AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Projekt dyplomowy

Opracowanie środowiska obliczeniowego ECS (Encja-Komponent-System) do symulacji ekosystemu, modelowanie i analiza

Development of an ECS (Entity-Component-System) computing environment for ecosystem simulation; modeling and analysis

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

Kraków, 2026

Spis Treści

1. Wstęp

Złożone systemy biologiczne, takie jak ekosystemy, charakteryzują się silnymi sprzężeniami zwrotnymi, nieliniowością oraz wrażliwością na warunki środowiskowe. Analiza ich zachowania metodami analitycznymi jest w wielu przypadkach niewystarczająca lub wręcz niemożliwa. Z tego powodu w badaniach nad dynamiką populacji oraz zjawiskami ekologicznymi coraz częściej wykorzystuje się modele komputerowe i symulacje numeryczne, które pozwalają na obserwację emergentnych wzorców zachowania w kontrolowanych warunkach [1].

Jednym z podejść dobrze przystosowanych do opisu takich systemów są modele oparte na automatach komórkowych oraz systemach wieloagentowych. Automaty komórkowe umożliwiają dyskretną reprezentację przestrzeni i lokalnych reguł oddziaływania, natomiast modele agentowe pozwalają na odwzorowanie zróżnicowanych strategii przetrwania, zachowań oraz mechanizmów adaptacyjnych poszczególnych organizmów [2]. Połączenie tych dwóch perspektyw tworzy elastyczne środowisko do badania procesów ekologicznych, takich jak kształtowanie nisz, konkurencja o zasoby czy ewolucja cech dziedzicznych [1].

Z perspektywy inżynierskiej realizacja tego typu modelu wymaga wydajnego, skalowalnego i łatwo rozszerzalnego środowiska symulacyjnego. Wraz ze wzrostem liczby agentów oraz złożoności reguł dynamiki środowiska rosną wymagania obliczeniowe oraz potrzeba zachowania czytelnej, modularnej architektury oprogramowania [1]. Oznacza to konieczność zaprojektowania rozwiązania zdolnego do obsługi dużych populacji i złożonych map środowiskowych, a jednocześnie umożliwiającego swobodne modyfikowanie modelu oraz prowadzenie serii eksperymentów numerycznych bez ingerencji w kod źródłowy.

Niniejsza praca łączy oba te aspekty: z jednej strony koncentruje się na opracowaniu modelu ekosystemu opartego na automacie komórkowym i populacji agentów, z drugiej zaś na zaprojektowaniu i implementacji wieloplatformowego środowiska symulacyjnego, wykorzystującego nowoczesną architekturę oprogramowania opartą na wzorcu *Entity–Component–System* (ECS) [3]. Model ekosystemu uwzględnia mechanizmy gospodarki energetycznej, presji selekcji abiotycznej oraz adaptacji międzypokoleniowej.

1.1. Architektura Entity-Component-System

Architektura Entity–Component–System (ECS) stanowi wzorzec projektowy w którym następuje konsekwentne rozdzielenie stanu od zachowania. W odróżnieniu od klasycznego

podejścia obiektowego, w którym dane i logika są enkapsulowane w ramach pojedynczych obiektów, ECS organizuje system wokół przetwarzania danych [4].

Podstawową jednostką identyfikacji w ECS jest encja, rozumiana jako lekki identyfikator reprezentujący byt w świecie symulacji. Encje same w sobie nie posiadają stanu ani zachowania, a ich właściwości są określane przez przypisane komponenty – niezależne struktury danych opisujące pojedyncze aspekty stanu, takie jak pozycja, energia czy parametry środowiskowe. Systemy realizują logikę obliczeniową, operując na zbiorach encji posiadających określone zestawy komponentów [4, 5].

Taki podział sprzyja jednorodnemu przetwarzaniu dużych zbiorów danych i w efekcie lepszemu wykorzystaniu pamięci podręcznej procesora oraz ograniczeniu kosztów pośrednich związanych z wywołaniami wirtualnymi i rozproszonym dostępem do danych. Z tego względu podejście to znalazło szerokie zastosowanie w nowoczesnych silnikach gier komputerowych oraz systemach symulacyjnych czasu rzeczywistego, w których istotna jest wydajność i możliwość elastycznego rozszerzania logiki [4].

W kontekście systemów wieloagentowych ECS umożliwia naturalne modelowanie populacji złożonych z wielu bytów o zróżnicowanych cechach poprzez dynamiczne komponowanie encji z komponentów, bez konieczności tworzenia rozbudowanych hierarchii klas.

1.2. Automaty komórkowe

Automaty komórkowe stanowią klasę dyskretnych modeli dynamicznych, w których przestrzeń reprezentowana jest przez regularną siatkę komórek, a czas przebiega w krokach. Każda komórka posiada stan, a jego wartość w kolejnym kroku czasu zależy wyłącznie od aktualnego stanu tej komórki oraz stanów komórek w jej lokalnym sąsiedztwie, według ustalonej reguły przejścia. Dzięki prostocie lokalnych reguł oraz bogactwu możliwych zachowań emergentnych automaty komórkowe są powszechnie stosowane do modelowania zjawisk przestrzennych, takich jak rozprzestrzenianie się populacji, dyfuzja substancji czy dynamika ekosystemów [2].

W niniejszej pracy automat komórkowy wykorzystywany jest do reprezentacji środowiska, natomiast organizmy modelowane są jako agenci osadzeni w tej przestrzeni.

1.3. Systemy wieloagentowe

Systemy wieloagentowe (ang. *multi-agent systems*) stanowią klasę modeli, w których złożone zachowanie całego układu wynika z interakcji wielu autonomicznych jednostek podejmujących

decyzje na podstawie lokalnych informacji. W kontekście ekologii rolę agentów pełnią zazwyczaj organizmy lub ich uproszczone reprezentacje, wyposażone w zestaw cech, strategii oraz reguł zachowania. Takie podejście umożliwia badanie zjawisk emergentnych, takich jak samoorganizacja, powstawanie struktur przestrzennych czy adaptacja do zmiennych warunków środowiskowych [1].

Zastosowanie modeli wieloagentowych w połączeniu z automatem komórkowym pozwala na naturalne odzwierciedlenie przestrzennej heterogeniczności środowiska oraz lokalnych interakcji między organizmami. W projektowanej w ramach niniejszej pracy symulacji agenci reprezentują organizmy podlegające presji selekcji abiotycznej i gospodarce energetycznej. Ich zachowanie determinowane jest przez genotyp, na który wpływ mają doświadczenia życiowe przodka, modelowane za pomocą mechanizmu plastyczności międzypokoleniowej [1].

1.4. Struktura pracy

W rozdziale drugim przedstawiono cel pracy, z podziałem na zadanie inżynierskie związane z budową środowiska symulacyjnego oraz zadanie modelowe dotyczące konstrukcji i analizy modelu ekosystemu. Rozdział trzeci poświęcono opisowi zastosowanego stosu technologicznego oraz zaprojektowanej architektury oprogramowania, ze szczególnym uwzględnieniem hybrydowego podejścia łączącego elementy programowania obiektowego z architekturą opartą na wzorcu ECS.

W rozdziale czwartym omówiono szczegółowo model ekosystemu, w tym dynamikę środowiska, agenta oraz mechanizmy adaptacji i dziedziczenia. Rozdział piąty dotyczy metodologii testowej, natomiast w rozdziale szóstym zaprezentowano i omówiono wyniki wybranych eksperymentów symulacyjnych. W rozdziale siódmym dokonano oceny modelu ekosystemu na podstawie uzyskanych wyników, a rozdział ósmy zawiera podsumowanie pracy oraz wnioski końcowe.

2. Cel pracy

Celem pracy jest opracowanie oraz analiza komputerowego modelu ekosystemu, a także implementacja środowiska symulacyjnego umożliwiającego prowadzenie eksperymentów numerycznych z jego wykorzystaniem.

W ramach realizacji celu sformułowano dwa główne zadania, obejmujące część inżynierską oraz modelowo-badawczą.

2.1. Zadanie inżynierskie

Zadanie inżynierskie polega na zaprojektowaniu i implementacji wieloplatformowego, wydajnego oraz modularnego środowiska symulacyjnego. Oprogramowanie powinno umożliwiać łatwe rozszerzanie logiki systemu oraz zapewniać szeroką konfigurowalność parametrów eksperymentów, pracę w trybie graficznym i konsolowym wspierającą automatyzację badań, a także możliwie deterministyczny przebieg symulacji w obrębie danej platformy sprzętowo-kompilacyjnej.

2.2. Zadanie modelowe

Zadanie modelowe polega na implementacji oraz analizie modelu ekosystemu opartego na automacie komórkowym i populacji agentów. Automat komórkowy służy do dyskretnej reprezentacji środowiska i jego parametrów abiotycznych, natomiast agenci stanowią autonomiczne byty podlegające mechanizmom gospodarki energetycznej oraz presji selekcji abiotycznej. Analiza obejmuje ocenę wpływu tych mechanizmów na kształtowanie nisz ekologicznych, różnicowanie genotypów oraz dynamikę migracji populacji.

3. Architektura i technologie

W rozdziale omówiono zastosowane technologie oraz architekturę oprogramowania symulacyjnego, ze szczególnym uwzględnieniem hybrydowego podejścia łączącego programowanie obiektowe z architekturą opartą na wzorcu ECS.

3.1. Wykorzystane technologie

Projekt realizowano w ramach 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 [3]:

- ***EnTT*** – biblioteka dostarczająca wysokowydajne, nowoczesne i elastyczne narzędzia do programowania w paradygmacie ECS [6],
- ***SFML*** – biblioteka multimedialna zapewniająca obsługę renderowania obrazu oraz zdarzeń wejściowych użytkownika [7],
- ***Dear ImGui*** – biblioteka do tworzenia graficznego interfejsu użytkownika w trybie *immediate mode* [8],
- ***CLI11*** – biblioteka pełniąca funkcje parsera argumentów wiersza poleceń [9],
- ***nlohmann/json*** – narzędzie przeznaczone do przetwarzania danych w formacie JSON [10].

Do zarządzania procesem budowania aplikacji wykorzystano wieloplatformowe narzędzie ***CMake***, pozwalające na kompilację 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 [11],
- ***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* [11, 12],

- ***Clang-format*** – parser zapewniający automatyczne i spójne formatowanie kodu [11],
- ***Clangd*** – serwer językowy (ang. *Language Server Protocol*) wspierający proces programowania przez inteligentne uzupełnianie kodu, nawigację oraz analizę semantyczną w czasie rzeczywistym [11, 13].

Zastosowanie wymienionych technologii pozwala na ograniczenie długu technicznego, zwiększenie utrzymywalności projektu oraz usprawnienie procesu wytwarzania i wdrażania oprogramowania.

3.2. 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*) [14] oraz architektury zorientowane na dane. Wśród tych architektur szczególną popularność uzyskał wzorzec ECS powszechnie obecny w nowoczesnych silnikach gier komputerowych, gdzie wysoka wydajność jest priorytetowa [4].

Wybór odpowiedniego modelu architektury ma istotny wpływ na stabilność i skalowalność 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 [5], silne powiązanie danych z logiką wewnętrzną obiektów może prowadzić do usztywnienia struktury i utraty elastyczności, utrudniając rozwój, utrzymanie i ponowne wykorzystywanie kodu w dynamicznie rozwijającym się oprogramowaniu [4, 5].

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 [4], 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 separację stanu od zachowania. co pozwala uniknąć problemów wynikających ze sztywnych

hierarchii typów oraz silnej enkapsulacji danych charakterystycznych dla OOP [5]. 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 [4], 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 wskazują, 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.

W związku z tym w projektowanej aplikacji przyjęto architekturę hybrydową, gdyż najlepiej łączy wysoką wydajność przetwarzania i skalowalność charakterystyczną dla ECS z czytelną i naturalną obsługą podsystemów infrastrukturalnych oferowaną przez paradygmat obiektowy.

3.3. Architektura modularna oprogramowania

Projektowane oprogramowanie zostało oparte na dwupoziomowej architekturze hybrydowej, w ramach której wyróżniono dwie główne warstwy [3]:

- **warstwę silnika** – dostarczającą niezbędną infrastrukturę systemową 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*) [15].

Zastosowane podejście pozwala na wykorzystanie zalet paradygmatu obiektowego w zarządzaniu zasobami systemowymi przy jednoczesnym zachowaniu wydajności modelu ECS w przetwarzaniu populacji agentów.

3.3.1. Mechanizm serwisów i zarządzanie cyklem życia aplikacji

Fundamentem silnika [3] jest rejestr biblioteki EnTT [6], który wykorzystując mechanizm kontekstu, pozwala na realizację wzorca lokalizatora usług (ang. *service locator*) [16]. Dzięki temu każda usługa i system mają dostęp do globalnych zasobów bez konieczności stosowania antywzorca *Singleton* [16] lub nadmiarowego przekazywania wskaźników przez kolejne poziomy hierarchii wywołań (ang. *dependency drilling*) [16].

W literaturze wzorzec Service Locator bywa określany jako kontrowersyjny, gdyż może prowadzić do ukrywania zależności oraz utrudniać testowanie w tradycyjnych aplikacjach biznesowych [16]. W przypadku silników gier i systemów symulacyjnych kompromis ten jest jednak powszechnie akceptowany [17], 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 [6, 16]. Implementację wykorzystującą opisany mechanizm lokalizatora usług przedstawiono w fragmencie kodu 1.

```
const auto& input = m_registry.ctx().get< InputService >();
auto& camera      = m_registry.ctx().get< Camera >();

using namespace keyboard;

camera.keyboardMovementInput = { 0.f, 0.f };
if ( input.isDown( Key::W ) ) camera.keyboardMovementInput.y -= 1.f;
if ( input.isDown( Key::S ) ) camera.keyboardMovementInput.y += 1.f;
if ( input.isDown( Key::A ) ) camera.keyboardMovementInput.x -= 1.f;
if ( input.isDown( Key::D ) ) camera.keyboardMovementInput.x += 1.f;
```

Fragment kodu 1. Użycie wzorca service locator i serwisu
Źródło: opracowanie własne.

Serwisy odpowiadają za przetwarzanie niskopoziomowych sygnałów takich jak zdarzenia użytej biblioteki graficznej (SFML [7]) na wysokopoziomowe obiekty dostępne dla systemów aplikacji [3]. Pozwala to na minimalizację powtarzalności kodu i poprawia jego czytelność. We fragmencie kodu 1 przedstawiono przykładową taką relację. `InputSystem` wykorzystuje usługę `InputService` udostępniającą wysokopoziomowy interfejs wejścia [3]. Zadaniem tego systemy jest wywoływanie odpowiedniej logiki na podstawie danych wejściowych od użytkownika .

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`. Serwisy rejestrowane w silniku są przechowywane w kontenerze a ich referencje następnie umieszczane są w kontekście rejestru [3]. Pętla główna, przedstawiona we fragmencie kodu 2, zajmuje się cyklicznym wywoływaniem metod aktualizujących stan serwisów oraz systemów.

```
auto Engine::run() -> void
{
        auto& dispatcher = m_registry.ctx().get< entt::dispatcher >();
        while ( m_isRunning )
        {
                for ( auto& service : m_services ) service->beginFrame();
                for ( auto& system : m_systems ) system->update();
                for ( auto& service : m_services ) service->endFrame();

                dispatcher.update();
        }
}
```

Fragment kodu 2. Pętla główna silnika
Źródło: opracowanie własne.

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.

Takie podejście umożliwia sekwencyjne i przewidywalne zarządzanie stanem aplikacji w głównej pętli sterującej (fragment kodu 2). Ponadto, separacja logiki od infrastruktury znacząco podnosi rozszerzalność (ang. *extensibility*) systemu [15, 16]. 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 [3].

Dodatkowym atutem wynikającym z zastosowania polimorficznego kontenera serwisów jest możliwość bardzo łatwego warunkowego i selektywnego zarządzania komponentami [15]. Pozwala to na uruchomienie aplikacji w różnych trybach operacyjnych poprzez pominięcie

instancji serwisów odpowiedzialnych za renderowanie obrazu czy obsługę okna. W takim scenariuszu główna pętla sterująca pozostaje niezmieniona [3].

Modularność ta pozytywnie wpływa również na utrzymywalność (ang. *maintainability*) kodu, ułatwiając diagnostykę błędów poprzez izolację poszczególnych serwisów i eliminację ukrytych zależności globalnych [16].

3.3.2. 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 (kodu 2) [3].

Fundamentalnym mechanizmem budowania logiki aplikacji jest wzorzec wstrzykiwania zależności (ang. *Dependency Injection*) [15]. 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 co zostało przedstawione we fragmencie kodu 3 [3].

```
auto App::initSystems() -> void
{
        auto& registry = m_engine.registry();
        if ( m_cliOptions.gui )
        {
                m_engine.addSystem< InputSystem >( registry );
                m_engine.addSystem< CameraMovementSystem >( registry );
                m_engine.addSystem< SimRunnerSystem >( registry, m_cliOptions );
                m_engine.addSystem< UISystem >( registry );
                m_engine.addSystem< RenderSystem >( registry );
        }
        else
        {
                m_engine.addSystem< SimRunnerSystem >( registry, m_cliOptions );
        }
}
```

Fragment kodu 3. Inicjalizacja systemów aplikacji
Źródło: opracowanie własne.

Zastosowanie wspólnej klasy bazowej i polimorficznego kontenera w silniku niesie ze sobą te same korzyści co w przypadku serwisów w zakresie modularności. Możliwe jest selektywne zarządzanie funkcjonalnościami aplikacji poprzez warunkowe instancjonowanie konkretnych systemów. We fragmencie kodu 3 przedstawiono jak w trybie konsolowym (*headless*), systemy takie jak `RenderSystem` po prostu nie są dodawane do silnika. Pozwala to 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. Jednostka ta nie powinna udostępniać swoich metod ani stanu innym elementom aplikacji. Taka restrykcyjna hermetyzacja zapobiega powstawaniu niekontrolowanych powiązań [3] pomiędzy systemami.

Definicja systemów jako klas, a nie wolnych funkcji jest rozwiązaniem nietypowym na tle wielu popularnych implementacji architektury ECS. Zazwyczaj systemy implementowane są jako funkcje, lambdy lub funktory operujące na danych komponentów [6, 17]. Podejście obiektowe pozwala 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 [3].

Podejście to rozwiązuje następujące problemy:

- **Eliminacja kosztownych alokacji**: wybrane systemy obliczeniowe wymagają dodatkowej pamięci na operacje pośrednie. Zastosowanie klas pozwala na jednorazową alokację kontenerów na etapie konstrukcji systemu [3]. Dzięki temu unika się cyklicznego przydzielania i zwalniania pamięci w każdej klatce, co w systemach czasu rzeczywistego jest krytyczne z perspektywy wydajności [4].
- **Unikanie zanieczyszczania rejestru**: dane które nie są wykorzystywane przez więcej niż jeden element programu pozostają prywatnymi składowymi klasy systemu. Umieszczanie ich w rejestrze i późniejsze pobieranie poprzez mechanizm Service Locator byłoby architekturalnie nieuzasadnione i wprowadzałoby niepotrzebny narzut logiczny [3].
- **Jasność interfejsów:** rejestr pozostaje czytelnym zbiorem danych domenowych 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 [3].

3.3.3. Orkiestracja systemów symulacyjnych

Podczas dodawania pierwszych systemów symulacyjnych do projektu zidentyfikowano dwa istotne wyzwania [3]:

- **Złożoność**: pewne systemy zajmują się logiką aplikacji, inne zaś logiką symulacji. Instancjonowanie 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, staje się niemożliwe wprowadzenie mechanizmów pozwalających na kontrolę prędkości symulacji, w tym mechanizmu pauzy. Mechanizmy te są bardzo przydatne zarówno z perspektywy programisty, w celu debugowania oraz z perspektywy użytkownika, w celu weryfikacji konkretnego modelu w trybie graficznym.

Problem rozwiązano poprzez wykorzystanie, że systemy są zdefiniowane jako klasy. Pozwala to im na przechowywanie własnego stanu i posiadanie złożonej logiki wewnętrznej.

Wprowadzono unikalny system – `SimRunnerSystem`. Z perspektywy silnika nie różni się on pod żadnym względem od innych systemów aplikacyjnych, również implementując bazowy interfejs `ISystem`. Pełni on rolę warstwy orkiestrującej (ang. *orchestrator*) [15], która z punktu widzenia systemów symulacyjnych działa również jako uproszczona fasada, udostępniając jeden, spójny punkt wejścia do wykonywania całego potoku symulacji [3].

Niesie to ze sobą następujące korzyści:

- **Enkapsulacja potoku symulacji**: system ten posiada własny wektor podsystemów symulacyjnych. Tworzy on wewnętrzną pętlę, która – analogicznie do pętli głównej silnika – iteruje po wszystkich systemach biologicznych. Dzięki temu logika symulacji jest odseparowania od logiki infrastrukturalnej aplikacji [3].
- **Autonomia czasu**: ponieważ system ten manualnie wywołuje swoje podsystemy biologiczne, możliwym stało się wprowadzenie logiki kontroli symulacji, w tym prędkości i pauzowania [3].
- **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 [3].

3.3.4. Komunikacja między systemami

Rygorystyczna separacja systemów oraz ich hermetyzacja wymagają wprowadzenia ustandaryzowanych mechanizmów wymiany informacji. W projekcie zrezygnowano z bezpośrednich powiązań między systemami na rzecz dwóch uzupełniających się podejść: obiektów kontekstowych oraz magistrali zdarzeń (ang. *event dispatcher*) [3].

W sytuacjach, gdy komunikacja jest rzadsza i ma charakter reaktywny, zastosowano wzorzec publikuj-subskrybuj (ang. *publish-subscribe*), realizowany przez `entt::dispatcher`. Mechanizm ten pozwala na całkowite odizolowanie nadawcy od odbiorcy. Fragment kodu 4 przedstawia kod wykonywany po kliknięciu przycisku restartu symulacji. System zajmujący się interfejsem użytkownika, wyszukuje obiekt `entt::dispatcher` i kolejkuje zdarzenie. Zdarzanie `event::ResetSim` jest zdefiniowane jako pusta struktura danych (*struct*). Pozwala to na komunikację między różnymi systemami bez tworzenia żadnych powiązań – w tym na przekazywanie danych, a nie wyłącznie sygnałów [3].

```
ImGui::BeginDisabled( simRunnerData.iteration == 0 );
if ( ImGui::Button( labels.restartButton.data() ) )
{
        auto& dispatcher = registry.ctx().get< entt::dispatcher >();
        dispatcher.enqueue< event::ResetSim >();
}
ImGui::EndDisabled();
```

Fragment kodu 4. Wywoływanie wydarzenia
Źródło: opracowanie własne.

W bibliotece EnTT zdarzeniem może być dowolny typ danych, w tym typy trywialne takie jak `int` lub `float`. Należy jednak pamiętać, że to właśnie typ służy do rozróżniania komunikatów [6]. Z tego powodu odradza się stosowanie prymitywów na rzecz dedykowanych struktur, które poprawiają czytelność kodu i zapewniają unikalność zgodnie z zasadą identyfikacji nominalnej języka *C++* (ang. *nominal typing*) [12].

Obiekt `entt::dispatcher` pozwala zarówno na kolejkowanie oraz natychmiastowe wywoływanie eventów. W wielu wypadkach jednak rozsądniejszym jest drugie rozwiązanie, co pozwala na zachowanie lepszej stabilności oprogramowania [16]. W zaimplementowanej architekturze obie opcje są dostępne a wywoływanie zakolejkowanych wydarzeń następuje na końcu klatki, po zakończeniu wszystkich aktualizacji systemów i serwisów (fragment kodu 4). Zmniejsza to ryzyko korupcji danych wynikającej z zakłócenia potoku logicznego [3].

W literaturze systemy zdarzeniowe są wskazywane jako naturalny sposób realizacji luźno powiązanej komunikacji pomiędzy niezależnymi podsystemami silnika. Nystrom opisuje centralną kolejkę zdarzeń jako „kręgosłup nerwowy” pozwalający przekazywać komunikaty między modułami bez bezpośrednich zależności oraz z możliwością opóźnionego

przetwarzania, co poprawia rozszerzalność i modularność architektury. Choć pomimo tego autor zwraca uwagę, że centralna kolejka zdarzeń pozostaje de facto globalną zmienną, co może wprowadzać ukryte zależności i komplikować architekturę [16].

Innym sposobem komunikacji między systemami i modułami są obiekty kontekstowe. Są one zdefiniowane jako struktury inicjalizowane przez główną aplikacje klasy i umieszczane w kontekście rejestru, co pozwala na ich dostęp we wszystkich modułach aplikacji. Przykładem takiego obiektu jest struktura `Preset`, która przechowuje ustawienia użytkownika odczytane z pliku konfiguracyjnego podczas uruchamiania aplikacji. Aplikacja ma dziesiątki ustawień, które modyfikują działania licznych systemów co sprawia, że ustawienia tę muszą być ogólnodostępne. Podobną strukturą jest `SimRunnerData`, której rolą jest upublicznianie informacji na temat stanu symulacji do zainteresowanych systemów. W tym systemów zajmujących się interfejsem użytkownika oraz logowaniem wyników. Nieco innym przykładem jest struktura `TickDataCollection`, używana przez różne systemy. Struktura ta zbiera dane statystyczne które mogłyby zostać utracone pod koniec klatki. Przykładowo: określenie powodu śmierci agenta nie jest możliwe, jeżeli ten agent został już zdealokowany [3].

W kontekście rejestru można znaleźć jeszcze jeden typ obiektów. Podobnie jak wcześniej omówione serwisy, nie wpasowują się one nigdzie w purystyczną definicje modelu ECS. Już na początku prac nad oprogramowaniem problematycznym elementem okazał się automat komórkowy stanowiący podstawę świata symulacji. Siatka automatu, zależna od danych wejściowych podanych przez użytkownika, może składać się z tysięcy komórek. Komórki same w sobie mają bardzo proste reguły przejścia o zerowym promieniu. Definicja tychże komórek jako encje w rejestrze byłaby bardzo nierozsądna pod względem optymalizacyjnym. Rejestr ECS, pomimo bycia wydajnym, wprowadza dodatkowy narzut związany z zarządzaniem identyfikatorami i rozproszoną alokacją komponentów w pamięci [4]. Znacznie lepszym rozwiązaniem jest umieszczenie tych komórek w pojedynczej, ciągłej tablicy i opakowanie jej w strukturę umieszczoną w kontekście rejestru co zapewnia lokalność danych (ang. *cache locality*) i zwiększa wydajność [4]. W tym konkretnym przypadku, w miarę rozrostu symulacji, struktura zmieniła się w klasę z dodatkową funkcjonalnością [3].

Ze względu na to, że symulacja zawiera elementy losowości w postaci mutacji genetycznej, istotne było zapewnienie możliwie deterministycznego przebiegu obliczeń. Język C++ definiuje generatory takie jak `std::mt19937`, mające możliwość odtworzenia dokładnie tych

samych wyników, o ile programista jest w stanie zaopatrzyć je w to samo ziarno [18]. Plik konfiguracyjny daje użytkownikowi możliwość przekazania tej wartości, natomiast umieszczona w kontekście rejestru klasa `Randomizer` pozwala udostępnić deterministyczne źródło liczb pseudolosowych wszystkim systemom, które tego potrzebują [1]. Należy jednak podkreślić, że pełna deterministyczność całej symulacji jest ograniczona przez własności arytmetyki zmiennoprzecinkowej. Standard *C++* nie gwarantuje bitowo identycznych wyników operacji na liczbach typu `float` i `double` między różnymi platformami sprzętowymi ani przy odmiennych ustawieniach kompilatora (np. poziomach optymalizacji, wykorzystaniu instrukcji wektorowych czy FMA) [18, 19, 20]. W praktyce oznacza to, że model zachowuje się deterministycznie w obrębie tej samej konfiguracji sprzęt–kompilator, natomiast wyniki nie muszą być w pełni odtwarzalne na innych architekturach lub przy innej konfiguracji procesu kompilacji.

3.3.5. Konfiguracja aplikacji i interfejsy wejścia/wyjścia

Warstwa odpowiedzialna za komunikację z użytkownikiem została zaprojektowana tak aby wspierać automatyzację badań oraz elastyczność w definiowaniu warunków początkowych [3].

Konfiguracja aplikacji następuje w trzech krokach. Do obsługi parametrów linii komend wykorzystano bibliotekę CLI11 [9]. Zdefiniowanych jest kilka podstawowych parametrów, w tym opcja `--help` oferująca pomoc użytkownikowi. Parametrem wymaganym do inicjalizacji aplikacji jest `-preset`, która służy do wskazania pliku konfiguracyjnego w formacie JSON.

Plik konfiguracyjny zawiera kilkadziesiąt różnych opcji pozwalających na modyfikacje symulacji. Opcje te pozwalają na zmianę rzeczy takich jak:

- lokalizacje definicji siatki automatu komórkowego,
- docelowy katalog na wyniki,
- logowanie danych,
- ziarno dla generatora liczb losowych,
- liczba iteracji do zasymulowania,
- zmianę niemal wszystkich parametrów symulacyjnych, w tym:
    - startowych genów,
    - ustawień rozrostu wegetacji,
    - wpływu środowiska na agentów,
    - różnych innych modyfikatorów.

Plik wczytywany jest z pomocą biblioteki nlohmann/json [10]. Zapisane w pliku opcje są parsowane i tłumaczone na wewnętrzną reprezentację w obiekcie kontekstowym `Preset`. Jeżeli dane są niekompletne lub format jest niepoprawny to program zgłosi błąd i zatrzyma inicjalizacje aplikacji. Podejście z niezależnym plikiem konfiguracyjnym pozwala na wersjonowanie ustawień i szybkie przełączanie się pomiędzy różnymi scenariuszami badawczymi bez potrzeby rekompilacji kodu [3].

Środowisko symulacji jest wczytywane z zestawu obrazków w formacie PNG, z pomocą biblioteki stb_image [21]. Oprogramowanie sprawdza zawartość katalogu wskazanego w pliku konfiguracyjnym i wyszukuje w nim cztery mapy w postaci obrazków:

- `temperature.png` – definiuje temperaturę każdej komórki siatki,
- `humidity.png` – definiuje wilgotność każdej komórki siatki,
- `elevation.png` – definiuje elewacje każdej komórki siatki,
- `population.png` – definiuje populację startową każdej komórki siatki.

Program oczekuje, że wszystkie obrazki będą w ośmiobitowej skali szarości (ang. *grayscale*). Wartości pikseli następnie mapowane są na wewnętrzna reprezentację tych atrybutów, na podstawie których tworzone są komórki. Oczekuje się, że wartości pojedynczego piksela będzie znajdować się w przedziale `<0,255>`, z wartością zero oznaczająca kolor w pełni czarny. Program tłumaczy piksele na liczby zmiennoprzecinkowe w przedziale `<0,1>`. Im intensywniejsza czerń piksela tym większa będzie wartość danego atrybutu odpowiadającej mu komórki. Jedynym wyjątkiem jest populacja, gdzie zastosowano uproszczenie. Piksel o maksymalnej intensywności odpowiada instancjowaniu pojedynczego agenta w wybranej komórce. Jeżeli obrazki nie będą o tych samych wymiarach lub zestaw będzie niekompletny, program uzna to za błąd użytkownika, zgłosi i przerwie inicjalizację [3].

Rozwiązanie to pozwala na osiągnięcie precyzji, w projektowaniu środowisk symulacyjnych, poprzez możliwość „namalowania mapy” w dowolnym programie graficznym wspierającym eksport w formacie PNG. Uznano, że adaptacja istniejących już, zaawansowanych programów graficznych jest preferowana ponad tworzenie własnych, prostych narzędzi od podstaw.

Aplikacja działa w dwóch trybach użytkowych. Tryb graficzny wykorzystuje biblioteki SFML [7] i Dear ImGui [8]. Przechodzi się do niego poprzez przekazanie flagi `--gui` przy uruchamianiu programu. W trybie tym, użytkownik może w czasie rzeczywistym analizować przebieg symulacji. Dostępne są mechanizmy kontrolne, w tym pauza, restart i ustawienie

prędkości. Liczne filtry renderują komórki siatki w różnych kolorach, mapując różne statystyki środowiska oraz populacji na dwuwymiarową reprezentację. Dane statystyczne zmieniające się w czasie rzeczywistym pozwalają na wczesną ocenę wyników uzyskiwanych ze zdefiniowanej konfiguracji. Dodatkowo, możliwym jest przybliżanie, oddalanie i przemieszczanie kamery oraz zmiana rozmiaru czcionki interfejsu użytkownika. Mimo wszystko użytkownik nie ma opcji wpływu na przebieg symulacji. Jest to uproszczenie jakie było konieczne ze względu na ograniczenia czasowe. Niemniej jednak, zastosowana architektura, pozwala na relatywnie szybką i łatwą rozbudowę funkcjonalności oprogramowania co pozwala na dalsze rozszerzenie aplikacji o tę możliwości [3].

Alternatywnym trybem działania aplikacji jest tryb konsolowy. Tryb ten maksymalizuje wydajność poprzez pominięcie instancjonowania wszelkich systemów, serwisów i obiektów odpowiedzialnych za aspekty graficzne aplikacji co jest przydatne w przypadku długich symulacji. Dane statystyczne uzyskiwane w trakcie przebiegu symulacji są buforowane i zapisywane w postaci plików formatu CSV do wskazanego przez użytkownika folderu docelowego.

4. Model ekosystemu

Zrezygnowano z klasycznego modelu drapieżnictwa (relacja agent-agent) na rzecz drapieżnictwa abiotycznego, w którym głównym czynnikiem selekcyjnym jest presja nieożywionego środowiska oraz ograniczona podaż energii [3, 22].

4.1. Dynamika środowiska

Środowisko symulacji stanowi automat komórkowy o zerowym promieniu, w którym każda komórka opisana jest przez trzy niezależne parametry: temperaturę, wilgotność i wysokość. Wartości te definiują lokalną niszę ekologiczną [3].

4.1.1. Pojemność środowiska i regeneracja

Każda komórka posiada dynamicznie odnawialny poziom wegetacji. Maksymalny limit pożywienia oraz tempo wzrostu są obliczane na podstawie odchylenia parametrów komórki od wartości idealnych, zdefiniowanych przez użytkownika w konfiguracji [3].

Do wyznaczenia optymalnych warunków wykorzystano rozkład normalny. Im parametry komórki są bliższe średniej, tym wyższa jest produktywność danej jednostki. Wysokość terenu pełni rolę dodatkowego modyfikatora (kary), symulującego trudne warunki wysokogórskie [3].

4.1.2. Model agenta i gospodarka energetyczna

Agent w symulacji jest traktowany jako autonomiczny system dążący do maksymalizacji sukcesu reprodukcyjnego przy jednoczesnym minimalizowaniu kosztów metabolicznych [3].

Każdy agent posiada zestaw genów określających jego preferencje środowiskowe oraz maksymalny poziom energii. Zastosowano mechanizm plastyczności międzypokoleniowej: genotyp potomka nie jest identyczny z genotypem rodzica z chwili jego narodzin. Doświadczenia rodzica (np. przebywanie w skrajnych temperaturach) modyfikują parametry potencjalnego potomstwa przed etapem mutacji losowej [3, 23].

Agent ponosi stały koszt energetyczny związany z bazowym metabolizmem oraz koszty zmienne wynikające z ruchu i niedopasowania do lokalnych warunków środowiska. Śmierć występuje w wyniku wyczerpania zasobów energii (wygłodzenia) lub osiągnięcia limitu wieku [3]. Proces prokreacji jest warunkowany nie tylko posiadaną energią, ale również analizą otoczenia. Agent dokonuje oceny dostępności zasobów w najbliższej okolicy, co na celu ma zapobieganie nadmiernej eksploatacji lokalnego ekosystemu i stymuluje migracje w poszukiwaniu lepszych nisz [3].

4.2. Mechanizm adaptacji i dziedziczenia

Sercem procesu ewolucyjnego jest model dziedziczenia, który łączy klasyczną darwinowską mutację z mechanizmem plastyczności międzypokoleniowej [23]. Zastosowane podejście pozwala na badanie nie tylko losowych zmian w populacji, ale także kierunkowej odpowiedzi organizmów na konkretne warunki środowiskowe [3].

4.2.1. Plastyczność międzypokoleniowa

W odróżnieniu od modeli, w których genotyp jest statyczny, zaimplementowany system wprowadza mechanizm akumulacji doświadczenia. Agent w trakcie życia modyfikuje parametry, które przekaże potomstwu [3]:

- **adaptacja do niszy środowiskowej** - jeżeli agent przebywa w środowisku odbiegającym od jego optymalnych preferencji (np. w strefie o niższej temperaturze), wartości genów potencjalnego potomstwa są powoli przesuwane w stronę tych warunków;
- **przekazywanie doświadczenia środowiskowego** – mechanizm ten odzwierciedla biologiczną zdolność organizmów do lokalnej adaptacji jeszcze przed wystąpieniem reprodukcji, co wpływa na cechy dziedziczone przez kolejne pokolenia.

Mechanizm ten nie stanowi bezpośredniej implementacji procesów epigenetycznych, lecz ich modelowe przybliżenie, którego celem jest uchwycenie kierunkowego wpływu doświadczeń środowiskowych rodzica na parametry dziedziczone przez kolejne pokolenie [23].

4.2.2. Selekcja przedrozrodcza i bariery energetyczne

Zanim dojdzie do aktu prokreacji, system weryfikuje zdolność rodzica do zapewnienia bytu nowemu osobnikowi. Mechanizm usiłuje zapobiegać niekontrolowanemu przyrostowi naturalnemu w skrajnie ubogich warunkach [3]:

- **Kara za głód:** próby rozmnożenia w warunkach niedosytu skutkują osłabieniem genetycznym potomka. Maksymalna energia zostaje obniżona, co jest mechanizmem adaptacji do ubogich warunków.
- **Wymóg sytości**: prokreacja możliwa jest wyłącznie po osiągnięciu pełnego nasycenia energetycznego oraz zapewnieniu zapasu energetycznego w lokalnej okolicy. Agent weryfikuje obecność wegetacji, wymaganą do wyżywienia siebie, swojego potomstwa oraz innych lokalnych osobników.

4.2.3. Mutacja losowa i ekonomia przetrwania

Ostatnim etapem reprodukcji jest nałożenie losowego szumu (mutacji) na wypracowany przez rodzica genotyp. Każdy gen mutuje niezależnie w zdefiniowanym w konfiguracji przedziale. Każda zmiana genetyczna w modelu niesie za sobą pewne korzyści oraz koszty [3]:

- **Preferencje środowiskowe:** pozwalają agentowi na lepszą adaptację do lokalnych warunków zmniejszając zużycie energetyczne. Kosztem jednak jest mniejsza kompatybilność z innymi niszami ekologicznymi.
- **Wysoki zapas energii:** pozwala agentowi na dłuższą przeżywalność w okresach przejściowego braku żywności oraz eksploracje, ale jednocześnie znacząco utrudnia prokreację. Potomek o wysokim zapotrzebowaniu rzadziej osiąga stan pełnego nasycenia, co ogranicza jego sukces reprodukcyjny.
- **Niski zapas energii:** ułatwia zaspokojenie wymogu energetycznego potrzebnego do prokreacji, ale pozostawia agentów skrajnie wrażliwych na fluktuacje biomasy w lokalnym środowisku i utrudnia eksploracje.

Zasady te definiują pętle drapieżnictwa abiotycznego – środowisko nie „atakuje" agentów fizycznie, lecz poprzez bezlitosną ekonomię energii eliminuje nieefektywne kombinacje genów, promując te, które najlepiej balansują między kosztem metabolizmu a zdolnością do reprodukcji [3, 22].

4.3. Ograniczenia biologiczne

Poza mechanizmami dziedziczenia, każda jednostka podlega zestawowi sztywnych reguł cyklu życia, które determinują jej dynamikę w ekosystemie [3].

W modelu przyjęto model prokreacji bezpłciowej, co upraszcza symulacje i eliminuje konieczność poszukiwania partnera. Aby zapobiec niekontrolowanemu przyrostowi populacji, wprowadzono konfigurowalny czas regeneracji (parametr `refractory period`), określający minimalną liczbę iteracji, jakie muszą upłynąć pomiędzy uzyskaniem możliwości wydania kolejnego potomstwa [3].

Ilość energii, jaką agent może przyswoić w pojedynczym kroku symulacji jest odgórnie ograniczona przez użytkownika. Parametr ten krytycznie wpływa na organizmy o wysokim zapasie energii. Mimo posiadania „większego żołądka", nie są one w stanie napełnić go natychmiastowo co wymusza na nich dłuższe przebywanie w zasobnych niszach i zwiększa ryzyko śmierci przed osiągnięciem sytości niezbędnej do prokreacji [3].

Każdy agent posiada zdefiniowany maksymalny wiek ustawiany przez użytkownika w konfiguracji co zapobiega przeludnieniu i ochrania dynamikę ekosystemu poprzez stałą rotacje żyjących organizmów [3].

Warto zaznaczyć, że pomimo iż wszystkie z opisanych powyżej parametrów mogłyby zostać zaimplementowane jako geny podlegające mutacji, celowo z tego zrezygnowano na rzecz uproszczenia i utrzymania stabilności symulacji [3].

4.4. Założenia badawcze

Zastosowanie wieloparametrowego opisu komórek oraz rozkładu normalnego do wyznaczania produktywności biologicznej miało na celu umożliwienie obserwacji konkretnych zjawisk ekologicznych. Konstrukcja modelu opiera się na hipotezie, że tak zdefiniowane środowisko pozwoli na:

- **Teoretyczne wykształcenie nisz ekologicznych:** dążono do stworzenia niejednorodnej areny działań. Zamysłem było sprawdzenie czy populacja, zamiast zmierzać do jednego, uniwersalnego genotypu, zacznie różnicować się na grupy wyspecjalizowane w eksploatacji skrajnie odmiennych obszarów,
- **Weryfikację mechanizmu drapieżnictwa abiotycznego:** model został zaprojektowany tak, aby sprawdzić, czy samo tempo regeneracji biomasy może pełnić funkcję selekcyjną [22]. Założono, że niedopasowanie metaboliczne agenta do lokalnej wydajności niszy środowiskowej powinno prowadzić do jego eliminacji, wymuszając ewolucyjną optymalizację gospodarki energetycznej.
- **Badanie dynamiki migracji**: projektowanie zróżnicowanych map pomóc zbadać czy populacja zagrożona przez wygłodzenie jest w stanie wyemigrować w kierunku odkrywania nowych, zasobnych obszarów.

5. Środowisko testowe

W celu zapewnienia spójności i powtarzalności wyników wszystkie eksperymenty przeprowadzono w jednolitym środowisku testowym. W niniejszym rozdziale przedstawiono szczegółową specyfikację sprzętu, oprogramowania oraz konfiguracji kompilacji wykorzystywanych podczas realizacji badań. Opisane tu środowisko stanowi odniesienie dla wszystkich testów prezentowanych w dalszej części pracy. Wyniki, dane wejściowe oraz skrypty użyte podczas przeprowadzania testów zostały zawarte w publicznym repozytorium poświęconym wynikom [24].

5.1. Konfiguracja sprzętowa

Testy wykonano na komputerze osobistym o następującej konfiguracji sprzętowej:

- procesor – Core i5 11400H 2.70 GHz,
- procesor graficzny – Intel UHD Graphics 11th Gen 128MB,
- pamięć operacyjna RAM – 16GB DDR4.

GPU komputera wykorzystane było wyłącznie do renderowania interfejsu graficznego aplikacji. Obliczenia związane z przebiegiem symulacji były wykonane w pełni na pojedynczym rdzeniu CPU.

5.2. Konfiguracja oprogramowania i kompilacja

Oprogramowanie symulacyjne zostało skompilowane i uruchomione w następującym środowisku:

- system operacyjny – Windows 10 Home 22H2 64-bit,
- kompilator C++ – Clang 21.1.7 x86_64-pc-windows-msvc,
- konfiguracja kompilacji: tryb Release z optymalizacją -O3,
- commit (Git) – 85fe91021557b272a230b086f315078f5ac3e8ec [3].

6. Ocena architektury

Celem niniejszego rozdziału jest ocena zaprojektowanej architektury oprogramowania pod kątem jej własności niefunkcjonalnych, w szczególności wydajności obliczeniowej oraz deterministyczności działania. Przedstawiono wyniki testów wydajnościowych przeprowadzonych dla różnych liczebności populacji agentów, analizę wpływu warstwy graficznej na czas przetwarzania, a także test deterministyczności symulacji dla tych samych danych wejściowych.

6.1. Testy wydajnościowe

Podobnie jak w pracach poświęconych zastosowaniu architektury ECS w symulacjach agentowych, nacisk położono tu na ocenę skalowalności obliczeniowej [4]. Wykonano szereg testów wydajnościowych polegających na przeprowadzeniu symulacji dla zmiennej liczby agentów. Przygotowano specjalny zestaw konfiguracyjny, w którym:

- ustawiono czas trwania symulacji na sto jeden tysięcy iteracji,
- zapotrzebowanie agentów na energię zredukowano do zera,
- długość życia oraz liczbę iteracji wymaganą do osiągnięcia możliwości prokreacji zwiększono do wartości przekraczających całkowitą liczbę iteracji symulacji.
- operacje zapisu związane z logowaniem wyników opóźniono do zakończenia symulacji.

Celem tych zabiegów było zapewnienie stałej liczby agentów podczas całego przebiegu każdej symulacji, co umożliwiło uzyskanie powtarzalnego i stabilnego obciążenia obliczeniowego. Należy zaznaczyć, że takie uproszczenia w pewnym stopniu obniżyły koszt obliczeniowy symulacji, ponieważ część ścieżek logicznych i mechanizmów nie była w pełni aktywowana. Mimo to najbardziej wymagające systemy, odpowiedzialne za logikę zachowania i procesy decyzyjne agentów, nadal wykonywały pełen zakres swojej pracy.

Szereg testów przeprowadzono w sposób zautomatyzowany w trybie konsolowym z wykorzystaniem skryptów napisanych w języku Python [25].

6.1.1. Wyniki testów wydajnościowych

Jako *czas iteracji* definiuje się czas, jaki aplikacja przeznacza wyłącznie na wykonanie modelu symulacyjnego: obejmuje to aktualizację automatu komórkowego, wszystkich agentów oraz buforowanie wyników wydajnościowych danej iteracji.

Z kolei *czas klatki* określa się jako czas potrzebny na pełne przetworzenie pojedynczej klatki aplikacji, zawierający zarówno wykonanie iteracji symulacji, jak i wszystkie dodatkowe operacje warstwy silnika i serwisów oraz niesymulacyjnej warstwy logicznej aplikacji. W przypadku trybu graficznego uwzględniany jest również czas obsługi wejścia użytkownika, odświeżania okna oraz renderowania elementów interfejsu użytkownika. Pierwszy tysiąc iteracji służył jako rozgrzewka pomiarowa i nie został uwzględniony w analizie. Zestawienie wartości średnich i median oraz odpowiadającej im przepustowości podano w tabeli 1.

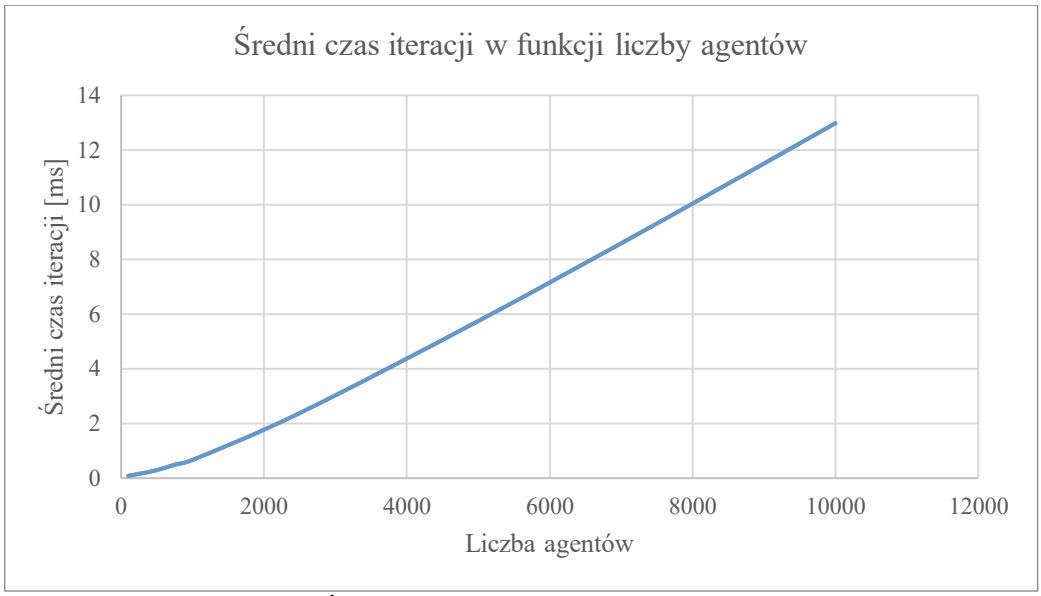
Tab. 1. Wyniki testu wydajnościowego - czas iteracji zależny od liczby agentów

Liczba agentów	**Średni czas iteracji [ms]**	**Mediana czasu iteracji [ms]**	**Przepustowość [agentów / s]**
100	0.088	0.086	1 134 658
250	0.158	0.156	1 584 541
500	0.293	0.286	1 707 894
750	0.491	0.482	1 527 406
1000	0.667	0.663	1 500 058
2500	2.380	2.376	1 050 579
5000	5.745	5.715	870 286
7500	9.309	9.281	805 706
10000	12.978	12.967	770 529

Źródło: opracowanie własne

6.1.2. Analiza czasu iteracji

Na rysunku 1 przedstawiono zależność średniego czasu pojedynczej iteracji symulacji od liczby agentów.



Rys. 1. Średni czas iteracji w funkcji liczby agentów
Źródło: opracowanie własne na podstawie tabeli 1.

Czas trwania iteracji rośnie monotonicznie wraz z liczebnością populacji i wykazuje charakter zbliżony do liniowego. Dla 100 agentów średni czas iteracji wynosi ok. 0.09 ms, natomiast dla 10 000 agentów ok. 13 ms, co odpowiada skalowaniu jedynie nieznacznie szybszemu niż liniowe i pozbawionemu nieliniowego „załamania" wydajności.

6.1.3. Analiza przepustowości

Na rysunku 2 przedstawiono przepustowość, rozumianą jako liczbę aktualizowanych agentów na sekundę, w funkcji liczebności populacji.

Rys. 2. Przepustowość w funkcji liczby agentów
Źródło: opracowanie własne na podstawie tabeli 1.

Dla małych wartości *N* obserwuje się wzrost przepustowości – od ok. 1.13 mln agentów/s dla 100 agentów do maksymalnie ok. 1.71 mln agentów/s dla 500 agentów. W tym zakresie istotną rolę odgrywa narzut stałych operacji niezależnych od liczby agentów (m.in. obsługa pętli głównej i orkiestracja systemów), który przy większej populacji ulega lepszej amortyzacji.

Powyżej 500–1000 agentów przepustowość stopniowo maleje, osiągając wartość ok. 0.77 mln agentów/s dla 10 000 agentów. Spadek ten ma łagodny charakter i wynika głównie z rosnącego obciążenia pamięci oraz kosztów dostępu do danych komponentów przy dużej liczbie encji. Pomimo tego, w całym badanym zakresie utrzymywany jest poziom setek tysięcy aktualizacji agentów na sekundę na pojedynczym rdzeniu CPU, co wskazuje, że zastosowana architektura ECS zapewnia dobrą skalowalność obliczeniową.

Niewielkie odchylenia od idealnie monotonicznego przebiegu w zakresie 250–1000 agentów wynikają z wpływu stałego narzutu oraz szumu pomiarowego i nie mają istotnego znaczenia z punktu widzenia ogólnego trendu.

6.1.4. Analiza wpływu trybu graficznego na wydajność

W celu oceny wpływu warstwy graficznej na wydajność symulacji porównano średni czas trwania klatki oraz średni czas trwania pojedynczej iteracji modelu w trybie konsolowym (CLI) i graficznym (GUI) dla populacji liczącej 500 agentów. Wyniki zestawiono w tabeli 2.

Tab. 2. Wzrost wydajności w trybie konsolowym

Metryka	**CLI (średnia) [ms]**	**GUI (średnia) [ms]**	**Przyspieszenie CLI**
Czas klatki	0.2929	0.8275	**2.83**× (182.5%)
Czas iteracji	0.2927	0.3365	**1.15**× (15.0%)

Źródło: opracowanie własne.

Zgodnie z tabelą 2, przejście z trybu graficznego do konsolowego pozwala na około 2.8-krotne (≈183%) przyspieszenie czasu klatki, co wynika z wyeliminowania kosztów renderowania oraz przetwarzania interfejsu użytkownika. Jednocześnie występuje niewielka różnica w czasie trwania samej iteracji – tryb konsolowy jest średnio szybszy o ok. 15%. Powstały dodatkowy narzut iteracyjny w trybie GUI wynika wyłącznie z instancjonowania systemu odpowiedzialnego za przygotowanie danych diagnostycznych wyświetlanych w interfejsie użytkownika, który tworzony jest jedynie w tym trybie.

Oznacza to, że obciążenie wynikające z warstwy graficznej jest w dużym stopniu odseparowane od logiki symulacji, a architektura hybrydowa skutecznie izoluje rdzeń symulacji od kosztów

renderowania. W efekcie wydajność części obliczeniowej pozostaje niemal identyczna w obu trybach. Uzyskane wyniki uzasadniają osobny tryb konsolowy jako rozwiązanie optymalizacyjne.

6.2. Test deterministyczności

Aby ocenić deterministyczność implementacji, przeprowadzono serię pięciu pełnych przebiegów symulacji z wykorzystaniem jednakowych danych wejściowych. Cały eksperyment został uruchomiony w sposób zautomatyzowany za pomocą skryptu w języku Python, co zapewniło pełną powtarzalność warunków wykonania. W przeciwieństwie do testów wydajnościowych zastosowano tu w pełni funkcjonalną konfigurację modelu ekosystemu: agenci podlegali normalnej dynamice populacji, zużyciu energii, śmierci, prokreacji i presji selekcji. Celem było sprawdzenie, czy przy stałych warunkach wejściowych identyczny jest nie tylko stan końcowy, lecz również cały przebieg ewolucji układu. Każdy przebieg obejmował 100 000 iteracji. W trakcie działania symulacja zapisywała do pliku wynikowego pełny zestaw danych z każdej iteracji, co w efekcie dawało 100 000 rekordów na jeden przebieg [24].

Wszystkie pięć przebiegów wygenerowało identyczne skróty kryptograficzne (SHA-256), co oznacza, że pełne sekwencje wyników były bitowo zgodne. Wskazuje to na deterministyczność implementacji symulacji w obrębie danej konfiguracji sprzętowo–kompilacyjnej przy użyciu jednakowych danych wejściowych [24].

Należy przy tym zaznaczyć, że ze względu na właściwości arytmetyki zmiennoprzecinkowej deterministyczność ta nie musi być zachowana na innych platformach sprzętowych lub przy odmiennych ustawieniach kompilatora [19, 20], jednak w praktyce jest wystarczająca do prowadzenia powtarzalnych eksperymentów numerycznych na docelowej platformie.

6.3. Wnioski z oceny architektury

Na podstawie przeprowadzonych testów można sformułować następujące wnioski dotyczące zaprojektowanej architektury oprogramowania:

- Czas trwania pojedynczej iteracji symulacji rośnie w przybliżeniu liniowo wraz ze wzrostem liczby agentów, co wskazuje na dobrą skalowalność obliczeniową modelu opartego na wzorcu ECS. W badanym zakresie liczby agentów nie zaobserwowano gwałtownego „załamania” wydajności. Jest to spójne z wynikami badań nad wydajnością architektur ECS w symulacjach agentowych [4].

- Przepustowość, liczona jako liczba aktualizowanych agentów na sekundę, utrzymuje się na poziomie setek tysięcy aktualizacji na pojedynczym rdzeniu CPU. Spadek przepustowości dla dużych populacji ma łagodny charakter i wynika głównie z rosnącego obciążenia pamięci i kosztów dostępu do danych komponentów.
- Uruchomienie aplikacji w trybie konsolowym pozwala na około 2.8-krotne zmniejszenie czasu trwania klatki względem trybu graficznego. Oznacza to, że obciążenie wynikające z warstwy graficznej jest w dużym stopniu odseparowane od logiki symulacji, a tryb CLI stanowi użyteczny wariant optymalizacyjny do długotrwałych eksperymentów.
- Test deterministyczności wykazał, że przy tych samych danych wejściowych i niezmiennym środowisku uruchomieniowym pełne przebiegi symulacji są bitowo powtarzalne. Deterministyczność ta jest w praktyce wystarczająca do prowadzenia powtarzalnych eksperymentów numerycznych na docelowej platformie sprzętowej.
- Zastosowanie dwupoziomowej architektury hybrydowej (warstwa silnika + warstwa aplikacji), mechanizmu serwisów oraz wzorca ECS zapewnia wysoką modularność systemu i możliwość łatwego rozszerzania logiki poprzez dodawanie nowych systemów bez ingerencji w istniejący kod (rozdział 3.3.2).
- Wykorzystanie zewnętrznego pliku konfiguracyjnego w formacie JSON oraz zestawu map wejściowych w formacie PNG umożliwia szeroką konfigurowalność parametrów eksperymentów i definiowanie zróżnicowanych scenariuszy badawczych bez konieczności rekompilacji oprogramowania. Dodatkowo połączenie trybu konsolowego z obsługą parametrów linii poleceń ułatwia automatyzację serii eksperymentów [24].
- Dobór narzędzi (*CMake, Clang*, biblioteki wieloplatformowe) oraz brak zależności od specyficznych funkcji systemu operacyjnego sprawiają, że środowisko symulacyjne ma charakter wieloplatformowy (rozdział 3.1), pomimo iż w ramach niniejszej pracy szczegółowe testy przeprowadzono na jednej konfiguracji sprzętowo-systemowej.

Uzyskane wyniki potwierdzają, że przyjęta architektura hybrydowa spełnia założenia zadania inżynierskiego: zapewnia efektywne przetwarzanie dużych populacji agentów, wyraźną separację warstwy symulacyjnej i graficznej oraz praktyczną powtarzalność przebiegu symulacji. Zastosowanie mechanizmu systemów i serwisów umożliwia łatwe rozszerzanie logiki, a konfiguracja eksperymentów za pomocą zewnętrznych plików JSON oraz wejściowych map środowiskowych w formacie PNG zapewnia wysoką elastyczność i

niezależność od kodu źródłowego. W połączeniu z trybem konsolowym i obsługą parametrów linii poleceń architektura wspiera automatyzację badań, natomiast dobór wieloplatformowych narzędzi i bibliotek gwarantuje przenośność rozwiązania między różnymi środowiskami programistycznymi.

7. Ocena modelu ekosystemu

W niniejszym rozdziale oceniono jakość działania opracowanego modelu ekosystemu, w szczególności tego, czy zaproponowane mechanizmy gospodarki energetycznej, selekcji abiotycznej oraz migracji populacji prowadzą do spójnej i interpretowalnej dynamiki w skali globalnej oraz lokalnej. Analiza została przeprowadzona w podejściu mieszanym: ilościowo, na podstawie statystyk rejestrowanych w trakcie symulacji oraz jakościowo, na podstawie obserwacji przestrzennego rozkładu populacji i parametrów agentów w interfejsie użytkownika.

Zastosowanie obserwacji jakościowej jest szczególnie ważne w odniesieniu do zjawisk o charakterze lokalnym (np. zachowanie populacji na granicach regionów środowiskowych), których redukcja do postaci pojedynczych miar globalnych może prowadzić do utraty istotnych informacji. W analizie wyników szczególną uwagę poświęcono genowi energii, ponieważ stanowi on syntetyczny wskaźnik jakości warunków środowiskowych oraz skuteczności strategii życiowej organizmu, a jednocześnie jego interpretacja jest jednoznaczna w kontekście mechanizmów modelu.

We wszystkich wizualizacjach w skali szarości przedstawionych w pracy ciemniejsza barwa odpowiada wyższej wartości analizowanej wielkości.

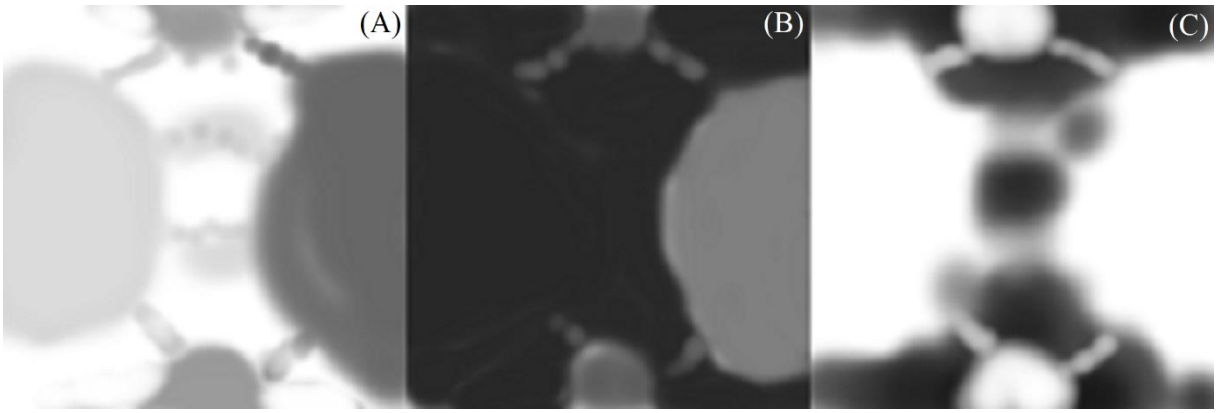
7.1. Konfiguracja eksperymentów

Przeprowadzono dwa eksperymenty w odmiennych środowiskach. W obu przypadkach symulacja była wykonywana na siatce o rozmiarze 100×100 przez 100 000 iteracji. Wykorzystano standardowy zestaw parametrów konfiguracyjnych aplikacji. W odróżnieniu od rozdziałów poświęconych testom wydajnościowym, konfiguracja nie była tu dobierana specyficznie pod pomiar wydajności [24].

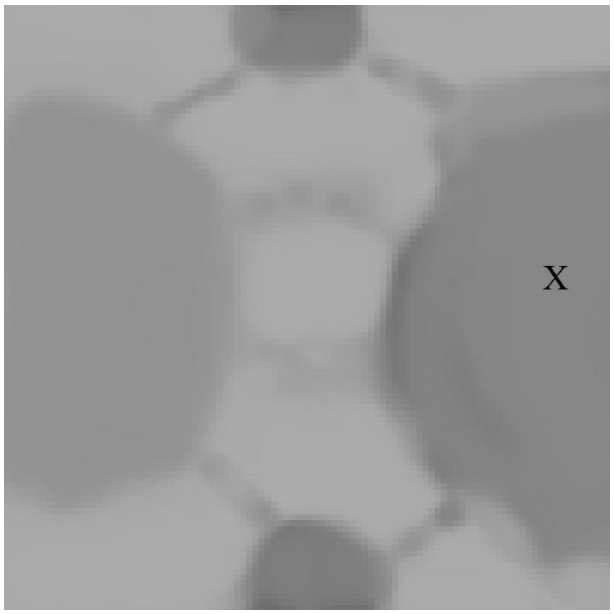
Pierwszy eksperyment został zaprojektowany jako środowisko o wysokiej restrykcyjności przestrzennej, zawierające silnie ograniczone korytarze migracyjne oraz kilka wyraźnych, odseparowanych nisz o odmiennej jakości. Drugi eksperyment pełnił rolę walidacyjną i wykorzystywał mapę o łagodnych gradientach środowiskowych, bez jednoznacznych barier przestrzennych – w tym przypadku ograniczenie stanowi stopniowy spadek jakości środowiska aż do obszarów niepodtrzymujących populacji [24].

7.2. Pierwszy eksperyment

Zastosowano mapę o strukturze zbliżonej do „labiryntu”: znaczna część przestrzeni ma charakter górski, a przejścia pomiędzy regionami są możliwe jedynie wąskimi korytarzami o niskiej jakości środowiska. Układ mapy tworzy kilka wyraźnych nisz [24]: dwa obszary o bardzo wysokiej jakości w części północnej i południowej, obszar średniej jakości po stronie wschodniej oraz obszar niskiej jakości po stronie zachodniej. Jednocześnie ograniczona łączność między regionami sprzyja izolacji populacji i stabilizacji lokalnych warunków środowiskowych, co wzmacnia wyraźny podział na nisze. Populację startową zdefiniowano jako pojedynczego agenta w obszarze wschodnim. Mapy wejściowe użyte w eksperymencie połączono zbiorczo w rysunek 3. Uzyskana wegetacja startowa przedstawiona jest na rysunku 4.



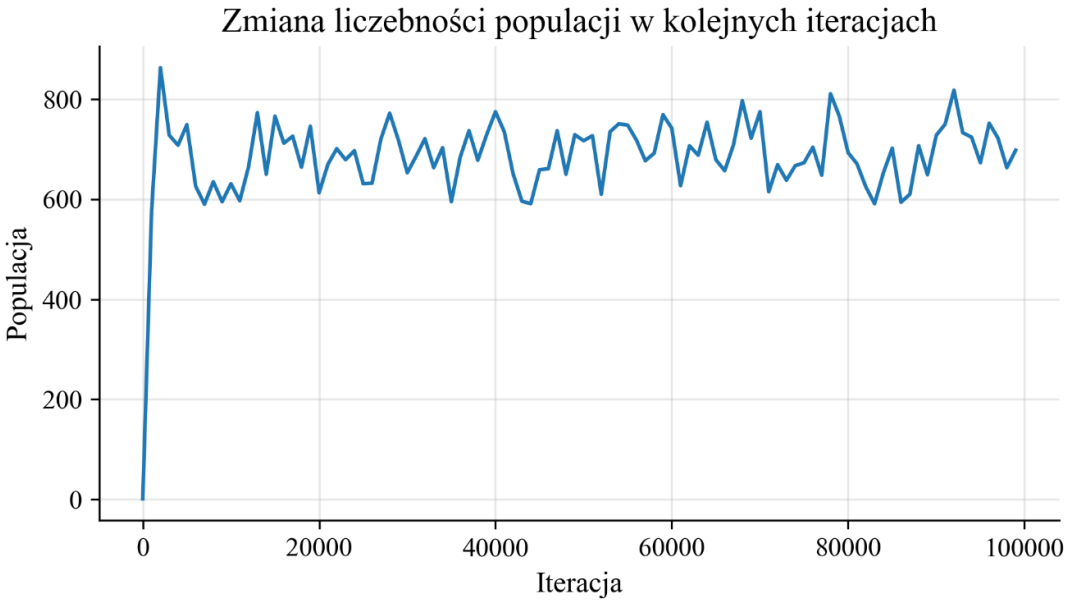
Rys. 3. Mapy wejściowe pierwszego eksperymentu modelu. (A–C): (A) temperatura, (B) wilgotność, (C) wysokość
Źródło: opracowanie własne.



Rys. 4. Mapa wegetacji startowej powstałej z map wejściowych przedstawionych na rysunku 3. Znakiem X została zaznaczona pozycja startowa agenta
Źródło: opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji).

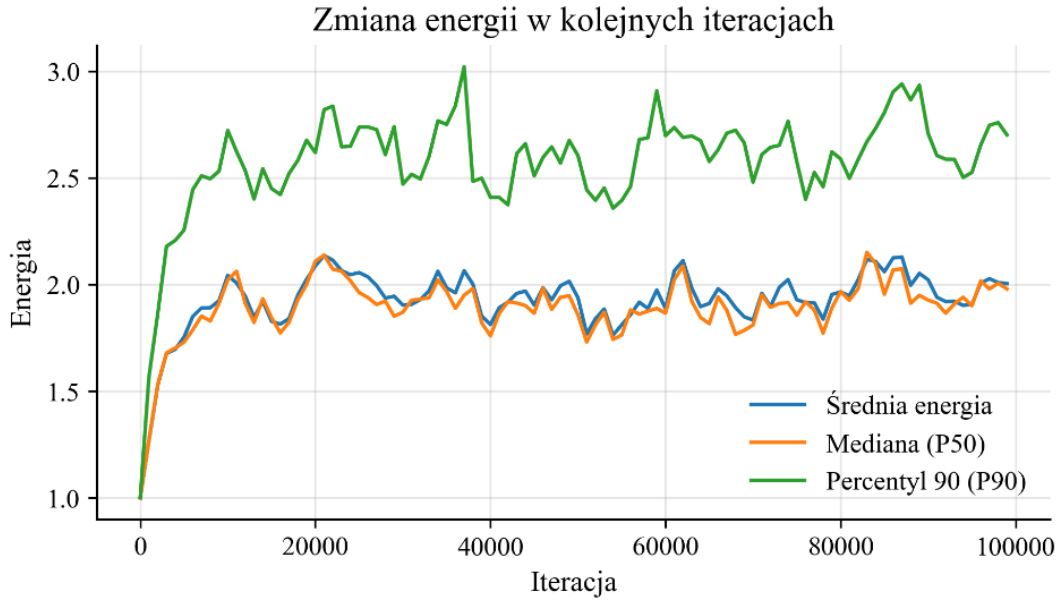
7.2.1. Szybki wzrost populacji i dojrzewanie energetyczne

W regionie startowym obserwowano gwałtowny wzrost liczebności populacji. W pierwszym tysiącu iteracji dominowała intensywna eksploatacja lokalnie dostępnych zasobów, co sprzyjało szybkiemu rozmnażaniu i zagęszczaniu populacji w obrębie początkowej niszy. Zjawisko to jest widoczne na rysunku 5.



Rys. 5. Zmiana liczebności populacji w kolejnych iteracjach - pierwszy eksperyment
Źródło: opracowanie własne.

Równolegle z ekspansją liczebności populacji zaobserwowano stopniową zmianę strategii energetycznych, interpretowaną jako „dojrzewanie" do bardziej kosztownych zachowań eksploracyjnych. Na rysunku 6 można zaobserwować narastanie zróżnicowania agentów pod względem maksymalnych zapasów energii, w tym stały wzrost wśród nielicznej grupy organizmów.



Rys. 6. Zmiana energii w kolejnych iteracjach - pierwszy eksperyment
Źródło: opracowanie własne.

W tym samym okresie obserwowano liczne próby eksploracji terenów odległych i niekorzystnych, które jednak często kończyły się niepowodzeniem, co wskazuje na ważną rolę ograniczeń energetycznych w początkowej fazie symulacji.

7.2.2. Nieudane ekspedycje i przełomy kolonizacyjne

W środowisku „labiryntowym" ekspedycje w wielu przypadkach kończyły się niepowodzeniem, co wynikało z konieczności przejścia przez wąskie korytarze o niskiej jakości środowiska oraz wysokich kosztów energetycznych związanych z ruchem w terenie niekorzystnym. Przełom nastąpił po tysięcznej iteracji w momencie, gdy nieliczna grupa osobników o wysokich zapasach energii była w stanie dotrzeć do niszy południowej. Zaobserwowano szybkie zasiedlanie tego obszaru i gwałtowny przyrost populacji w nowym

regionie. Zdarzenie to korelowało również jednym z najwyższych poziomów liczebności populacji w całym przebiegu symulacji, co jest widoczne jako pierwszy pik na rysunku 5.

Kolonizacja nowej, wysoce zasobnej niszy chwilowo zwiększa dostępność pożywienia w przeliczeniu na jednego osobnika, co umożliwia szybkie rozmnażanie i intensywną ekspansję lokalną. Jednocześnie środowisko o ograniczonej łączności sprzyja temu, że sukces kolonizacji jest zdarzeniem rzadkim i zależnym od pojawienia się osobników o wystarczającym potencjale energetycznym, co stanowi naturalny filtr selekcyjny w warunkach wysokiego kosztu migracji.

7.2.3. Kolonizacja północy i wtórna ekspansja na region zachodni

W dalszym przebiegu symulacji, po ok. siedmiu tysiącach iteracji zaobserwowano kolejne zdarzenie przełomowe – zasiedlenie regionu północnego. Podobnie jak w przypadku południa, początkowo próby dotarcia do tego obszaru były nieudane, a skuteczna kolonizacja nastąpiła dopiero po dłuższym czasie, gdy w populacji pojawiły się osobniki zdolne do poniesienia kosztów energetycznych związanych z przejściem przez niekorzystne korytarze.

W kolejnej fazie rozwoju układu zaobserwowano wtórną ekspansję: po ok. dziesięciu tysiącach iteracji, osobniki wykształcone w bardzo sprzyjających warunkach regionu północnego były w stanie przemieścić się na region zachodni, stanowiący najsłabszą niszę w całym środowisku.

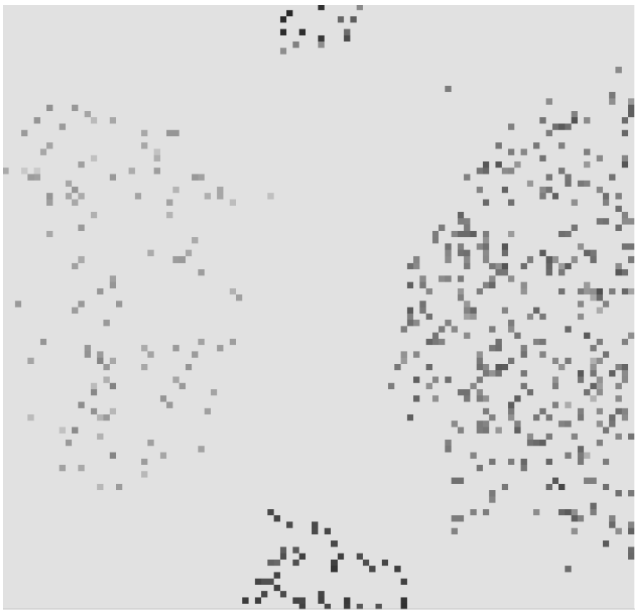
Nastąpiło utworzenie nowej, licznej kolonii na zachodzie, przy czym warunki przestrzenne powodowały, że migracja powrotna była ograniczona – populacja zachodnia „utknęła” w obrębie własnej niszy. Skutkiem tego był stopniowy rozwój w kierunku lepszego dopasowania do lokalnych warunków, obserwowany na przestrzeni kolejnych pokoleń [23]. Jednocześnie zaobserwowano istotną zmianę w rozkładzie energii w układzie. Początkowo, bezpośrednio po utworzeniu kolonii zachodniej, średni poziom energii w tym regionie był relatywnie wysoki, co wynikało z chwilowo korzystnego stosunku zasobów do liczby kolonistów oraz faktu, że organizmy prowadzące kolonizację pochodziły z bardzo korzystnej i bogatej w surowce północy. Po kilkudziesięciu pokoleniach jednak, wraz ze wzrostem liczebności populacji zachodniej, poziom energii mieszkańców tego regionu obniżył się do najniższych wartości w całym układzie. W konsekwencji spadek energii w tej niszy istotnie ograniczył możliwość powrotu osobników, które przystosowały się do gorszych warunków lokalnych. Zjawisko to znajduje również odzwierciedlenie w rozkładzie energii w stanie końcowym symulacji, omówionym w kolejnym podrozdziale (rysunek 7). W praktyce oznacza to, że model utrzymał jednocześnie populację funkcjonujące w warunkach bardzo sprzyjających (północ/południe) oraz populację zmuszoną do strategii niskonakładowej w warunkach ubogich (zachód) [22].

7.2.4. Stabilizacja długoterminowa

Po zakończeniu kolonizacji układ osiągnął stan względnej stabilizacji w skali globalnej. Mimo że w późniejszym okresie symulacji dostępność zasobów w całym układzie wzrosła w porównaniu do fazy początkowej (co wynikało z zajęcia dodatkowych, zasobnych regionów), nie zaobserwowano znacznego przekroczenia maksymalnej liczebności populacji osiągniętej w momencie kolonizacji południa. Populacja pozostała względnie stabilna przy jednoczesnych fluktuacjach charakterystycznych dla lokalnego przejadania zasobów oraz okresowych migracji. Taki przebieg wskazuje, że dalszy wzrost zasobów globalnych nie musi bezpośrednio przekładać się na wzrost liczebności populacji, jeżeli ograniczeniem stają się mechanizmy konkurencji lokalnej, koszty energetyczne eksploracji oraz restrykcje reprodukcyjne opisane w modelu.

W analizie zdolności adaptacyjnych populacji zaobserwowano wysokie wartości wskaźników dopasowania do warunków abiotycznych w ujęciu globalnym. Średnia adaptacja utrzymywała się w pobliżu ~95%, przy czym brak osiągania wartości idealnej interpretowany jest jako naturalna konsekwencja obecności mutacji oraz stałego mieszania się genotypów na granicach regionów. Jednocześnie obserwacje z interfejsu użytkownika wskazują, że osobniki sukcesywnie migrujące do nowych regionów wykazywały chwilowo niższy poziom adaptacji w porównaniu do populacji lokalnych, co jest spójne z intuicyjnym mechanizmem modelu: nagła zmiana środowiska powoduje czasowe niedopasowanie cech do nowych warunków, a dopiero kolejne pokolenia podlegają selekcji sprzyjającej lepszemu dostosowaniu do nowej niszy; efekt ten może być dodatkowo wzmacniany przez międzypokoleniowy wpływ warunków środowiskowych uwzględniony w modelu [23]. Ponieważ jednak migranci stanowili niewielki odsetek populacji globalnej, zjawisko to miało minimalny wpływ na miary uśrednione w skali całego układu.

Na rysunku 7 przedstawiono stan długoterminowej stabilizacji rozkładu energii w populacji. Rozkład pozostał wyraźnie zróżnicowany, co odzwierciedla trwałe istnienie nisz o odmiennej jakości środowiska. Najwyższe wartości energii utrzymywały się w regionach północnym i południowym, podczas gdy populacje funkcjonujące w obszarach mniej sprzyjających, w szczególności w regionie zachodnim, charakteryzowały się istotnie niższym poziomem energii, przy jednoczesnym zachowaniu stabilności liczebnej.



Rys. 7. Średni stan energii agentów w komórkach. Ostatnia iteracja symulacji
Źródło: opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji).

Na rysunku 8 przedstawiono rozmieszczenie przestrzenne populacji, które pozostawało silnie zróżnicowane pomiędzy poszczególnymi niszami środowiskowymi. Największa liczba agentów koncentrowała się w regionie wschodnim o umiarkowanych warunkach, podczas gdy regiony północny i południowy charakteryzowały się mniejszą populacją o zbliżonej gęstości, a region zachodni – dużą populacją o niskim zagęszczeniu.

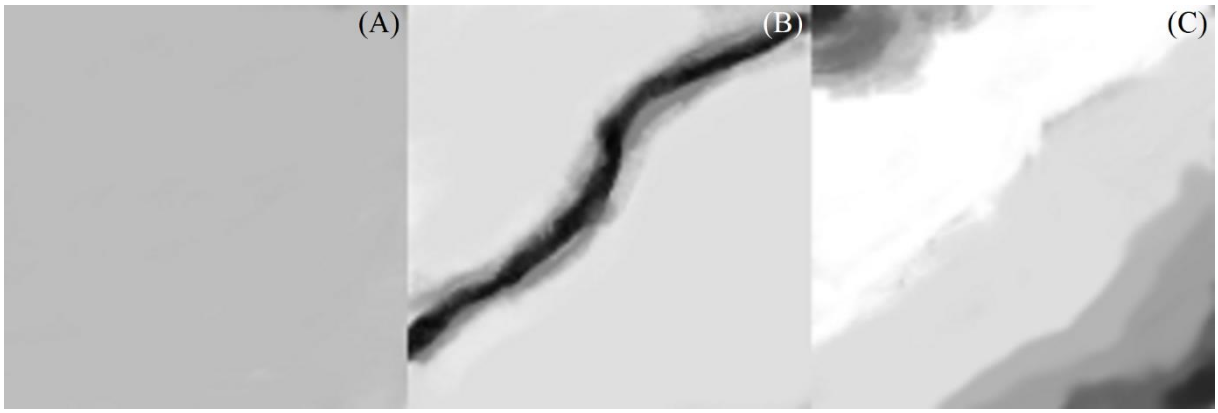
Rys. 8. Rozkład populacji. Ostatnia iteracja symulacji
Źródło: opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji).

7.2.5. Podsumowanie

Przebieg symulacji w środowisku o wysokiej restrykcyjności przestrzennej pokazał, że model generuje sekwencję faz obejmującą szybki wzrost populacji w niszy startowej, następnie wielokrotne nieudane próby eksploracji, aż do momentu przełomowych kolonizacji odległych, zasobnych regionów. Ograniczona łączność pomiędzy niszami sprawiała, że migracja miała charakter zdarzeń rzadkich i była zależna od pojawienia się osobników o wystarczającym potencjale energetycznym, co pełniło rolę naturalnego filtra selekcyjnego. W stanie długoterminowym układ osiągał stabilność globalną przy jednoczesnym utrzymaniu trwałego zróżnicowania przestrzennego energii i liczebności populacji pomiędzy regionami, co wskazuje na uformowanie się kilku nisz ekologicznych o odmiennych warunkach życia.

7.3. Drugi, walidacyjny eksperyment

W drugim eksperymencie zastosowano mapę o strukturze gradientowej, w której głównym elementem środowiska jest rzeka płynąca z północnego wschodu w kierunku południowego zachodu. W północno-zachodniej oraz południowo-wschodniej części mapy występują wzgórza, natomiast jakość warunków środowiskowych maleje wraz z oddalaniem się od rzeki. Rzeka stanowi obszar szczególnie zasobny, a populację startową zdefiniowano jako pojedynczego agenta umieszczonego na jej brzegu [24]. Mapy wejściowe użyte w eksperymencie połączono zbiorczo w rysunek 9. Uzyskana wegetacja startowa przedstawiona jest na rysunku 10.



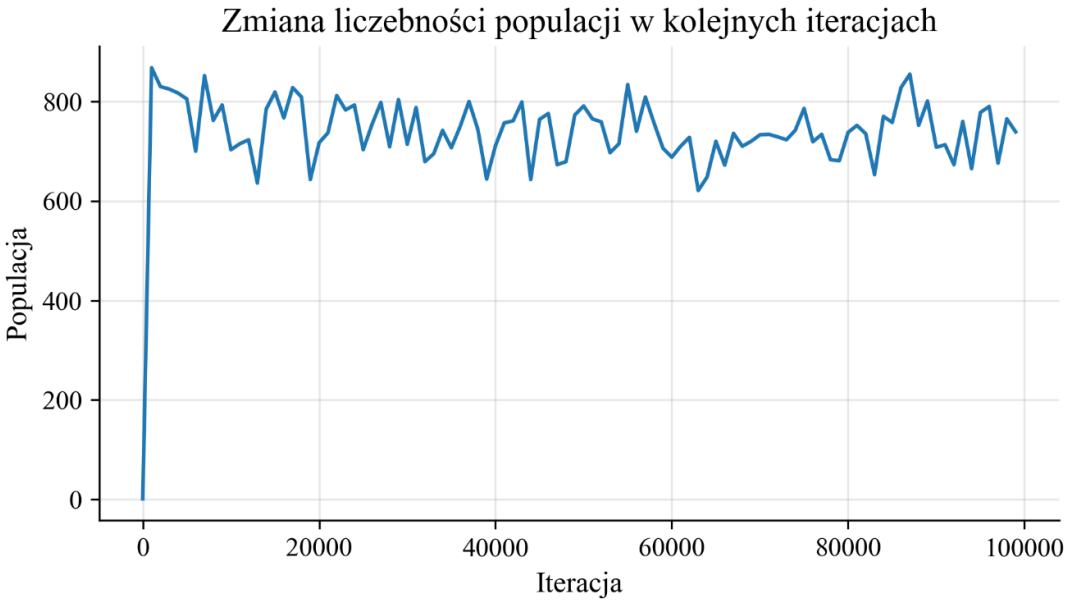
Rys. 9. Mapy wejściowe drugiego eksperymentu modelu. (A–C): (A) temperatura, (B) wilgotność, (C) wysokość
Źródło: opracowanie własne.

Rys. 10. Mapa wegetacji startowej powstałej z map wejściowych przedstawionych na rysunku 9. Znakiem X została zaznaczona pozycja startowa agenta
Źródło: opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji).

7.3.1. Wzrost populacji i szybka stabilizacja układu

W początkowej fazie symulacji zaobserwowano gwałtowny wzrost liczebności populacji, przy czym główna aktywność eksploracyjna przebiegała wzdłuż rzeki. W porównaniu do pierwszego eksperymentu kolonizacja przestrzeni następowała znacznie szybciej, co wynikało z braku wąskich gardeł migracyjnych i łatwej dostępności terenu.

Na rysunku 11 obserwujemy, że symulacja szybko weszła w stan quasi-stacjonarny, w którym zarówno liczebność populacji, statystyki energii i adaptacji ulegały jedynie niewielkim wahaniom. W dalszym przebiegu obserwowano oscylacje o małej amplitudzie, bez istotnych zmian w strukturze zasiedlenia mapy, co wskazuje na szybkie osiągnięcie równowagi między presją zasobową a mechanizmami reprodukcji i śmiertelności.

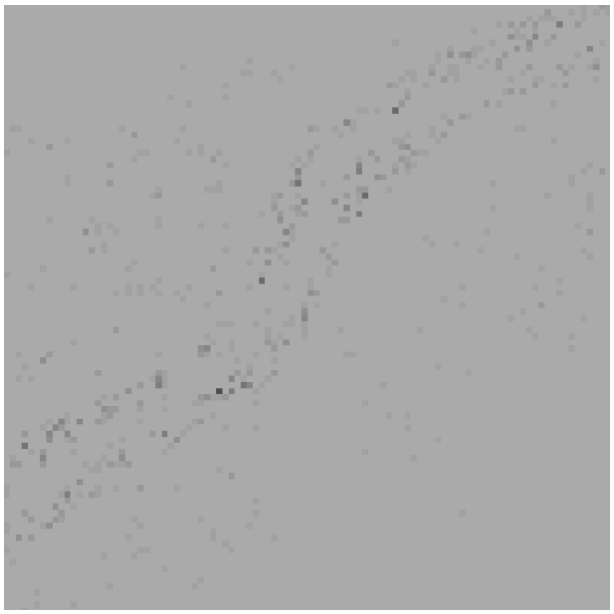


Rys. 11. Zmiana liczebności populacji w kolejnych iteracjach – eksperyment walidacyjny
Źródło: opracowanie własne.

7.3.2. Rozmieszczenie przestrzenne populacji i rola gradientu środowiskowego

W odróżnieniu od środowiska „labiryntowego", w drugim eksperymencie brak fizycznych barier powodował, że granice pomiędzy obszarami zasiedlenia nie były determinowane geometrią mapy, lecz przede wszystkim jakością środowiska.

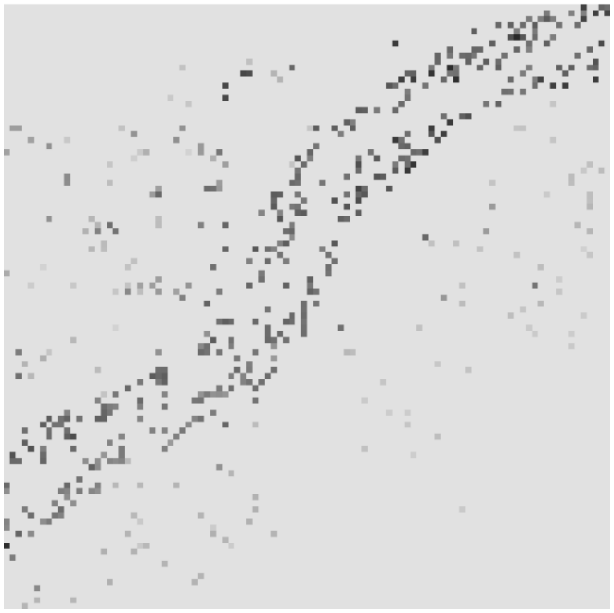
Na rysunku 12 przedstawiono końcowy rozkład populacji, który potwierdza, że największą gęstość zaludnienia zaobserwowano wzdłuż rzeki. Wraz z oddalaniem się od rzeki liczebność populacji malała, a w skrajnie niekorzystnych obszarach (zwłaszcza w rejonach wzgórz) populacja nie utrzymywała się w sposób trwały. Charakterystyczne było również to, że agenci niechętnie eksplorowali skrajne obszary mapy – w kierunku regionów o bardzo niskich zasobach – mimo że nie istniały przeszkody uniemożliwiające fizyczne dotarcie do tych miejsc. W praktyce widoczne było więc powstanie „pasa życia" wzdłuż rzeki oraz stopniowe rozrzedzenie populacji w miarę pogarszania się warunków. Jest to spójne z interpretacją modelu: przy braku barier topologicznych ostatecznym ograniczeniem staje się bilans energetyczny wynikający z jakości zasobów, kosztów ruchu oraz konkurencji lokalnej.



Rys. 12. Rozkład populacji. Ostatnia iteracja symulacji
Źródło: opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji)

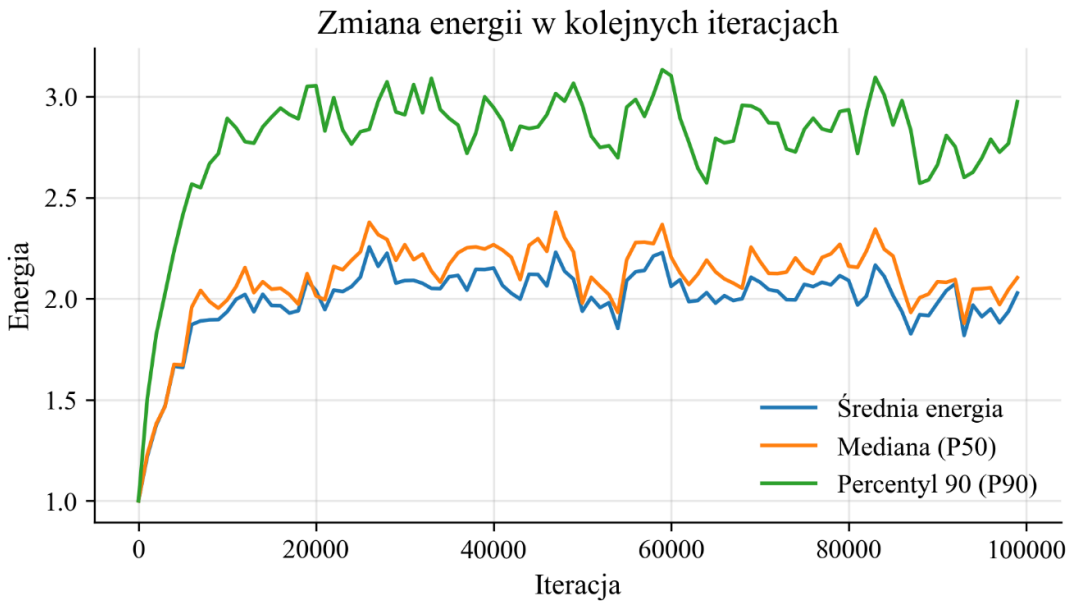
7.3.3. Zróżnicowanie energii i mieszanie populacji

Najwyższe wartości energii występowały bezpośrednio w pobliżu rzeki, natomiast w obszarach pozarzecznych obserwowano wyraźny spadek energii agentów. Oddalając się od rzeki można było zauważyć wyraźne obniżenie typowego poziomu energii (silniejsze na południu niż na północy), co potwierdza, że rzeka pełni rolę dominującej niszy o wysokiej jakości. Stan ten został odzwierciedlony na rysunku 13, przedstawiającym ostatnią iterację symulacji.



Rys. 13. Średni stan energii agentów w komórkach. Ostatnia iteracja symulacji. Ciemniejszy kolor oznacza wyższą wartość
Źródło: Opracowanie własne (zmodyfikowany zrzut ekranu z aplikacji)

Jednocześnie interesującą cechą środowiska gradientowego była duża zmienność energii w regionach ubogich: obok agentów o niskiej energii pojawiały się również osobniki o energii bardzo niskiej, co wskazuje na współistnienie wielu strategii przetrwania w warunkach ograniczonych zasobów. Na rysunku 14 można zaobserwować jeszcze większy rozrzut wartości energii niż w pierwszym eksperymencie (rysunek 6). 10% najsilniejszych osobników jeszcze bardziej zawyża średnia, która mimo wszystko pozostaje poniżej mediany co sugeruje obecność mniejszej populacji agentów o skrajnie niskiej energii.



Rys. 14. Zmiana energii w kolejnych iteracjach - drugi eksperyment
Źródło: opracowanie własne.

Zaobserwowano także zjawisko asymetrycznej migracji: częste były „wycieczki" mieszkańców rzeki w kierunku obszarów ubogich, po których następował powrót lub śmierć z głodu, natomiast osobniki z terenów ubogich relatywnie rzadko przemieszczały się w kierunku rzeki. Jedną z możliwych interpretacji jest rola konkurencji i ryzyka: agenci funkcjonujący w zasobnej niszy wzdłuż rzeki budują większy margines energetyczny, który pozwala im ponosić koszty czasowego niedopasowania do środowiska oraz okresowego braku pożywienia podczas eksploracji. Dla osobników z terenów ubogich margines ten jest mniejszy, a wejście w strefę wysokiej konkurencji w pobliżu rzeki może oznaczać brak dostępu do pożywienia przez kilka tur, co w warunkach niskiej energii istotnie zwiększa ryzyko śmierci.

Brak barier fizycznych skutkował również silniejszym „mieszaniem” osobników pomiędzy obszarami o zbliżonej jakości (np. na równinach pozarzecznych). W porównaniu do pierwszego eksperymentu, gdzie izolacja przestrzenna sprzyjała utrzymaniu wyraźnych podziałów populacji, w środowisku gradientowym granice te były mniej ostre: mimo że populacja preferowała obszary o korzystniejszych warunkach, ruch i eksploracja prowadziły do częstszego występowania osobników „nietypowych” dla danego miejsca (np. jednostek o bardzo niskiej energii pojawiających się okresowo w pobliżu rzeki).

7.3.4. Adaptacja abiotyczna w środowisku bez barier

W drugim eksperymencie średni poziom adaptacji globalnej był nieznacznie niższy niż w pierwszym (około 93% wobec ~95%) [24]. Analiza rozkładu przestrzennego adaptacji wskazuje jednak, że obniżenie to nie wynika z prostego gradientu odległości od rzeki, lecz z odmiennego charakteru funkcjonowania populacji poza główną niszą zasobową. W obszarach pozarzecznych poziom adaptacji był wyraźnie bardziej zróżnicowany i „wymieszany” przestrzennie, co można interpretować jako efekt bardziej koczowniczego trybu życia: agenci konsumują lokalne zasoby, po czym przemieszczają się dalej, często wchodząc w regiony o odmiennych parametrach środowiskowych, co prowadzi do obniżenia dopasowania.

Odmiennie kształtowała się sytuacja wzdłuż rzeki, gdzie wysoka dostępność zasobów oraz szybki odrost wegetacji sprzyjały osiadłemu trybowi życia. W tym przypadku presja selekcyjna nie polegała na ciągłej adaptacji do zmieniających się warunków, lecz na zdolności przetrwania okresów intensywnej konkurencji poprzez gromadzenie zapasów energii. W efekcie w skali całego układu obserwowany jest niższy poziom adaptacji uśrednionej, wynikający ze współistnienia stabilnej, dobrze dopasowanej populacji rzecznej oraz bardziej mobilnych subpopulacji funkcjonujących w warunkach mniej korzystnych.

7.3.5. Wniosek walidacyjny

Drugi eksperyment potwierdził, że obserwowane w pierwszej symulacji zjawiska (kolonizacja, stabilizacja globalna, różnicowanie nisz) wynikają z mechanizmów modelu, a nie ze specyfiki pojedynczej mapy. W środowisku gradientowym, pozbawionym barier geometrycznych, struktura zasiedlenia była determinowana głównie jakością warunków środowiskowych, co stanowi spójny i oczekiwany rezultat w kontekście przyjętych założeń modelu.

7.4. Wnioski z oceny modelu ekosystemu

Przeprowadzone eksperymenty symulacyjne pozwoliły ocenić zachowanie zaprojektowanego modelu ekosystemu w dwóch odmiennych scenariuszach środowiskowych: środowisku silnie restrykcyjnym przestrzennie oraz środowisku o łagodnych gradientach i wysokiej dostępności migracyjnej. W obu przypadkach zaobserwowano spójną i interpretowalną dynamikę populacji, wynikającą bezpośrednio z zaimplementowanych mechanizmów gospodarki energetycznej, selekcji abiotycznej oraz migracji.

W pierwszym eksperymencie model wykazał zdolność do formowania wyraźnych nisz ekologicznych oraz trwałego zróżnicowania populacji w przestrzeni [22]. Ograniczona łączność między regionami środowiskowymi prowadziła do izolacji populacji, lokalnej adaptacji oraz powstawania stabilnych strategii przetrwania dostosowanych do jakości środowiska. Kolonizacja nowych nisz następowała rzadko i była uzależniona od pojawienia się osobników o odpowiednim potencjale energetycznym, co potwierdza rolę kosztów energetycznych jako skutecznego mechanizmu selekcyjnego. Jednoczesne współistnienie populacji funkcjonujących w warunkach bardzo sprzyjających oraz populacji przystosowanych do środowisk ubogich wskazuje na zdolność modelu do utrzymywania długoterminowej różnorodności strategii życiowych [23].

Drugi eksperyment, pozbawiony barier topologicznych, ujawnił odmienny tryb funkcjonowania ekosystemu. Brak fizycznych ograniczeń migracji sprzyjał intensywnemu mieszaniu się populacji oraz częstszej eksploracji środowisk przy niedopasowanych parametrach. W konsekwencji zaobserwowano niższy poziom adaptacji globalnej, interpretowany jako efekt bardziej koczowniczego trybu życia organizmów oraz mniejszej presji na ścisłą specjalizację lokalną. Jednocześnie model poprawnie odtworzył koncentrację populacji w obszarach o najwyższej produktywności.

W obu scenariuszach gen energii okazał się skutecznym, syntetycznym wskaźnikiem jakości niszy oraz strategii przetrwania organizmów. Jego rozkład przestrzenny był spójny z rozkładem zasobów środowiskowych, a obserwowane fluktuacje energii i populacji wskazują na stabilną, nieliniową dynamikę typową dla systemów ekologicznych. Istotne jest również to, że wiele kluczowych zjawisk — takich jak kolonizacja, izolacja populacji, adaptacja migrantów czy lokalne strategie przetrwania — ujawnia się wyraźnie dopiero w obserwacji przestrzennej, co uzasadnia zastosowanie jakościowej analizy wizualnej jako uzupełnienia danych statystycznych.

Podsumowując, zaproponowany model spełnia założenia badawcze: umożliwia kształtowanie się nisz ekologicznych, różnicowanie strategii energetycznych oraz badanie wpływu struktury środowiska na dynamikę migracji i adaptacji populacji. Uzyskane wyniki wskazują, że nawet przy relatywnie prostych lokalnych regułach model generuje złożone, emergentne zachowania w skali globalnej, co czyni go użytecznym narzędziem eksploracyjnym do dalszych badań symulacyjnych.

8. Podsumowanie

Celem niniejszej pracy było opracowanie oraz analiza komputerowego modelu ekosystemu, a także implementacja środowiska symulacyjnego umożliwiającego prowadzenie eksperymentów numerycznych z jego wykorzystaniem. W ramach pracy zaprojektowano i zaimplementowano wieloplatformową, wydajną i rozszerzalną aplikację symulacyjną, a następnie wykorzystano ją do przeprowadzenia serii eksperymentów w celu oceny właściwości modelu.

8.1. Osiągnięcia inżynierskie

Z perspektywy inżynierskiej kluczowym rezultatem pracy jest powstanie modularnego środowiska symulacyjnego o architekturze ułatwiającej rozbudowę i eksperymentowanie z logiką modelu. Zastosowane podejście hybrydowe, łączące model ECS w warstwie symulacyjnej z podejściem obiektowym w podsystemach infrastrukturalnych, umożliwiło jednoczesne zachowanie wydajności przetwarzania populacji agentów oraz utrzymanie przejrzystej organizacji kodu w obszarach związanych z obsługą zasobów, wejścia/wyjścia i interfejsu użytkownika. Istotnym elementem opracowanego narzędzia jest również możliwość uruchamiania symulacji zarówno w trybie graficznym, jak i konsolowym, co wspiera automatyzację badań oraz pozwala integrować symulacje z zewnętrznymi procesami analizy danych [3, 24].

8.2. Wnioski modelowe

Z perspektywy modelowej praca doprowadziła do opracowania modelu ekosystemu, w którym dynamika populacji agentów wynika z lokalnych reguł gospodarki energetycznej, selekcji abiotycznej oraz interakcji ze środowiskiem reprezentowanym w formie automatu komórkowego. Przeprowadzona ocena wykazała, że model generuje spójne i interpretowalne zachowania emergentne, a jego dynamika jest silnie zależna od struktury środowiska [22].

W eksperymentach model konsekwentnie generował wiarygodną dynamikę ekosystemu: kolonizację przestrzeni, selekcję wynikającą z ograniczeń energetycznych, lokalne wyczerpywanie i odnawianie zasobów oraz przejście do stanu quasi-stacjonarnego. Jednocześnie obserwowano trwałe zróżnicowanie populacji w zakresie energii i strategii przetrwania, a także efekt migracji i konkurencji prowadzący do różnic w zagęszczeniu oraz stopniu „mieszania" osobników [23]. Wyniki wskazują, że mechanizmy modelu są spójne i pozwalają analizować zależność struktury zasiedlenia od jakości środowiska i kosztów ruchu [24].

8.3. Ograniczenia i kierunki dalszego rozwoju

Analiza zachowań przestrzennych w części opierała się na obserwacji jakościowej w GUI, gdyż wybrane zjawiska lokalne nie są bezpośrednio uchwytne w postaci miar globalnych bez dodatkowych narzędzi analitycznych. Model nie był również kalibrowany względem danych empirycznych, gdyż jego celem było badanie ogólnych mechanizmów dynamiki populacji w środowisku abstrakcyjnym.

W dalszych pracach zasadne jest rozszerzenie warstwy analitycznej o metryki przestrzenne i regionalne (m.in. identyfikację subpopulacji oraz analizę przepływów migracyjnych) oraz rozwinięcie przestrzeni cech agentów, tak aby możliwe było obserwowanie bardziej skrajnych adaptacji do warunków niekorzystnych. Z perspektywy narzędzia symulacyjnego istotnym kierunkiem rozwoju są również usprawnienia zwiększające ergonomię użytkowania i programowania, w szczególności w obszarze konfiguracji, automatyzacji i interpretacji danych.

8.4. Podsumowanie pracy

W pracy opracowano kompletne środowisko symulacyjne oraz model ekosystemu umożliwiający prowadzenie eksperymentów numerycznych i obserwację złożonych zjawisk emergentnych wynikających z prostych lokalnych reguł. Uzyskane rezultaty wskazują, że zaproponowane podejście stanowi stabilną podstawę do dalszej rozbudowy zarówno w zakresie architektury oprogramowania, jak i w zakresie samego modelu i metod jego analizy.

Bibliografia

[1] Muci A. L., Jorquera M. A., Ávila Á. I., Rengel Z., Crowley D. E., de la Luz Mora M., *A combination of cellular automata and agent-based models for simulating the root surface colonization by bacteria*, Ecological Modelling, vol. 247, s. 1–10, 2012.

[2] Breckling B., Pe'er G., Matsinos Y. G., *Cellular Automata in Ecological Modelling*, [w:] *Modelling Complex Ecological Dynamics*, Springer, Berlin 2011.

[3] Rak F., *ecosystem-cc – repozytorium GitHub*, https://github.com/Filip-Rak/ecosystem-cc, [dostęp: 10.01.2026].

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

[5] Wiebusch D., Latoschik M. E., *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.

[6] Caini M., *EnTT Documentation* , https://skypjack.github.io/entt/, [dostęp: 09 01 2026].

[7] Gomila L., *Documentation for SFML 3.0.1*, https://www.sfml-dev.org/documentation/3.0.1/, [dostęp: 09.01.2026].

[8] Cornut O., *Dear ImGui – repozytorium GitHub,* https://github.com/ocornut/imgui, [dostęp: 09.01.2026].

[9] Schreiner H., *Introduction – CLI11 Tutorial*, https://cliutils.github.io/CLI11/book/, [dostęp: 09.01.2026].

[10] Lohmann N., *Overview – JSON for modern C++,* https://json.nlohmann.me/api/basic_json/, [dostęp: 09.01.2026].

[11] Lattner C., *LLVM and Clang: Next Generation Compiler Technology*, https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf, 17.05.2008, [dostęp: 08.01.2026].

[12] Stroustrup B., Sutter H., *C++ Core Guidelines*, https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, [dostęp: 08.01.2026].

[13] LLVM Foundation, *The LLVM Compiler Infrastructure*, https://clangd.llvm.org/, [dostęp: 08.01.2026].

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

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

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

[17] Bevy Developer Team, *Bevy Engine Documentation*, https://bevy.org/learn/quick-start/introduction/, [dostęp: 09.01.2026].

[18] cppreference.com contributors, *cppreference.com*, https://cppreference.com, [dostęp: 10.01.2026].

[19] IEEE Standards Association, *IEEE Standard for Floating-Point Arithmetic*, IEEE, New York 2019.

[20] LLVM Project Contributors, *LLVM Language Reference Manual*, https://llvm.org/docs/LangRef.html#llvm-language-reference-manual, [dostęp: 12.01.2026].

[21] Barrett S., *stb – GitHub repository*, https://github.com/nothings/stb, [dostęp: 11.01.2026].

[22] Holt R. D., *Bringing the Hutchinsonian niche into the 21st century: Ecological and evolutionary perspectives*, Proceedings of the National Academy of Sciences (PNAS), vol. 106, suppl. 2, 2009

[23] Jablonka E., Raz G., *Transgenerational Epigenetic Inheritance: Prevalence, Mechanisms, and Implications for the Study of Heredity and Evolution*, The Quarterly Review of Biology, vol. 84, no. 2, 2009.

[24] Rak F., *ecosystem-cc-results – repozytorium GitHub*, https://github.com/Filip-Rak/ecosystem-cc-results, [dostęp: 20.01.2026].