AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ
KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Projekt dyplomowy

"Modelowanie trójskładnikowego ekosystemu z wykorzystaniem automatów komórkowych i badanie jego ewolucji"

„Modelling of a three-component ecosystem with the use of cellular automata and study of its evolution"

Autor: *Filip Rak*
Kierunek studiów: *Inżynieria Obliczeniowa*
Opiekun projektu: *prof. dr hab. Inż. Dmytro Svyetlichnyy*

1. Kraków, 2026

Spis Treści

2. Wstęp

2.1. Modelowanie systemów biologicznych

2.1.1. Systemy wieloagentowe

Przykładowa treść

3. Cele

3.1. Cel inżynierski

3.2. Cel modelowy

4. Architektura i technologie

4.1. Wybór technologii

W ramach realizacji projektu zdecydowano się na wykorzystanie nowoczesnego stosu technologicznego opartego na języku *C++ 23*. Wybór ten podyktowany był koniecznością zapewnienia najwyższej wydajności obliczeniowej przy jednoczesnym dostępie do nowoczesnych mechanizmów abstrakcji oraz dojrzałego ekosystemu bibliotek.

W projekcie wykorzystano następujące główne biblioteki [1]:

- ***EnTT*** – biblioteka dostarczająca wysokowydajne, nowoczesne i elastyczne narzędzia do programowania w paradygmacie ECS (ang. *Entity Component System*) [2],
- ***SFML*** – biblioteka multimedialna zapewniająca obsługę renderowania obrazu oraz zdarzeń wejściowych użytkownika [3],
- ***Dear ImGui*** – biblioteka do tworzenia graficznego interfejsu użytkownika w trybie *immediate-mode* [4],
- ***CLI11*** – biblioteka pełniąca funkcje parsera argumentów wiersza poleceń [5],
- ***nlohmann/json*** – narzędzie przeznaczone do przetwarzania danych w formacie JSON [6].

Do zarządzania procesem budowania aplikacji wykorzystano wieloplatformowe narzędzie ***CMake***, pozwalające na kompilacje kodu źródłowego niezależnie od platformy sprzętowej oraz systemu operacyjnego. Narzędzie to umożliwia ponadto scentralizowane zarządzanie i pobieranie zewnętrznych zależności wymaganych przez projekt.

W celu zapewnienia wysokiej jakości kodu źródłowego oraz zgodności z nowoczesnymi standardami programistycznymi wykorzystywano narzędzia analizy i automatyzacji:

- **LLVM Clang** – kompilator oferujący precyzyjne komunikaty o błędach będący podstawą innych wykorzystywanych narzędzi [7],

- **Clang-tidy** – narzędzie do statycznej analizy kodu (ang. *static analysis*), służące do wykrywania błędów logicznych, naruszeń standardów bezpieczeństwa oraz egzekwowania reguł nowoczesnego C++ zgodnych z *C++ Core Guidelines* [7] [8],
- **Clang-format** – parser zapewniający automatyczne i spójne formatowanie kodu [7],
- **Clangd** – serwer językowy (ang. *Language Server Protocol*) wspierający proces programowania przez inteligentne uzupełnianie kodu, nawigacje oraz analizę semantyczną w czasie rzeczywistym [7] [9].

Zastosowanie wymienionych narzędzi pozwala na zminimalizowanie długu technicznego, zwiększenie utrzymywalności (ang. *maintainability*) projektu oraz znaczące usprawnienie i przyspieszenie procesu wytwarzania i wdrażania oprogramowania.

4.2. Porównanie OOP i ECS

Projektowanie oprogramowania symulacyjnego może być realizowane w oparciu o różne paradygmaty programowania. W praktyce inżynierskiej, zwłaszcza w dziedzinie gier i symulacji czasu rzeczywistego szczególnie często stosuje się dwa podejścia: klasyczne programowanie obiektowe (ang. *OOP, Object-Oriented Programming*) [10] oraz architektury zorientowane na dane, wśród których szczególną popularność uzyskał wzorzec ECS (ang. *Entity–Component–System*), powszechnie obecny w nowoczesnych silnikach gier komputerowych, gdzie wysoka wydajność jest priorytetowa [11].

Wybór odpowiedniego modelu architektury ma istotny wpływ na stabilność i wydajność oprogramowania, w tym systemów symulacyjnych. Tradycyjne podejście obiektowe opiera się na odwzorowaniu bytów świata rzeczywistego na obiekty, co sprzyja enkapsulacji i czytelnemu modelowaniu skomplikowanych relacji. Jednakże, jak zauważa się w badaniach nad systemami czasu rzeczywistego [12], silne powiązanie danych z logiką wewnętrzną obiektów może prowadzić do usztywnienia struktury i utraty elastyczności, utrudniając rozwój, utrzymanie i ponowne wykorzystanie kodu w dynamicznie rozwijającym się oprogramowaniu [11] [12].

Znaczącą zaletą OOP jest niewątpliwie łatwość zarządzania logiką na poziomie pojedynczych obiektów. Niemniej jednak, w przypadku symulacji o dużej skali, gdzie uzyskanie wysokiej wydajności stanowi istotne wyzwanie, model ten ujawnia wady wynikające z nieoptymalnych wzorców dostępu do pamięci. Jak wskazują badania [11], enkapsulacja w OOP utrudnia optymalizację lokalności danych i efektywnego wykorzystania pamięci podręcznej procesora, co prowadzi do obniżenia wydajności przy dużej liczbie bytów.

W odpowiedzi na te ograniczenia, w nowoczesnych systemach coraz częściej stosuje się architekturę zorientowaną na dane realizowaną przez wzorzec ECS. Podejście to promuje pełną separacje stanu od zachowania. co pozwala uniknąć problemów wynikających ze sztywnych hierarchii typów oraz silnej enkapsulacji danych charakterystycznych dla OOP [12]. W przeciwieństwie do obiektowego modelu reprezentacji, w którym dane są rozproszone w wielu obiektach, ECS porządkuje je w struktury zoptymalizowane pod jednorodne przetwarzanie.

Jak wykazano w analizach wydajnościowych [11], taki układ umożliwia znacznie wyższy stopień zrównoleglenia obliczeń - szczególnie w symulacjach o dużej liczbie bytów - oraz poprawia lokalność danych (ang. *cache locality*), co przekłada się na bardziej efektywne wykorzystanie pamięci podręcznej procesora i wyraźny wzrost skalowalności. Badania te potwierdzają, że architektura ECS lepiej odpowiada potrzebom współczesnych symulacji, w

których krytyczne znaczenie ma efektywne przetwarzanie dużych, jednorodnych zbiorów danych.

W praktyce czysta architektura ECS, mimo wysokiej wydajności, bywa mniej intuicyjna w zarządzaniu unikalnymi podsystemami infrastrukturalnymi (np. obsługą okna graficznego czy integracją z systemem plików). Zadania te często naturalnie wpisują się w paradygmat obiektowy, przez co ich implementacja w czystym ECS może prowadzić do nadmiernego komplikowania kodu.

4.3. Architektura modularna oprogramowania

Projektowane oprogramowanie zostało oparte na dwupoziomowej architekturze hybrydowej, w której wyróżniono dwie główne warstwy [1]:

- **warstwę silnika –** dostarczającą w wysokopoziomowej formie niezbędną infrastrukturę niskopoziomową oraz izolującą logikę aplikacji od szczegółów implementacyjnych zaplecza graficznego (*ang. backend*);
- **warstwę aplikacji –** stanowiącą warstwę logiczną oprogramowania, realizowaną poprzez rejestrację systemów w silniku z wykorzystaniem wzorca wstrzykiwania zależności (ang. *dependency injection*) [13].

Zastosowane podejście pozwala na wykorzystanie zalet paradygmatu obiektowego w zarządzaniu zasobami systemowymi przy jednoczesnym zachowaniu wydajności modelu ECS (ang. *Entity Component System*) w przetwarzaniu populacji agentów.

4.3.1. Mechanizm serwisów i zarządzanie cyklem życia aplikacji

Fundamentem silnika [1] jest rejestr biblioteki EnTT (*entt::registry*) [2], który wykorzystując mechanizm kontekstu, pozwala na realizacje wzorca lokalizatora usług (ang. *service locator*) [14]. Dzięki temu każda usługa i system mają dostęp do globalnych zasobów bez konieczności stosowania antywzorca *singleton* [14] lub nadmiarowego przekazywania wskaźników przez kolejne poziomy hierarchii wywołań (ang. *dependency drilling*) [14].

W literaturze wzorzec *Service Locator* bywa określany jako kontrowersyjny, gdyż może prowadzić do ukrywania zależności oraz utrudniać testowanie w tradycyjnych aplikacjach biznesowych [14]. W przypadku silników gier i systemów symulacyjnych kompromis ten jest jednak powszechnie akceptowany [15], ponieważ wiele komponentów infrastrukturalnych (np. wejście, czas, renderowanie, zarządzanie zasobami) ma naturalnie globalny zakres oraz wspólny cykl życia powiązany z główną pętlą aplikacji. W takich architekturach jawne wstrzykiwanie zależności prowadziłoby do istotnego wzrostu złożoności kodu i obniżenia jego czytelności. Zachowując jednak odpowiednie zasady *Service Locator* staje się nie tylko wygodnym ale także najbardziej rozsądnym rozwiązaniem [2] [14].

Serwisy odpowiadają za przetwarzanie niskopoziomowych sygnałów takich jak zdarzenia użytej biblioteki graficznej (SFML [3]) na wysokopoziomowe obiekty dostępne dla systemów aplikacji [1]. Pozwala to na minimalizację powtarzalność kodu i poprawiając jego czytelność. Przykładem takiej relacji jest „InputSystem”, którego zadaniem jest wywoływanie odpowiedniej logiki na podstawie wejścia od użytkownik. System ten wykorzystuje usługę, „InputService”, udostępniającą wysokopoziomowy interfejs wejścia [1].

[[KOD ŹRÓDŁOWY: Fragment InputSystem::update()]]

Ze względu na to, że wiele serwisów wymaga wykonania dodatkowej logiki na początku lub końcu klatki (np. odczyt wejścia, czyszczenie buforów, renderowanie) - zdefiniowany został bazowy interfejs „IService". Dostarcza on metody wywoływane w sposób cykliczny przez silnik. Serwisy rejestrowane w silniku są przechowywane w kontenerze *std::vector<std::unique_ptr<IService>>*), *a* ich referencje następnie umieszczane są w kontekście rejestru [1]. Warto zaznaczyć, że w projektowanej architekturze serwisy nie powinny pobierać się nawzajem z kontekstu. Mechanizm, w tym wypadku, *Service Locator* przeznaczony jest wyłącznie do udostępniania infrastruktury warstwie aplikacji, natomiast serwisy powinny pozostawać względem siebie niezależne.

[[KOD ŹRÓDŁOWY: Engine::run]]

Takie podejście umożliwia sekwencyjne i przewidywalne zarządzanie stanem aplikacji w głównej pętli sterującej. Ponadto, separacja logiki od infrastruktury znacząco podnosi rozszerzalność (ang. *extensibility*) systemu [14] [13]. Zastosowanie abstrakcyjnych interfejsów sprawia, że warstwa zaplecza graficznego jest w pełni wymienna; przykładowo, zastąpienie biblioteki SFML innym rozwiązaniem wymaga jedynie implementacji nowej klasy serwisu, bez ingerencji w logikę systemów aplikacji. Dodanie nowej funkcjonalności, takiej jak moduł dźwiękowy czy alternatywny system renderowania, ogranicza się do implementacji interfejsu „IService" i rejestracji obiektu w silniku, bez konieczności modyfikacji istniejących mechanizmów sterujących [1].

Dodatkowym atutem wynikającym z zastosowania polimorficznego kontenera serwisów jest możliwość bardzo łatwego warunkowego i selektywnego zarządzania komponentami [13]. Pozwala to na uruchomienie aplikacji w różnych trybach operacyjnych (np. w trybie tekstowym pozbawionym interfejsu graficznego) poprzez pominięcie instancji serwisów odpowiedzialnych za renderowanie obrazu czy obsługę okna. W takim scenariuszu główna pętla sterująca pozostaje niezmieniona [1].

Modularność ta pozytywnie wpływa również na utrzymywalność (ang. *maintainability*) kodu, ułatwiając diagnostykę błędów poprzez izolację poszczególnych serwisów i eliminację ukrytych zależności globalnych [14].

4.3.2. Warstwa aplikacji, systemy i ich rejestracja

Podobnie jak w przypadku serwisów, wszystkie systemy logiczne posiadają wspólną bazową klasę abstrakcyjną „ISystem". Definiuje ona ustandaryzowany interfejs z metodą „update()", co umożliwia silnikowi traktowanie zróżnicowanych modułów logiki w sposób polimorficzny [1].

Fundamentalnym mechanizmem budowania logiki aplikacji jest wzorzec wstrzykiwania zależności (ang. *Dependency Injection*) [13]. Takie podejście zapewnia pełną separacje odpowiedzialności, ułatwia testowalność oraz zwiększa elastyczność konfiguracji, przy uzyskaniu luźnego powiązania (ang. *loose coupling*) pomiędzy silnikiem a konkretną logiką symulacji. Tworzenie systemów odbywa się w głównej klasie aplikacji na etapie inicjalizacji [1].

[[KOD ŹRÓDŁOWY: App::initSystems]]

Zastosowanie wspólnej klasy bazowej i polimorficznego kontenera w silniku (*std::vector<std::unique_ptr<ISystem>>*) niesie ze sobą te same co w przypadku serwisów w zakresie modularności. Możliwe jest selektywne zarządzanie funkcjonalnościami aplikacji poprzez warunkowe instancjonowanie konkretnych systemów. Przykładowo, w trybie konsolowym (*headless*), systemy takie jak „RenderSystem" po prostu nie są dodawane do silnika, co pozwala na nietrywialne przyspieszenie obliczeń bez modyfikacji logiki sterującej.

Istotną różnicą pomiędzy serwisami a systemami jest ich widoczność wewnątrz oprogramowania. Systemy nigdy nie są rejestrowane w kontekście rejestru. Wynika to z założenia, że system stanowi zamkniętą jednostkę wykonawczą, pracującą nieustannie w każdej kolejnej klatce, która nie powinna udostępniać swoich metod ani stanu innym elementom aplikacji. Taka restrykcyjna hermetyzacja zapobiega powstawaniu niekontrolowanych powiązań [1].

Zaimplementowanie systemów jako klas, a nie wolnych funkcji jest rozwiązaniem nietypowym na tle wielu popularnych implementacji architektury ECS, w których systemy są zazwyczaj implementowane jako funkcje, lambdy lub funktory operujące na danych komponentów [2] [15]. Pozwala ono jednak na wykorzystanie mechanizmu lokalnego buforowania danych (ang. *caching*). Ponieważ instancja systemu istnieje przez cały cykl życia aplikacji, może ona posiadać własne składowe (*ang. member variables*), które nie są częścią globalnego rejestru [1].

Podejście to rozwiązuje następujące problemy:

- **eliminacja kosztownych alokacji** - wybrane systemy obliczeniowe wymagają dodatkowej pamięci na operacje pośrednie. Zastosowanie klas pozwala na jednorazową alokację kontenerów (np. *std::vector::reserve*) na etapie konstrukcji systemu [1]. Dzięki temu unika się cyklicznego przydzielania i zwalniania pamięci w każdej klatce, co w systemach czasu rzeczywistego jest krytyczne z perspektywy wydajności [11];
- **unikanie zanieczyszczania rejestru** - dane, które nie są wykorzystywane przez więcej niż jeden element programu (np. tymczasowe bufory do obliczeń statystycznych w „TickLogSystem" czy uchwyty do zasobów graficznych w „RenderSystem"), pozostają prywatnymi składowymi klasy systemu. Umieszczanie ich w rejestrze i późniejsze pobieranie poprzez mechanizm *Service Locator* byłoby architekturalnie nieuzasadnione i wprowadzałoby niepotrzebny narzut logiczny [1];
- **jasność interfejsów -** rejestr pozostaje czytelnym zbiorem danych domenowych (agentów) oraz globalnych usług. Dzięki izolacji danych prywatnych wewnątrz klas systemów, programista ma pewność, że stan znajdujący się w rejestrze jest istotny dla całej aplikacji, a nie jest jedynie tymczasowym produktem ubocznym jednego z procesów [1].

4.3.3. Orkiestracja systemów symulacyjnych

Podczas dodawania pierwszych systemów symulacyjnych do projektu zidentyfikowano dwa istotne wyzwania:

- **złożoność** – pewne systemy zajmują się logiką aplikacji, inne zaś logiką symulacji. Instancja wszystkich tych systemów w głównej klasie aplikacji zmniejsza czytelność kodu;
- **brak kontroli nad główną pętlą** – ze względu na to, że główna pętla sterująca ukryta jest przed aplikacją w warstwie silnika, niemożliwym staje się wprowadzenie mechanizmów pozwalających na kontrolę prędkości symulacji, w tym mechanizmu

pauzy. Mechanizmy tę są bardzo przydatne zarówno z perspektywy programisty, w celu debugowania, jak i również z perspektywy użytkownika, w celu weryfikacji konkretnego modelu w trybie graficznym.

Rozwiązaniem powyższych problemów stało się wykorzystanie fakt, iż systemy są zdefiniowane jako klasy, co pozwala im na przechowywanie własnego stanu i posiadanie złożonej logiki wewnętrznej. Wprowadzono unikalny system – „SimRunnerSystem". Z perspektywy silnika system ten nie różni się kompletnie niczym od innych systemów aplikacyjnych, również implementując bazowy interfejs „ISystem". Pełni on jednak istotną rolę fasady oraz orkiestratora, co niesie ze sobą następujące korzyści:

- **enkapsulacja potoku symulacji** – system ten posiada własny wektor podsystemów symulacyjnych. Tworzy on wewnętrzną pętle, która – analogicznie do pętli głównej silnika – iteruje po wszystkich systemach biologicznych. Dzięki temu logika symulacji jest odseparowania od logiki infrastrukturalnej aplikacji;
- **autonomia czasu** – ponieważ system ten manualnie wywołuje swoje podsystemy biologiczne, możliwym stało się wprowadzenie logiki kontroli symulacji, w tym prędkości i pauzowania;
- **centralizacja zarządzania stanem** – jako zarządca, system ten bierze na siebie odpowiedzialność resetowania symulacji do stanu początkowego. Zabieg ten pozwala na zachowanie czystości w głównej klasie aplikacji.

5. Implementacja modelu ekosystemu

6. Metodologia testowa

7. Analiza wyników

Bibliografia

[1] F. Rak, „ecosystem-cc - repozytorium GitHub," [Online]. Available: https://github.com/Filip-Rak/ecosystem-cc. [Data uzyskania dostępu: 10 1 2026].

[2] M. Caini, „EnTT Documentation," [Online]. Available: https://skypjack.github.io/entt/. [Data uzyskania dostępu: 09 01 2026].

[3] L. Gomila, „Documentation for SFML 3.0.1," [Online]. Available: https://www.sfml-dev.org/documentation/3.0.1/. [Data uzyskania dostępu: 09 01 2026].

[4] O. Cornut, „Dear ImGui - GitHub Repository,” [Online]. Available: https://github.com/ocornut/imgui. [Data uzyskania dostępu: 09 01 2026].

[5] H. Schreiner, „Introduction - CLI11 Tutorial,” [Online]. Available: https://cliutils.github.io/CLI11/book/. [Data uzyskania dostępu: 09 01 2026].

[6] N. Lohmann, „Overview - JSON for modern C++,” [Online]. Available: https://json.nlohmann.me/api/basic_json/. [Data uzyskania dostępu: 09 01 2026].

[7] C. Lattner, „LLVM and Clang: Next Generation Compiler Technology,” 17 05 2008. [Online]. [Data uzyskania dostępu: 08 01 2026].

[8] B. Stroustrup i H. Sutter, „C++ Core Guidelines,” [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. [Data uzyskania dostępu: 08 01 2026].

[9] L. Foundation, „The LLVM Compiler Infrastructure,” [Online]. Available: https://clangd.llvm.org/. [Data uzyskania dostępu: 08 01 2026].

[10] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin i R. Yannow, „Seeking Grace: A New Object-Oriented Language for Novices,” w *SIGCSE ’13*, Denver, Colorado, 2013.

[11] A. Ambrosio, D. De Vinco, F. Foglia, C. Spagnuolo i V. Scarano, „The impact of ECS logic on parallel performance in agent-based model simulations,” w *BigHPC2025*, Turin, Italy, 2025.

[12] D. Wiebusch i M. E. Latoschik, „Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems,” w *Proceedings of the IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS 2015)*, Arles, France, 2015.

[13] R. C. Martin, Clean Architecture: A Craftsman’s Guide to Software Structure and Design, Boston: Pearson Education, 2017.

[14] R. Nystrom, Game Programming Patterns: Software Design for Games, Geneva, IL: Genever Benning, 2014.

[15] B. D. Team, „Bevy Engine Documentation - Resources,” [Online]. Available: https://bevy.org/learn/quick-start/getting-started/resources/. [Data uzyskania dostępu: 09 01 2026].