The C++ of EnTT

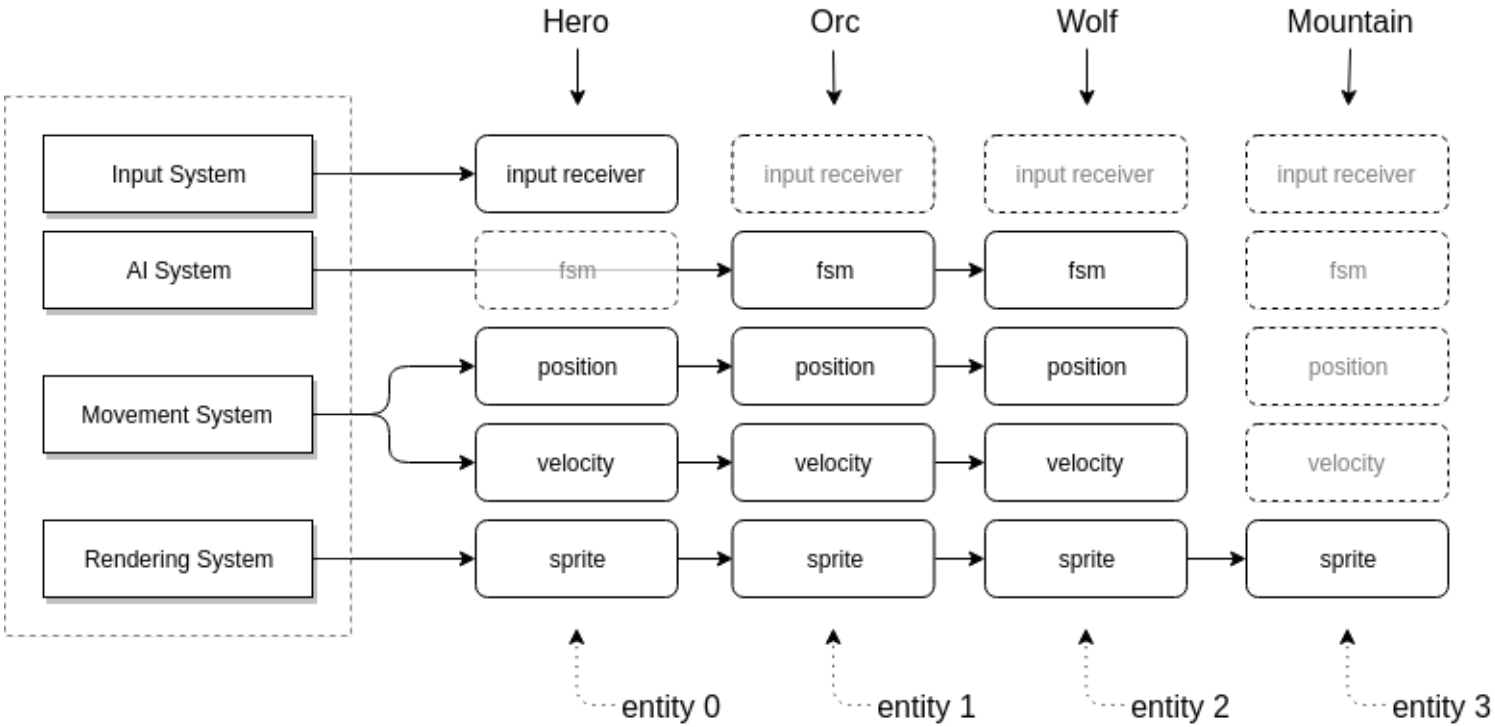
Michele Caini

skypjack

May 11, 2019

From hierarchies to components

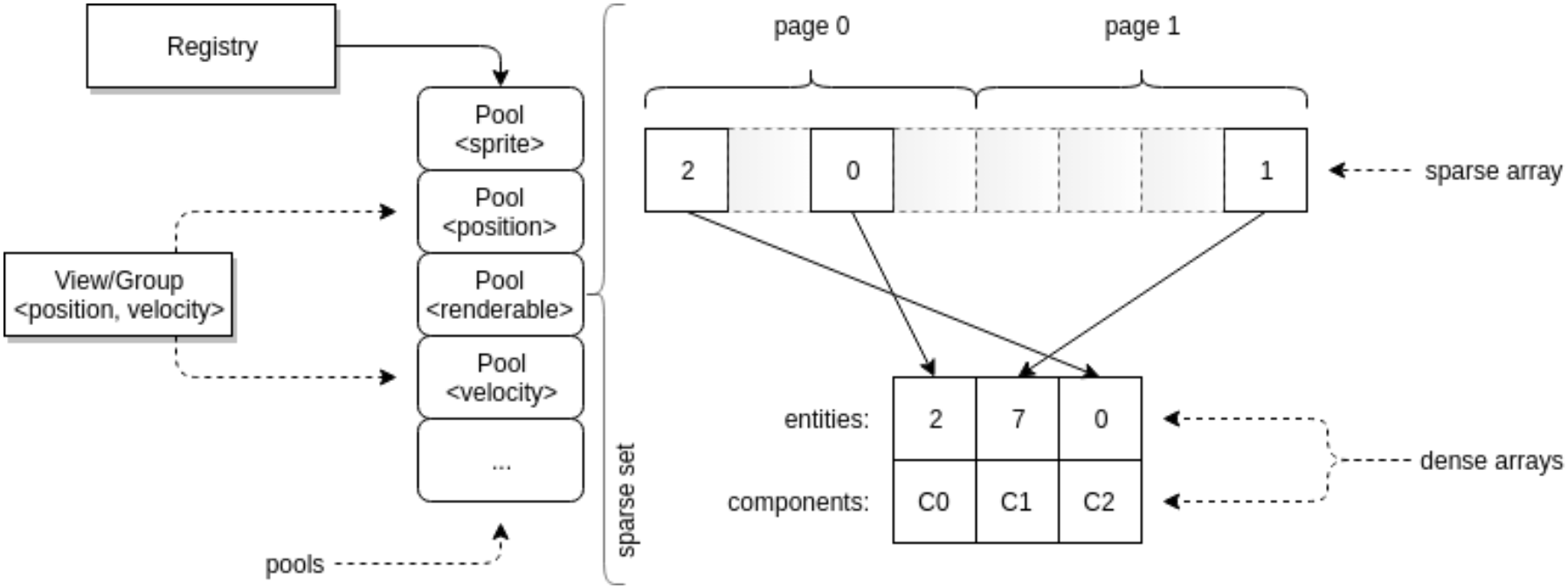
Entity–Component–System (**ECS**) is an architectural pattern.



It favors **composition** over inheritance and sacrifices encapsulation.

EnTT - Gaming meets modern C++

`EnTT` is a C++ framework mainly known for its **ECS** model.



Multiple access patterns supported, from **perfect SoA** to fully random.

The C++ of EnTT

`EnTT` is first and foremost a framework in **modern C++**:

> *Using `EnTT` and diving into it taught me all I know about templates, which was zero last year, so I'd say it's a good example, especially since it's pretty clean actually thanks to all the type folding and C++17 features. – gitter channel*

Keep note

SFINAE, type erasure, tag dispatching, fold expressions, type traits, ...

Who needs types?

Type erasure in a nutshell:

- Polymorphism - yeah, **that** polymorphism.
  - Mostly hierarchies and `virtual` member functions.
- `void *`, `std::shared_ptr<void>` and the others for data.
- Function pointers, inheritance and template machinery for behaviors.

Who needs types?

Type erasure in a nutshell:

- Polymorphism - yeah, **that** polymorphism.
  - Mostly hierarchies and `virtual` member functions.
- `void *`, `std::shared_ptr<void>` and the others for data.
- Function pointers, inheritance and template machinery for behaviors.

Type erasure and `EnTT`

Pools of components, `delegate` class, runtime reflection system, ...

Delegate: a minimal example

A zero-cost abstraction to wrap callable targets.

```
template<typename>
struct delegate;

template<
    typename Ret,
    typename... Args
> struct delegate<Ret(Args...)> {
    // ...

    Ret operator()(Args... args) {
        return fn(data, args...);
    }

private:
    Ret(* fn)(void *, Args...);
    void *data;
};
```

```
struct my_type {
    int member(int i) {
        return i*i;
    }
};

// ...

my_type instance;
delegate<int(int)> op;
op.connect<&my_type::member>(&instance);

// ...

const auto res = op(42);
```

Not a drop-in replacement for `std::function` but it has also drawbacks.

Delegate: C++14 vs C++17

C++14: void(int) is not void(double)

```
template<Ret(*Function)(Args...)>
void connect() noexcept {
    fn = +[](void *, Args... args) -> Ret {
        return Ret((Function)(args...));
    };
} // del.connect<&my_function>()
```

C++17: welcome auto and invoke

```
template<auto Function>
void connect() noexcept {
    fn = +[](void *, Args... args) -> Ret {
        return Ret(std::invoke(Function, args...));
    };
} // del.connect<&my_function>()
```

Delegate: C++14 vs C++17

C++14: no auto, no party

```
template<typename Type, Ret(Type::*Member)(Args...)>
void connect(Type *instance) noexcept {
    // ...
} // del.connect<my_type, &my_type::member>(&instance)
```

C++17: const, non-const, noexcept, ...

```
template<auto Member, typename Type>
void connect(Type *instance) noexcept {
    data = instance;

    fn = +[](void *instance, Args... args) -> Ret {
        return Ret(std::invoke(Member, static_cast<Type *>(instance), args...));
    };
} // del.connect<&my_type::member>(&instance)
```

Soft errors on sale

SFINAE: **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**error

> *If a substitution results in an invalid type or expression, type deduction fails. [...] Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure.*

Thankfully C++17 gave us also `if constexpr`.

SFINAE and `EnTT`

Sparse sets, `registry` class, groups and views, signalling part, ...

Iterate a view

`if constexpr` allows us to save typing and reduce the boilerplate:

```
template<typename Entity, typename... Comp>
class basic_view { /* ... */ };

template<typename Entity, typename Comp>
class basic_view<Entity, Comp> {
    // ...

    template<typename Func>
    void each(Func func) const {
        if constexpr(std::is_invocable_v<Func, Comp &>) {
            // ...
        } else {
            // ...
        }
    }
};
```

What would it be like without `if constexpr` instead?

Old fashioned SFINAE

A not reusable one-off: `decltype` and trailing return type.

```
template<typename Func>
auto each(Func func) const
// (soft) error for [](auto entity, auto &comp) { ... }
-> decltype(func(std::declval<Comp &>()), void()) {
    // ...
}

template<typename Func>
auto each(Func func) const
// (soft) error for [](auto &comp) { ... }
-> decltype(func(Entity{}, std::declval<Comp &>()), void()) {
        // ...
}
```

Scattered around, more difficult to read and to maintain.

Old fashioned SFINAE

Directly from the 90s: dear old `std::enable_if_t`.

```
template<typename Func>
// (soft) error for [](auto entity, auto &comp) { ... }
// std::enable_if_t<std::is_invocable_v<Func, Comp &> /*, void */>
typename std::enable_if<std::is_invocable_v<Func, Comp &> /*, void */>::type
each(Func func) const {
    // ...
}

template<typename Func>
// (soft) error for [](auto &comp) { ... }
// std::enable_if_t<std::is_invocable_v<Func, Entity, Comp &> /*, void */>
typename std::enable_if<std::is_invocable_v<Func, Entity, Comp &> /*, void */>::type
each(Func func) const {
    // ...
}
```

It does exactly what it says, but doesn't improve things much.

That's not all

Remember: `EnTT` is a **C++ framework**.

Some things you can spot around if you pay attention:

- CRTP (curiously recurring template pattern): `emitter` class.
- Tag dispatching: `process` and `scheduler` classes.
- Type traits: *named types* to make `EnTT` work across boundary.
- Small object optimization: `meta_any` class.
- Fold expressions: almost everywhere.

```
pool<Comp>().construction().template connect<&Registry::creating<
    &handler_family::type<Component...>,
    std::tuple_element_t<
        (Indexes < Pivot ? Indexes : (Indexes+1)),
        std::tuple<Component...>
    >...
>>(); // Fortunately EnTT v3 no longer contains this!!
```

An much more...

Questions?

*C++ has indeed become too **expert friendly**. – Bjarne Stroustrup*

Links

- EnTT - Gaming meets modern C++
- ECS back and forth series
  - Introduction
  - Where are my entities?
  - Sparse sets and grouping functionalities
  - To be continued...
- Andrzej's C++ blog
  - Type Erasure - Part 1 (and all other parts)
  - Clever overloading
- More C++ Idioms