum class HumanoidAnimationState { ... };
class HumanoidItem { ... };

struct Physics {
    Vec2F position;
    Vec2F velocity;
    float mass;
};

struct HumanoidState {
    HumanoidAnimationState animation_state;
    HumanoidItem left_hand_item;
    HumanoidItem right_hand_item;
    Vec2F aim_position;
};

struct Player {
    Physics physics;
    HumanoidState humanoid;

    float health;
    EntityId focused_entity;
    float food_level;
    bool admin;

    ...
};

enum class MonsterAnimationState { ... };
struct DamageRegion { ... };

struct Monster {
    Physics physics;

    MonsterAnimationState animation_state;

    float health;
    EntityId current_target;
    DamageRegion damage_region;

    ...
};

struct Npc {
    Physics physics;
    HumanoidState humanoid;

    float health;

    ...
};

struct WorldTile { ... };

struct World {
    List<EntityId> player_ids;
    HashMap<EntityId, void*> entities;

    MultiArray2D<WorldTile> tiles;

    ...
};
```

So far this isn’t going too bad, but this is still just describing the data structures in our game. Remember, one of the OO principles we talked about is encapsulation, you want to expose a minimal interface to each of these structs (we really should call them classes!) and methods that only expose what is necessary. Also, there’s that void pointer in entities inside world, we probably shouldn’t have that, we’re going to need to keep our entities stored somehow so let’s make an Entity interface while we’re at it for things that are common to all entities.

```
typedef uint32_t EntityId;

// Forward declare World to pass it to Entity
struct World;

struct InputState { ... };
struct RenderState { ... };

// Pure virtual interface!
class Entity {
public:
    // Okay, *definitely* all entities will have a position, and it's probably
    // fine for this to be const.
    virtual Vec2F position() const = 0;

    // Do all entities have a velocity?  Probably not actually, there are
    // probably going to be stationary entities, so let's skip velocity.  Any
    // other common fields?  Honestly probably there are a few that deserve to
    // go here, but there can't be *too* many without immediately thinking of
    // cases where an entity doesn't have them.  Maybe they all return Maybe or
    // something, but that doesn't sound very useful as an interface?  We'll
    // just make more interfaces for those and stick to OO design!
    // So, we're going to need to update each entity somehow and probably render
    // it, so let's define a few methods for this.

    // Well, we definitely have to pass the entity's world to its update method,
    // because for example a Player has to be able to do things like spawn
    // projectile entities, and so do monsters and NPCs probably.  Also,
    // monsters have to know where the player is to attack them!
    void update(World* world) = 0;

    // Let's assume that these aren't toooo stateful and messy and each entity
    // can just "render themselves" in some reasonable way.
    void input(InputState const& input_state) = 0;
    void render(RenderState& render_state) = 0;

private:
    // No private data!  We definitely know that we should favor composition
    // over inheritance because we've been told this before, so only pure
    // virtual interfaces for us!  This is one facet of OO design that's
    // *probably* basically dead now?  I don't actually know for sure, but I
    // know this has been talked about to death so we don't have to beat THIS
    // dead of a horse.
};

class Player : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

private:
    Physics m_physics;
    HumanoidState m_humanoid;

    ...
};

class Monster : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

private:
    Physics m_physics;

    ...
};
```

```
};

class NPC : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

private:
    Physics m_physics;
    HumanoidState m_humanoid;

    ...
};

struct WorldTile { ... };

struct World {
    List<EntityId> player_ids;
    HashMap<EntityId, shared_ptr<Entity>> entities;

    MultiArray2D<WorldTile> tiles;

    ...
};
```

So one point before we go on, why am I bothering with this `EntityId` stuff here? This is C++, why can't we use pointers? Well, it turns out that doing this is *very* unsafe, so the pattern that all game engines I've ever (in languages without a fancy garbage collector) seen adopt is actually to have some kind of map from some form of "entity id" to an actual entity pointer and keep that in a single location. The reason for this is that, say in C++, let's say every entity keeps a shared_ptr or some kind of downcasted shared_ptr. The problem then becomes that World is now a giant ball of reference-cycles and entities may never be destructed, and this is a huge problem. On the other side of this, if they kept raw pointers, they would be continually invalidated and this tends to lead to hard to solve ephemeral bugs and so more or less nobody does this (least, nobody keeps raw pointers / references around for "very long"). You *could* use for example weak_ptr, and this is sometimes used, but there tend to be other reasons to use ids because they're useful for networking and can more easily be saved and loaded from disk. This is interesting, because this is the major change that we had to make in our "UR-game-architecture" to make it compatible with Rust, but it's very *very* common in game engines in general!

(In game engines, EntityId is usually something like an int that counts up, possibly a uint64_t that doesn't round or a uint32_t that does round, and this is done so that an EntityId is never or very rarely re-used. This way, if an entity goes away, generally anything monitoring it will notice it gone rather than some other entity immediately taking its place. There's another pattern here called "generational indexes" which is important and I'll talk about later)

Okay so this seems like it… could actually work to make a complex game with? I can sort of imagine a *very* simple game you could make with these classes and interfaces as they are right now, but let's flesh it out a bit to see what sort of problems we'll run into.

Let's think about what a monster needs to track a player. They might go through all the entities in the world, filter by only the players, sort by distance (you'd use some kind of spatial hash / kd-tree in reality), and then track the closest one. That actually seems basically possible with this as it is! Okay, new requirement: monsters should track players with the lowest health first. Uh-oh, okay player health is private, so we better make a public accessor for this:

```
class Player : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

    float health() const;

private:
    ...
};
```

We *could* have just made the health public, but OO design! What if there's an invariant being held, where if the health drops below some value, some animation state is triggered? What about damage? Certainly the single responsibility principle states that a *monster* shouldn't be in charge of setting a *player's* health.

Okay, new requirement: Monsters should not go after players marked as "admin". Alright, add an accessor!

```
class Player : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

    float health() const;
    bool is_admin() const;

private:
    ...
};
```

Okay, we've made a little prototype game but no damage is happening yet, so let's go ahead and get damage set up. Hmm, where should the damage system for players go? Probably a player should reduce their health since it's *their* health… or maybe monsters could do it since it's *their* damage region? I dunno, I'll say it's the player's job. I guess that means we need accessors on the monsters for their damage region:

```
class Monster : Entity {
public:
    Vec2F position() const override;

    void input(InputState const& input_state) override;
    void update(World* world) override;
    void render(RenderState& render_state) override;

    DamageRegion const& damage_region() const;

private:
    ...
};
```

This is turning into a lot of accessors. We *could* just start making lots of things public, but what do OO principles state? That we should expose as little and finite an interface as possible so that we can change our implementation without refactoring. Maybe this won't get *so* bad.

Okay, new requirements: A certain monster will only aggro the player when the player touches the ground near them. Okay, this is going to require yet more accessors, except because we've been following good OO principles about code re-use, we have that internal member m_physics of type `Physics`, and it is the only thing that can know whether an entity is touching the ground. So, I guess Physics needs an accessor to *its* internal state `Physics::onGround` And then the *Player* needs an accessor `Player::onGround` which in turn will use `Physics::onGround`. Okay, this is turning into LOTS of accessors.

It feels like every time a new requirement comes in, you have to take what may have once been sane interfaces and "poke more holes through them". 8 months later, most of a semi-game has been built this way and there are LOTS of these holes poked through. Lots of code has no clear place to be because it concerns multiple entities at once, and a lot of logically similar functionality is split across several files. "The problem with OO is that everything happens somewhere else".

A new requirement comes in: I have an idea for a special kind of item that when the player holds it and its near a specific kind of enemy, it will glow. The enemies that trigger this should be scared of the item and back away, but have a special animation where they're mesmerized by the glowing item. This should only work for players that have achieved some specific quest goal.

You throw up your hands in frustration, this would require 4 separate modules to know about all of the other's internals (Player, Items, Monsters, Animation). You add a lot more accessors and special interfaces. Things are a mess.

So, these are made up examples, but they are not much different than the real things that I went through a *lot*. This is why it is now considered common knowledge now that for games, OO *mostly* just gets in the way. Data hiding has very limited utility for a game outside of just maintaining invariants at the edges of the code, where things are smaller and more contained. A lot, possibly even the vast majority of the interesting behavior in your game ends up spanning many data types, and does not naturally "belong" to any specific entity. Lots of entity types are 80% or 60% similar to others and its hard to re-use code, and the more modules we add inside our entities to help reuse code, the more layers are added. What have we gained vs our UR-architecture? That not everything is visible to all our functions, even though we keep adding more and more and more accessors so it sort of is? Maybe some marginally better code organization, but also sometimes *worse* code organization?

Okay, so obviously this has a lot of downsides even in C++, and obviously you know where we're eventually going to end up because I told you at the beginning of the talk (Just use ECS!). You CAN make a game this way, but you'll have a lot of problems and poke a LOT of holes and probably end up with some mega-objects. Just for a nice visceral reaction, let's look at the actual, *for-real* Player class inside Starbound as of the current version:

```
class Player :
  public virtual ToolUserEntity,
  public virtual LoungingEntity,
  public virtual ChattyEntity,
  public virtual DamageBarEntity,
  public virtual PortraitEntity,
  public virtual NametagEntity,
  public virtual PhysicsEntity,
  public virtual EmoteEntity {

public:
  Player(PlayerConfigPtr config, Uuid uuid = Uuid());
  Player(PlayerConfigPtr config, Json const& diskStore);
  Player(PlayerConfigPtr config, ByteArray const& netStore);

  ClientContextPtr clientContext() const;
  void setClientContext(ClientContextPtr clientContext);

  StatisticsPtr statistics() const;
  void setStatistics(StatisticsPtr statistics);

  QuestManagerPtr questManager() const;

  Json diskStore();
  ByteArray netStore();

  EntityType entityType() const override;

  void init(World* world, EntityId entityId, EntityMode mode) override;
  void uninit() override;

  Vec2F position() const override;
  Vec2F velocity() const override;

  Vec2F mouthPosition() const override;
  Vec2F mouthOffset() const;
  Vec2F feetOffset() const;
  Vec2F headArmorOffset() const;
  Vec2F chestArmorOffset() const;
  Vec2F legsArmorOffset() const;
  Vec2F backArmorOffset() const;

  // relative to current position
  RectF metaBoundBox() const override;
```

```
// relative to current position
RectF collisionArea() const override;

pair<ByteArray, uint64_t> writeNetState(uint64_t fromStep = 0) override;
void readNetState(ByteArray data, float interpolationStep = 0.0f) override;

void enableInterpolation(float extrapolationHint = 0.0f) override;
void disableInterpolation() override;

virtual Maybe<HitType> queryHit(DamageSource const& source) const override;
Maybe<PolyF> hitPoly() const override;

List<DamageNotification> applyDamage(DamageRequest const& damage) override;
List<DamageNotification> selfDamageNotifications() override;

void hitOther(EntityId targetEntityId, DamageRequest const& damageRequest) override;
void damagedOther(DamageNotification const& damage) override;

List<DamageSource> damageSources() const override;

bool shouldDestroy() const override;
void destroy(RenderCallback* renderCallback) override;

Maybe<EntityAnchorState> loungingIn() const override;
bool lounge(EntityId loungeableEntityId, size_t anchorIndex);
void stopLounging();

void revive(Vec2F const& footPosition);

List<Drawable> portrait(PortraitMode mode) const override;
bool underwater() const;

void setShifting(bool shifting);
void special(int specialKey);

void moveLeft();
void moveRight();
void moveUp();
void moveDown();
void jump();

void dropItem();

float toolRadius() const;
float interactRadius() const override;
List<InteractAction> pullInteractActions();

uint64_t currency(String const& currencyType) const;

float health() const override;
float maxHealth() const override;
DamageBarType damageBar() const override;
```

```cpp
float healthPercentage() const;

float energy() const override;
float maxEnergy() const;
float energyPercentage() const;

float energyRegenBlockPercent() const;

bool energyLocked() const override;
bool fullEnergy() const override;
bool consumeEnergy(float energy) override;

float foodPercentage() const;

float breath() const;
float maxBreath() const;

float protection() const;

bool forceNude() const;

String description() const override;

List<LightSource> lightSources() const override;

Direction walkingDirection() const override;
Direction facingDirection() const override;

Maybe<Json> receiveMessage(ConnectionId sendingConnection, String const& message, Jso

void update(uint64_t currentStep) override;

void render(RenderCallback* renderCallback) override;

PlayerInventoryPtr inventory() const;
// Returns the number of items from this stack that could be
// picked up from the world, using inventory tab filtering
size_t itemsCanHold(ItemPtr const& items) const;
// Adds items to the inventory, returning the overflow.
// The items parameter is invalid after use.
ItemPtr pickupItems(ItemPtr const& items);
// Pick up all of the given items as possible, dropping the overflow.
// The item parameter is invalid after use.
void giveItem(ItemPtr const& item);

void triggerPickupEvents(ItemPtr const& item);

bool hasItem(ItemDescriptor const& descriptor, bool exactMatch = false) const;
size_t hasCountOfItem(ItemDescriptor const& descriptor, bool exactMatch = false) cons
// altough multiple entries may match, they might have different
// serializations
ItemDescriptor takeItem(ItemDescriptor const& descriptor, bool consumePartial = false
void giveItem(ItemDescriptor const& descriptor);

// Clear the item swap slot.
void clearSwap();

// Refresh worn equipment from the inventory
void refreshEquipment();

PlayerBlueprintsPtr blueprints() const;
bool addBlueprint(ItemDescriptor const& descriptor, bool showFailure = false);
bool blueprintKnown(ItemDescriptor const& descriptor) const;

bool addCollectable(String const& collectionName, String const& collectableName);

PlayerUniverseMapPtr universeMap() const;

PlayerCodexesPtr codexes() const;

PlayerTechPtr techs() const;
void overrideTech(Maybe<StringList> const& techModules);
bool techOverridden() const;

PlayerCompanionsPtr companions() const;

PlayerLogPtr log() const;

InteractiveEntityPtr bestInteractionEntity(bool includeNearby);
void interactWithEntity(InteractiveEntityPtr entity);

// Aim this player's target at the given world position.
void aim(Vec2F const& position);
Vec2F aimPosition() const override;

Vec2F armPosition(ToolHand hand, Direction facingDirection, float armAngle, Vec2F off
Vec2F handOffset(ToolHand hand, Direction facingDirection) const override;

Vec2F handPosition(ToolHand hand, Vec2F const& handOffset = {}) const override;
ItemPtr handItem(ToolHand hand) const override;

Vec2F armAdjustment() const override;

void setCameraFocusEntity(Maybe<EntityId> const& cameraFocusEntity) override;

void playEmote(HumanoidEmote emote) override;

bool canUseTool() const;

// "Fires" whatever is in the primary (left) item slot, or the primary fire
// of the 2H item, at whatever the current aim position is.  Will auto-repeat
// depending on the item auto repeat setting.
void beginPrimaryFire();
// "Fires" whatever is in the alternate (right) item slot, or the alt fire of
// the 2H item, at whatever the current aim position is.  Will auto-repeat
// depending on the item auto repeat setting.
void beginAltFire();

void endPrimaryFire();
void endAltFire();

// Triggered whenever the use key is pressed
void beginTrigger();
void endTrigger();

ItemPtr primaryHandItem() const;
ItemPtr altHandItem() const;

Uuid uuid() const;

PlayerMode modeType() const;
void setModeType(PlayerMode mode);
PlayerModeConfig modeConfig() const;

ShipUpgrades shipUpgrades();
void setShipUpgrades(ShipUpgrades shipUpgrades);

String name() const override;
void setName(String const& name);

Maybe<String> statusText() const override;
bool displayNametag() const override;
Vec3B nametagColor() const override;

void setBodyDirectives(String const& directives);
void setHairType(String const& group, String const& type);
void setHairDirectives(String const& directives);
void setEmoteDirectives(String const& directives);
void setFacialHair(String const& group, String const& type, String const& directives)
void setFacialMask(String const& group, String const& type, String const& directives)

String species() const override;
void setSpecies(String const& species);
Gender gender() const;
void setGender(Gender const& gender);
void setPersonality(Personality const& personality);

void setAdmin(bool isAdmin);
bool isAdmin() const override;

bool inToolRange() const override;
bool inToolRange(Vec2F const& aimPos) const override;
bool inInteractionRange() const;
bool inInteractionRange(Vec2F aimPos) const;

void addParticles(List<Particle> const& particles) override;
```

```cpp
void addSound(String const& sound, float volume = 1.0f) override;

bool wireToolInUse() const;
void setWireConnector(WireConnector* wireConnector) const;

void addEphemeralStatusEffects(List<EphemeralStatusEffect> const& statusEffects) over
ActiveUniqueStatusEffectSummary activeUniqueStatusEffectSummary() const override;

float powerMultiplier() const override;

bool isDead() const;
void kill();

void setFavoriteColor(Vec4B color);
Vec4B favoriteColor() const override;

// Starts the teleport animation sequence, locking player movement and
// preventing some update code
void teleportOut(String const& animationType = "default", bool deploy = false);
void teleportIn();
void teleportAbort();

bool isTeleporting() const;
bool isTeleportingOut() const;
bool canDeploy();
void deployAbort(String const& animationType = "default");
bool isDeploying() const;
bool isDeployed() const;

void setBusyState(PlayerBusyState busyState);

// A hard move to a specified location
void moveTo(Vec2F const& footPosition);

List<String> pullQueuedMessages();
List<ItemPtr> pullQueuedItemDrops();

void queueUIMessage(String const& message) override;
void queueItemPickupMessage(ItemPtr const& item);

void addChatMessage(String const& message);
void addEmote(HumanoidEmote const& emote);

List<ChatAction> pullPendingChatActions() override;

float beamGunRadius() const override;

bool instrumentPlaying() override;
void instrumentEquipped(String const& instrumentKind) override;
void interact(InteractAction const& action) override;
void addEffectEmitters(StringSet const& emitters) override;
void requestEmote(String const& emote) override;
```

```
ActorMovementController* movementController() override;
StatusController* statusController() override;

List<PhysicsForceRegion> forceRegions() const override;

SongbookPtr songbook() const;

void finalizeCreation();

float timeSinceLastGaveDamage() const;
EntityId lastDamagedTarget() const;

bool invisible() const;

void animatePortrait();

bool isOutside();

void dropSelectedItems(function<bool(ItemPtr)> filter);
void dropEverything();

bool isPermaDead() const;

bool interruptRadioMessage();
Maybe<RadioMessage> pullPendingRadioMessage();
void queueRadioMessage(Json const& messageConfig, float delay = 0);
void queueRadioMessage(RadioMessage message);

// If a cinematic should play, returns it and clears it.  May stop cinematics
// by returning a null Json.
Maybe<Json> pullPendingCinematic();
void setPendingCinematic(Json const& cinematic, bool unique = false);

void setInCinematic(bool inCinematic);

Maybe<pair<Maybe<StringList>, float>> pullPendingAltMusic();

Maybe<PlayerWarpRequest> pullPendingWarp();
void setPendingWarp(String const& action, Maybe<String> const& animation = {}, bool d

Maybe<pair<Json, RpcPromiseKeeper<Json>>> pullPendingConfirmation();
void queueConfirmation(Json const& dialogConfig, RpcPromiseKeeper<Json> const& result

AiState const& aiState() const;
AiState& aiState();

// In inspection mode, scannable, scanned, and interesting objects will be
// rendered with special highlighting.
bool inspecting() const;

// Will return the highlight effect to give an inspectable entity when inspecting
```

```
  EntityHighlightEffect inspectionHighlight(InspectableEntityPtr const& inspectableEnti

  Vec2F cameraPosition();

  using Entity::setTeam;

private:
  // ...
};
```

Interesting fact, there are many more methods than there are fields in Player, somehow. If you handed me a game and asked me to add a feature to the engine, and asked me whether it would be easier to add in the “too much OO architecture” vs “UR-architecture”, I’d say it’s easier in the “UR-architecture” almost *every* time. BUT it IS doable with effort obviously, because lots of Starbound is like this :(.

So, this has downsides in C++, but not every game has as many crazy one-off features as Starbound. Maybe this is just an extreme example, and usually this is not quite as bad of a problem? Let’s see what happens if you try this in Rust! *Right off the bat*, things start to become *hard*. Let’s go back to the simplest OO version in C++:

```
// typedefs...

class Entity {
public:
    virtual Vec2F position() const = 0;

    void input(InputState const& input_state) = 0;
    void update(World* world) = 0;
    void render(RenderState& render_state) = 0;
};

// entity definitions...

struct World {
    List<EntityId> player_ids;
    HashMap<EntityId, shared_ptr<Entity>> entities;
    ...
};
```

Partially translating into Rust:

```
pub trait Entity {
    fn position(&self) -> Vec2F;

    fn input(&mut self, input_state: &InputState);
    fn update(&mut self, world: &mut World);
    fn render(&mut self, render_state: &mut RenderState);
}

pub struct World {
    player_ids: Vec<EntityId>,
    entities: HashMap<EntityId, Rc<Entity>>,
    ...
}
```

Even in this tiny example you can already see this is going to be a huge pain. First of all, World owns each Entity, but each Entity has a set of mutable methods, one of which must take a mutable reference to World. That won't work, because you will mutably borrow an Entity, and then have to mutably borrow it *again* to pass in a World reference (which would presumably contain the self Entity). In order to do this, probably *every* Entity implementation would need internal mutation because otherwise they would have no way to take a World reference, and also World would probably need internal mutation because it too would need to be passed by immutable reference.

```
pub trait Entity {
    fn position(&self) -> Vec2F;

    fn input(&self, input_state: &InputState);
    fn update(&self, world: &World);
    fn render(&self, render_state: &mut RenderState);
}

pub struct World {
    player_ids: RefCell<Vec<EntityId>>,
    entities: RefCell<HashMap<EntityId, Rc<Entity>>>,
    ...
}
```

Okay, now we won't have borrow errors during entity updates, but now everything has to be inside a RefCell? Maybe we can place our entire state inside a RefCell and it's not so hard? We've discussed the tendency for complex behaviors to be very cross-cutting, so what if we have the following situation:

A monster damages a player, this triggers the player to play an audio as a hurt sound effect, but for gameplay reasons this *also* triggers a logical sound that other creatures can react to. Maybe there is a swarming behavior for some particular monster type? So, hurting a player triggers an external mutation (audio), internal mutation (health), which then maybe signals monsters which have their own internal mutations (target entity). Maybe this all was started from a Monster, so the control flow goes from Monster to Player and *back into* Monster. If you're debugging Monster, you now have triggered *spooky action at a distance* by indirectly triggering mutation inside your own struct method. Except, rust doesn't allow this, (and this wouldn't even type check probably without RefCell), so in rust the version of this is that you simply get a RefCell panic.

Let's keep going, say entities have some kind of tag or dynamic set of tags, and let's add that to our C++ Entity interface:

```
class Entity {
public:
    virtual Vec2F position() const = 0;
    virtual List<Tag> tags() const = 0;
};
```

Say you profile your game and simply returning a copy of `List<Tag>` is eating up huge CPU time, so you change the interface to this:

```
class Entity {
public:
    virtual Vec2F position() const = 0;
    virtual List<Tag> const& tags() const = 0;
};
```

This is assuming such a change is possible and not a use after free bug waiting to happen, which it often is. Well, Rust is supposed to save us from use after free, so what happens if you translate this to the Rust version:

```
pub trait Entity {
    fn position(&self) -> Vec2F;
    fn tags<'a>(&'a self) -> &'a Vec<Tag>;
}
```

I've made the lifetimes un-elided to make it obvious, but what this says is that this method returns a single Vec reference that borrows *the entire entity*. That's.. helpful but also not at all helpful if you want to later call any other method that might mutate. If this has internal mutation and is inside a RefCell, this is also impossible and would have to instead return `std::cell::Ref`. Everything is hard, much harder than it is in C++. You go onto IRC to ask for help, you get the well meaning but potentially unhelpful answer: "You're just still in the phase where you're fighting the borrow checker."

It can get even worse. Entity is pretty sparse, even the real `Entity` inside Starbound, but say you apply these same principles to `World`:

```
pub struct World { ... }

impl World {
    fn tile<'a>(&'a self, index: Vector2<i32>) -> &'a WorldTile { ... }

    // HUGE number of additional members...
}
```

If you have a big enough `World` structure, even in the absence of things like pure virtual interfaces (or in Rust, traits), you can still run into problems. Should accessing one tile borrow the *entire world*? Things would be much easier if you could somehow borrow only *part* of the world and thus allow you to mutate some other part, and that's exactly what you could do were all the fields of World simply public. Rust is quite good at splitting borrows of a struct, but by design, it cannot do that for such an opaque method. This would be vastly easier if simply World was a struct with plain public members. This becomes MORE important the larger the structure is, and data hiding becomes less important the more into the "application" level you are. With games this becomes worse and worse, because your games become more and more complicated over time as you add features. We're not trying to write an engine, we're just trying to write a simple game directly in Rust, all of that data has to go somewhere!

If you're writing a method like this, it's more obvious that you can make fields of a large compound structure public and split borrows will help, but imagine that you didn't design things this way. Imagine instead that you have *two* kinds of Worlds, one for the Server and one for the Client. This is precisely how Starbound works, and there is a large complex interface for a “world” that is common between the client and server.

```
pub trait World {
    fn tile<'a>(&'a self, index: Vector2<i32>) -> &'a WorldTile;

    // HUGE number of additional trait methods...
}
```

Now it’s *impossible* to “just allow the fields to be public” regardless of “OO” principles or not, just like Entity. More “fighting the borrow checker”. OO principles guide you to having the loosest coupling you can, and a very common strategy for this is to, rather than having objects depend on each other directly, that they would depend on each other through a pure interface. Starbound is *chock full* of these, Entity and World are only two. They *amplify* borrowing problems by *amplifying* the amount you are forced to borrow!

I don’t think I can *completely* fully justify everything I’m saying inside a 30 minute talk, this is really only scratching the surface, but hopefully I’ve given you at least an *idea* of where I’m coming from. I know that for some of you this is going to sound VERY subjective and possibly only sound like useful advice for a specific type of software (games). Also, for a lot of you, this is maybe boring and you already knew all this, but it’s still hopefully helpful to see this as it pertains to an industry which not everyone is part of.

But, that being said, here are some of my takeaways:

- For games, OO doesn’t really help, at all. The parts of OO that are useful which I listed above are fine, but at the point where you’re not writing “library” code and are actually writing a game, data hiding is NOT generally useful, and just wastes a lot of effort and time. The more data your game has, the more it changes because you’re constantly experimenting, the worse this gets. This might be old news to you!
- Thinking about “objects” vs data types in a game sounds superficially appealing, but it’s actually actively harmful. Most behavior doesn’t “attach” to any data, and if you start thinking about things that way it can be hard to stop. Stop conflating the data representation of your game with the systems that operate on it!

- Sometimes I think I would rather deal with a single 12k line mega-procedure than a tangled ball of objects.
- Carmack quote: "Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function."
- Sometimes if you have to write a lot of messy procedural state changes, just being honest about it with long (maybe not 12k lines) procedures is the best strategy. Just be honest about the messiness of what's happening, hiding it in other functions or methods doesn't help, it only makes it more confusing when it goes wrong. If you can pull things out that are simple and pure functions, do it, but leave the messy procedural truth as it is! Another Carmack thing

- I find it really easy and enlightening to think about the types of structures that determine the state of a game. You can often learn almost everything you need to know from taking a code base and just looking at all the type definitions. If you're looking at C++, if you just look at the struct members and their relationships and ignore all the code, this will often tell you more than looking at say, the names of functions. I spent a lot of time reading Haskell code and they had this pattern that's often mimicked in Rust where they would have a module called "Types" that just had… all the types that were required in some library. I kind of like that, and I think Rust is really great at this kind of programming.

  I'm being a bit cheeky here, but what tells you more, this interface:

  ```
  class World {
      List<EntityPtr> entityQuery(RectF const& boundBox) const = 0;
      ...
  }
  ```

  or this struct definition:

  ```
  struct World {
      SpatialHash2D<EntityId, float, EntityPtr> entitySpatialMap;
      ...
  };
  ```

  Obviously a lot of the information here comes from the *names* of the types, and clearly `SpatialHash2D` is some struct of its own with its own sub-types, but

> if you name your types well the point still stands, you see an obvious trail of bread crumbs to follow to understand what an interface may hide from you (That your spatial entity queries are probably going to be fast and you shouldn't expect a query to take linear time in the number of entities).

I asked if we could use OO to improve our UR-architecture, and surprisingly we largely *can't*. Useful things like the dot operator and data hiding to maintain invariants in library-ish code aside, the large scale structure of our game is not really helped by OO. We need some way to deal with the downsides of a single giant mutable public nested struct without making an OO mess. We're now approaching the ECS answer from the top down.

Back to the beginning.

Our "OO-architecture" was a complete bust in Rust, so let's start over with something we know will work, our "UR-architecture" and try to improve it. This is a pretty standard ECS transformation, and you may have seen this before. I've read a bunch of ECS explanations before and I know of one fairly good one, and *lots* of really confusing bad ones, and hopefully I can explain this in a nice way that focuses on the *data* representation, and is useful to Rust specifically.

Let's start from the beginning with our simplistic "Starbound", writing down what the state representation of our game is again, but this time in Rust. No methods, no functions, only the data types. This will be very similar to the Mario 64 example "UR-architecture" above, with a few small additions:

```rust
type usize = EntityIndex;

struct Physics {
    position: Vector2<f32>,
    velocity: Vector2<f32>,
    mass: f32,
}

struct HumanoidAnimationState { ... }
struct HumanoidItem { ... }

struct HumanoidState {
    animation_state: HumanoidAnimationState,
    left_hand_item: HumanoidItem,
    right_hand_item: HumanoidItem,
    aim_position: Vector2<f32>,
}

struct Player {
    physics: Physics,
    humanoid: HumanoidState,

    health: f32,
    focused_entity: EntityIndex,
    food_level: f32,
    admin: bool,

    ...
}

enum MonsterAnimationState { ... }
struct DamageRegion { ... }

struct Monster {
    physics: Physics,
    animation_state: MonsterAnimationState,

    health: f32,
    current_target: EntityIndex,
    damage_region: DamageRegion,

    ...
}

struct NpcBehavior { ... }

struct Npc {
    physics: Physics,
    humanoid: HumanoidState,

    health: f32,
    behavior: NpcBehavior,

    ...
}

enum Entity {
    Player(Player),
    Monster(Monster),
    Npc(Npc),
}

struct Assets { ... }

struct GameState {
    assets: Assets,

    entities: Vec<Option<Entity>>,
    players: Vec<EntityIndex>,

    ...
}

fn main() {
    let mut game_state = initial_game_state();

    loop {
        let input_state = capture_input_state();

        player_control_system(&mut game_state, &input_state);
        npc_behavior_system(&mut game_state);
        monster_behavior_system(&mut game_state);

        physics_system(&mut game_state);

        // ... lots more systems

        render_system(&mut game);
        audio_system(&mut game);

        wait_vsync();
    }
}
```

There are a few changes from the Mario 64 example. For one, we're trying to use Rust features like enums and data types like Option, rather than trying so hard to mirror C. The great thing about this, is suddenly the structure of our game state has less possible invalid states! For example, here if we had tried to use a unified "Entity" with a type code, like Mario 64, there would be invariants that had to be kept manually stating that certain fields were in sensible states depending on the type of the entity. Now, many of these invariants just fall naturally out because sum types are so great. This is interesting, because it makes encapsulation less important, even with everything public, there are certain invariants that cannot be broken. There ARE invariants that aren't expressible in this way though, an example being that the array of player ids probably need to point to Player type entities only!

The `entities: Vec<Option<Entity>>` is interesting as well. Since some of our entities keep indexes to this array as "pointers" to other entities, it makes sense that we should never move an entity in the array. If we allocate a bunch of entities then remove the first entity we allocated, instead of moving every entity, we need to set it to None instead so that the rest of the array stays in place. When allocating, we probably would go through the array looking for the first None empty slot, and if none are found, push a new entity onto the end. This is interesting, because this is very similar to how the “Mario 64” example worked with a static array.

Other than these changes, this is not really a huge departure. All of the game state is still morally global, and each system is still potentially large and procedural. Here’s the thing though… I actually don’t think this is very bad at all? In all honesty, if I were doing a game jam game, this might be how I would write my game!

I wouldn’t write low level graphics or audio code, but if I were doing a simple 2D game or very simple 3D game, this would work for me. My game state might be a pretty complex struct full of other structs, and I might have many “systems” (which remember here are just regular functions) in different files, and this would be fine. I would choose a very simple graphics API with as little state as possible, to reduce as much as possible the headache of “loading” and “unloading” graphics data and the same with the audio API, and this would be the structure of my game. If I had to use a very messy and stateful graphics or audio system, I would pre-load everything when creating the game state initially, stick all of it inside some Assets structure, and that would be it. There are plenty of high level level graphics and sound APIs for Rust that actually make this really easy.

But, this pattern is obviously not perfect, even other than everything being globally public. For one, there’s a lot of repeated data in each of our entity types, e.g. with `Physics` being repeated across Player, Monster, and Npc. In our `physics_system` function, probably there is some commonality in this system which is broken down further into another function, but the `physics_system` definitely *for sure* has to understand that there are 3 separate entity types that all have physics. Whenever we add an entity type, probably this system has to change its implementation, and honestly probably *very many* systems have to change when you add entity types.

Also, as we add more entities, we’ll probably start to find out that a large part of them will be repeated over and over. Maybe “Monster” is too specific, and you split this up between “FlyingMonster” and “GroundMonster”, but they share 80% of the same fields, and in turn both of those share 50% of their fields with Player.

This is in some ways better than having all entities be one unified type, but in some ways actually it's kind of worse? It's not terrible, but let's see if we can do better. Let's go back to having a unified Entity type like the Mario 64 example:

```
type usize = EntityIndex;

// All the different types of fields that an Entity can have, grouped somewhat
// logically...

struct Physics {
    position: Vector2<f32>,
    velocity: Vector2<f32>,
    mass: f32,
}

struct HumanoidAnimationState { ... }

struct HumanoidItem { ... }

enum MonsterAnimationState { ... }

struct DamageRegion { ... }

struct NpcBehavior { ... }

struct HumanoidState {
    animation_state: HumanoidAnimationState,
    left_hand_item: HumanoidItem,
    right_hand_item: HumanoidItem,
    aim_position: Vector2<f32>,
}

struct PlayerState {
    focused_entity: EntityIndex,
    food_level: f32,
    admin: bool,
}

struct MonsterState {
    current_target: EntityIndex,
    animation_state: MonsterAnimationState,
}

struct NpcState {
    behavior: NpcBehavior,
}

// An entity is a collection of all the possible entity fields, we let every one
// of them be optional.  In this case, we have lost some type safety, because
// this can express more invalid states than our previous example, some of these
// combinations probably don't make sense.  Also, maybe right now it doesn't
// make sense for an entity to be missing a position, so all entities have to
// have physics even though it's optional here.
struct Entity {
    physics: Option<Physics>,
    health: Option<f32>,
    humanoid: Option<HumanoidState>,
    player: Option<PlayerState>,
    monster: Option<MonsterState>,
    npc: Option<NpcState>,

    ...
}

struct Assets { ... }

struct GameState {
    assets: Assets,

    entities: Vec<Option<Entity>>,
    players: Vec<EntityIndex>,

    ...
}

fn main() {
    let mut game_state = initial_game_state();

    loop {
        let input_state = capture_input_state();

        player_control_system(&mut game_state, &input_state);
        npc_behavior_system(&mut game_state);
        monster_behavior_system(&mut game_state);

        physics_system(&mut game_state);

        // ... lots more systems

        render_system(&mut game);
        audio_system(&mut game);

        wait_vsync();
    }
}
```

Okay, so this is interesting. It’s clear that we lose some invariants here in that more potentially invalid entities could be created, but there are some big advantages here. One that stands out is that our implementation of physics_system is probably vastly simplified, just loop over all the entities and mutate the physics field if they have one. This is much simpler than looping over the entities and having to match on the entity type.

There are still structures that are only valid for each of our “logical” entity types though, and clearly it probably wouldn’t make sense for something to be both an NPC *and* a Monster at the same time, so this is I guess another invariant to keep. It’s interesting though, because the amount of per-“type” data got smaller. Let’s change this a bit more and separate out more fields:

```
struct Physics {
    position: Vector2<f32>,
    velocity: Vector2<f32>,
    mass: f32,
}

struct HumanoidAnimation { ... }

struct HumanoidItems {
    left_hand_item: HumanoidItem,
    right_hand_item: HumanoidItem,
    aim_position: Vector2<f32>,
}

struct MonsterAnimation { ... }

struct NpcBehavior { ... }

struct Aggression {
    current_target: EntityIndex,
}

// Just for symmetry, let's let Health be a struct type
struct Health(f32);

struct Hunger {
    food_level: f32,
}

struct PlayerState {
    focused_entity: EntityIndex,
    admin: bool,
}

struct Entity {
    physics: Option<Physics>,
    huamnoid_animation: Option<HumanoidAnimation>,
    humanoid_items: Option<HumanoidItems>,
    monster_animation: Option<MonsterAnimation>,
    npc_behavior: Option<NpcBehavior>,
    health: Option<Health>,
    hunger: Option<Hunger>,
    player: Option<PlayerState>,

    ...
}
```

So there are lots and lots of ways to express the fields that may go into each entity, and they all have their advantages and disadvantages. In this case, after transforming our field types this way, a couple of things become expressible that weren't before! For example, since we've broken out the `food_level` field from Player and called it `Hunger`, we can now express entity types that aren't Players that are hungry, so now we can describe NPCs with hunger. Also, after breaking out `Aggression`, we can express hostile NPCs! However, maybe we can express something with monster-type animation that also can carry humanoid items, and maybe this is actually logically invalid, so there is a give and take here. Still, this is interesting, and solves the problem where there is much commonality between our entity types that requires repeated code to express or extract from monolithic enums.

Now, it may be very clear where I'm going with this already, but this pattern, where entities are composed of one or more of a set of named parts, and they are specified a la carte, is very common. In fact, these parts are normally called "components"! We now have all of Entities, Components, and Systems, so if you squint this is everything needed to be an "ECS" system, but this is actually very simple! By focusing on the data representation of our state, it is really not very many steps in between this and what we started with.

Let's make one more actually sort of inconsequential change, but this will help us in a second. Also, we'll start calling all of these pieces we've been defining "components" just to drive the point home:

```
struct PhysicsComponent { ... }
struct HumanoidAnimationComponent { ... }
struct HumanoidItemsComponent { ... }
struct MonsterAnimationComponent { ... }
struct NpcBehaviorComponent { ... }
struct AggressionComponent { ... }
struct HealthComponent { ... }
struct HungerComponent { ... }
struct PlayerComponent { ... }

type EntityIndex = usize;
struct Assets { ... }

struct GameState {
    assets: Assets,

    // All of these component vecs must be the same length, which is the current
    // number of entities.
    physics_components: Vec<Option<PhysicsComponent>>,
    humanoid_animation_components: Vec<Option<HumanoidAnimationComponent>>,
    humanoid_items_components: Vec<Option<HumanoidItemsComponent>>,
    monster_animation_components: Vec<Option<MonsterAnimationComponent>>,
    npc_behavior_components: Vec<Option<NpcBehaviorComponent>>,
    aggression_components: Vec<Option<AggressionComponent>>,
    health_components: Vec<Option<HealthComponent>>,
    hunger_components: Vec<Option<HungerComponents>>,
    player_components: Vec<Option<PlayerComponents>>,

    players: Vec<EntityIndex>,

    ...
}
```

This is the classic “array of structs” to “struct of arrays” transform. It should be clear that (nearly) exactly the same information is expressible between this and the previous representation. There are a few new invariants to maintain, namely that each component Vec is the same length, but there’s not really new information. When you just write out types like this, these kinds of changes actually look pretty simple, instead of in our OO architecture where something like this might be considered an impossible refactor. Obviously we’d have to change the systems a lot here too potentially every time we made these changes, but there’s value in being able to think about the data independently.

The transformation to “structs of arrays” is generally the thing that ECS introductions really focus on, and then they start talking about performance concerns and cache behavior and it can be a little confusing and overwhelming. They’re not wrong, but I really don’t think this is the important part. This is why I’ve introduced ECS this way, because I think a lot of explanations miss this point, that from a certain perspective, these changes are actually pretty boring and normal. You might have this structure or you might have the previous structure, maybe one is better performing than the other, but you have not radically changed programming paradigms by choosing one vs the other. From a data-oriented perspective, these things are not earth shattering, they're just performance optimizations. There are lots MORE potential optimizations you can do here as well, but I think we've reached the point where we have a bare-minimum that somebody might call an "ECS system".

There are some takeaways here for Rust users in general, not just game developers:

- Thinking about JUST the structure of state is really powerful, often times "methods" and "objects" get in the way. Nobody really thinks that having methods on your structs is "too much OO", but it DOES couple your procudures to your data, even if only organizationally. Both data structure design and simple module organization are important, but don't conflate them! I'm convinced that "ECS design" in games is so great because it forces you to think about your data rather than be stuck in OO mode, not because of the magic of "structs of arrays". This is a lesson that can be applied to Rust in general.
- You can do a LOT with indexes into Vecs. This is way easier than self borrowing or Rc<RefCell>.

Okay, let's stop for a second and talk about `EntityIndex`. I genuinely think that most of the time when you find yourself running into self borrowing with Rust, plain Vecs and indexes should first tool you reach for. There are OTHER tools, like various arena crates, rental etc, but I think that generally you should try Vec first. Generally these problems happen when trying to represent some kind of graph structure (and an ECS is just a really flat graph… kind of), and *even in C++* this is the advice generally given! (See Andrei Alexandrescu's performance talks, his favorite data structure in the world is std::vector, "just use vector!") Things like Arena allocators are great and actually quite performant, and there are versions that work with multiple types, but they can't be 'static without the addition of self-borrowing. Self borrowing solutions like rental are tools of *last resort*. It is not worth it, move on with your life, and just use Vecs and indexes.

THAT BEING SAID, they have do have some problems. In our entity examples, we've been glossing over the question of how to find "free" EntityIndexes, and how "deletion" works, because it's not actually terribly great. The cost of allocating an "entity" is not constant because we have to scan through a Vec looking for free entries. Deleting an entity is cheaper, but it has bad properties.

We can delete an entity which frees up a slot in a Vec, but then it's possible that the very next allocated entity will use the same index. This is fine if we make sure that there are no outstanding "index references" to this entity before deletion, but what if we mess that up? We'll get a "random other entity" in its place without being able to tell that it was in fact removed out from under us!. Also, both of these situations are now much worse after the "struct of arrays" transform, because we now have multiple arrays instead of one and how we manage "entity indexes" is kind of up in the air now. We're going to solve both of these problems at once.

Generational indexes are awesome.

So this is one of my favorite patterns that I'm not sure is widely known in the Rust community but I believe is widely known in the gamedev one.

We've been using `EntityIndex` to identify and look up entities stored inside a Vec. This is very similar to the pattern I used in the C++ mini-Starbound, I used an `EntityId` which was an increasing integer id to store entities like this:

```
HashMap<EntityId, shared_ptr<Entity>> entities;
```

Both are abstract mappings from integers to some value. In the Vec case, it is basically the fastest POSSIBLE data structure, indexing into an array, and in the case of `HashMap` it is much slower, but the `HashMap` is much more flexible! For one thing, there is only an upper bound really on the total number of entries, not on, say the largest size of the index. `HashMap` works just as well if you start the keys at value 1,000,000,000 or 1, certainly a billion entry Vec is different than a 1 sized one. This matters for the deletion problem above. If we indexed using a HashMap and a key type of `u32` or even `u64`, we could just have new indexes always increment (with eventual wrap-around). In this way, you could more or less guarantee that no index would ever be re-used, or at the VERY least it would be a "very long time" before an index would be reused. In the `u64` case, if you ever re-used an index, your game technology probably has "cloud" in the name, so you might have bigger problems anyway :). BUT HashMap is much slower than Vec :(

Is there a way to get this property with integer indexes into a Vec? There is, and it's called "generational indexes"!

Instead of using just an integer index, we make a “generational index” type like this:

```
// You can use other types that usize / u64 if these are too large
#[derive(Eq, PartialEq, etc...)]
pub struct GenerationalIndex {
    index: usize,
    generation: u64,
}

impl GenerationalIndex {
    pub index(&self) -> usize { ... }
}
```

Then, we make something called an “GenerationalIndexAllocator”:

```
struct AllocatorEntry {
    is_live: bool,
    generation: u64,
}

pub struct GenerationalIndexAllocator {
    entries: Vec<AllocatorEntry>,
    free: Vec<usize>,
}

impl GenerationalIndexAllocator {
    pub fn allocate(&mut self) -> GenerationalIndex { ... }

    // Returns true if the index was allocated before and is now deallocated
    pub fn deallocate(&mut self, index: GenerationalIndex) -> bool { ... }

    pub fn is_live(&self, index: GenerationalIndex) -> bool { ... }
}
```

(There are faster ways to implement this, but this one is actually pretty much fine. Also note that this is an obviously GOOD use of data hiding).

The basic idea is that you can “allocate” indexes for a vector just like if you had `Vec<Option<Entry>>`, but you will instead never re-use indexes. It works like this, you allocate an index and get back a `GenerationalIndex` with real index 0, it will also have “generation” 0. If you delete that index, it will go into a pool of free indexes, so next time you allocate an index you might get back another generational index with real index 0, but crucially the generation will now be 1.

Generational indexes are never re-used because the generation will always increment, yet the "real indexes" will always be "small", on the order of the largest total number of entries ever allocated. This way, you can use fast indexing into a Vec without many of the bad "pointer-like" properties of simple indexes!

I said this pattern is not widely known inside the Rust community, but that's at least a bit of a lie because there is a recently released crate built on this idea called "slotmap" and it's great! However, it's missing a CRUCIAL feature for our example, which is that in "slotmap" you can only allocate indexes for a specific SlotMap, you can't allocate indexes and re-use them for different SlotMaps. Useful, but it would be vastly MORE useful if these concepts were separate, which is what we've sketched out here. Indexes into a Vec are already a great pattern when encountering "self borrowing", and generational indexes make it much much better. "slotmap" beat me to releasing a crate for it :(, so consider this a feature request :)

We'll go ahead and make a type that's a bit easier to use than `Vec<Option<T>>` to store our actual data as well:

```rust
struct ArrayEntry<T> {
    value: T,
    generation: u64,
}

// An associative array from GenerationalIndex to some Value T.
pub struct GenerationalIndexArray<T>(Vec<Option<ArrayEntry<T>>>);

impl<T> GenerationalIndexArray<T> {
    // Set the value for some generational index.  May overwrite past generation
    // values.
    pub fn set(&mut self, index: GenerationalIndex, value: T) { ... }

    // Gets the value for some generational index, the generation must match.
    pub fn get(&self, index: GenerationalIndex) -> Option<&T> { ... }
    pub fn get_mut(&mut self, index: GenerationalIndex) -> Option<&mut T> { ... }
}
```

So with this new abstraction, let's change our engine some more:

```rust
struct PhysicsComponent { ... }
struct HumanoidAnimationComponent { ... }
struct HumanoidItemsComponent { ... }

struct MonsterAnimationComponent { ... }
struct NpcBehaviorComponent { ... }
struct AggressionComponent { ... }
struct HealthComponent { ... }
struct HungerComponent { ... }
struct PlayerComponent { ... }

// We're dropping the index or id suffix, because there is no other "Entity"
// type to get confused with.  Don't forget though, this doesn't "contain"
// anything, it's just a sort of index or id or handle or whatever you want to
// call it.
type Entity = GenerationalIndex;

// Map of Entity to some type T
type EntityMap<T> = GenerationalIndexArray<T>;

struct GameState {
    assets: Assets,

    entity_allocoator: GenerationalIndexAllocator,

    physics_components: EntityMap<PhysicsComponent>,
    humanoid_animation_components: EntityMap<HumanoidAnimationComponent>,
    humanoid_items_components: EntityMap<HumanoidItemsComponent>,
    monster_animation_components: EntityMap<MonsterAnimationComponent>,
    npc_behavior_components: EntityMap<NpcBehaviorComponent>,
    aggression_components: EntityMap<AggressionComponent>,
    health_components: EntityMap<HealthComponent>,
    hunger_components: EntityMap<HungerComponents>,
    player_components: EntityMap<PlayerComponents>,

    players: Vec<Entity>,

    ...
}
```

Neat! We're in the home stretch now, this is almost a full blown ECS system.

Take away for users of Rust in general: obviously generational indexes are awesome but for some reason they're more popular only in C++, but they're potentially even MORE useful for Rust! To the author of "slotmap", please expose the allocator separately, it's hugely useful! If that doesn't seem in scope or the kind of API you want to provide, I can release a similar crate with the version I have that does do this.

Dynamic typing is actually kind of nice in VERY controlled quantities.

Okay, we're really close now, this is very close to how a "real" ECS system (like specs!) might work. The biggest problem we haven't addressed from earlier is still that everything is kind of global still. More so, every "system" (for us, this is still just a fancy name for plain functions) depends on *all* of the types that go into our game state, which may be quite large. Most of a game is going to live inside one crate and the dependency graph between modules such as this is really hardly anything to fret over, but still changing anything inside "GameState" potentially would affect every system at least *theoretically*. Let's see what we can do about this?

I want to emphasize before we go on that this, like many steps before, is *optional*. You may balk at this as unnecessary complication, and you may be right! This IS something that is in most ECS implementations though, so it's worth covering if only to understand them, and it's more or less unavoidable when trying to build a library to do this sort of thing.

For this we're going to need the `anymap` crate, but also the `mopa` crate would do as well. What we need is a container that can store exactly one of every type that we put into it:

```
pub struct AnyMap { ... }

impl AnyMap {
    pub fn insert<T>(&mut self, t: T) { ... }
    pub fn get<T>(&mut self) -> Option<&T> { ... }
    pub fn get_mut<T>(&mut self) -> Option<&mut T> { ... }
}
```

How might we use this to store our components?

```
struct PhysicsComponent { ... }
struct HumanoidAnimationComponent { ... }
struct HumanoidItemsComponent { ... }
struct MonsterAnimationComponent { ... }
struct NpcBehaviorComponent { ... }
struct AggressionComponent { ... }
struct HealthComponent { ... }
struct HungerComponent { ... }
struct PlayerComponent { ... }

type Entity = GenerationalIndex;
type EntityMap<T> = GenerationalIndexArray<T>;
```

```
struct GameState {
    assets: Assets,

    entity_allocoator: GenerationalIndexAllocator,
    // We're assuming that this will contain only types of the pattern
    // `EntityMap<T>`.  This is dynamic, so the type system stops being helpful
    // here, you could use `mopa` crate to make this somewhat better.
    entity_components: AnyMap,

    players: Vec<Entity>,

    ...
}
```

Now instead of storing game-specific data, let's just keep going with dynamic typing! We'll say that our game state is a dynamic collection of entities with components, and *also* a dynamic collection of other types, one of each type. We'll call these "resources". Also we'll change the name from `GameState` to something more accurate.

```
type Entity = GenerationalIndex;
type EntityMap<T> = GenerationalIndexArray<T>;

struct ECS {
    entity_allocoator: GenerationalIndexAllocator,
    // Full of types like `EntityMap<T>`.
    entity_components: AnyMap,

    resources: AnyMap,
}
```

And here we have arrived at what an actual ECS data structure might look like. I've added a "resources" AnyMap because this is also a very common pattern, and it means that more or less your entire game state can be expressed inside this "ECS" structure. We're calling it an "ECS", but importantly, really there aren't any mentions of "systems" in sight. I don't actually like describing ECSes in terms of systems, because I think while it can be important, it's really incidental. If our "systems" are pure functions in a loop or they're something much fancier, BOTH of these capture the important part of ECS design.

So, you may be balking now at the egregious and sudden introduction of dynamic typing. Let's stop and think though what this buys us. Say you get a new feature request for your game, say you need a new crazy special monster that has some kind of counter inside it. Every time you kill the monster, it copies itself into two and decrements the counter, duplicating like the heads on a hydra. This means you might need a new component type, say `EnemyDuplicationLevel` or something. With dynamic typing, you can add this component without “disturbing” your other systems, because without importing the new module, they can’t possibly “see” that the ECS has such a component anyway. The same is true for resources, you can add new data types to your model without “disturbing” existing systems.

The justification for this may seem pretty weak, and it kind of is. In order to get the full picture we need to go a bit further. I’m going to speed up a bit so we can get to the end and I can show a full-ish picture, something which I consider a “modern” game engine design that you might use for a medium or large project. It’s actually not much further along than this, but the last few features are all related and are not as useful alone.

The “registry” pattern

Now that we’ve introduced dynamic typing, there’s a design pattern that I like in Rust that I haven’t really seen in practice yet (I’m sure it exists and I have just not seen it). I’m going to call it the “registry pattern”.

In ECS implementations like specs, there’s a step where you “register” a type with your ECS, which inserts an entry into some AnyMap or equivalent to an AnyMap. Using a component type that is unregistered is usually an error. Let’s take this a bit further, and not tie “registering” to the ECS per se, let’s make our own “registry”.

```rust
pub struct ComponentRegistry { ... }

impl ComponentRegistry {
    // Registers a component, components must implement a special trait to allow
    // e.g. loading from a JSON config.
    pub fn register_component<T: Component>(&mut self) { ... }

    // Sets up entries for all registered components to the given ECS
    pub fn setup_ecs(&self, ecs: &mut ECS) { ... }

    // Loads a given entity into the given ECS, loading all the components from
    // the given config
    pub fn load_entity(&self, config: Json, ecs: &mut ECS) -> Entity { ... }
}
```

We’ll also make one for “resources”

```
pub struct ResourceRegistry { ... }

impl ResourceRegistry {
    // The Resource trait provides loading from JSON and other things.
    pub fn register_resource<T: Resource>(&mut self) { ... }

    // Sets up entries for all registered resources to the given ECS
    pub fn setup_ecs(&self, ecs: &mut ECS) { ... }

    // Adds a resource to the given ECS by loading from the given config.
    pub fn load_resource(&self, config: Json, ecs: &mut ECS) { ... }
}
```

Then, we’ll tie them together in one big global constant with lazy_static!

```
// When we add a component to our project, there are two steps.  First, add the
// component somewhere as a Rust module, THEN add it to this list here.  For
// added convenience, this function could go in the lib.rs which contains the
// component modules themselves.  If you were very fancy, you could have some
// kind of "plugin architecture" for this as well, grouping related components /
// resources together into "plugins".
fn load_component_registry() -> ComponentRegistry {
    let mut component_registry = ComponentRegistry::new();

    component_registry.register::<PhysicsComponent>();
    component_registry.register::<PlayerComponent>();
    ...
}

// Ditto
fn load_resource_registry() -> ResourceRegistry { ... }

pub struct Registry {
    pub components: ComponentRegistry,
    pub resources: ResourceRegistry,
}

lazy_static! {
    pub static ref REGISTRY: Registry = Registry {
        components: load_component_registry(),
        resources: load_resource_registry(),
    };
}
```

I like this pattern, because this is, if you squint, almost like a sort of global type registry like you might find in Java. It feels a bit “enterprisey”, but it’s quite useful! It is *also* similar (again, if you squint) to very full-featured all encompassing engines with built-in editors, where you might say add a component type through some menu or GUI and it ends up being stored in some project config file. In this case, instead of a project config file it’s simply a bit of auxiliary rust code.

This pattern ends up being very useful! Imagine you have some game state that can be loaded from a JSON config file. Each resource and component can be loaded from JSON, so you can just add a component or resource type and then add the entry to the data format, load the game and see it (except you need a new system that understands the new types, we’ll get there in a sec).

So at this point, this is a sketch of something that is unmistakably turning into a “real” ECS game engine. I like this, because so far, except for some brief required library-ish functionality, I have barely talked about functions or systems! The reason for this is, I don’t like introducing this concept by talking about behavior, I really think that thinking just about how we describe our state is a much more useful way to approach this.

With that out of the way, it’s pretty easy to see how you could also just as easily add a `SystemRegistry` to go along with this. Our systems are just functions, so all this would do is allow you to add functions to your main loop in a slightly more complicated way. One possible addition here is to allow systems to have some kind of config, so that maybe you can configure tunable parameters for them or even use the same function multiple times with different parameters. It’s generally advised though that you try very hard not to give your systems *state*, so you can limit your game state to just components and resources. If your game state is just components and resources, you can often do cool things like clone them (save state like in an emulator!) or easily serialize it, and this is harder with something like system closures.

Takeaways for regular Rust: The registry pattern is actually quite nice. In Starbound there is a “Root” object which is sort of like this, where every type is registered, but it’s more stateful than what I’ve described here. It’s read only, but it’s constructed by reading assets. I actually like the idea of a “type registry”, and something like this is necessary as soon as you want to use this sort of dynamic typing with AnyMap, and the two patterns go together well to limit the problem of "everything depending on everything else".

ECS is SQL for games

Let me pull the curtain back a bit. What have we described so far: A way of declaring a special kind of key (Entity), and a series of records that go with those keys (components), and also for good measure a way to define records that don't pair with any of those keys (resources). We've added dynamic typing into the mix, and shown a pattern so that you can define all the record types you have in one place (kind of like a schema?). In the real ECS I have built, I even have ways of performing primitive "queries" on components, saying "give me all entities with a position, optionally a velocity, but NOT a mass".

This probably sounds achingly familiar, and if it does there's a good reason for it. ECS is just SQL for games. A very, very, very limited form of SQL, but spiritually very similar. Define the schema for your data, load it, run queries on it, and update it. Very limited SQL where every query might have to run in no more than a few microseconds.

I said earlier that I've read a lot of bad introductions to ECS and one good one. I'm not the first to mention this parallel between ECS and SQL, all the way back in 2009, there was an article about this very point with a focus on MMOs.

So, if this is true, why can't we just use something like sqlite to store our game state? The funny thing is, you might actually be able to do this, but it's tough to run queries fast enough in 16ms. Let's think about what kind of benefits this would bring, though! Oh man, I need to make a save format for the game.. nope it's SQL I'm already done. Oh, I need to update the format of everyone's save file.. write some external SQL update scripts and you're done! Oh, I'd like to give the ability to save the game's state at any arbitrary point? Done. I noticed a bug and I want to roll back the game state to 30 seconds ago and step through it frame by frame, running queries to see when a bad state was produced? done. All this careful introduction of the generational indexes concept, SQL has this already, they're keys with auto increment (more like `HashMap<EntityId, T>` really).

At that point, each of your systems would basically be a series of queries to get data out of SQL, some updates, and then writing it back out to SQL. We've turned sexy game development into boring old web development! (web development can be sexy and game development can be boring, just for the record)

The only issue is that I'm *pretty* sure it's about two orders of magnitude too slow to work, but still it's a neat idea, and it's useful to draw parallels!

I think this is important, because I see a lot of debate about "what makes something a pure ECS" that I think is ultimately pretty silly.

- Is it an ECS if entities can have multiples of a component? Well, the differences between multiples of a component and a component that contains just a Vec are very very few, but yes it's still ECS. Is it still SQL if you have two tables that are 1 to N instead of 1 to 1? Yes, of course it is.
- Is it still an ECS if my components have methods? Yes, adding a component method is not the end of the world, especially if there is a small local invariant to maintain. SQL is all about data, but it also has stored procedures to maintain invariants.
- Is it still an ECS if I need two different ECS sets, or I put lots of data into resources? Should I be using "singleton entities" or resources? Is it still SQL if I have two different databases, or split a top level table apart? Yes, of course!

… and so on

The analogy is not perfect, but I think this is really kind of illuminating. There is research currently about how to make more capable ECS systems that have things like component graphs with child relationships between components and entities and all kinds of crazy stuff, and the thing is that's of course fine. SQL has all of this already, it's an entire language and set of software for expressing *all kinds* of data relationships, it's just that games have generally simpler needs, will basically always be in-memory only, and have timing requirements on the order of nanoseconds or microseconds, so we compromise and make new tools.

This is the other reason I'm doing this talk, to help further de-mystify "ECS" and show how we ended up here, and to give perspective.

The one place where it all goes wrong

(NOTE: I don't know if I need this section at all, I'm just hawking the OTHER talk I wanted to do. Maybe this is unimportant? This talk is already potentially very very long)

I claimed at the beginning that the attitude I see of being overly skeptical of the borrow checker, of considering it too restrictive, is bunk. I still absolutely believe this, but I want to talk quickly about the one place where I kept running into issues, really the *only* place where I really had trouble, where I constantly felt limited. Certainly as a project grows in size, *something* must go wrong right? Well, I have one example of this, though this problem isn't really limited to Rust at all, it's just more *immediately* painful (noticing a pattern?).

Language boundaries are hard.

So, the issues here are all… pretty hard to explain succinctly. I'm not sure I can come up with small limited examples like before without getting way into the weeds and doubling the length of this already lengthy talk. I wrote an entire crate (rlua) just to try and solve this problem, and after a ridiculous amount of work I can confidently say that I've *sort of* half-solved it.

I have a representation of components where you choose to read or write each component (they're stored in RwLocks for system concurrency). I want Lua to be able to read a set of systems, so I have a lua script perform a "query" on the ECS store. All I need to do is to be able to hand Lua the "RwLockReadGuard" (really a structure containing this) that the query returns, except… RwLockReadGuard is not static. Oh I guess I'll just have the Lua query lock inside the query API rather than once at the beginning… well that would be very slow and also not thread safe, a query should lock for the whole time. I guess I could use the rental crate.. oh god the rental crate is HARD and this is TERRIBLE. (I was later able to solve this problem with rlua's "scope" system, but it's still bad).

It's best if you don't have your systems be stateful, right? Generally systems should store data either in the components they operate on if they're logically related, or maybe in a custom resource, or simply store some cache-like values but be prepared if the system is reloaded and the cache is reset. Don't rely on data that can't be saved / restored / serialized etc. This is great until you write a Lua system, because exfiltrating values outside of Lua is actually really hard. You can't store a Lua state inside a resource because it's not Sync, and you can't store Lua data separate from the state because the Lua internal state and external handles are REALLY not Sync. You can sort of solve this with magic Lua registry keys, but then if each system has its own Lua instance, you've made a footgun if you ever try to use Lua values in the wrong context, and they're not serializable anyway.

You can make a data type which is limited to just data and not an internal Lua type, but then you must write marshalling between it and the non-lua representation, so every time your script system reads or writes this storage it's *dog slow*. Some of the slowness is from copying data, but most of it is from the intrinsically slow Lua API.

Everything is hard, it works if you really try but you feel like it should be simpler than this. It's harder even than the equivalent in C++, but the equivalent in C++ is actually just as bad, you just notice way faster in Rust. At least there are not constant crashes and shared_ptr cycles now?

Seems a bit similar to our situation before, where we were trying the wrong approach with OO. Unfortunately, in the case of rlua, there are pretty hard limitations based on how the internal Lua C API works, and it might be close to the best I can offer.

Language boundaries are hard, especially between languages that have very different sets of limitations like Rust and Lua (or C++ and Lua). Especially in the presence of a garbage collector in the contained language, if the garbage collector in the contained language ever makes contact with one in the host language, nothing will work, nothing will get collected. (In C++ / Rust this is shared_ptr / Arc, yes these are a sort of garbage collector).

I mention this because I think Lua is very popular for games, and I get questions a lot about how to combine rlua with things like specs, and I think it's just really hard, and goes against the larger point I'm making with this talk. I think this is probably *uniquely* hard due to language boundaries being so tricky, and the situation is not vastly worse or better than the one in C++, it's slightly more difficult and vastly safer, just slightly more difficult is not a great answer when in C++ it's still very difficult.

I also mention this because I might have an answer for this soon! The OTHER talk I thought about doing was describing what I believe is a novel way of implementing language runtimes safely in rust that have garbage collection that is zero cost, and comes with the ability to have a fast, mostly pain-free (or as much as possible) bindings experience.

In the meantime, think really really hard before you add a scripting layer to your game engine. The problem is, I LOVE scripting layers in game engines (for modability and many other reasons), so I do this anyway, but it is not a decision to be taken lightly and it can eat up a lot of time and effort. I want to make this more painless though, and make a safe, fast Lua that feels at home in Rust in the same way that PUC-Rio's Lua feels at home in C. I will talk about this in the future!

Summary

I had multiple purposes with this talk:

- To walk through the design of a medium-scale rust project for those with only experience in small-scale ones.
- To show some examples of traps people might fall in that cause them to "fight the borrow checker", and to help make more concrete how to move past this phase, at least in part.
- To show how awesome data-oriented programming is and how well of a fit it is for games and for Rust
- To show how patterns in game dev that appeared long before Rust, when applied to Rust, still seem to be a good fit.

I talked a lot about patterns that in some cases were hard fought in other languages, only appearing after laboriously exploring the space of solutions that DIDN'T ultimately work. Interestingly, those solutions that ultimately don't work very well often times are painful more quickly in Rust, they're louder, MORE annoying. I think this is a good thing, and I like having tools that make bad patterns feel as bad as they are. Rust is *great* at this.

But could you take most of the lessons here and apply them just as easily in C++? In C? Yes, absolutely. Even if you're the sort of person that doesn't make these mistakes even when there isn't this sort of language pressure to avoid them (you're way better than me), there are tons of other great benefits. I didn't really get a chance to talk about all the benefits of Rust or talk about my experiences with Rust in general, because I only have 30 minutes and this is probably way longer than 30 minutes at this point.