NEURO-SYMBOLIC AI: KOOPERACJA LOGIKI Z SIECIAMI NEURONOWYMI

Index: 285358
www.programistamag.pl
Magazyn programistów i liderów zespołów IT

programista

4/2026 (125)
Cena 33.90 zł (w tym VAT 8%)

ESP-NOW

Protokół komunikacyjny dla drona

JUŻ W EMPIKU

ISSN 2084-9400 04

- Komputer niebinarny. Czy to jest możliwe?
- Programista wspomagany sprzętowo
- Czy można zastąpić płatne AI lokalnymi?
- Własny bot na Discordzie
- ECS w silnikach gier i symulacji
- Strefa CTF

PERFECTION IN PROTECTION LICENSING SECURITY
WIBU SYSTEMS

ECS w silnikach gier i symulacji

Za każdym razem, gdy z wielką pompą ogłaszana jest nowa gra klasy AAAA, na kolejnej wersji Nierzeczywistego® silnika, w szumie prasowym przewija się między innymi zapewnienie, że dzięki nowej architekturze software (i przy wydatnym wsparciu blaskomiotnego hardware firmy Zawiść™) na ekranie zobaczymy nieprzeliczone rzesze bytów „bez najmniejszego spadku płynności". Jest w tym pewna przesada, ponieważ sam wzorzec encja–komponent–system (ang. Entity–Component–System, ECS) niczego nie przyspiesza: o zysku decyduje rozmieszczenie danych w pamięci i to, czy czas klatki ogranicza procesor. W niniejszym artykule opisano, czym ECS jest w praktyce, jak silniki przechowują komponenty i planują pracę systemów oraz kiedy lepiej po ten wzorzec nie sięgać.

DLACZEGO ECS?

W zamieszłych czasach krążyło po IRC-u i Usenecie takie powiedzenie, że tak jak każdy prawdziwy mężczyzna powinien w życiu zasadzić drzewo, postawić dom i spłodzić syna, tak każdy prawdziwy informatyk powinien napisać kompilator, bazę danych i grę. Kiedy studiowałem, choć zdarzało się popełnić pierwsze dwa, to i tak wszyscy chcieli pisać to trzecie. Było to w czasach, gdy głównym produktem małego, lokalnego studia Techland był program „ English Translator" (bo tak żartobliwie nazwano ten zaawansowany generator lolcontentu), a gaming wydawał się czymś fajnym i romantycznym. Współcześnie mamy z jednej strony mało romantyczną, ogromną, bogatą gałąź przemysłu rozrywkowego, a z drugiej mnóstwo niewielkich „indikowych" studiów, mniejszych i większych projektów OSSW i pojedynczych hobbystów, którzy robią fajne rzeczy. I mali, i duzi korzystają jednak z podobnego zestawu narzędzi i technik.

W oprogramowaniu, gdzie setki albo miliony bytów zmieniają się w czasie rzeczywistym – w grach, symulacjach fizycznych, wizualizacjach, robotyce czy środowiskach z wieloma agentami – klasyczne drzewo obiektów z dziedziczeniem i rozproszonymi polami szybko przestaje być wygodne. Najpierw bolą iteracje: każda klatka animacji to przejście po strukturach rozrzuconych w pamięci, a nie po zwartych tablicach danych. Potem bolą zmiany w projekcie-grze: nowy typ „potworka" dokleja się do hierarchii klas albo wymusza refaktoryzację w miejscach, które wcale nie powinny wiedzieć o nowej mechanice. Układ encja–komponent–system (ang. *Entity–Component–System*, ECS) to odpowiedź inżynierska, która oddziela tożsamość bytu od jego atrybutów i od reguł przetwarzania. Dane trzymane są raczej w układzie struktura tablic (ang. *Structure of Arrays*, SoA) niż tablica struktur (ang. *Array of Structures*, AoS): zamiast jednej tablicy rekordów „cały byt naraz" (w środku rzeczy takie jak pozycja, prędkość, charyzma, punkty życia i mnóstwo innych pól obok siebie), mamy osobno ciągłą tablicę pozycji wszystkich bytów, osobno ciągłą tablicę prędkości itd., a ten sam indeks w każdej z nich oznacza ten sam byt; system liczący np. ruch może wtedy przeiterować pętlą po jednym typie danych naraz, zamiast odwiedzać kolejne pełne obiekty rozrzucone po pamięci. Nie oznacza to porzucenie czy walkę z paradygmatem programowania obiektowego – to świadomy kompromis pod konkretny profil obciążenia i czytelności kodu w dużych, interaktywnych systemach.

W poprzednim numerze „Programisty" ukazał się mój artykuł pt. „Randka z LOLA". Opisane w nim narzędzie bierze surowe dane wysokościowe z badań powierzchni Księżyca, wybiera z nich najbardziej urozmaicony fragment terenu i zapisuje go jako trójwymiarową siatkę w pliku glTF (otwarty format zapisu i wymiany scen oraz modeli 3D; jego binarny wariant ma rozszerzenie `.glb`). Samo narzędzie było zaprojektowane oszczędnie: orkiestrator (ang. *orchestrator*) w stylu `run()`, czyli po prostu uruchamiający po kolei poszczególne kroki, moduły wczytujące dane i generację glTF: jeden czytelny potok przetwarzania (ang. *pipeline*) od wejścia do pliku wyjściowego. Jego wizualizację trzeba było oglądać przy pomocy zewnętrznego viewera. Moim ostatecznym celem jest jednak „latanie" nad powierzchnią Księżyca na własnym silniku – żeby była to prawdziwa wycieczka – tytułowa „randka" po modelu w miejsce statycznego podglądu pliku. Zmierza to w kierunku, który coraz bardziej przypomina prostą interakcję w czasie rzeczywistym, szkic gry – mamy scenę, kamerę, ruch, kilka rzeczy, które mogłyby żyć obok siebie w tej samej przestrzeni. Właśnie do tego może przydać się ECS – gdy wokół kamery pojawia się coś więcej, niż jedna siatka (ang. *mesh*) i jedna ścieżka skryptu, architektura ECS staje się naturalnym językiem na opisanie świata, który co klatkę się przesuwa, oświetla i przelicza, zamiast raz przejść drogę od wczytania surowych danych do zapisania pliku wynikowego.

ECS – DEFINICJA OPERACYJNA

W wielu tutorialach ECS jest opisywane jako „nowoczesniejsze" programowanie obiektowe (ang. *Object-Oriented Programming*, OOP). Ta wygodna metafora jest myląca, jeśli traktować ją dosłownie. W klasycznym OOP obiekt gry to zwykle instancja klasy, która łączy dane z zachowaniem: ma pola pozycji, prędkości, grafiki i metodę w stylu `update()`, która w każdej klatce sama siebie przelicza; nowe cechy dokleja się dziedziczeniem albo coraz grubszą klasą bazową. W ECS te role się rozdzielają: encja to tylko tożsamość, komponenty to same dane, a systemy to osobne przebiegi kodu, które wybierają sobie ze świata wszystkie byty spełniające warunek (np. mają pozycję i prędkość) i aktualizują je hurtem – kompozycja (ang. *composition*)

zamiast drzewa klas, zapytanie (ang. *query*) zamiast „każdy obiekt po kolei woła swoją metodę". To inny, niż w OOP podział pracy: osobną tożsamość bytu w świecie, jego atrybuty zapisane jako dane oraz osobne procedury, które te dane przekształcają według reguł. Silnik ECS dostarcza narzędzi, które pozwalają to robić wydajnie i przewidywalnie – o szczegółach dalej, a na początek operacyjna definicja trzech słów z akronimu.

Encja (ang. *entity*) to w praktyce sam identyfikator bytu w świecie – nie klasa z metodami, nie paczka pól. Nie ma w sobie pozycji ani grafiki, i w ogóle żadnych danych; mówi tylko: „w świecie jest byt o takim numerze". W wielu silnikach ten numer to para: indeks slotu (ang. *slot* – stałe miejsce, wiersz w tablicy świata) oraz generacja (ang. *generation* – licznik, ile razy to miejsce było już ponownie zajęte po usunięciu poprzedniego bytu).

Wyobraźmy sobie slot o indeksie 7. Tworzymy pocisk – dostaje identyfikator (7, 0). Pocisk znika, miejsce 7 wraca do puli wolnych slotów. Później w slocie 7 pojawia się skrzynia – dostaje identyfikator (7, 1): ten sam indeks, ale wyższa generacja. Zapis (7, 0) nie powinien już nic zmieniać w świecie; środowisko wykonawcze (ang. *runtime*) może odrzucić takie nieaktualne ID, zanim kod przypadkiem potraktuje skrzynię jak wciąż żyjący pocisk. To ochrona przed pomyłką w logice gry – nie to samo co wiszący wskaźnik (ang. *dangling pointer*) w C++, gdzie cieknie nam pamięć, ale gorsza obawa: czy uchwyt wciąż opisuje ten sam byt, co chwilę temu.

Komponent (ang. *component*) to zwykle zwykła struktura danych, na przykład pozycja w przestrzeni, prędkość, promień kolizji, znacznik „chce skoczyć" (krótkotrwała informacja od wejścia – gracza: ten byt ma skoczyć w tej klatce – odczytuje ją system ruchu i zwykle od razu usuwa). Opisuje jeden aspekt encji; wiele komponentów składa się na to, czym dany byt jest w danej chwili. W języku Rust naturalnie chce się, żeby takie struktury (`struct`) były tanie w kopiowaniu tam, gdzie to ma sens – czyli miały cechy (ang. *trait* – nazwany zestaw zachowań, który typ deklaruje, że spełnia) `Copy` (kopia wartości w miejscu, bez alokacji) i `Clone` (jawne klonowanie przez `.clone()`, także gdy wymaga alokacji). Przy wielowątkowości chcielibyśmy, żeby typy były świadomie oznaczone jako `Send` (można bezpiecznie przenieść do innego wątku) i `Sync` (można bezpiecznie współdzielić między wątkami przez referencje), jeśli planista (ang. *scheduler* – moduł silnika ECS, który w każdej klatce ustala kolejność systemów i decyduje, które mogą działać równolegle) ma przenosić pracę między rdzeniami procesora. Logika „co z tym zrobić" nie mieszka w komponencie jak w metodzie `update()` klasy bazowej – komponent jest raczej kolumną w arkuszu symulacji.

System (ang. *system*) to kod, który działa na zbiorze encji spełniających warunek w stylu „ma komponenty A i B, nie ma C", albo mających znacznik danej fazy. Każdy system deklaruje z góry, które komponenty tylko czyta, a które zapisuje, i tej deklaracji się trzyma. Z tych deklaracji planista odczytuje, które systemy nie sięgają po te same dane i mogą pracować równolegle, a my zyskujemy jasną odpowiedź, kto w danej klatce zmienia którą wartość. Dlatego stan globalny – zmienna albo obiekt, do którego zagląda i w którym pisze każdy, kto ma ochotę – psułby ten układ: dane zmieniałyby się poza deklaracją, więc ani planista, ani autor testu nie miałby już z czego wnioskować. Zamiast „każdy obiekt sam się aktualizuje" dostajemy jawne przebiegi, np. najpierw wejście, potem fizyka, na końcu animacja – dokładny układ ustala projekt i planista silnika, a nie przypadkowa kolejność listy wskaźników.

ECS A PROJEKTOWANIE ZORIENTOWANE NA DANE

Dlaczego więc nie programowanie obiektowe? Oczywiście obiektówka w grach działała latami i nadal działa w wielu miejscach. Kłopot zaczyna się, gdy w każdej klatce trzeba odwiedzić tysiące instancji rozrzuconych po stercie (ang. *heap*): pozycja jednego aktora leży obok zupełnie innych pól niż pozycja następnego, a procesor bardzo chciałby czytać pamięć w zwartych porcjach. To jeden z aspektów zależności od sprzętu, a konkretnie fragmentacji pod kątem pamięci podręcznej (ang. *cache*) – wrócimy do niego w dalszej części artykułu.

Drugą bolączką są głębokie hierarchie dziedziczenia. Klasyczny problem diamentu (ang. *diamond problem* – gdy klasa dziedziczy z dwóch gałęzi, które same dziedziczą z tej samej bazy, i nie wiadomo, którą ścieżką „dojść" do wspólnego przodka przy wywołaniach metod) i przechodzenie po drzewie klas przy każdej nowej mechanice (latanie, pływanie, bycie skrzynią i jednocześnie przyciskiem) prowadzą do plątaniny zależności. Często wychodzi z tego antywzorzec architektoniczny, wariacja na temat God Object. Potworek obiektowy (ang. *monster object*): jedna klasa bazowa wie za dużo, a każda pochodna dokleja kolejne macki, kolce, pazury, flagi i nadpisania. Zmiana typu nadrzędnego potrafi rozchwiać połowę projektu – krucha klasa bazowa w czystej postaci.

Trzeci temat to stan ukryty w polach prywatnych i efektach ubocznych wywołań: trudniej wtedy odtworzyć deterministyczną symulację, zreprodukować klatkę w debuggerze albo napisać test, który sprawdza tylko jedną regułę bez postawienia całego świata. ECS nie rozwiązuje tych problemów magią, ale wymusza inny nawyk: najpierw widzimy dane i przebieg systemów, dopiero potem „kim jest" encja w sensie narracyjnym.

Projektowanie zorientowane na dane (ang. *Data-Oriented Design*, DOD) mówi w uproszczeniu: najpierw planujemy przepływ danych i transformacje, a dopiero potem ubieramy to w typy i moduły. ECS jest jedną z popularnych implementacji tego podejścia w silnikach gier: świat to zestaw tabel, a klatka symulacji to seria przejść, które je aktualizują. Ograniczenia sprzętu (jak pamięć, pamięć podręczna, potok procesora) pomagają zrozumieć, dlaczego DOD w ogóle się pojawia, ale same z siebie nie dowodzą, że „ECS zawsze szybszy". Do tematu pomiarów czasu i porównania wydajności (ang. *benchmarks*) jeszcze wrócimy, na razie pozostańmy przy definicji i przyjrzyjmy się jej prostej implementacji. Kod pokazuję w języku Rust, jako tym, który wyjątkowo wyraźnie obnaża napięcia między wzorcem ECS a regułami bezpiecznego dostępu do danych, o których będzie mowa dalej. Znakiem rozpoznawczym Rust jest model własności i pożyczania (ang. *ownership* i *borrowing*) – kompilator pilnuje już na etapie budowania, kto i jak długo może dane czytać, a kto je zmieniać. W Listingu 1 pokazano tylko suchy, czytelny kontrakt jako ilustrację podziału ról. W prawdziwym ECS tablice tych struktur żyją w magazynie komponentów (ang. *storage*) wybranym przez bibliotekę, a funkcja jest rejestrowana u planisty wraz z informacją, jakie komponenty czyta i które zapisuje. Znających Rust z góry przepraszam za dość obszerne komentarze w kodzie.

Listing 1. Encja, komponenty pozycji i prędkości oraz funkcja integrate_motion jako najprostszy system ruchu

```
/// Id encji -  w prawdziwym silniku często
/// (index, generation).
// #[derive(...)] zleca kompilatorowi
// wygenerowanie standardowych cech dla typu.
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub struct Entity(pub u32);

/// Komponent: tylko dane.
#[derive(Clone, Copy, Debug, Default)]
pub struct Position {
    pub x: f32,
    pub y: f32,
    pub z: f32,
}

#[derive(Clone, Copy, Debug, Default)]
pub struct Velocity {
    pub dx: f32,
    pub dy: f32,
    pub dz: f32,
}

/// System w najprostszym ujęciu: funkcja
/// na parach (Position, Velocity).
/// `&mut` daje prawo zapisu (tu zmieniamy
/// pozycję), samo `&` dawałoby tylko odczyt.
pub fn integrate_motion(
    pos: &mut Position,
    vel: Velocity,
    dt: f32,
) {
    pos.x += vel.dx * dt;
    pos.y += vel.dy * dt;
    pos.z += vel.dz * dt;
}
```

Warto od razu rozróżnić dwie rzeczy, które w dyskusji bywają mylone. Funkcja `integrate_motion` z Listingu 1 może być czystym krokiem na wycinku danych: dostaje bufory, liczy wynik, koniec. System w silniku ma dodatkowo miejsce w grafie planisty (ang. *scheduler graph* – uporządkowany układ systemów pokazujący, co musi wykonać się przed czym, a co może iść równolegle), dostęp do zasobów świata (czas, losowość z kontrolowanego źródła, uchwyt do fizyki itp.), sąsiedztwo z innymi systemami i czasem wyłączność na cały stan, gdy trzeba coś przearanżować bez wyścigów. Ten drugi poziom to już nie tylko wzorzec z podręcznika, lecz konkretna polityka silnika (a czasem i samej gry).

ECS A CPU

Przyjrzyjmy się teraz aspektowi sprzętowemu – nie po to, by ogłaszać „ECS == szybciej”, tylko by zrozumieć, dlaczego układ danych w ogóle ma znaczenie dla procesora. Ułatwi nam to później orientację w magazynie komponentów i w zapytaniach (ang. *query*), ale będzie to tylko model mentalny, a nie *benchmark*.

Jak powszechnie wiadomo, procesor jest szybki, a pamięć operacyjna wolna, zaś odległość w hierarchii pamięci (pamięć podręczna L1 i L2 przy rdzeniu, współdzielona L3 i dopiero na końcu RAM) robi różnicę rzędu wielkości. Procesor nie ściąga sobie pojedynczych bajtów „na żądanie” w izolacji – przenosi blok o rozmiarze linii pamięci podręcznej (ang. *cache line*), zwykle jakąś „okrągłą” liczbę, np. 64 bajty. Jeśli następne dane leżą obok w RAM, trafiają do tego samego bloku i kolejne odczyty są tanie. Jeśli kod skacze po pamięci co kilka kilobajtów, każdy krok kończy się nietrafieniem w *cache* (ang. *cache miss*) i czekaniem – potok instrukcji stoi, a koszt opóźnienia jest wyższy, niż koszt samej arytmetyki. W kontekście sprzętowym mówi się wtedy o lokalności przestrzennej (ang. *spatial locality*): im bardziej sekwencyjny i przewidywalny dostęp do sąsiednich adresów, tym lepiej wykorzystuje się to, co już wpadło do *cache*.

Jako człowiek siedzący całe życie w systemach wbudowanych, pozwolę sobie tutaj zrobić przystanek na marudzenie. Tzw. programiści aplikacji często lekceważą ograniczenia sprzętowe platformy, na której działają, i w ogóle to, jak działa sprzęt – zakładają, że zawsze można dołożyć pamięci albo procesora, więc prawdziwy (ciasny) budżet zostawia się tym od embeddedu. Gra albo inna („bardziej poważna”) duża symulacja szybko przypomina, że hierarchia pamięci nie rozróżnia „świata enterprise” od silnika na komputerze gracza. Zdarzało mi się jako architektowi nie raz walczyć z developerami, którzy za nic w świecie nie chcieli słyszeć słowa „optymalizacja”, bo przecież docelowa platforma jest potężna (co w języku marketingowym brzmiało jak „nieskończona”). Miałem nawet kolegę, który zagroził, że się zwolni, jeśli każą mu zoptymalizować kod pod względem zajętości pamięci. Oczywiście kazali i, co ciekawe, kolega rzeczywiście się zwolnił. Jak na ironię, poszedł do gamingu.

W klasycznym OOP aktorzy, pociski i skrzynie często żyją jako osobne obiekty na stercie. Każdy obiekt to paczka pól: pozycja, prędkość, grafika, AI, dźwięk itp. System ruchu potrzebuje pozycji i prędkości, ale przy tablicy struktur (AoS) przechodzi przez całe rekordy, żeby dotrzeć do kilku pól – albo skacze wskaźnikami między obiektami rozrzuconymi w pamięci. W obu przypadkach łatwo zmarnować linie *cache* na dane, których w tej pętli i tak się nie używa. To dokładnie ten sam argument co w sekcji o programowaniu obiektowym, tylko wyrażony językiem sprzętu. Nie ma jednej magicznej instrukcji „włącz ECS” w zestawie rozkazów procesora, od której sprzęt „sam przyspieszy”. Przyspieszenie daje układ danych i przebiegów, który ułatwia lokalność danych (ang. *data locality*). Komponenty jednego typu leżą w zwartych tablicach; system ruchu idzie po kolei po pozycjach i prędkościach, zamiast budzić całe „potworki”. W Listingu 2 pokazano ten sam przykład w dwóch wersjach pętli: najpierw skok po dużych rekordach, potem zwarty przebieg po równoległych kolumnach. Dlatego SoA jest tu korzystniejsze od AoS: w jednej pętli przechodzimy kolejno przez obie wąskie tablice – indeks po indeksie – więc w linii *cache* ląduje głównie to, czego system faktycznie używa, zamiast ściągać z RAM całe rekordy z polami zbędnymi w tej klatce.

Listing 2. Aktualizacja pozycji w pętli: tablica struktur (AoS) i osobne tablice pól (SoA)

```
// AoS: każdy krok dotyka całego rekordu aktora
for i in 0..n {
    actors[i].pos.x += actors[i].vel.dx * dt;
}

// SoA: kolejne i czytają sąsiednie komórki
// w dwóch tablicach
for i in 0..n {
    pos_x[i] += vel_dx[i] * dt;
}
```

Zamów Programistę w prenumeracie