Rust'owe Opowieści: Rust, ECS i webassembly

Intro

"Bitowy rycerzu, król wzywa cię do służby. Jeśli odzyskasz księżniczkę - czeka cię nagroda". Artykuł ten skupiać się będzie na poziomie designu/architektury gry typu turowe roguelike. Po szczegóły implementacyjne zapraszam do mojego repozytorium lub repo autora książki *Hands-on Rust* (link do repo). Gra bazuje na autorskiej bibliotece bracket-lib, jednak wiedza zawarta w książce może zostać przeniesiona na inny silnik, np. bevy (praca w toku :)).

Chapter 1: Pomysł

Od najmłodszych lat byłem zafascynowany grami video. Po opanowaniu sztuki władania bitami, jedną z pierwszych moich myśli było: "zostanę programistą w game dev". Los jednak chciał, że pracuję w "web dev" i głównie zajmuję się frontendem. Myślami jednak błądziłem do czasów, kiedy to z wypiekami na twarzy czytałem "*Język C++. Szkoła programowania*" od deski do deski. Trzy lata temu dowiedziałem się o języku Rust. Jest dla mnie idealny, ponieważ łączy wiele konceptów, które znam z programowania funkcyjnego w JS oraz siłę niskopoziomowego języka. Od razu pomyślałem o zrobieniu gry. Sam pomysł odłożyłem jednak w czasie, chcąc najpierw wynieść moje Rust'owe skille na odpowiedni poziom. To udało się, gdy znalazłem książkę "*Hands-on Rust: Effective Learning through 2D Game Development and Play*". Roguelike + Rust? Piękniejsze mogło być chyba tylko połączenie RPG + Rust. Moja bitowa głowa krzyknęła: "do dzieła"!

"Rycerz odwiedził wszechmocną wyrocznię i zapytał:

- Z jakimi potworami przyjdzie mi się mierzyć?
- Widzę, widzę jasno i wyraźnie, to najpotężniejsze istoty na ziemi, Rust, ECS i webassembly".

Chapter 2: Rust

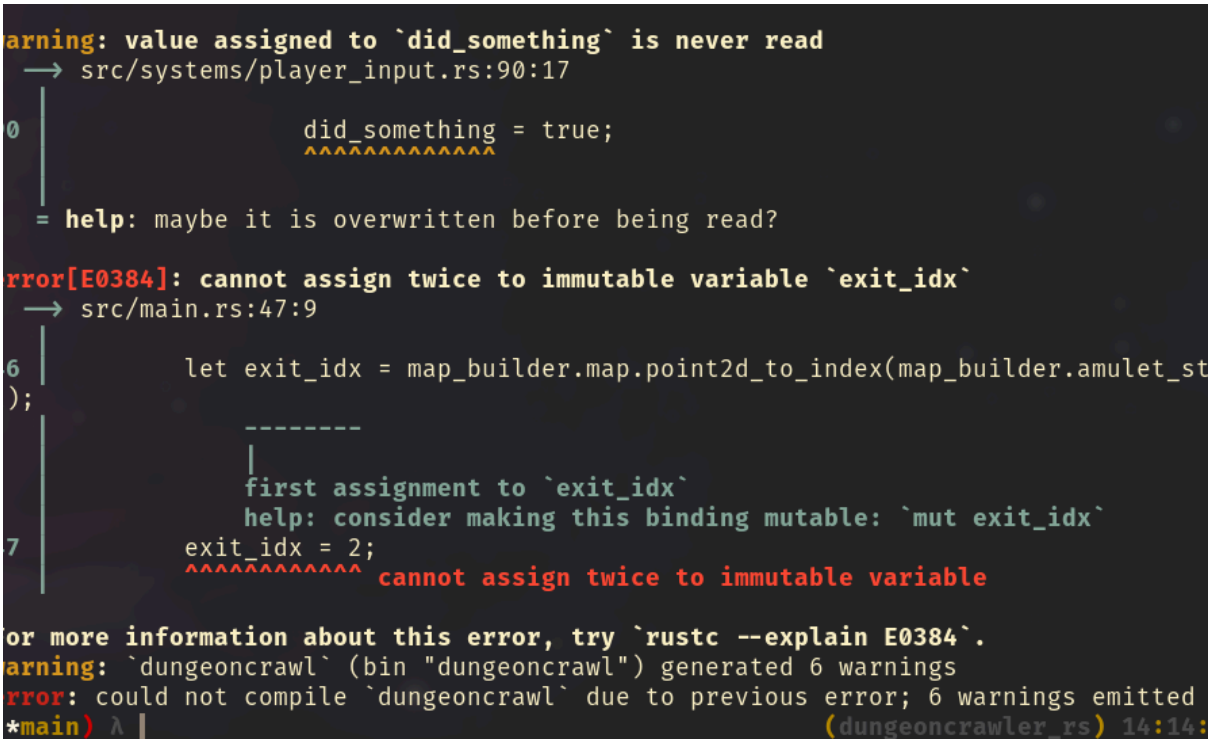
Rust jest całkiem nowym językiem. Pierwsza jego oficjalna wersja została wydana w 2010 roku. Jest to język wieloparadygmatowy i ogólnego przeznaczenia. Fascynującym faktem o języku Rust jest to, że nie wynajduje koła na nowo. Wręcz przeciwnie, zapożycza on wiele konceptów z różnych języków, np:

Abstract Machine Model	C
Data Types	C, SML, OCaml, Lisp, Limbo
Optional Bindings	Swift
Hygienic Macros	Scheme
Functional Programming	Haskell, OCaml, F#
Attributes	ECMA-335
Memory Model / Management	C++, ML Kit, Cyclone
Type Classes	Haskell
Channels & Concurrency	Newsqueak, Alef, Limbo
Message Passing & Thread Failure	Erlang

źródło: https://devopedia.org/rust-language
Główne cechy języka (źródło https://en.wikipedia.org/wiki/Rust_(programming_language)):

- bezpieczeństwo pamięci,
- zarządzanie pamięcią poprzez Resource Acquisition Is Initialization,
- bezpieczeństwo współbieżności w koncepcji Ownership i Borrowing,
- system typów zwany cechami obsługujący mechanizm podobny do klas typów, inspirowany językiem Haskell.

Rust nie należy do prostych języków. Aby stać się biegłym w sztuce władania Rust'owym mieczem, należy poświęcićsporo czasu. W dzisiejszym rozumieniu nie znajdziemy tutaj "odśmiecacza" (w dalszej części tekstu będę operować angielską wersją tego słowa :)). Istnieje pojęcie 'static garbage collector' (ciekawskich zapraszam tu) i w pewnym sensie Rust również taki mechanizm posiada. Jest to nic innego jak kompilator, który dba i pomaga programiście zrozumieć często popełniane błędy (dangling pointer anyone?).

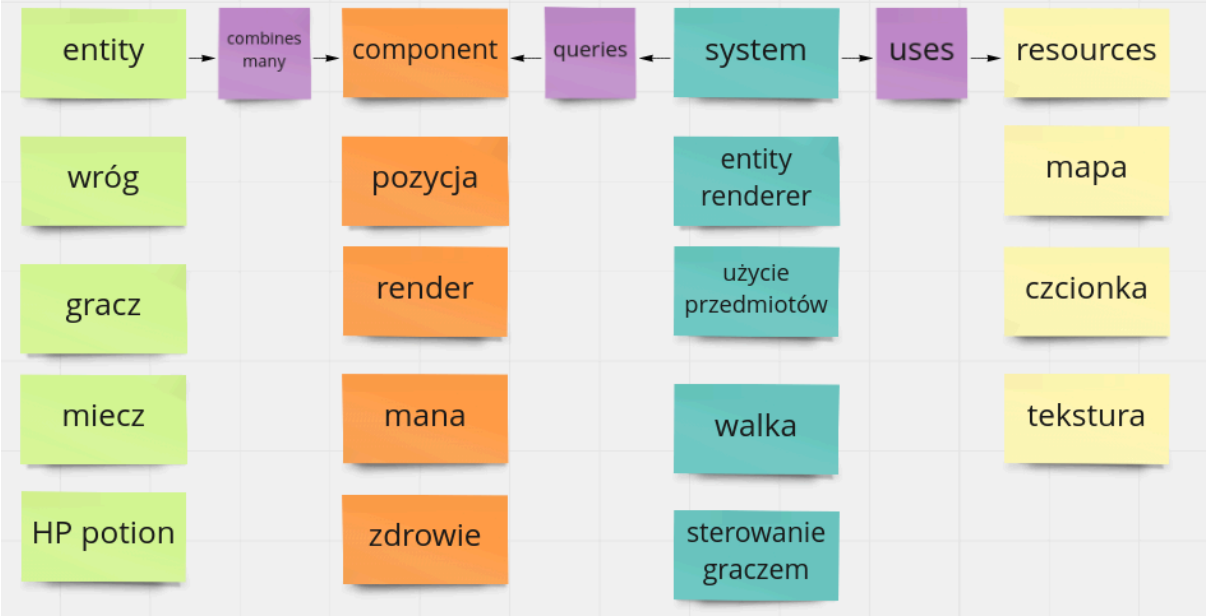


Przy pierwszym kontakcie z Rust'em kompilator jest dla nas wrogiem. Radzę mimo wszystko zaprzyjaźnić się z nim jak najszybciej, ponieważ narzędzie to jest nieocenionym przewodnikiem po świecie dobrych programistycznych praktyk.

Chapter 3: ECS

Pracując jako programista wielokrotnie czytałem kod w języku, którego nie znałem. Już nauka Rusta była trudna, a ECS okazał się potworem, o którym wcześniej nie słyszałem. Dowiedziałem się, że ECS jest wzorcem architektonicznym, który polega na modelowaniu poprzez kompozycje. Znalazłem również informację o tym, że w świecie game dev dziedziczenie jest sporym problemem. Tutaj mój mentalny model zaczął powoli się klarować. Jako web developer wiele razy miałem do czynienia z dziedziczeniem i wiem, jak problematyczne może ono być. Tym bardziej, że C++ pozwala na dziedziczenie po wielu bazowych klasach, co może doprowadzić do "diamond problem".

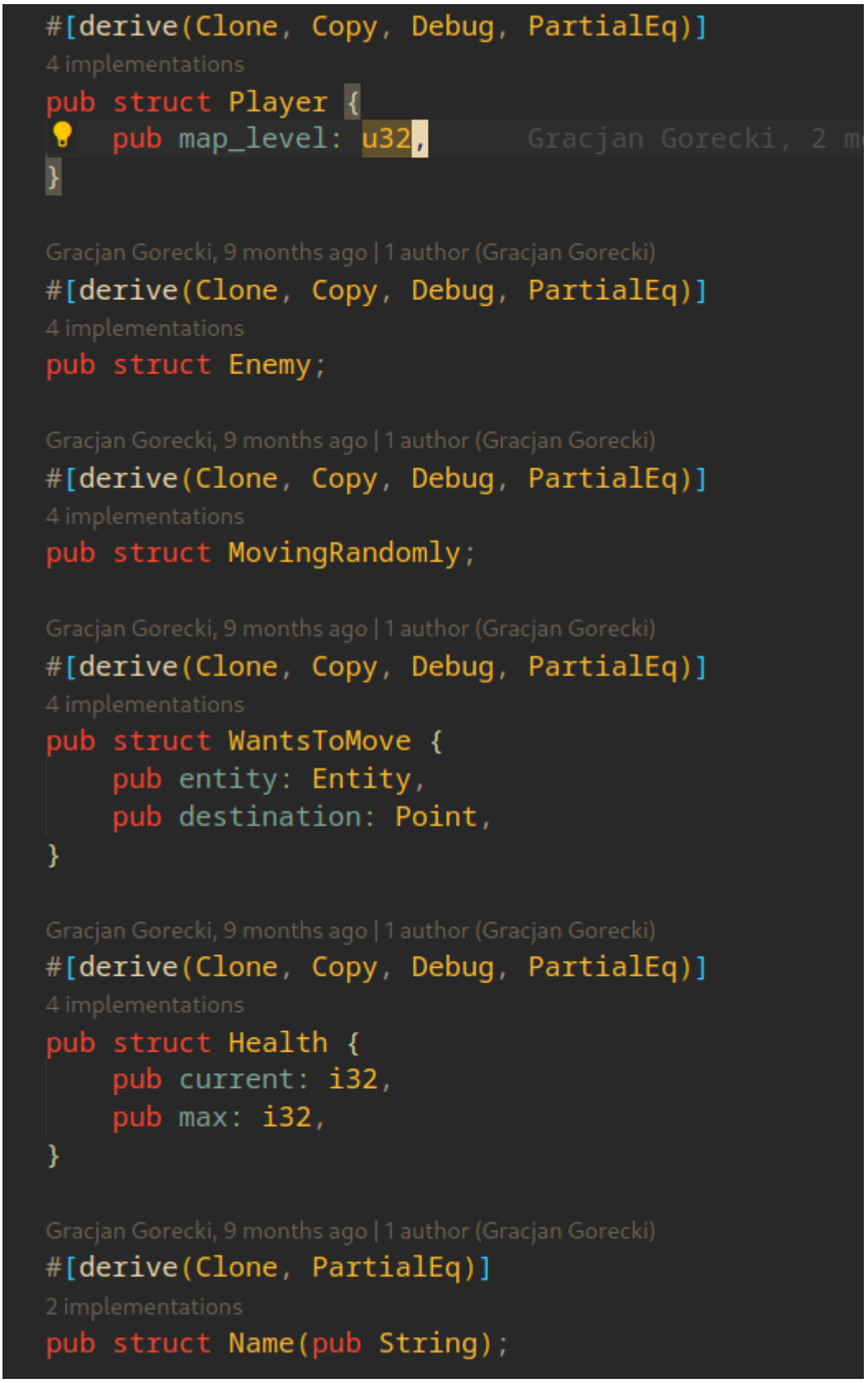
Aby dokładniej zrozumieć koncept ECS, sięgnąłem do mojego toolbox'a i wyciągnąłem z niego technikę wizualizacji (skorzystałem z miro i zwykłych kolorowych karteczek). Uważam, że to doskonałe narzędzie do zrozumienia nowych konceptów, ponieważ obrazy przemawiają do nas o wiele bardziej, niż sam tekst. Wynik mojej pracy widać poniżej:



- Entity: nie posiada logiki, zbudowany jest z komponentów. Dobrą analogią jest encja bazodanowa lub frontendowy store, przechowujący dane o np. graczu, przeciwnikach, przedmiotach. Przykładowa funkcja rejestracji wygląda następująco:

```rust
pub fn spawn_player(ecs: &mut World, pos: Point) {
    ecs.push(components: (
        Player { map_level: 0 },
        pos,
        Render {
            color: ColorPair::new(fg: WHITE, bg: BLACK),
            glyph: to_cp437('@'),
        },
        Health {
            current: 10,
            max: 10,
        },      Gracjan Gorecki, 2 months ago • added templat
        FieldOfView::new(radius: 8),
        Damage(1),
    ));
}
```

- Component: myślę, że przykład wyrazi więcej niż tysiąc słów, więc oto i on:



Jak widać, komponent jest niczym innym, jak typem danych zdefiniowanym przez programistę. Rust nie posiada klas, w zamian mamy do dyspozycji struktury. Jeżeli

jeszcze nie poznałaś/eś czym jest macro “derive”, dobre źródło znajduje się tutaj.

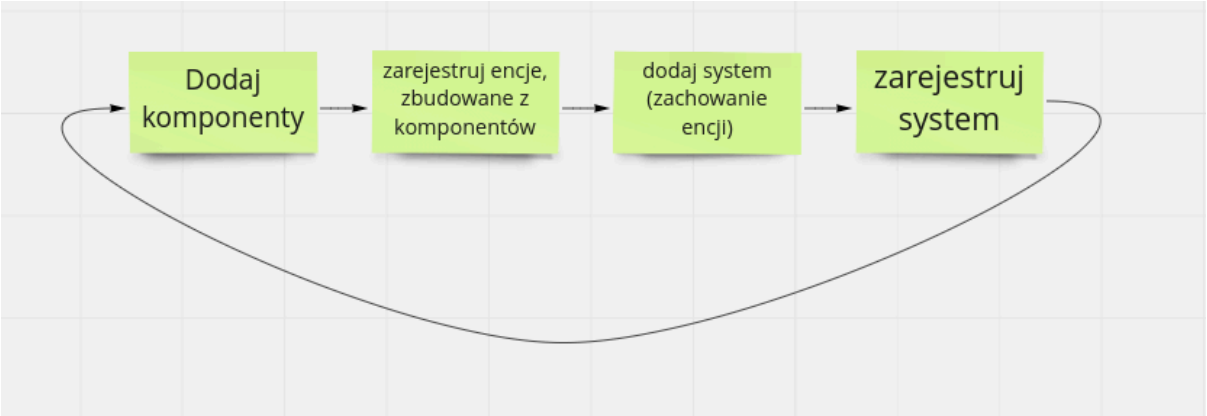
- System: tutaj zawarta jest logika gry. System tworzy zapytanie do “ecs” o komponenty, następnie zwracana jest lista encji, którą możemy modyfikować, np. zmienić położenie.

```
3  #[system]
4  #[read_component(Point)]
5  #[read_component(Render)]
6  #[read_component(FieldOfView)]
7  #[read_component(Player)]
8  pub fn entity_render(ecs: &SubWorld, #[resource] camera: &Camera) {
9      let mut renderables: Query<(&Point, &Render), …> = <(&Point, &Render)>::query();
10     let mut fov: Query<&FieldOfView, EntityFilterTuple<…>> = <&FieldOfView>::query().filter(component::<Player>());
11
12     let mut draw_batch: Reusable<DrawBatch> = DrawBatch::new();
13     draw_batch.target(console: 1);
14     let offset: Point = Point::new(camera.left_x, camera.top_y);
15     let player_fov: &FieldOfView = fov.iter(world: ecs).nth(0).unwrap();
16
17     renderables Query<(&Point, &Render), …>
18         .iter(world: ecs) impl Iterator<Item = (&Point, …)>
19         .filter(|(pos: &&Point, _)| player_fov.visible_tiles.contains(&pos)) impl Iterator<Item = (&Point, …)>
20         .for_each(|(pos: &Point, render: &Render)| {
21             draw_batch.set(pos: *pos - offset, render.color, render.glyph);
22         });
23     draw_batch.submit(z_order: 5000).expect(msg: "Batch error");
24 }
25
```

Tutaj po raz kolejny makra okazują się pomocne. Od linii 4 do 7 zdefiniowane są komponenty, do których funkcja “entity_render” ma mieć dostęp. Warto zauważyć, że jako pierwszy argument funkcji dostaniemy obiekt typu “SubWorld”. Nazwa wskazuje, że jest to część świata (bazy encji), która ogranicza się do typów zdefiniowanych przez makra. W linii 9 wywołana jest funkcja “query”, która zwróci listę encji zawierającą komponenty “Point” i “Render”, np. encja “Player” jest zbudowana z komponentów “Point” i “Render”, a więc encja “Player” zostanie zwrócona z zapytania. Analogią niech będzie pseudo kod SQL:
“Select * front Entities where entity.type includes Point **and** Render”.

- Resources: to dodatkowa pula encji, które powinny być dostępne wszędzie, niezależnie od “świata ecs”. Takim zasobem jest typ “Camera” ze zrzutu powyżej. Warto odnotować, że obiekty zarejestrowane jako “zasoby” są przekazywane przez argument funkcji, a nie przez wywołanie funkcji “query”.

Mając wyjaśnione podstawowe pojęcia “ECS”, algorytm tworzenia gry wygląda następująco:



Chapter 4: Webassembly

Webassembly jest dla mnie nową technologią, z którą do tej pory nie miałem do czynienia. Stwierdziłem więc, że postaram się zrozumieć sam koncept kompilacji i uruchomienia projektu w Rust. Natomiast zrozumienie działania webassembly w przeglądarce zostawiłem na później.

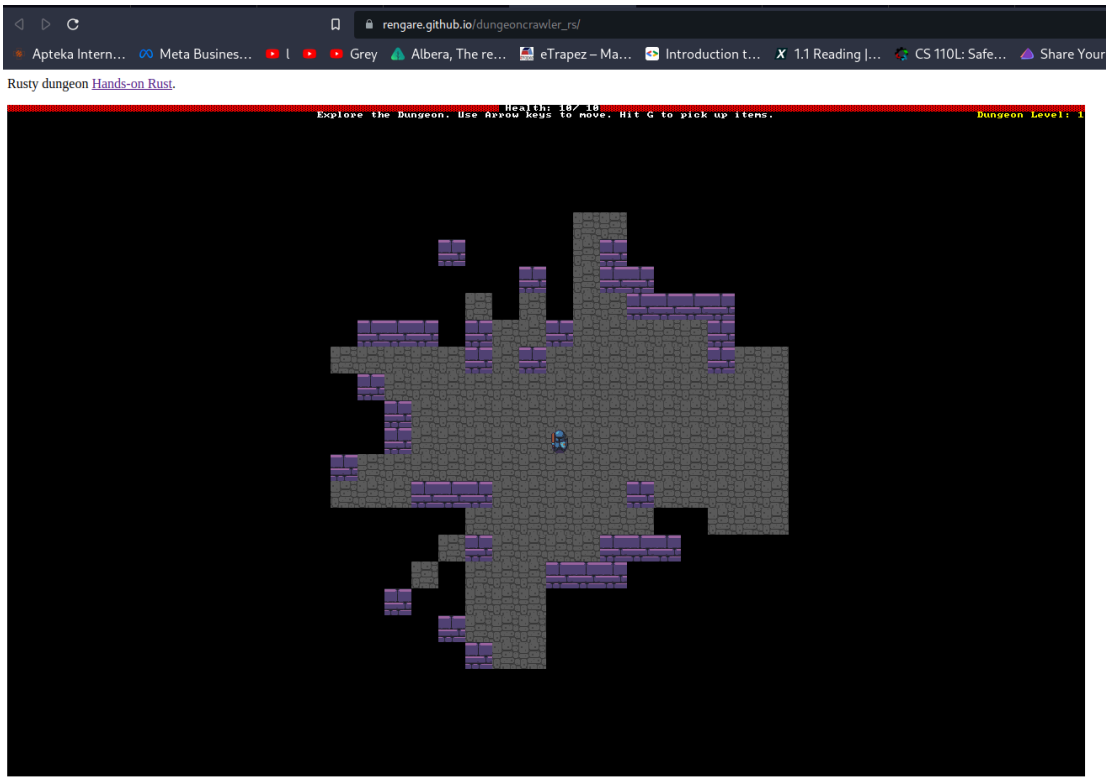
Pracę nad webassembly rozpoczynamy od gotowej gry z rozdziału 3, która wygląda tak:

Ekosystem języka Rust cały czas się rozwija. Jedną z rzeczy, która od razu zwróciła uwagę społeczności zainteresowanej językiem Rust, była możliwość kompilacji kodu do webassembly. Wynika to m.in z faktu, że Rust nie ma maszyny wirtualnej (dzięki czemu prościej jest o wsparcie nowej architektury) lub własnego środowiska uruchomieniowego (NodeJS) - samo środowisko nie jest przeszkodą, C# również można uruchomić w

webassembly, jednak wynikowy kod musi zawierać kod środowiska .net. Rust jest językiem kompilowanym do kodu maszynowego. Uogólniając, webassembly jest swego rodzaju kodem maszynowym, wbudowanym w przeglądarki (więcej tutaj). Biblioteki, których będziecie używać, również muszą wspierać budowanie do webassembly. Mówiąc inaczej, biblioteka, która będzie mieć zależności związane z konkretną architekturą lub systemem (np. taka, która działa tylko na Linux), nie skompiluje się do webassembly bez wcześniejszego przygotowania.

Tak więc pierwszą czynnością, jaką należy wykonać, jest instalacja odpowiedniego wsparcia do kompilacji:
“rustup target add wasm32-unknown-unknown”.
Następnie instalujemy
“cargo install wasm-bindgen-cli”.
Jest to narzędzie służące do komunikacji pomiędzy JavaScript i Rust. Pozwala na wywołanie funkcji JavaScript z poziomu Rust i odwrotnie (więcej tutaj).
Oczywiście kod webassembly uruchamia się w przeglądarce, dlatego potrzebujemy plik html. Aplikację budujemy za pomocą komendy:
“cargo build --target wasm32-unknown-unknown --release”, a następnie wiążemy kod wynikowy:
“wasm-bindgen target/wasm32-unknown-unknown/release/dungeoncrawl.wasm --out-dir ./docs --no-modules --no-typescript”.

Tak przygotowany folder “docs” możemy wrzucić do statycznego serwera plików. Ja wykorzystałem github pages (https://rengare.github.io/dungeoncrawler_rs/).



Skompilowana gra do webassembly, uruchomiona w przeglądarce

Dobrnęliśmy do końca, a nasz dzielny rycerz  uratował księżniczkę  . Nasz bohater pokonał Rust, ECS i webassembly. Jednak to nie koniec, bowiem przygoda wciąż trwa, a smoki i potwory żyją i mają się dobrze. Dzielny bohater przeznaczył swój skarb, aby zrobić upgrade swojego ekwipunku. Uzbrojony w “Bevy i Tokio” jest gotowy na nowe przygody.

Prologue

Webassembly pozwala na wykonywanie kodu napisanego w takich językach, jak C++, Rust czy C#. Dzięki udostępnionym API JavaScript, możliwa jest komunikacja między JS a Webassembly. Jednym z zastosowań jest możliwość portowania gier natywnych i uruchamianie ich w przeglądarce.
Przygoda z pisaniem gry nie należała do najprostszych. W głowie pojawiało mi się wiele pytań, np.

- czy moja znajomość Rust’a jest wystarczająca,
- czym jest ECS,
- jak pisać kod zgodny z ECS w Rust,
- jak powinna wyglądać struktura plików,
- jak działa webassembly i jak skompilować projekt.

Na szczęście wyżej wymieniona książka przeprowadziła mnie przez proces. Jednak nie zawsze było łatwo i musiałem włożyć sporo pracy w to, by zrozumieć nowe koncepty. W tym celu często korzystałem z wizualizacji. Swoim artykułem pragnę zachęcić Ciebie czytelniku do przetestowania języka Rust technologii webassembly oraz eksploracji nieznanych tematów ;)

Bio

Inżynier specjalizujący się w systemach informatycznych, głównie internetowych. Ukończył PWSZ w Nysie oraz Politechnikę Wrocławską. Komercyjną przygodę rozpoczął 8 lat temu, jednak samą informatyką interesuje się od najmłodszych lat. Na co dzień pracuje w Synergy Codes jako frontend developer. Jego hobby to informatyka oraz nauka w rozumieniu ogólnym.