Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему: «Розробка та програмна реалізація алгоритму моделювання зграйної поведінки в Unity»

Виконав: студент групи К23-1М

Спеціальність 122 Комп'ютерні науки

Зубко В.Д.
(прізвище та ініціали)

Керівник к.ф.-м.н., доц. Моромуль М.Ф.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Дніпровський державний технічний університет
(місце роботи)

доцент кафедри математичного моделювання та системного аналізу
(посада)

к.т.н., доц. Волосова Н.М.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Зубко В.Д. Розробка та програмна реалізація алгоритму моделювання зграйної поведінки в Unity.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 122 «Комп'ютерні науки». – Університет митної справи та фінансів, Дніпро, 2025.

Об'єктом дослідження є алгоритми моделювання поведінки зграй. Предмет дослідження – алгоритм Boids для симуляції зграйної поведінки в Unity.

Метою роботи є розробка алгоритму симуляції зграйної поведінки для середовища Unity з використанням алгоритму Boids, який забезпечує реалістичну поведінку агентів у віртуальному просторі.

Кваліфікаційна робота присвячена розробці та дослідженню алгоритму моделювання зграйної поведінки об'єктів у Unity з використанням алгоритму Boids. У роботі проведено аналіз методів моделювання групової поведінки, зокрема алгоритмів рою частинок, клітинних автоматів та ройового інтелекту. Визначено принципи уникнення зіткнень, вирівнювання руху та когезії як ключові елементи алгоритму Boids. Алгоритм реалізовано у середовищі Unity з використанням компонентно-орієнтованої архітектури ECS та мови програмування C#. Проведено тестування працездатності та продуктивності алгоритму, що підтвердило його ефективність у симуляціях великої кількості агентів.

Практична цінність роботи полягає у створенні алгоритму, який може бути застосований для розробки відеоігор, анімаційних фільмів, навчальних симуляцій, досліджень у сфері штучного інтелекту та симуляцій натовпів.

Ключові слова: алгоритм Boids, Unity, ECS, C#, моделювання, симуляція.

ABSTRACT

Zubko V. Development and software implementation of the swarm behavior modeling algorithm in Unity.

Diploma project for obtaining a master's degree in speciality 122 "Computer Science." - University of Customs and Finance, Dnipro, 2025.

The object of the study is swarm behavior modeling algorithms. The subject of the study is the Boids algorithm for simulating swarm behavior in Unity.

The purpose of the work is to develop a swarm behavior simulation algorithm for the Unity environment using the Boids algorithm, which ensures realistic agent behavior in a virtual space.

The qualification work is dedicated to the development and research of a swarm behavior modeling algorithm in Unity using the Boids approach. The study includes an analysis of group behavior modeling methods, such as particle swarm optimization, cellular automata, and swarm intelligence algorithms. The principles of collision avoidance, alignment, and cohesion are identified as the key elements of the Boids algorithm. The algorithm was implemented in the Unity environment using an Entity Component System (ECS) architecture and the C# programming language. The developed algorithm underwent functionality and performance testing, which confirmed its effectiveness in simulating a large number of agents.

The practical significance of the work lies in creating an algorithm applicable for video game development, animated films, educational simulations, artificial intelligence research, and crowd behavior modeling.

Keywords: Boids algorithm, Unity, ECS, C#, modeling, simulation.

ЗМІСТ

ВСТУП

У сучасному світі програмування та комп'ютерних технологій важливим завданням є створення ефективних та оптимізованих програмних рішень, здатних обробляти великі обсяги даних у реальному часі. Це особливо актуально у розробці програмних систем, орієнтованих на візуалізацію процесів, симуляцію взаємодії об'єктів або керування складними обчислювальними алгоритмами.

Актуальність даного дослідження полягає у необхідності створення продуктивних алгоритмічних рішень для моделювання поведінки об'єктів, що діють у групах або зграях, таких як частинки у фізичних симуляціях, агенти у комп'ютерних іграх або моделювання природних явищ. Використання принципів зграйної поведінки дозволяє моделювати динамічні системи, де кожен об'єкт діє незалежно, але зважає на сусідні елементи, створюючи складну групову динаміку.

Груповий інтелект тварин часто перевершує когнітивні здібності окремих особин. Наприклад, поведінка мурах може створювати враження, що ці комахи діють згідно з чітким планом, усвідомлюючи свої цілі та напрямок руху. Однак, насправді кожна окрема мураха не володіє значними знаннями або навичками. При виникненні нової ситуації вона може бути дезорієнтованою. Колонії мурах демонструють здатність вирішувати складні завдання, недосяжні для окремих особин, зокрема пошук оптимального маршруту до джерела їжі, розподіл обов'язків між членами спільноти та захист території. Джерело цього явища пов’язане з фундаментальною проблемою біології: як із простих дій окремих особин виникає складна поведінка групи? Ця здатність до узгоджених дій, коли кінцева мета невідома для індивіда, залишається однією з ключових загадок науки. Проте останні десятиліття принесли значний прогрес у розумінні цього феномена.

Одним із ключових відкриттів є те, що в колоніях мурах відсутні чіткі ролі керівників і підлеглих. Комунікація між мурахами відбувається за

допомогою тактильних і хімічних сигналів. Принцип ройового інтелекту полягає у слідуванні простим правилам, заснованим на обмеженій інформації, доступній кожній окремій особині. Така поведінка надихнула бельгійського дослідника Марко Дориго на розробку математичних алгоритмів, що базуються на поведінці мурах. Ці алгоритми використовуються для вирішення складних задач оптимізації, таких як планування транспортних маршрутів, складання графіків авіаперевезень і управління військовими роботами.

Одним із прикладів успішного застосування таких алгоритмів є система логістики компанії American Air Liquide, що займається постачанням газів для промислових і медичних цілей. Для оптимізації доставки до тисяч споживачів компанія використовує алгоритми, розроблені на основі досліджень аргентинських мурах, які прокладають феромонні траси до джерел їжі.

Ройова поведінка демонструє універсальні принципи: відсутність централізованого управління, координація через прості правила та адаптація до умов середовища. Це дозволяє спільнотам ефективно діяти в складних ситуаціях і забезпечувати виживання всієї групи.

У сучасній ігровій індустрії методи колективного руху агентів використовуються для створення реалістичних моделей поведінки натовпу, управління групами персонажів та моделювання складних сценаріїв взаємодії. Принципи ройового інтелекту застосовуються у симуляціях масових битв, руху натовпу та анімації зграйних структур.

Одним із ключових алгоритмів, що використовується у цій сфері, є метод *Boids*, розроблений Крейгом Рейнольдсом. Він ґрунтується на трьох простих правилах: уникнення зіткнень, вирівнювання відносно сусідів та тяжіння до центру групи. Цей підхід було адаптовано для моделювання поведінки зграйних об'єктів у серії ігор *Assassin's Creed*, де NPC реагують на дії гравця, створюючи ефект живого міста.

Новизна дослідження полягає у використанні алгоритму Boids для симуляції зграйної поведінки в середовищі Unity. Алгоритм заснований на трьох базових принципах: уникнення зіткнень, вирівнювання руху та когезії,

забезпечує реалістичне моделювання взаємодії агентів у просторі. Використання компонентно-орієнтованої архітектури (Entity Component System) у Unity дозволяє підвищити продуктивність та масштабованість симуляції.

Метою дослідження є розробка алгоритму симуляції зграйної поведінки для Unity з використанням алгоритму Boids, що включає ключові принципи зграйної динаміки та оптимізацію обчислювальних ресурсів.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Провести аналіз наукових джерел та існуючих алгоритмів моделювання зграйної поведінки.
2. Дослідити принципи алгоритму Boids та адаптувати його для реалізації у середовищі Unity.
3. Розробити програмну реалізацію алгоритму з використанням компонентно-орієнтованої архітектури.
4. Виконати тестування програмного продукту для перевірки його функціональності та продуктивності.

Об'єкт дослідження – алгоритми моделювання зграйної поведінки.

Предмет дослідження – алгоритм Boids для симуляції зграйної поведінки в Unity.

Практичне значення роботи полягає у створенні програмного рішення, яке може бути використане у різних сферах: від відеоігор та анімації до наукових симуляцій. Розроблений алгоритм може слугувати основою для більш складних систем керування натовпами, симуляцій руху частинок або досліджень у сфері штучного інтелекту.

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Обсяг роботи становить 60 сторінок основного тексту, 38 рисунків та 1 таблиці.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Аналіз публікацій щодо моделювання поведінки зграї

Груповий інтелект тварин часто перевершує когнітивні здібності окремих особин. Наприклад, поведінка мурах може створювати враження, що ці комахи діють згідно з чітким планом, усвідомлюючи свої цілі та напрямок руху. Однак, насправді кожна окрема мураха не володіє значними знаннями або навичками. При виникненні нової ситуації вона може бути дезорієнтованою. Саме завдяки скоординованим діям у межах колонії, що об'єднує тисячі особин, мурахи змогли адаптуватися та успішно існувати на планеті протягом понад 140 мільйонів років.

Колонії мурах демонструють здатність вирішувати складні завдання, недосяжні для окремих особин, зокрема пошук оптимального маршруту до джерела їжі, розподіл обов'язків між членами спільноти та захист території. Незважаючи на те, що окрема мураха є досить примітивною істотою, вся колонія здатна ефективно реагувати на загрози та нові умови завдяки феномену колективного інтелекту [1].

Джерело цього явища пов’язане з фундаментальною проблемою біології: як із простих дій окремих особин виникає складна поведінка групи? Ця здатність до узгоджених дій, коли кінцева мета невідома для індивіда, залишається однією з ключових загадок науки. Проте останні десятиліття принесли значний прогрес у розумінні цього феномена.

Одним із ключових відкриттів є те, що в колоніях мурах відсутні чіткі ролі керівників і підлеглих. Навіть при кількості особин, що перевищує півмільйона, колонія функціонує ефективно. Це пояснюється безперервною комунікацією між особинами, які слідують простим правилам взаємодії, формуючи систему самоорганізації.

Комунікація між мурахами відбувається за допомогою тактильних і хімічних сигналів. Наприклад, під час зустрічі мураха за допомогою вусиків визначає, чи належить інший мураха до тієї ж колонії, а також може дізнатися, де той працював (мурахи, які повернулися з пошуку їжі, пахнуть інакше, ніж ті, хто залишався в гнізді). Перед виходом з гнізда фуражири (збирачі їжі) контактують із патрульними мурахами, які оцінюють безпеку зовнішнього середовища. Якщо контактів із патрульними достатньо, фуражири приймають рішення покинути мурашник. Відповідно, рішення щодо збору їжі приймається колективно, а не окремими особинами.

Цей принцип ройового інтелекту полягає у слідуванні простим правилам, заснованим на обмеженій інформації, доступній кожній окремій особині. Така поведінка надихнула бельгійського дослідника Марко Дориго на розробку математичних алгоритмів, що базуються на поведінці мурах. Ці алгоритми використовуються для вирішення складних задач оптимізації, таких як планування транспортних маршрутів, складання графіків авіаперевезень і управління військовими роботами.

Одним із прикладів успішного застосування таких алгоритмів є система логістики компанії American Air Liquide, що займається постачанням газів для промислових і медичних цілей. Для оптимізації доставки до тисяч споживачів компанія використовує алгоритми, розроблені на основі досліджень аргентинських мурах, які прокладають феромонні траси до джерел їжі.

Ройовий інтелект притаманний не лише мурахам, а й іншим тваринам, зокрема медоносним бджолам. Коли колонія бджіл стає перенаселеною, вона ділиться: частина робочих бджіл із маткою залишає гніздо для пошуку нового житла. Процес вибору нового місця відбувається колективно: група бджіл досліджує потенційні варіанти і, обмінюючись інформацією, приходить до спільного рішення. Така стратегія дозволяє бджолам уникнути помилок, оскільки вибір житла є критично важливим для виживання колонії [2].

Бджолиний метод прийняття рішень, заснований на зборі даних, конкуренції ідей та усуненні невдалих варіантів, можна застосовувати навіть

у людських спільнотах. Наприклад, аналогічний принцип використовується у прогнозуванні результатів кінних перегонів, де ставки гравців формують колективний прогноз, часто дуже точний.

Групова поведінка спостерігається і серед інших тварин. Зграї птахів, косяки риб та стада копитних тварин діють узгоджено, що допомагає їм ефективно виявляти хижаків, знаходити їжу та координувати міграцію. У таких спільнотах виживання значною мірою залежить від здатності групи координувати свої дії [3].

Особливий інтерес викликає ройова поведінка птахів. Під час польоту кожен птах орієнтується на рух сусідів, дотримуючись трьох базових правил: уникати зіткнень, дотримуватися середньої траєкторії польоту і залишатися поблизу групи. Саме ці принципи стали основою для створення алгоритмів штучного інтелекту, зокрема програми Крейга Рейнолдса, яка моделювала рух рою роботів.

Ці роботи, подібно до птахів, координували свої дії без централізованого управління, що підвищувало ефективність їхньої роботи. Такий підхід може бути використаний у майбутньому для пошуково-рятувальних операцій, екологічного моніторингу та навіть військових завдань, оскільки рій здатний швидко адаптуватися до змін і замінити пошкоджених роботів без втрати працездатності всієї системи.

Таким чином, незалежно від того, чи йдеться про комах, птахів чи інших тварин, ройова поведінка демонструє універсальні принципи: відсутність централізованого управління, координація через прості правила та адаптація до умов середовища. Це дозволяє спільнотам ефективно діяти в складних ситуаціях і забезпечувати виживання всієї групи.

У сучасній ігровій індустрії методи колективного руху агентів використовуються для створення реалістичних моделей поведінки натовпу, управління групами персонажів та моделювання складних сценаріїв взаємодії. Принципи ройового інтелекту застосовуються у симуляціях масових битв, руху натовпу та анімації зграйних структур.

Одним із ключових алгоритмів, що використовується у цій сфері, є метод *Boids*, розроблений Крейгом Рейнольдсом. Він ґрунтується на трьох простих правилах: уникнення зіткнень, вирівнювання відносно сусідів та тяжіння до центру групи. Цей підхід було адаптовано для моделювання поведінки зграйних об'єктів у серії ігор *Assassin's Creed*, де NPC реагують на дії гравця, створюючи ефект живого міста.

У жанрі стратегій реального часу (RTS) алгоритми ройового інтелекту використовуються для управління великими групами бойових одиниць. У серії *Total War* застосовується система, що моделює поведінку тисяч солдатів, які координують свої дії через прості правила взаємодії: уникнення зіткнень, збереження формації та підтримка позицій. Подібні підходи використовуються у *StarCraft II* та *Age of Empires* для управління арміями.

Ігрові рушії *Unreal Engine* та *Unity* містять вбудовані інструменти для створення складних сценаріїв поведінки натовпу, зокрема системи *NavMesh* та *Crowd Simulation*, які використовують алгоритми ройового інтелекту для управління NPC.

У жанрі survival horror ройовий інтелект застосовується для моделювання масової поведінки ворогів. У серіях *Left 4 Dead* та *World War Z* створено системи управління ордами зомбі, де кожен елемент реагує на події у грі, створюючи атмосферу загрози та хаосу.

У мультиплеєрних іграх алгоритми ройового інтелекту використовуються для координації дій ботів. Наприклад, у *Battlefield 2042* NPC взаємодіють у складі бойових загонів, аналізуючи позиції союзників і противників.

1.2 Аналіз методів моделювання поведінки зграї

Моделювання є потужним інструментом, що дозволяє поєднувати знання з різних навчальних дисциплін. Як традиційне, так і комп'ютерне моделювання здатні виконувати важливі функції у вирішенні різноманітних проблем. Завдяки моделюванню можна оцінити наслідки впливу антропогенних факторів, що, своєю чергою, дає змогу запобігти небажаним, а інколи й катастрофічним явищам на глобальному рівні. Це стосується, наприклад, змін клімату на Землі або потенційних загроз від космічних об'єктів для існування людства. Отже, прогнозування сприяє формуванню змісту та різних підходів до мислення в умовах сучасності [4].

Для будь-якого об'єкта, залежно від цілей дослідження, можна створити різноманітні моделі. Наприклад, у випадку складної системи, такої як популяція тварин, для опису життєвих процесів окремої тварини як біологічного об'єкта може бути використана одна модель. Якщо ж досліднику необхідно проаналізувати поведінку тварин у зграї, він застосує етологічну (поведінкову) модель. Для прогнозування змін у чисельності популяції знадобиться вже інша – екологічна модель. Це підкреслює, що характеристики об'єкта, значущі для побудови однієї моделі, можуть виявитися несуттєвими для іншої. Завдання дослідника полягає у пошуку балансу: створити модель процесу, зберігаючи його ключові властивості.

У більшості сучасних моделей поведінки тварин індивіди розглядаються як частинки, що взаємодіють відповідно до фізичних законів. У процесі руху таких частинок діє сила притягання, тоді як під час зіткнень із перешкодами або іншими особинами активізується сила відштовхування. Подібні підходи ефективно описують динаміку малих груп тварин, однак зі збільшенням щільності потоку математичний апарат значно ускладнюється, оскільки для коректного опису явища виникає потреба у введенні додаткових параметрів.

Ось деякі з цих методів:

1) Алгоритм рою частинок

Метод оптимізації за допомогою рою частинок (Particle Swarm Optimization, PSO) базується на моделюванні поведінки множини агентів (частинок) у просторі параметрів задачі оптимізації. Даний підхід є ефективним завдяки простоті реалізації та відсутності потреби у використанні градієнтної інформації під час обчислень, що робить його придатним для розв’язання задач нелінійної оптимізації, навчання нейронних мереж, пошуку глобальних мінімумів функцій, а також вирішення задач комбінаторної оптимізації, часто застосовуваних у генетичних алгоритмах.

PSO належить до класу еволюційних алгоритмів, які засновані на принципах природних процесів і використовують стохастичний підхід до пошуку оптимальних рішень. Важливою характеристикою методу є те, що він не потребує обчислення градієнта, що забезпечує ефективність у задачах із складними, нерегулярними або недиференційовними цільовими функціями. Це робить PSO особливо цінним для оптимізаційних задач із високою обчислювальною складністю або за умов, коли градієнт важко або неможливо обчислити.

У даному алгоритмі кожен агент представлений частинкою, яка характеризується поточним положенням у просторі параметрів, вектором швидкості та відповідним значенням цільової функції. Процес пошуку оптимального розв’язку відбувається через ітеративне оновлення положень і швидкостей частинок. Положення частинок оновлюються відповідно до їхнього попереднього досвіду (інерційного компонента), найкращого знайденого рішення частинкою (компонент локального пошуку) та найкращого глобального рішення в популяції (компонент глобального пошуку). Вагові коефіцієнти кожного з цих факторів контролюють баланс між глобальним та локальним пошуком [5].

Концепція PSO натхненна колективною поведінкою тварин, зокрема зграї птахів або косяка риб. Ідея цього алгоритму бере початок із графічного моделювання непередбачуваних траєкторій руху птахів у зграї з метою дослідження механізмів синхронного польоту, уникнення зіткнень та

формування оптимальних структур. У такій груповій поведінці відсутній централізований контроль: кожен індивід коригує свою траєкторію на основі спостереження за сусідами та інформації про знайдені ресурси або загрози.

Поведінка зграї є прикладом ройового інтелекту, що демонструє децентралізовану координацію та самоорганізацію. У цьому контексті групова поведінка часто сприяє підвищенню загальної ефективності популяції, оскільки обмін інформацією між особинами дозволяє швидше знаходити ресурси або уникати небезпек. Попри те, що конкуренція за ресурси може спричиняти локальні конфлікти, загальний ефект є вигідним для виживання виду, оскільки збільшує ймовірність виявлення критично важливих ресурсів для всієї популяції.

Алгоритм Boids

У 1986 році Крейг Рейнольдс, спостерігаючи за поведінкою зграї птахів, розробив комп'ютерну модель під назвою *Boids* (від англ. "bird-oid objects" — об'єкти, подібні до птахів). Модель була створена для імітації групової поведінки через керування окремими агентами (птахами) у системі. Основою алгоритму є три фундаментальні принципи, що визначають взаємодію між агентами:

- Уникнення зіткнень – кожен агент прагне уникнути фізичного контакту з сусідніми агентами.
- Вирівнювання – кожен агент коригує свою траєкторію відповідно до середнього напряму руху сусідніх агентів.
- Тяжіння – кожен агент намагається підтримувати рівномірну відстань до сусідніх агентів.

Попри простоту заданих правил, результати симуляції показали високу візуальну правдоподібність: агенти групувалися у компактні формації, координували рух, уникали зіткнень та демонстрували хаотичний, але скоординований рух, що наближений до поведінки реальних зграї птахів або косяків риб.

Первинною метою Рейнольдса було створення візуально реалістичної симуляції для використання у комп'ютерній графіці та анімації. Втім, у своїй науковій публікації він відзначив, що запропонований підхід може бути адаптований для моделювання складніших сценаріїв, таких як пошук ресурсів, уникнення хижаків або навігація в динамічних середовищах [6].

Важливо зазначити, що алгоритм Boids є основою для досліджень у таких галузях, як штучний інтелект, моделювання колективної поведінки та динаміка складних систем. Його принципи часто використовуються у створенні багатоагентних систем, включаючи робототехнічні рої та системи управління автономними транспортними засобами.

Алгоритм рою частинок GBEST

У 1995 році Джеймс Кеннеді та Рассел Еберхарт розробили метод оптимізації безперервних нелінійних функцій, який отримав назву алгоритм рою частинок (Particle Swarm Optimization, PSO). Цей підхід ґрунтувався на концепціях, представлених у моделі Boids Крейга Рейнольдса, а також роботах Хеппнера та Гренадера, присвячених колективній поведінці.

Основний принцип PSO полягає в моделюванні групи частинок (агентів), що рухаються у просторі можливих рішень з метою знайти глобальний оптимум функції. Як і у випадку моделі Boids, кожна частинка взаємодіє з іншими за допомогою простих правил:

- Частинки прагнуть наблизитися до найкращого знайденого рішення (глобального оптимуму).
- Кожна частинка рухається у напрямку, що поєднує її попередній досвід (локальний оптимум) та найкраще рішення всієї групи.

Алгоритм PSO є відносно простим для реалізації та може бути описаний лише кількома десятками рядків коду. Його ефективність підтверджена у широкому спектрі оптимізаційних задач, включаючи:

- навчання нейронних мереж,
- оптимізацію параметрів у технічних системах,
- мінімізацію функцій складних топологій.

Попри свою простоту, метод PSO має певні обмеження, зокрема:

- Схильність до передчасної збіжності: частинки можуть застрягти у локальному мінімумі.
- Вибір параметрів: значення коефіцієнтів інерції та навчання суттєво впливають на швидкість та якість збіжності.

Таким чином, алгоритм рою частинок демонструє ефективність у задачах оптимізації завдяки своїй простоті, здатності до глобального пошуку та біологічно натхненному підходу до обміну інформацією між агентами. Це робить PSO одним із ключових інструментів у галузі еволюційних обчислень та штучного інтелекту.

2) Алгоритм клітинних автоматів

Клітинні автомати (Cellular Automata, CA) — це дискретні математичні моделі, призначені для моделювання складних систем через розбиття простору на регулярну сітку клітин. Кожна клітина цієї сітки може перебувати в одному з декількох скінченних станів, а її динаміка визначається простими правилами, які залежать від її власного стану та станів сусідніх клітин. Всі клітини оновлюються одночасно на кожному кроці симуляції, що забезпечує узгоджену еволюцію системи в часі. Ключовими компонентами клітинних автоматів є дискретний простір, дискретний час, локальні правила оновлення стану та обмеженість взаємодії лише найближчими сусідами клітини. Простота математичної основи клітинних автоматів дозволяє моделювати широкий спектр складних явищ, включаючи колективну поведінку живих систем.

Алгоритм клітинних автоматів знайшов широке застосування у моделюванні поведінки зграї птахів, косяків риб, а також інших систем, що демонструють колективну самоорганізацію. Моделювання зграї ґрунтується на представленні простору як регулярної сітки, де кожна клітина може бути або порожньою, або містити агента (птаха чи рибу). Поведінка кожного агента

описується локальними правилами взаємодії з сусідніми агентами, що дозволяє досягти узгодженого групового руху. В основі алгоритму лежать три ключові принципи: уникнення зіткнень, вирівнювання напрямку руху та когезія (прагнення залишатися в межах групи). Уникнення зіткнень реалізується через зміну напрямку руху агента у разі наближення до сусідів на критично малу відстань. Принцип вирівнювання означає, що агент коригує свій напрямок у відповідності до середнього напряму руху сусідів, а когезія забезпечує прагнення агента залишатися на середній відстані від центру групи сусідніх агентів.

Формально динаміка клітинних автоматів у контексті моделювання зграї може бути представлена через функцію оновлення, де стан кожної клітини залежить від її поточного стану та станів сусідніх клітин у локальній околиці. Зокрема, новий стан клітини на наступному часовому кроці обчислюється як функція від поточного стану та середніх параметрів (позиції, швидкості, орієнтації) її сусідів. Важливим аспектом є синхронне оновлення всієї системи на кожному кроці, що забезпечує одночасне врахування взаємодії всіх агентів у зграї.

Основною перевагою клітинних автоматів у моделюванні колективної поведінки є їхня здатність до децентралізованого управління системою. Оскільки стан кожної клітини визначається лише локальною взаємодією з сусідами, модель ефективно відображає природні системи, де індивіди приймають рішення на основі інформації, доступної у безпосередньому оточенні. Простота правил та обчислювальна ефективність роблять клітинні автомати зручними для симуляцій великих груп агентів із мінімальними обчислювальними витратами [7].

Клітинні автомати демонструють високу гнучкість у застосуванні для різноманітних завдань моделювання. В екологічних дослідженнях вони використовуються для симуляції міграційних процесів у популяціях птахів та косяків риб, а також для вивчення біорізноманіття у складних екосистемах. У комп’ютерній графіці клітинні автомати застосовуються для створення

реалістичних анімацій руху натовпів, зграї птахів у фільмах та відеоіграх. У сфері фізики вони застосовуються для моделювання поширення вогню, хвильових процесів та динаміки рідин.

Таким чином, клітинні автомати є ефективним інструментом для моделювання руху зграї, оскільки забезпечують децентралізоване управління системою, високу обчислювальну ефективність та здатність відображати складну колективну поведінку на основі простих локальних правил. Простота реалізації та можливість застосування у багатьох сферах роблять їх універсальним інструментом у галузі моделювання складних систем.

3) Застосування ШІ

Метод графових нейронних мереж (GNN) може бути використаний для моделювання руху зграї птахів шляхом представлення кожного агента (птаха) у вигляді вузла графа, де зв'язки між вузлами описують локальні взаємодії з найближчими сусідами. Цей підхід дозволяє моделювати децентралізовану самоорганізовану поведінку, характерну для зграї птахів у природі.

Основний принцип GNN для моделювання зграї полягає у побудові графа, де вершини представляють окремих агентів, а ребра — взаємозв'язки між ними на основі заданого радіусу взаємодії. Кожен вузол зберігає інформацію про поточний стан агента, включаючи його швидкість, положення та орієнтацію. Оновлення стану кожного вузла відбувається за допомогою нейронної мережі, яка враховує інформацію з локального оточення.

Основними правилами взаємодії між агентами у такій моделі є:

Уникнення зіткнень (Separation): агенти знижують швидкість або змінюють напрямок руху при надто близькому наближенні один до одного.

Вирівнювання напрямку руху (Alignment): кожен агент коригує свій напрямок відповідно до середнього напряму сусідніх агентів.

Когезія (Cohesion): агенти прагнуть рухатися в напрямку центра мас локальних сусідів, зберігаючи згуртованість зграї.

Метод GNN дозволяє агентам адаптувати свою поведінку залежно від локальних факторів. Для цього кожен вузол графа використовує функцію агрегації, яка збирає інформацію від найближчих сусідів, та функцію оновлення, яка обчислює новий стан агента на основі цієї інформації. Такий підхід є ефективним у випадках, коли агенти мають обмежену зону огляду або комунікації.

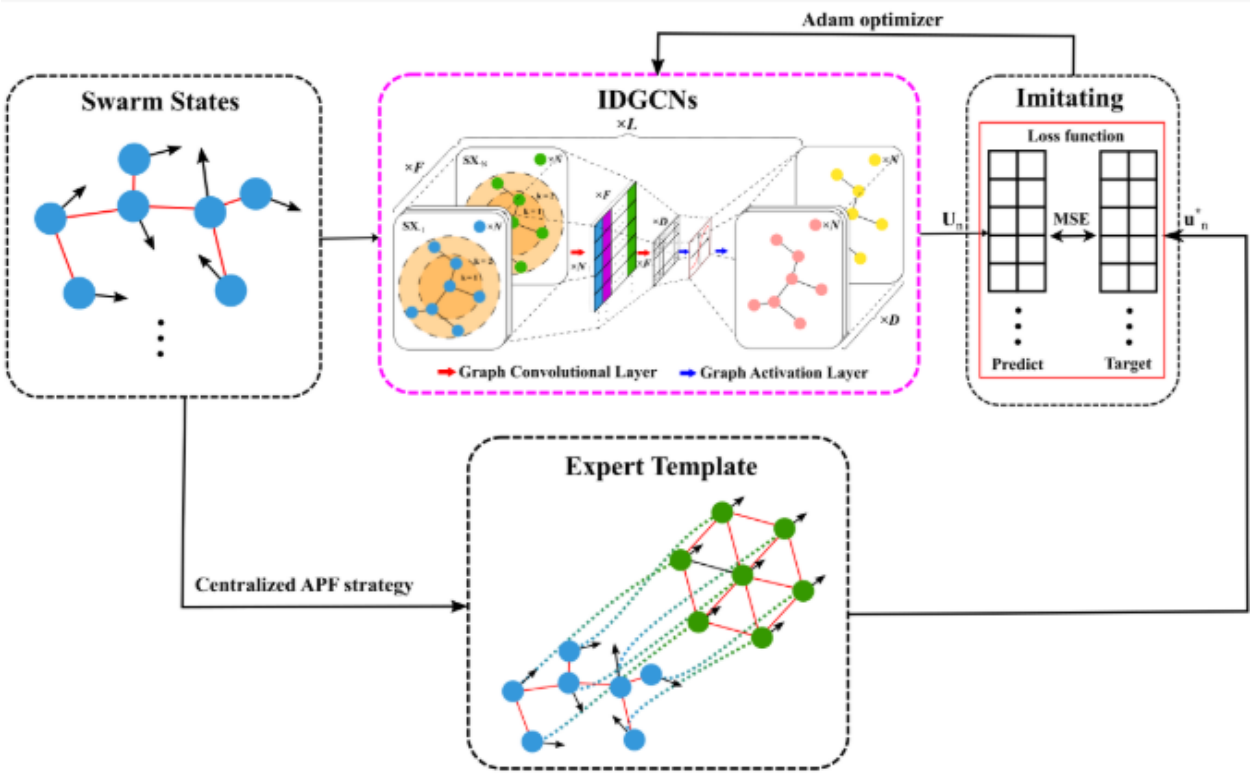


Рисунок 1.1 – Метод GNN

Використання GNN у моделюванні руху зграї дозволяє досягти реалістичних результатів завдяки здатності мережі адаптивно оновлювати зв'язки між агентами у процесі навчання. Алгоритм може бути розширений через включення додаткових факторів, таких як стохастичність руху, вагові коефіцієнти впливу сусідів, часові затримки та врахування зовнішніх факторів, наприклад, вітру або наявності перешкод у просторі.

Метод GNN також може бути використаний для моделювання руху роїв дронів або автономних транспортних систем, де необхідна координація

великої кількості агентів без централізованого контролю. Завдяки здатності до самоорганізації, такі системи демонструють високу стійкість до зовнішніх збурень та можуть використовуватися для моделювання поведінки у складних середовищах.

1.3 Висновок до першого розділу

У даному розділі було проведено аналіз методів та алгоритмів моделювання поведінки зграї, включаючи алгоритми Boids, рою частинок (PSO) та клітинних автоматів. Проаналізовано ключові підходи до симуляції колективної поведінки, зокрема правила вирівнювання, когезії та уникнення зіткнень, які лежать в основі алгоритму Boids. Було розглянуто принципи децентралізованої координації в природних системах, таких як колонії мурах, зграї птахів та косяки риб, а також їх використання у комп'ютерному моделюванні.

Метою дослідження є розробка алгоритму симуляції зграйної поведінки для Unity на основі принципів Boids для реалістичної візуалізації поведінки груп агентів у 2D-середовищі.

Для досягнення поставленої мети було визначено та виконано такі завдання дослідження:

1. Проведено аналіз наукових джерел щодо алгоритмів зграйної поведінки та методів їх моделювання.
2. Розроблено алгоритм Boids із застосуванням правил вирівнювання, когезії та уникнення зіткнень.
3. Виконано програмну реалізацію алгоритму у середовищі Unity.
4. Проведено тестування програмного продукту для перевірки його відповідності функціональним і нефункціональним вимогам.

Вхідними даними для алгоритму є набір агентів, які взаємодіють за допомогою локальних правил, зберігаючи згуртованість, синхронність руху та запобігаючи зіткненням.

РОЗДІЛ 2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ АЛГОРИТМУ МОДЕЛЮВАННЯ ПОВЕДІНКИ ЗГРАЇ

2.1 Вибір програмних засобів

Програмне забезпечення для симуляції зграйної поведінки має відповідати ряду ключових вимог, щоб забезпечити ефективність і точність моделювання. Воно повинно забезпечувати високу продуктивність, зокрема оптимізовану обробку великих обсягів даних, що є критично важливим при роботі з сотнями або тисячами агентів у реальному часі. Для цього необхідна підтримка багатопотоковості та використання обчислювальних оптимізацій, таких як SIMD-інструкції та GPU-прискорення [11].

Масштабованість системи повинна забезпечувати стабільну роботу як із малими, так і великими групами агентів без суттєвого зниження продуктивності. Гнучкість у налаштуванні параметрів, таких як швидкість, радіус огляду, сили впливу та взаємодії агентів, дозволяє адаптувати симуляцію під різні сценарії досліджень та тестів.

Інтерактивність системи є важливою для наукових експериментів та візуалізацій — користувач повинен мати можливість змінювати параметри симуляції без перезапуску сцени.

Окремо слід виділити важливість наявності засобів налагодження, зокрема візуальних інструментів для демонстрації векторів сил та зон огляду агентів.

Програмне забезпечення має бути зручним у використанні, з інтуїтивним інтерфейсом, та підтримувати інтеграцію з популярними ігровими рушіями, такими як Unity або GodotEngine, що спрощує його використання у різних сферах, від розваг до наукових досліджень.

1. Unity

Unity є потужним багатоплатформовим рушієм для розробки інтерактивних додатків та відеоігор, створеним компанією Unity Technologies.

Завдяки компонентно-орієнтованій архітектурі, він дозволяє створювати як прості 2D-ігри, так і складні 3D-симуляції з підтримкою фізики, анімації, штучного інтелекту та кросплатформеності.

Основою архітектури Unity є система об'єктів сцени (*GameObject*), до яких додаються компоненти, що визначають їх функціональність. Наприклад, компоненти *Transform* відповідають за координати об'єкта у просторі, *Rigidbody* — за фізичні параметри, а *Mesh Renderer* — за візуалізацію. Unity використовує мову програмування C#, яка дозволяє створювати скрипти для керування поведінкою об'єктів. Скрипти інтегруються у середовище як компоненти об'єктів, що спрощує розробку складної логіки гри.

Графічний рушій Unity підтримує кілька варіантів рендерингу: *Built-in Render Pipeline*, *Universal Render Pipeline (URP)* для мобільних платформ і середніх ПК, а також *High Definition Render Pipeline (HDRP)* для високоякісної графіки, що часто використовується в AAA-проєктах. Для обробки фізики використовується рушій PhysX від NVIDIA, що забезпечує реалістичні зіткнення, гравітацію та симуляцію твердих тіл [12, 13].

Система анімації Unity включає редактор *Animator*, який підтримує скелетну анімацію (*Skeletal Animation*), дерева змішування (*Blend Trees*) для плавних переходів між анімаціями та зворотну кінематику (*Inverse Kinematics*). Для реалізації штучного інтелекту рушій містить інструменти *NavMesh* (для автоматизованої навігації) та Behavior Trees (для складної поведінки NPC).

Однією з ключових переваг Unity є підтримка кросплатформеності: розробники можуть експортувати додатки для понад 25 платформ, зокрема Windows, macOS, Linux, Android, iOS, PlayStation, Xbox, Nintendo Switch, WebGL, а також VR/AR-пристрої, такі як Oculus та Hololens.

Unity також включає *Asset Store* — маркетплейс ресурсів, де можна знайти готові 3D-моделі, текстури, звукові ефекти, а також скрипти для прискорення розробки. Інструмент *Package Manager* дозволяє інтегрувати

додаткові бібліотеки, як-от *Cinemachine* для керування камерами та *Post-Processing Stack* для покращення графічної якості.

Основними перевагами Unity є інтуїтивний інтерфейс, що підходить як для початківців, так і для досвідчених розробників, велика спільнота з доступом до навчальних матеріалів та форумів, а також гнучкість у розробці як 2D, так і 3D проєктів. Unity використовується не лише для створення відеоігор, а й у сферах архітектурної візуалізації, медичних симуляцій, навчальних програм і навіть кіновиробництва [8].

Таким чином, Unity залишається одним із найпотужніших та універсальних інструментів для створення інтерактивних додатків завдяки модульності, простоті використання та широкому спектру функціональних можливостей.

2. GodotEngine

Godot Engine — це сучасний відкритий рушій для створення відеоігор та інтерактивних додатків, що пропонує широкий спектр функціональних можливостей. Завдяки своїй відкритості та безкоштовній ліцензії MIT, рушій здобув значну популярність серед інді-розробників та освітніх проектів. У цьому контексті важливо розглянути його архітектуру, інструменти програмування, графічні можливості та основні переваги.

Однією з ключових особливостей Godot є його компонентно-орієнтована архітектура, що базується на системі вузлів (*Nodes*) та сцен (*Scenes*). Вузли є базовими елементами, що виконують певні функції, такі як відображення графічних об'єктів або обробка фізики. Сцени, своєю чергою, є ієрархічними структурами вузлів, які можна повторно використовувати у проекті.

Варто зазначити, що така архітектура забезпечує високу модульність. Наприклад, створення складного об'єкта можна здійснити шляхом вкладення однієї сцени в іншу, що сприяє ефективному управлінню ресурсами та знижує складність проекту.

Godot підтримує кілька мов програмування, зокрема GDScript, C#, VisualScript та C++. GDScript є основною мовою рушія, яка відзначається простотою у використанні та синтаксично подібна до Python. Ця мова була спеціально розроблена для потреб розробки ігор і дозволяє швидко створювати логіку гри.

У контексті великих проектів варто відзначити підтримку C#, що забезпечує розробникам додаткову гнучкість. Наприклад, використання C# дозволяє інтегрувати складні бібліотеки .NET, які можуть бути корисними для реалізації специфічних функціональних можливостей [9].

Графічний рушій Godot підтримує як 2D, так і 3D рендеринг. У випадку 2D-графіки використовується оптимізований рендерер, що включає підтримку шейдерів, анімацій та освітлення. Це робить Godot зручним для створення ігор, орієнтованих на простоту дизайну та високу продуктивність.

Крім того, у контексті 3D-графіки Godot пропонує PBR-рендеринг (*Physically Based Rendering*), який забезпечує реалістичне відображення матеріалів. Версія Godot 4.0, зокрема, впровадила підтримку Vulkan API, що значно покращило якість рендерингу та продуктивність.

Необхідно підкреслити, що обидві системи графіки інтегровані в редакторі, що дозволяє розробникам використовувати єдиний інструмент для роботи з різними типами проектів.

Фізичний рушій Godot забезпечує можливості для симуляції фізичних процесів як у 2D, так і в 3D середовищах. Наприклад, у 2D розробники можуть використовувати компоненти *RigidBody2D*, *KinematicBody2D* або *StaticBody2D* для створення різних типів об'єктів [14, 15].

Крім того, рушій має потужну систему анімації, що базується на використанні вузла AnimationPlayer. Вона дозволяє створювати анімації за допомогою ключових кадрів або застосовувати деревовидні структури для плавного переходу між станами. Ця функціональність особливо корисна для складних проектів, які потребують синхронізації багатьох елементів.

Окрему увагу слід звернути на кросплатформеність Godot. Рушій підтримує створення додатків для широкого спектра платформ, таких як Windows, macOS, Linux, Android, iOS, WebGL та HTML5. Це відкриває можливості для розробників створювати проекти, які можуть бути легко адаптовані до потреб різних користувачів.

До основних переваг рушія відносять

- відкритий вихідний код,
- проста у використанні архітектура
- підтримка кількох мов програмування

Ці фактори роблять Godot привабливим вибором для інді-розробників та освітніх проектів.

Незважаючи на численні переваги, Godot має певні обмеження. Зокрема, його можливості у сфері високобюджетної 3D-графіки поступаються рушіям типу Unity або Unreal Engine. Це може стати викликом для розробників AAA-проектів, які потребують більшої деталізації та високого рівня продуктивності.

Godot Engine є універсальним інструментом для створення інтерактивних додатків, що поєднує гнучкість, відкритість та простоту використання. Завдяки своїм функціональним можливостям та активній спільноті розробників, рушій стає оптимальним вибором для проектів середнього та малого масштабу. Водночас для більш складних проектів, які вимагають високої продуктивності та деталізації, розробникам може знадобитися розгляд альтернативних рішень [10].

Таблиця 2.1 – Порівняння Unity та Godot

Характеристика	Unity	Godot
Ліцензія	Пропрієтарна (умови зміни з Unity 2024)	Відкрита ліцензія MIT
Мова програмування	C#	GDScript, C#, C++
Графічний рушій	URP, HDRP, Built-in Render Pipeline	Vulkan, OpenGL
Підтримка 2D та 3D графіки	Так, з широкими можливостями	Так, оптимізована 2D графіка та PBR для 3D
Продуктивність	Висока, особливо у 3D	Вища у 2D, середня у 3D
Інструменти для анімації	Animator, Blend Trees, IK	AnimationPlayer, вбудовані в редактор
Фізичний рушій	NVIDIA PhysX	Вбудований 2D/3D рушій
Кросплатформеність	25+ платформ, зокрема мобільні та VR	10+ платформ, включаючи WebGL, Android, iOS
Придатність для ААА-проєктів	Висока	Обмежена для ААА-проєктів

Було обрано ігровий рушій Unity через його високу продуктивність, розширені можливості для роботи як з 2D, так і з 3D графікою, підтримку кросплатформеності та зручність для розробників завдяки компонентно-орієнтованій архітектурі. Unity також має велику спільноту, що забезпечує доступ до численних навчальних матеріалів та готових ресурсів на Asset Store, що пришвидшує розробку проєктів. Його багатофункціональність і підтримка C# робить його оптимальним вибором для створення складних інтерактивних симуляцій зграйної поведінки.

2.2 Вимоги до програмного забезпечення

Вимоги до алгоритму симуляції зграйної поведінки визначають набір характеристик, які необхідно реалізувати для забезпечення реалістичності, продуктивності та гнучкості візуалізації зграй птахів, риб або інших об'єктів у двовимірному середовищі Unity. Чітке формулювання цих вимог є важливим етапом розробки програмного забезпечення, оскільки воно гарантує відповідність технічним стандартам і потребам кінцевих користувачів.

Функціональні вимоги:

1. Алгоритм має підтримувати моделювання зграйної поведінки в реальному часі для великої кількості агентів.
2. Реалізація трьох ключових правил зграйної поведінки:
   - Вирівнювання (Alignment): агенти коригують напрям руху відповідно до середнього вектора швидкості сусідів.
   - Утримання разом (Cohesion): агенти тяжіють до центру мас сусідніх агентів.
   - Уникнення зіткнень (Separation): агенти уникають надто близького наближення один до одного.
3. Візуалізація агентів через об'єкт Prefab із підтримкою кількох варіантів спрайтів для візуального різноманіття.

4. Забезпечення можливості масштабування кількості агентів без значного зниження продуктивності.

5. Можливість регулювання параметрів поведінки (швидкість, радіус огляду, кількість агентів) через конфігураційний файл Settings.

6. Забезпечення управління агентами через систему Entity Component System (ECS).

7. Обмеження області руху агентів із можливістю перетинання меж екрану та з'явлення з протилежного боку сцени.

Нефункціональні вимоги:

1. Мінімізація навантаження на CPU та GPU при обробці симуляції.

2. Стабільність:
   - Алгоритм повинен забезпечувати стабільну частоту кадрів навіть при великій кількості агентів на сцені.
   - Відсутність аварійних завершень під час обробки великої кількості об'єктів.

3. Гнучкість:
   - Легка інтеграція алгоритму у нові проєкти Unity.
   - Модульність

4. Алгоритм має бути легко піддаватись модульному тестуванню.

5. Візуальна якість:
   - Відсутність графічних артефактів при русі агентів.
   - Збереження коректної поведінки при збільшенні кількості агентів або зміні параметрів симуляції.

2.3 Засоби реалізації алгоритму

Розробка алгоритмів симуляції зграйної поведінки є надзвичайно актуальною в сучасних умовах через зростаючий попит на реалістичне моделювання колективної поведінки у різних сферах. Такі алгоритми активно використовуються в комп'ютерній графіці для створення візуальних ефектів у фільмах, іграх та віртуальних середовищах, де важливою є правдоподібність взаємодії великих груп об'єктів. У наукових дослідженнях алгоритми зграйної поведінки застосовуються для моделювання поведінки тварин, зокрема косяків риб, зграй птахів та руху комах. Вони також знаходять застосування у моделюванні натовпів для симуляції евакуації людей у критичних ситуаціях. В інженерних та робототехнічних системах ці алгоритми використовуються для керування групами автономних дронів та інших колективних систем, де важливою є координація руху та уникнення зіткнень. Така актуальність пояснюється високим попитом на алгоритми, що здатні працювати у реальному часі, масштабуватися для великих груп об'єктів та демонструвати складну поведінку при мінімальному наборі правил.

Алгоритм симуляції зграйної поведінки розроблений на основі класичного підходу Boids, але розширений додатковими механізмами для забезпечення більшої реалістичності та гнучкості в управлінні. Він включає базові принципи вирівнювання, утримання та уникнення зіткнень, але також враховує фактори унікальних реакцій агентів на зовнішні загрози, такі як наближення хижака. Алгоритм включає адаптивні налаштування, зокрема змінні параметри швидкості, відстані огляду та інші. Важливою особливістю є обмеження меж симуляції, а також механізми повернення агентів при виході за ці межі. Реалізація базується на оптимізованих обчисленнях, що дозволяє досягти високої продуктивності навіть при значній кількості агентів у симуляції, зберігаючи при цьому природність і передбачуваність поведінки зграї.

Загальні кроки алгоритму симуляції зграї мають наступний вигляд:

1. Ініціалізація даних – створення початкових об'єктів, завантаження параметрів симуляції, таких як кількість агентів, швидкість, радіус взаємодії (рис. 2.1).

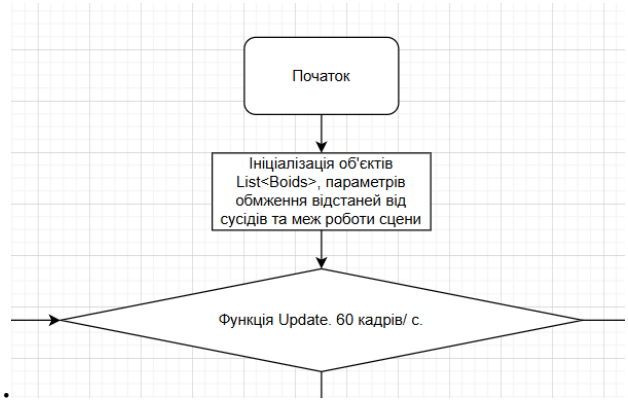


Рисунок 2.1 – Діаграма алгоритму (ч.1)

2. Оцінка об'єктів – визначення найближчих сусідів для кожного агента та аналіз їхніх параметрів (позиції, швидкості, напрямку руху) (рис. 2.2).

3. Утримання та вирівнювання – обчислення середнього напряму руху сусідів (вирівнювання), визначення центру для руху до групи (утримання) та обрахунок вектору для уникнення зіткнень із сусідами (розділення).

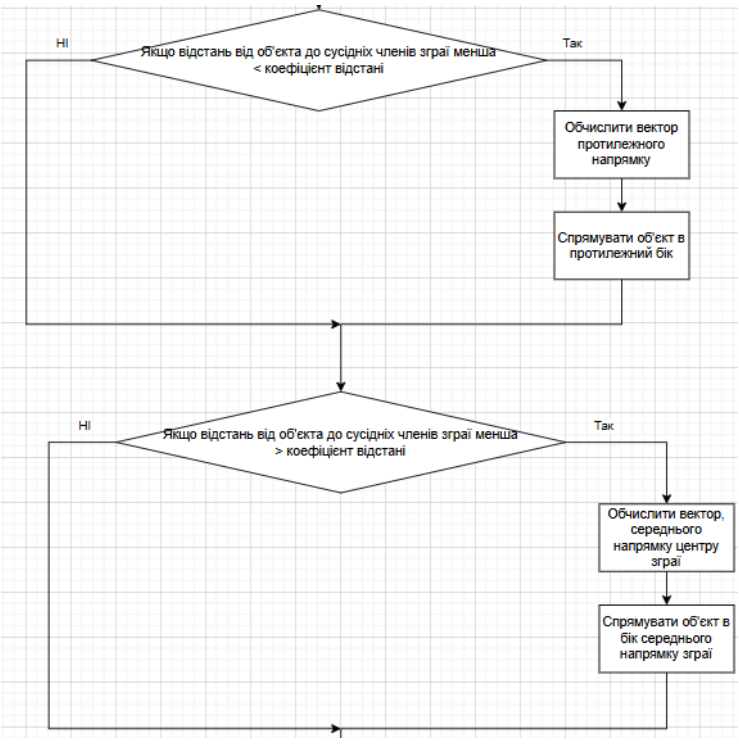


Рисунок 2.2 – Діаграма алгоритму (ч. 2,3)

4. Обмеження межі симуляції – перевірка меж симуляції, обмеження руху агентів та поступове поверненя на протилежний край сцени.

5. Оновлення параметрів – обчислення вектору швидкості та напряму руху агентів на основі отриманих результатів обчислень (рис. 2.3).

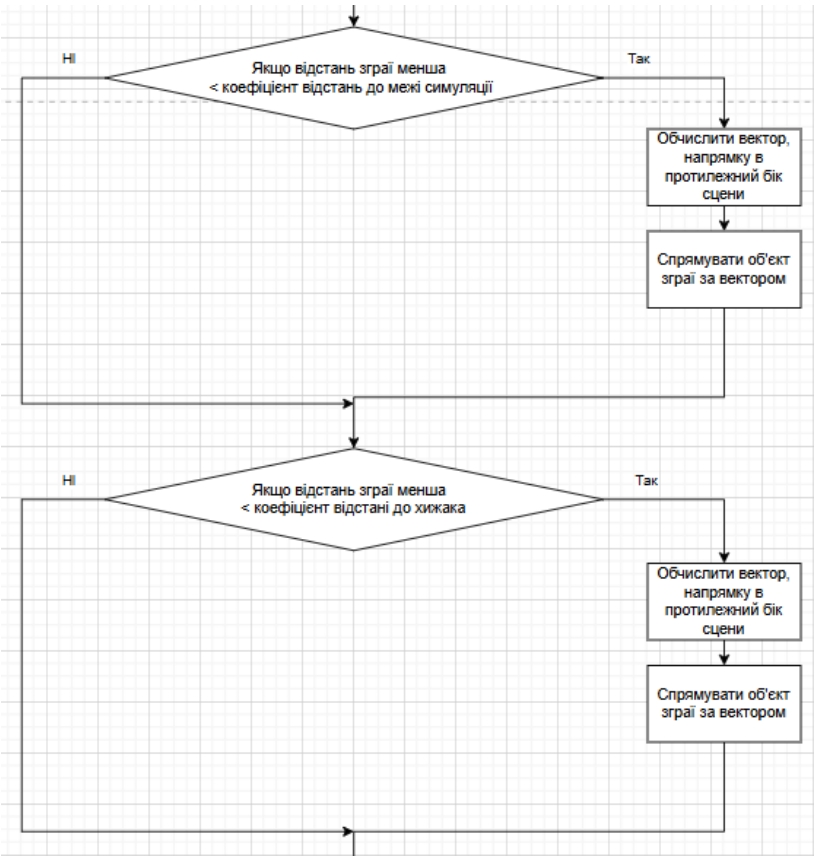


Рисунок 2.3 – Діаграма алгоритму (ч. 4,5)

2.4 Висновок до другого розділу

У другому розділі було проведено комплексний аналіз програмних засобів для реалізації алгоритму моделювання поведінки зграї. Було розглянуто можливості та функціональність таких середовищ, як Unity та Godot Engine, із детальним порівнянням їхніх технічних характеристик, зокрема продуктивності, підтримки анімації, фізики, компонентно-орієнтованої архітектури та кросплатформеності. Особливу увагу було

приділено здатності цих рушіїв ефективно обробляти великі обсяги даних під час симуляції поведінки численних агентів у реальному часі.

За результатами аналізу, для реалізації поставлених завдань було обрано Unity завдяки його розширеним функціональним можливостям, включаючи підтримку Entity Component System (ECS) для оптимізованої обробки великої кількості об'єктів, а також компонентно-орієнтовану архітектуру, яка полегшує розробку та налагодження симуляційних систем. Важливим фактором вибору також стала можливість використання мови програмування C#, яка дозволяє створювати гнучкі скрипти для управління поведінкою агентів та інтеграції алгоритму симуляції у різні проєкти. Unity забезпечує підтримку як 2D, так і 3D графіки, а також надає інструменти для створення складних візуалізацій та фізичних моделей.

Також було визначено вимоги до алгоритму симуляції зграйної поведінки, включаючи як функціональні, так і нефункціональні аспекти. Основні функціональні вимоги включають: підтримку моделювання в реальному часі, реалізацію ключових принципів зграйної поведінки (вирівнювання, утримання, уникнення зіткнень), можливість налаштування параметрів поведінки агентів через конфігураційний файл, а також візуалізацію агентів через об'єкти Prefab із різними варіантами спрайтів. Нефункціональні вимоги включають: високу продуктивність при обробці великих груп агентів, стабільність роботи алгоритму, гнучкість налаштувань та мінімізацію навантаження на апаратні ресурси (CPU, GPU).

Загалом, проведений аналіз програмних засобів, формулювання вимог до алгоритму та порівняння Unity та Godot дозволили обґрунтовано обрати Unity як основний інструмент для реалізації алгоритму симуляції зграйної поведінки. Його розширений функціонал, гнучкість у розробці, оптимізація для роботи з великими даними та активна спільнота розробників роблять його оптимальним рішенням для поставлених завдань у даному дослідженні.

РОЗДІЛ 3. РОЗРОБКА АЛГОРИТМУ СИСТЕМИ МОДЕЛЮВАННЯ ПОВЕДІНКИ ЗГРАЙ У СЕРЕДОВИЩІ UNITY

3.1 Актуальність розробки

Розробка алгоритму зграйної поведінки для Unity в сучасних умовах стрімкого розвитку комп’ютерної графіки, симуляційних моделей та інтерактивних додатків. Візуалізація поведінки зграя’ї використовується у різноманітних сферах, включаючи відеоігри, анімаційні фільми, симуляції для наукових досліджень та моделювання натовпів. Створення реалістичних моделей поведінки груп об'єктів, таких як зграї птахів, косяки риб або натовпи людей, дозволяє досягти більшої правдоподібності візуального контенту, покращуючи загальне сприйняття продукту кінцевим користувачем.

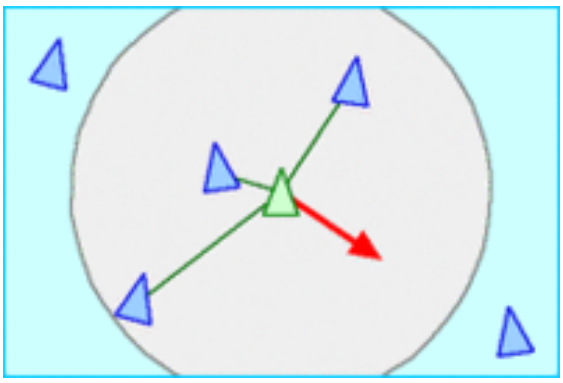


Рисунок 3.1 – Алгоритм симуляції зграї

Unity є однією з найпопулярніших платформ для розробки інтерактивного контенту, яка надає потужні інструменти для створення складних симуляцій, зокрема для моделювання алгоритму зграйної поведінки. Алгоритм стаї, або "boids" (bird-oid objects), заснований на правилах вирівнювання, утримання разом та уникнення зіткнень, що дозволяє створювати реалістичні анімації колективної поведінки об'єктів у віртуальних середовищах. Його застосування охоплює моделювання зграй птахів, косяків риб, натовпів людей, а також рух автономних роботів чи дронів.

Особливу актуальність алгоритм стаї набуває у поєднанні з архітектурою Entity Component System (ECS), яку пропонує Unity. ECS є підходом до програмування, що базується на розділенні логіки обробки даних (систем) і самих даних (компонентів). Така архітектура дозволяє зберігати великий обсяг даних у структурованому вигляді та забезпечує ефективне їх опрацювання за допомогою багатопотокової обробки. Це критично важливо для симуляцій, де велика кількість об’єктів повинна оброблятися одночасно, зберігаючи коректну взаємодію між ними.

Комбінація алгоритму стаї та ECS дозволяє досягти високої продуктивності під час обробки великої кількості об'єктів, знижуючи навантаження на процесор завдяки оптимізованим підходам до управління пам'яттю. Це робить алгоритм особливо важливим у сценаріях, що потребують великої кількості одночасних об'єктів на сцені, наприклад, при симуляції руху натовпу в іграх або візуалізації екологічних систем.

Розробка алгоритму зграйної поведінки має значення для освітніх та наукових проєктів. Цей алгоритм використовується для дослідження колективної поведінки у природі, наприклад, при вивченні біологічних моделей руху тварин або поведінки частинок у фізичних середовищах. У освітньому процесі він допомагає візуалізувати складні концепції координації, самоорганізації та взаємодії в динамічних системах.

Крім того, алгоритм знаходить застосування у прикладних проєктах, таких як симуляція евакуації з приміщень або управління автономними дронами. Точне відтворення групової поведінки у віртуальних середовищах є критично важливим для тестування систем безпеки, архітектурного планування та розробки штучного інтелекту. Це робить даний алгоритм важливим не лише у контексті комп’ютерної графіки, але й у ширших інженерних і наукових дослідженнях.

Таким чином, розробка алгоритму зграйної поведінки для Unity є важливою та актуальною задачею, яка сприяє як покращенню якості

інтерактивних додатків, так і розширенню можливостей для дослідження складних систем у різноманітних сферах застосування.

3.2 Структура алгоритму

Алгоритм повністю будується на структурі патерну ECS, тому має наступний вигляд:

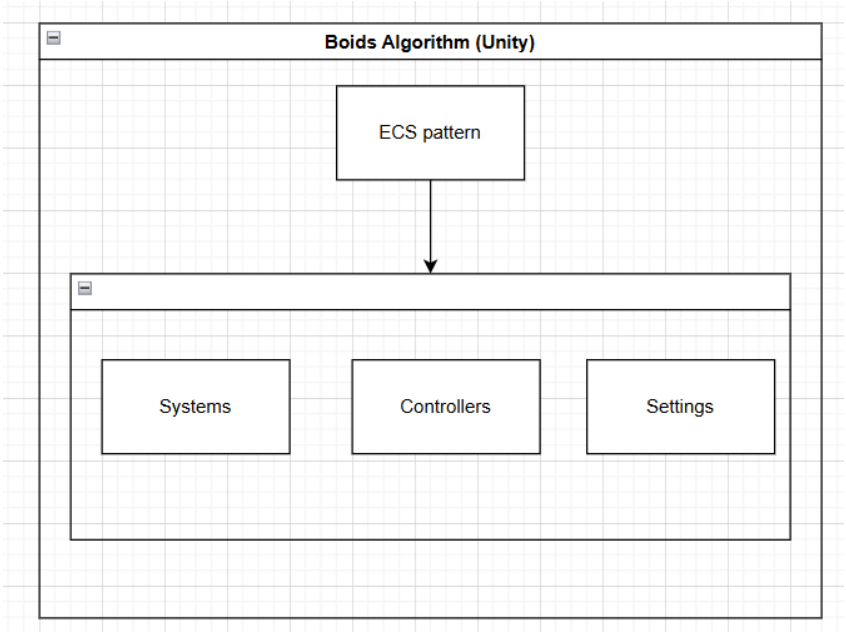


Рисунок 3.2 – Структура алгоритму

З урахуванням структури проєкту (Systems, Controller, Settings), архітектура, поділяється на наступні копоненти:

- Systems — відповідає за логіку обробки, наприклад, оновлення положення об'єктів або обчислення правил зграйної поведінки.
- Controller — керує загальною координацією симуляції, містить головні скрипти, що ініціює роботу систем.
- Settings — зберігає конфігураційні параметри для симуляції, такі як радіус взаємодії, швидкість тощо.

Такий підхід до структурування проєкту покращує підтримку, тестування та розширюваність коду.

3.3 Розробка алгоритму

Початок розробки даного проєкту передбачає створення алгоритму Boids для Unity з використанням структурного поділу на Systems, Controller, Settings. Основними вимогами є створення агентів (боїдів), які рухаються у 2D-просторі та дотримуються трьох правил алгоритму:

- вирівнювання (агенти рухаються у напрямку середньої швидкості сусідів),
- утримання разом (агенти тяжіють до центру мас групи)
- уникнення зіткнень (агенти уникають надто близького наближення один до одного).

Проєкт має бути побудований на компонентній архітектурі, де кожна частина алгоритму окремо відповідають за обробку дій щодо збереження даних, управління загальною системою та обробки окремих її частин.

Розробник повинен реалізувати: можливість масштабування кількості агентів, висока продуктивність при обробці великої кількості об'єктів, простота модифікації параметрів через налаштування та візуалізація поведінки в реальному часі у Unity 2D-середовищі.

Для виконання вищезазначених критеріїв, розроблений алгоритм має наступну структуру класів:

Рисунок 3.3 – Діаграма класів алгоритму

Для зручності було відокремлено кожен клас згідно з компонентом, до якого він належить, що дозволило чітко розділити функціональність системи та спростити подальшу підтримку коду.

Для початку потрібно розглянути найважливіші компоненти, що відповідають за функціональність алгоритму як системи в цілому, тобто класи Controllers (рис. 3.4):

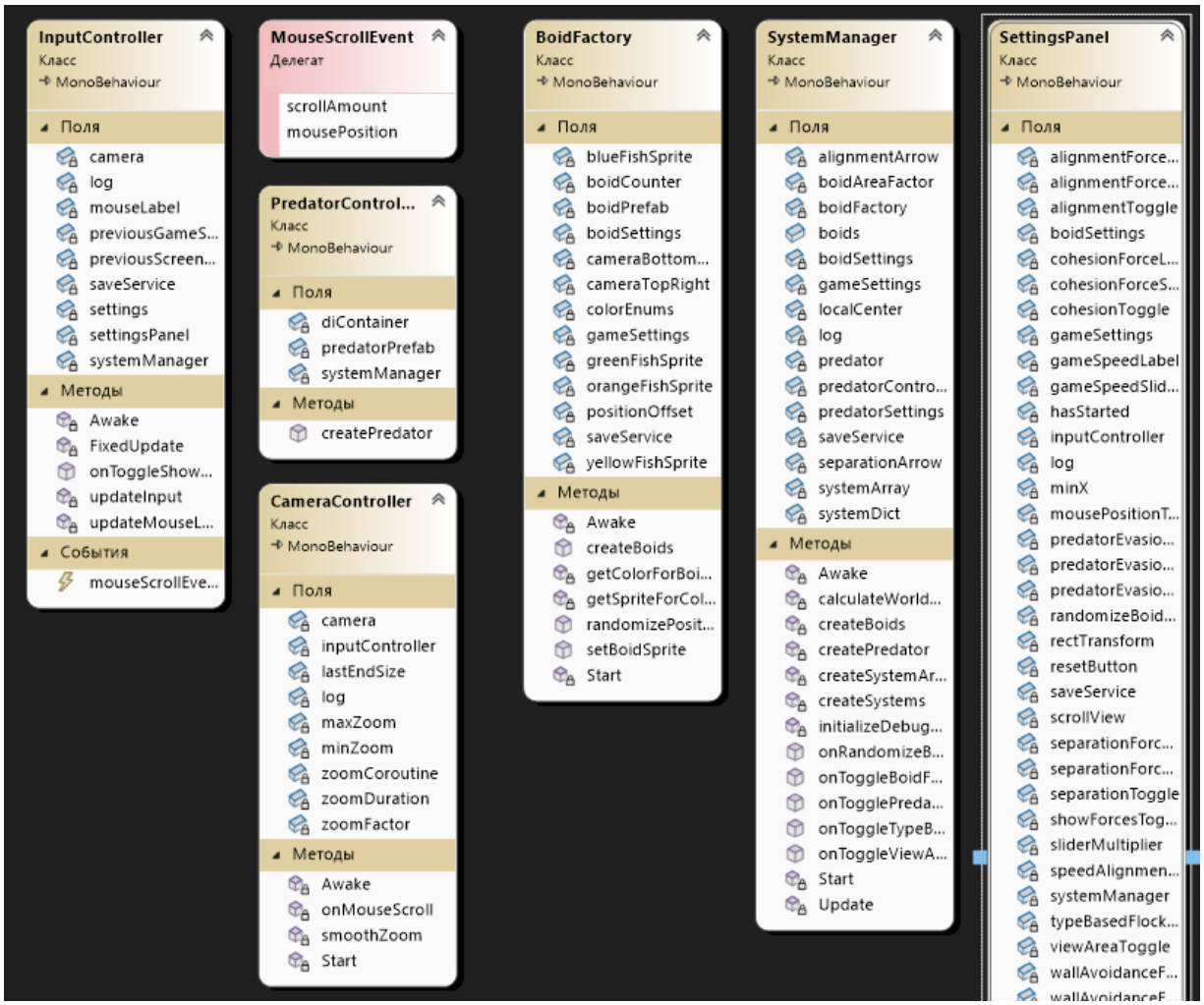


Рисунок 3.4 – Компоненти Controllers

Перший клас що потрібно розглянути це клас BoidFactory (рис. 3.5). Клас BoidFactory у Unity відповідає за створення та ініціалізацію агентів (боїдів) у симуляції зграйної поведінки. Основна функціональність цього класу полягає у підготовці параметрів для генерації агентів та їх керування поведінкою під час запуску сцени.

Поле positionOffset відповідає за зміщення позиції під час створення боїдів, використовується для уникнення накладання агентів один на одного під час генерації. Поле boidPrefab — це шаблонний об'єкт для створення кожного нового члена зграї у симуляції. Спрайти blueFishSprite, greenFishSprite, yellowFishSprite та orangeFishSprite використовуються для візуалізації із різними варіантами кольорів, за бажанням.

```
Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 2
public class BoidFactory : MonoBehaviour {
    [SerializeField] float positionOffset;
    [SerializeField] GameObject boidPrefab;
    [SerializeField] Sprite blueFishSprite;
    [SerializeField] Sprite greenFishSprite;
    [SerializeField] Sprite yellowFishSprite;
    [SerializeField] Sprite orangeFishSprite;

    [Inject] SaveService saveService;

    GameSettings gameSettings;
    BoidSettings boidSettings;
    Vector3 cameraBottomLeft;
    Vector3 cameraTopRight;
    int boidCounter;
    FishType[] colorEnums;

    Сообщение Unity | Ссылок: 0
    void Awake() {
        gameSettings = saveService.getSave().settings;
        boidSettings = gameSettings.boidSettings;
        colorEnums = (FishType[]) Enum.GetValues(typeof(FishType));
    }
```

Рисунок 3.5 – Клас BoidFactory

Залежність saveService (рис. 3.5) оголошена з атрибутом [Inject], що свідчить про використання механізму Dependency Injection (DI). Це підхід до управління залежностями, який спрощує тестування та підтримку коду, дозволяючи передавати залежності зовні під час створення об'єкта. У даному випадку saveService відповідає за завантаження збережених налаштувань гри.

Метод Awake() є стандартним методом Unity, що викликається при активації об'єкта. У цьому методі відбувається ініціалізація основних параметрів, зокрема завантаження налаштувань гри через saveService та підготовка масиву стаї. Це дозволяє підготувати всі дані ще до початку симуляції.

В другій частині класу функції відповідають за створення, ініціалізацію та налаштування окремих агентів у симуляції. Основна мета цього блоку коду — створення групи стаї, їх випадкове розміщення у сцені, задання параметрів швидкості, напряму руху та візуальних характеристик.

Метод createBoids (рис. 3.6) відповідає за масове створення агентів зграї відповідно до налаштувань boidSettings. Спочатку створюється список boids, в який додаються новостворені агенти. Далі, використовуючи параметри boidSettings.size (розмір члена стаї) та boidSettings.viewDistance (відстань огляду), обчислюється діаметр радіусу огляду агента (viewAreaDiameter) та формується viewAreaSize. Ці значення використовується для налаштування області, в якій агент може сприймати інших агентів.

Проходячи циклом for, створюється задана кількість членів зграї (boidCount), кожен із яких створюється за допомогою методу Instantiate з використанням шаблону boidPrefab.

```
Ссылок: 1
public List<Boid> createBoids() {
    var boids = new List<Boid>();
    var boidSize = boidSettings.size;
    var viewAreaDiameter = 2 * boidSettings.viewDistance / boidSize;
    var viewAreaSize = new Vector3(viewAreaDiameter, viewAreaDiameter);
    var boidCount = boidSettings.count;
    for (var i = 0; i < boidCount; i++) {
        var boid = Instantiate(boidPrefab).GetComponent<Boid>();
        boid.name = $"boid_{++boidCounter}";
        // sprite
        setBoidSprite(boid, boidSettings.typeBasedFlockingEnabled, i, boidCount);
        // size
        boid.transform.localScale = new Vector3(boidSize, boidSize);
        // position and velocity
        randomizePositionAndVelocity(boid);
        // view area size
        boid.viewArea.transform.localScale = viewAreaSize;

        boids.Add(boid);
    }
    if (boids.Count == 0) {
        throw new Exception("no boid was created");
    }
    return boids;
}
```

Рисунок 3.6 – Функція createBoids

Наступний метод randomizePositionAndVelocity(Boid boid) відповідає за створення випадкового напрямку руху та швидкості агента. Позиція: визначається випадковим чином у межах встановлених координат

(cameraBottomLeft та cameraTopRight), з урахуванням positionOffset, що дозволяє уникнути генерації агентів надто близько до країв камери. Цей метод забезпечує різноманітність поведінки агентів у симуляції, що додає реалістичності зграї.

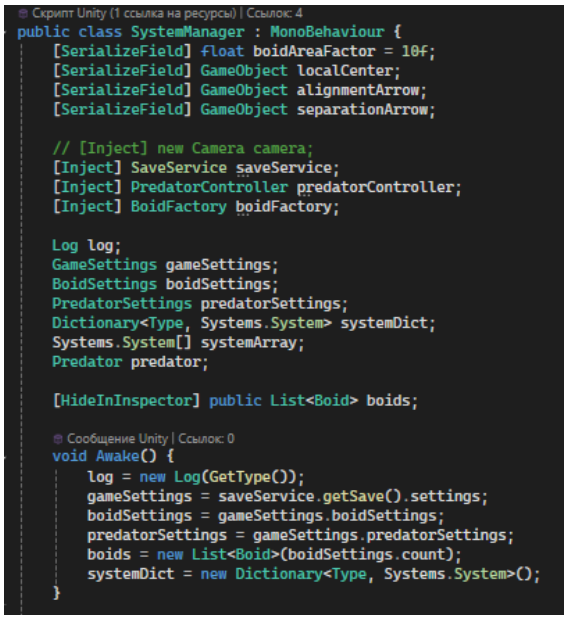
Метод setBoidSprite (рис. 3.7) відповідає за встановлення зовнішнього вигляду членів зграї:

```
Ссылок: 2
public void setBoidSprite(Boid boid, bool typeBasedFlockingEnabled, int index, int boidCount) {
    boid.fishType = typeBasedFlockingEnabled ? getColorForBoidIndex(index, boidCount) : FishType.Blue;
    boid.GetComponent<SpriteRenderer>().sprite = getSpriteForColor(boid.fishType);
}
```

Рисунок 3.7 – Метод setBoidSprite

Загалом клас BoidFactory відповідає за процеси генерацію об'єктів, випадкове розміщення, налаштування розміру, швидкості та зовнішнього вигляду агентів.

Наступний клас SystemManager виконує загальну роль в управлінні симуляції зграйної поведінки. Відповідає за координацію створення агентів, ініціалізацію систем та налаштування параметрів сцени. Він забезпечує управління агентами, так і контролює взаємодію між ними, використовуючи підхід централізованого управління. Такий підхід спрощує контроль за станом сцени, особливо у складних симуляціях із великою кількістю об'єктів.



```
Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 4
public class SystemManager : MonoBehaviour {
    [SerializeField] float boidAreaFactor = 10f;
    [SerializeField] GameObject localCenter;
    [SerializeField] GameObject alignmentArrow;
    [SerializeField] GameObject separationArrow;

    // [Inject] new Camera camera;
    [Inject] SaveService saveService;
    [Inject] PredatorController predatorController;
    [Inject] BoidFactory boidFactory;

    Log log;
    GameSettings gameSettings;
    BoidSettings boidSettings;
    PredatorSettings predatorSettings;
    Dictionary<Type, Systems.System> systemDict;
    Systems.System[] systemArray;
    Predator predator;

    [HideInInspector] public List<Boid> boids;

    Сообщение Unity | Ссылок: 0
    void Awake() {
        log = new Log(GetType());
        gameSettings = saveService.getSave().settings;
        boidSettings = gameSettings.boidSettings;
        predatorSettings = gameSettings.predatorSettings;
        boids = new List<Boid>(boidSettings.count);
        systemDict = new Dictionary<Type, Systems.System>();
    }
```

Рисунок 3.8 – Клас SystemManager

Поле boidAreaFactor визначає коефіцієнт, який впливає на розмір області, в якій можуть рухатися агенти. Даний параметр використовується для обмеження області симуляції або розрахунків відстаней між агентами. Поля localCenter, alignmentArrow та separationArrow представляють посилання на об'єкти сцени, що використовуються для візуалізації ключових векторів зграйної поведінки: центру зграї, напрямку вирівнювання та сили відштовхування.

У класі оголошено кілька важливих змінних для управління симуляцією. log використовується для створення журналу подій, використовується під час тестування та пошуку помилок. GameSettings — це основний об'єкт, що містить загальні налаштування гри, такі як параметри середовища та фізики симуляції, boidSettings і predatorSettings є підрозділами gameSettings і зберігають параметри для членів зграї. Наприклад, у boidSettings можуть зберігатися такі параметри, як швидкість, радіус огляду та кількість агентів.

Для управління системами використовується словник `systemDict`, де кожен тип системи асоціюється з конкретним екземпляром класу Systems.System. Це дозволяє зберігати різні налаштування, такі як система керування рухом, зіткнень.

Метод Awake(рис. 3.9) є ключовим для ініціалізації цього класу та викликається одразу після завантаження об'єкта. Він починається зі створення об'єкта log для реєстрації подій у процесі симуляції. Далі завантажуються налаштування гри (gameSettings) через`saveService.getSave().settings, що включає параметри середовища, та членів зграї.

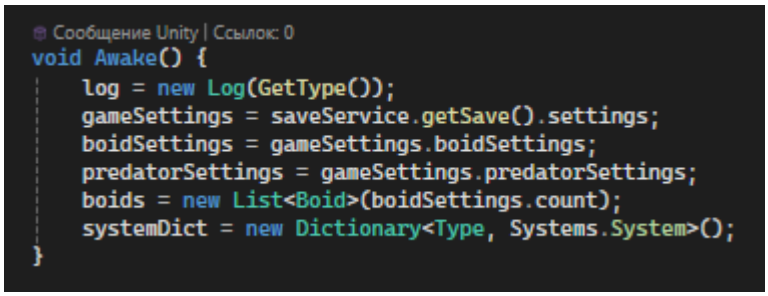


Рисунок 3.9 – Функція Awake

Ініціалізація списку boids здійснюється з використанням параметра boidSettings.count, який визначає кількість агентів, що беруть участь у симуляції. Це робиться для збереження пам’яті та уникнення зайвих об'єктів під час запуску. Також створюється systemDict, який буде використовуватися для зберігання та швидкого доступу до всіх систем, що беруть участь у симуляції.

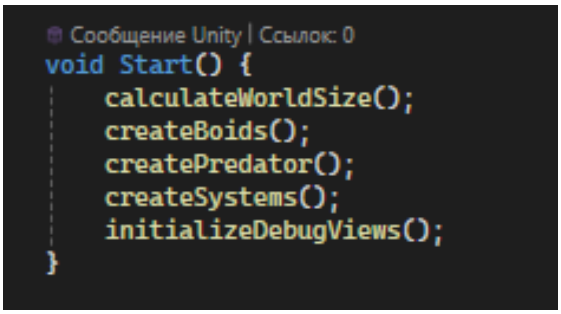


Рисунок 3.10 – Метод Start

Метод Start (рис. 3.10) відповідає за безпосереднє налаштування сцени перед початком симуляції. Він викликає кілька допоміжних методів: calculateWorldSize() (обчислення розміру світу для обмеження руху агентів), createBoids() (створення та ініціалізація зграї на сцені), createSystems()

(ініціалізація систем обробки взаємодії між агентами). Останнім етапом є виклик initializeDebugViews(), який, імовірно, активує візуальні елементи для налагодження сцени.

У другій частині класу SystemManager описані методи, що відповідають за обчислення розміру світу, створення агентів, та ініціалізацію систем, які керують поведінкою агентів у симуляції. Ці методи забезпечують належну підготовку сцени та її компонентів перед запуском симуляції, забезпечуючи контроль за простором та взаємодією між об'єктами.

Метод calculateWorldSize (рис. 3.11) відповідає за обчислення фізичних розмірів ігрового простору, в якому відбуватиметься симуляція. Розмір світу обчислюється на основі кількості агентів (boidSettings.count), їхнього розміру (boidSettings.size) та коефіцієнта boidAreaFactor, який визначає площу, що буде доступною для руху агентів. Далі обчислюється співвідношення сторін екрану (ratio) за допомогою поточних параметрів Screen.width та Screen.height. Висота (height) та ширина (width) сцени обчислюються з урахуванням цього співвідношення та обраної площі. У результаті gameSettings.worldRect зберігає прямокутник, що визначає межі світу у вигляді Rect. Це значення використовується для обмеження руху агентів.

```
Ссылок: 1
void calculateWorldSize() {
    var area = boidSettings.size * boidAreaFactor * boidSettings.count;
    if (area == 0) {
        throw new Exception("world area is 0");
    }
    var ratio = (float) Screen.width / Screen.height;
    var height = Mathf.Sqrt(area / ratio);
    var width = ratio * height;
    gameSettings.worldRect = new Rect(-width / 2, -height / 2, width, height);
    log.log($"world rect: {gameSettings.worldRect}, area: {area}, ratio: {ratio}");
}
```

Рисунок 3.11 – Метод calculateWorldSize

Метод createBoids викликає метод boidFactory.createBoids() для створення групи агентів у симуляції. Усі створені члени зграї зберігаються у змінній boids — списку об’єктів.

Метод createSystems (рис. 3.12) відповідає за ініціалізацію та реєстрацію всіх систем, що будуть керувати поведінкою агентів у симуляції. Кожна система додається до словника systemDict, де ключем є тип системи (Type), а значенням — відповідний об'єкт системи. Це дозволяє легко отримати доступ до конкретної системи за її типом. В кінці методу викликається createSystemArray (рис. 3.12), який перетворює словник систем у масив для зручнішого оновлення.

```
Ссылок: 1
void createSystems() {
    systemDict.Add(typeof(MovementSystem), new MovementSystem(boids));
    systemDict.Add(typeof(BorderSystem),
        new BorderSystem(boids, predator, gameSettings));
    systemDict.Add(typeof(FlockingSystem),
        new FlockingSystem(boids, gameSettings, boidSettings, localCenter, alignmentArrow, separationArrow));
    systemDict.Add(typeof(EvasionSystem), new EvasionSystem(boids, predator, gameSettings));
    systemDict.Add(typeof(DragSystem), new DragSystem(boids, predator, gameSettings));
    systemDict.Add(typeof(SpeedSystem), new SpeedSystem(boids, boidSettings));
    // systemDict.Add(typeof(BoidControlSystem), new BoidControlSystem(boids, gameSettings, camera));

    createSystemArray();
}

Ссылок: 1
void createSystemArray() {
    systemArray = new Systems.System[systemDict.Count];
    var i = 0;
    foreach (var pair in systemDict) systemArray[i++] = pair.Value;
}
```

Рисунок 3.12 – Методи createSystems та createSystemArray

Клас SystemManager зосереджений на підготовці середовища симуляції, включаючи розрахунок розмірів світу, створення агентів, а також ініціалізацію систем, що керують поведінкою об'єктів. Метод calculateWorldSize() гарантує, що симуляція відбувається в контрольованому просторі, тоді як createBoids() забезпечують створення та налаштування агентів. Найважливішим елементом є createSystems(), оскільки він визначає, як різні системи будуть взаємодіяти з агентами у симуляції, дотримуючись принципів модульності та компонентного підходу. Це забезпечує масштабованість, легкість у тестуванні та розширюваність симуляції.

Наступним кроком потрібно розглянути найважливіші класи System, які відповідають за реалізацію основних механік руху окремих компонентів системи. Кожен із цих класів виконує чітко визначену функцію, забезпечуючи

модульність та ефективність роботи симуляції, що робить їх ключовими компонентами всієї системи.

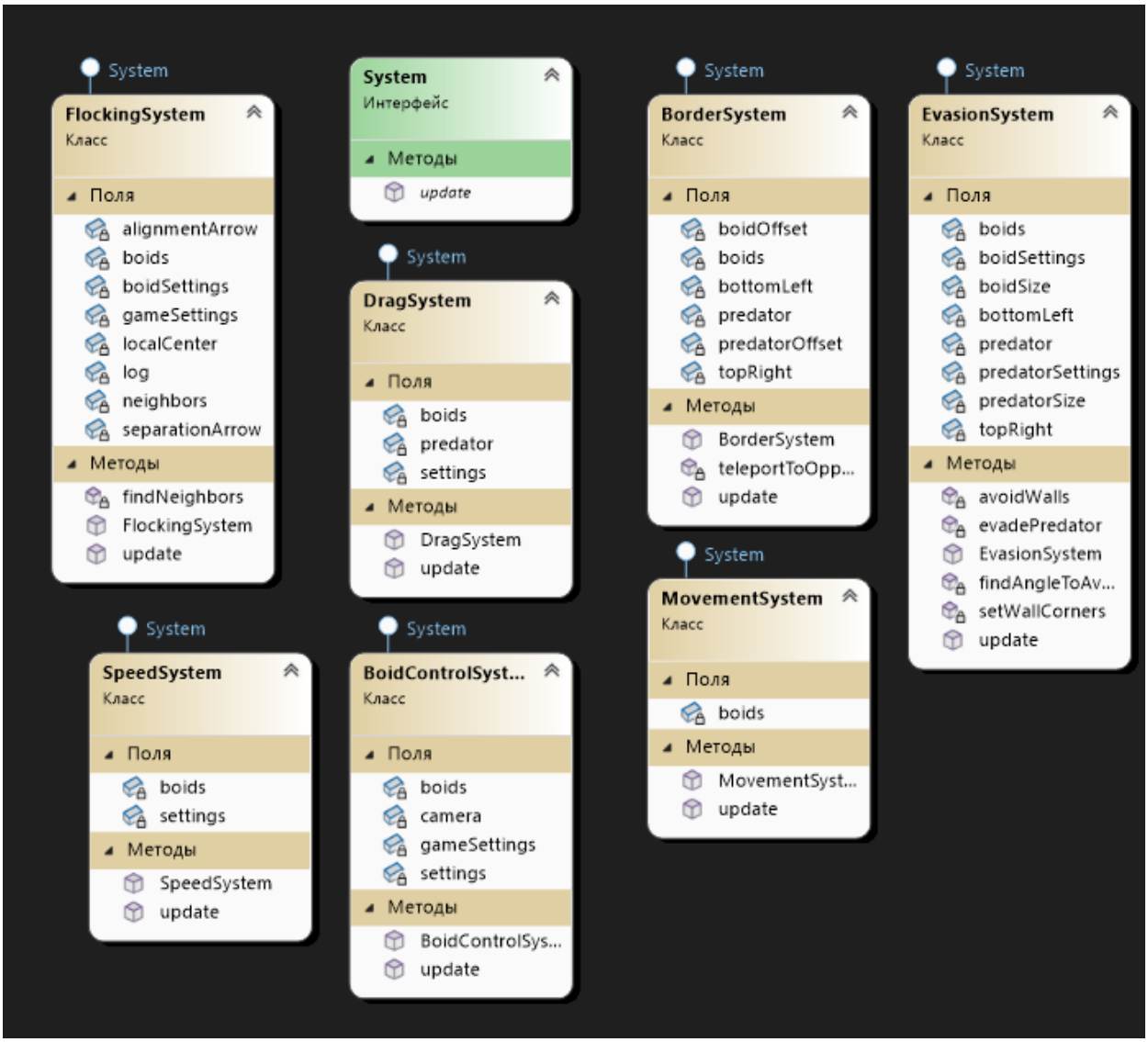


Рисунок 3.13 – Класи групи System

Спочатку потрібно розглянути клас BoidControlSystem. Даний клас реалізує функціональність прямого керування агентами за допомогою взаємодії користувача з мишею, додаючи інтерактивний елемент у симуляцію. Цей клас отримує список агентів, налаштування гри (GameSettings) та камеру (Camera) для обчислення позиції курсора в ігровому просторі.

```
public class BoidControlSystem : System {
    readonly List<Boid> boids;
    readonly GameSettings gameSettings;
    readonly BoidControlSettings settings;
    readonly Camera camera;

    Ссылок: 0
    public BoidControlSystem(List<Boid> boids, GameSettings gameSettings, Camera camera) {
        this.boids = boids;
        this.gameSettings = gameSettings;
        settings = gameSettings.boidControlSettings;
        this.camera = camera;
    }

    Ссылок: 2
    public void update(float deltaTime) {
```

Рисунок 3.14 – Клас BoidControlSystem

Основна функція системи (рис. 3.15) — оновлювати поведінку зграї у залежності від дій користувача: при натисканні лівої кнопки миші (GetMouseButton(0)) агенти притягуються до позиції курсора, а при натисканні правої кнопки миші (GetMouseButton(1)) — відштовхуються від неї. Це досягається обчисленням напряму до та від курсора (direction), перетворенням цього напряму в кут (MathUtils.vectorToAngle) і поступовим поворотом об’єкта у відповідному напрямі за допомогою Quaternion.RotateTowards.

```
Ссылок: 2
public void update(float deltaTime) {
    if (gameSettings.settingsPanelEnabled) return;

    var mousePosition = camera.ScreenToWorldPoint(Input.mousePosition);
    var attractingEnabled = Input.GetMouseButton(0);
    var repellingEnabled = Input.GetMouseButton(1);
    var attractingForce = settings.attractingForce * deltaTime;
    var repellingForce = settings.repellingForce * deltaTime;

    foreach (var boid in boids) {
        if (attractingEnabled) {
            var direction = mousePosition - boid.transform.position;
            var angle = MathUtils.vectorToAngle(direction);
            boid.transform.rotation = Quaternion.RotateTowards(
                boid.transform.rotation,
                Quaternion.Euler(0, 0, angle),
                attractingForce);
        }
        if (repellingEnabled) {
            var direction = boid.transform.position - mousePosition;
            var angle = MathUtils.vectorToAngle(direction);
            boid.transform.rotation = Quaternion.RotateTowards(
                boid.transform.rotation,
                Quaternion.Euler(0, 0, angle),
                repellingForce);
        }
    }
}
```

Рисунок 3.15 – Метод Update

Швидкість обертання визначається параметрами attractingForce і repellingForce, які масштабуються залежно від часу кадру (deltaTime) для забезпечення плавності. Важливою деталлю є перевірка парамтеру bool settingsPanelEnabled, який, якщо увімкнено, блокує керування, дозволяючи інтерактивність лише в активному режимі симуляції. Цей клас додає користувацький контроль до поведінки зграї, що робить симуляцію не лише цікавою, але й функціональною для демонстраційних або ігрових цілей.

```
namespace GameScene.Systems {
Ссылок: 3
public class MovementSystem : System {
    readonly List<Boid> boids;

    Ссылок: 1
    public MovementSystem(List<Boid> boids) {
        this.boids = boids;
    }

    Ссылок: 2
    public void update(float deltaTime) {
        foreach (var boid in boids) {
            var angleRadians = boid.transform.rotation.eulerAngles.z * Mathf.Deg2Rad;
            boid.velocity.x = boid.speed * Mathf.Cos(angleRadians);
            boid.velocity.y = boid.speed * Mathf.Sin(angleRadians);
            boid.transform.position += boid.velocity * deltaTime;
        }
    }
}
}
```

Рисунок 3.16 – Клас MovementSystem

Клас MovementSystem (рис. 3.16) є частиною системи симуляції та відповідає за оновлення позиції агентів у просторі на основі їхньої швидкості та напрямку руху. Цей клас реалізує базову механіку переміщення об'єктів у симуляції, гарантуючи, що всі члени зграї рухаються відповідно до своїх поточних налаштувань.

Основним параметром класу є список boids, переданий у конструкторі. Це дозволяє системі працювати з колекцією агентів, обробляючи кожного з них у циклі. Метод update(float deltaTime) викликається для оновлення стану об’єкту boid на кожному кадрі. В цьому методі обчислюється нова позиція залежно від швидкості (speed) і напрямку руху конкретного члена зграї, визначеного кутом повороту (rotation.eulerAngles.z).

Процес обчислення використовує кут у радіанах (angleRadians), який отримується шляхом конвертації кута з градусів у радіани (Mathf.Deg2Rad). На основі цього кута обчислюються складові вектору швидкості за осями x та y за допомогою тригонометричних функцій Mathf.Cos і Mathf.Sin. Значення швидкості по кожній осі множиться на загальну швидкість агента (boid.speed), формуючи вектор швидкості (velocity).

Далі нова позиція для кожного об’єкту обчислюється як сума поточної позиції (boid.transform.position) та вектора швидкості, помноженого на deltaTime. Це забезпечує плавність і точність переміщення, незалежно від частоти кадрів. Таким чином, об’єкти рухаються відповідно до своїх параметрів швидкості та напряму, що формує основну поведінку агентів у симуляції. Даний клас ізолює логіку переміщення в окремій системі.

Наступний клас SpeedSystem реалізує функціональність управління швидкістю зграї, забезпечуючи поступове збільшення швидкості агентів до заданого цільового значення. Він працює у вигляді окремої системи у симуляції, що відповідає за адаптацію швидкості кожного члена зграї на основі налаштувань, визначених у BoidSettings.

```
using System.Collections.Generic;
using GameScene.Settings;

namespace GameScene.Systems {
Ссылок: 3
public class SpeedSystem : System {
    readonly List<Boid> boids;
    readonly BoidSettings settings;

    Ссылок: 1
    public SpeedSystem(List<Boid> boids, BoidSettings settings) {
        this.boids = boids;
        this.settings = settings;
    }

    Ссылок: 2
    public void update(float deltaTime) {
        var targetSpeed = settings.targetSpeed;
        var acceleration = settings.baseAcceleration * deltaTime;
        foreach (var boid in boids) {
            var speed = boid.speed;
            if (speed < targetSpeed) {
                boid.speed = speed + acceleration;
            }
        }
    }
}
}
```

Рисунок 3.17 – Клас SpeedSystem

Клас отримує список об’єктів зграї (boids) і налаштування (BoidSettings) через конструктор (рис. 3.17). Основний метод класу, update(float deltaTime) використовується для поступового збільшення швидкості кожного об’єкту, якщо його поточна швидкість менша за цільову. Спочатку метод отримує цільову швидкість (targetSpeed) з налаштувань та обчислює значення прискорення, використовуючи добуток settings.baseAcceleration * deltaTime. Це гарантує, що зміна швидкості буде незалежною від частоти кадрів і забезпечить плавний перехід до цільової швидкості.

Цикл foreach проходить через кожного елемента зграї в списку boids. Для кожного елемента перевіряється його поточна швидкість (boid.speed). Якщо швидкість менша за targetSpeed, вона збільшується на обчислене значення прискорення.

```
Ссылок: 3
public class FlockingSystem : System {
    readonly Log log;
    readonly List<Boid> boids;
    readonly GameSettings gameSettings;
    readonly BoidSettings boidSettings;
    readonly List<Boid> neighbors;
    readonly GameObject localCenter;
    readonly GameObject alignmentArrow;
    readonly GameObject separationArrow;

    Ссылок: 1
    public FlockingSystem(List<Boid> boids,
        GameSettings gameSettings,
        BoidSettings boidSettings,
        GameObject localCenter,
        GameObject alignmentArrow,
        GameObject separationArrow) {
        log = new Log(GetType());
        this.boids = boids;
        this.gameSettings = gameSettings;
        this.boidSettings = boidSettings;
        neighbors = new List<Boid>(this.boidSettings.count);
        this.localCenter = localCenter;
        this.alignmentArrow = alignmentArrow;
        this.separationArrow = separationArrow;
    }
```

Рисунок 3.18 – Клас FlockingSystem

Наступний клас FlockingSystem, Він відповідає за керування взаємодією між боїдами згідно з класичними принципами алгоритму Boids: вирівнювання (alignment), утримання разом (cohesion) та уникнення зіткнень (separation). Цей клас обробляє поведінку боїдів у реальному часі, аналізуючи положення

сусідів, напрямок руху та відстань між агентами, що дозволяє досягти реалістичної динаміки руху зграї.

Конструктор FlockingSystem (рис. 3.18) приймає список об’єктів зграї (boids), налаштування гри (gameSettings), налаштування поведінки зграї (boidSettings) та декілька об'єктів (localCenter, alignmentArrow, separationArrow), які використовуються для візуалізації зграйної поведінки.

Метод update (рис. 3.19) є основним методом оновлення симуляції, який обробляє всі правила зграйної поведінки для кожного об’єкта.

```
public void update(float deltaTime) {
    // reset the debug boid indicators
    var boidCount = boids.Count;
    if (gameSettings.showBoidForces && boidCount > 0) {
        var debugBoid = boids[0];
        var debugBoidPosition = debugBoid.transform.position;
        var debugBoidRotation = debugBoid.transform.rotation;
        localCenter.transform.position = debugBoidPosition;
        alignmentArrow.transform.position = debugBoidPosition;
        separationArrow.transform.position = debugBoidPosition;
        alignmentArrow.transform.rotation = debugBoidRotation;
        separationArrow.transform.rotation = debugBoidRotation;
    }

    // cache values
    var minSpeed = boidSettings.minSpeed;
    var maxSpeed = boidSettings.maxSpeed;
    var acceleration = boidSettings.baseAcceleration * deltaTime;
    var separationDistance = boidSettings.separationDistance;
    var typeSeparationDistance = separationDistance * boidSettings.typeSeparationDistanceFactor;
    var alignmentEnabled = boidSettings.alignmentEnabled;
    var cohesionEnabled = boidSettings.cohesionEnabled;
    var separationEnabled = boidSettings.separationEnabled;
    var speedAlignmentEnabled = boidSettings.speedAlignmentEnabled;
    var typeBasedFlockingEnabled = boidSettings.typeBasedFlockingEnabled;
    var alignmentForce = boidSettings.alignmentForce * deltaTime;
    var cohesionForce = boidSettings.cohesionForce * deltaTime;
    var separationForce = boidSettings.separationForce * deltaTime;
```

Рисунок 3.19 – Метод update

На початку методу зберігаються значення з налаштувань (minSpeed, maxSpeed, alignmentForce, cohesionForce, separationForce тощо) у локальні змінні методу. Далі в методі обробляються три базові правила зграйної поведінки Boids (рис. 3.20):

- Вирівнювання (Alignment) - обчислюється середній напрямок руху сусідів (averageDirection). Об’єкт обертається в бік середнього напрямку, використовуючи Quaternion.RotateTowards. Це дозволяє агенту коригувати свій рух відповідно до сусідів, створюючи ефект синхронізованого руху.

- Утримання разом (Cohesion) - обчислюється середнє положення всіх сусідів (averagePosition). Потім обчислюється вектор до цього центру і член зграї обертається у відповідному напрямку вектора.
- Уникнення зіткнень (Separation) - для кожного сусіда перевіряється відстань до поточного об’єкту зграї. Якщо сусід знаходиться занадто близько (neighbor.distanceTemp < separationDistance), обчислюється напрямок відштовхування. Член зграї обертається у бік, протилежний найближчому сусідові, на заданий кут (separationForce).

```
// direction alignment
if (alignmentEnabled && hasSameTypeNeighbors) {
    var angle = MathUtils.vectorToAngle(averageDirection);
    currentBoid.transform.rotation = Quaternion.RotateTowards(
        currentBoid.transform.rotation,
        Quaternion.Euler(0, 0, angle),
        alignmentForce);

    if (i == 0) alignmentArrow.transform.rotation = Quaternion.Euler(0, 0, angle);
}
// cohesion
if (cohesionEnabled && hasSameTypeNeighbors) {
    var cohesionDirection = averagePosition - currentBoidPosition;
    var angle = MathUtils.vectorToAngle(cohesionDirection);
    currentBoid.transform.rotation = Quaternion.RotateTowards(
        currentBoid.transform.rotation,
        Quaternion.Euler(0, 0, angle),
        cohesionForce);

    if (i == 0) localCenter.transform.position = averagePosition;
}
// separation
if (separationEnabled && shouldSeparate) {
    var angle = MathUtils.vectorToAngle(separationDirection);
    currentBoid.transform.rotation = Quaternion.RotateTowards(
        currentBoid.transform.rotation,
        Quaternion.Euler(0, 0, angle),
        separationForce);

    if (i == 0) separationArrow.transform.rotation = Quaternion.Euler(0, 0, angle);
}
// speed alignment
if (speedAlignmentEnabled && hasSameTypeNeighbors) {
    var currentSpeed = currentBoid.speed;
    if (currentSpeed < averageSpeed) {
        currentSpeed += acceleration;
    } else if (currentSpeed > averageSpeed) {
        currentSpeed -= acceleration;
    }
    currentBoid.speed = Mathf.Clamp(currentSpeed, minSpeed, maxSpeed);
}
```

Рисунок 3.20 – Реалізація трьох правила зграйної поведінки

Метод findNeighbors використовується для виявлення сусідів поточного об’єкту. Він проходить через усіх агентів у симуляції, перевіряючи, чи потрапляють вони у зону огляду (boidSettings.viewDistance). Якщо агент знаходиться у межах видимості, він додається до списку neighbors.

Клас EvasionSystem зосереджується на налаштуванні параметрів уникнення зіткнень. Завдяки використанню параметрів GameSettings та BoidSettings, клас забезпечує гнучкість у налаштуванні поведінки агентів,

дозволяючи як активувати, так і деактивувати окремі механізми під час уникнення перешкод.

Метод setWallCorners (рис. 3.21) використовується для визначення координат кутів ігрового простору з урахуванням певного зміщення (offset). Це зміщення (predatorSize / 3) додається до координат нижнього лівого кута (bottomLeft) і віднімається від координат верхнього правого (topRight). Це запобігає розташуванню об'єктів надто близько до країв світу.

```
Ссылок: 1
void setWallCorners(Rect worldRect, float offset) {
    bottomLeft = worldRect.min;
    topRight = worldRect.max;
    bottomLeft.x += offset;
    bottomLeft.y += offset;
    topRight.x -= offset;
    topRight.y -= offset;
}
```

Рисунок 3.21 – Метод setWallCorners

Метод update(float deltaTime) є основним для цього класу. Використовуються для перебору всіх елементів зграї, щоб визначити наскільки агент наближений до кордонів, та викликає інші функції для відхилення об’єкту від кордонів сцени (рис. 3.22).

```
Ссылок: 2
public void update(float deltaTime) {
    var predatorPosition = predator.transform.position;
    var wallAvoidanceEnabled = boidSettings.wallAvoidanceEnabled;
    var predatorEvasionEnabled = boidSettings.predatorEvasionEnabled;
    var wallAvoidanceForce = boidSettings.wallAvoidanceForce;
    var boidSize = this.boidSize;
    var boidRayLength = 2 * boidSize;
    foreach (var boid in boids) {
        if (wallAvoidanceEnabled) {
            avoidWalls(boid.transform, boidSize, boidRayLength, deltaTime, wallAvoidanceForce);
        }
        if (predatorEvasionEnabled) {
            evadePredator(boid, predatorPosition, deltaTime);
        }
    }
    avoidWalls(predator.transform, predatorSize, predatorSize, deltaTime, predatorSettings.wallAvoidanceForce);
}

Ссылок: 2
void avoidWalls(Transform transform, float bodySize, float rayLength, float deltaTime, float avoidanceForce) {
    var currentAngle = transform.eulerAngles.z;
    if (currentAngle < 0) currentAngle += 360;
    var newAngle = findAngleToAvoidWalls(currentAngle, transform.position, bodySize, rayLength);
    if (Math.Abs(currentAngle - newAngle) < 0.1f) return;

    transform.rotation = Quaternion.RotateTowards(
        transform.rotation,
        Quaternion.Euler(0, 0, newAngle),
        avoidanceForce * deltaTime);
}
```

Рисунок 3.22 – Методи update та avoidWalls

Метод avoidWalls (рис. 3.22) відповідає за запобігання зіткнень агентів із межами ігрового простору, забезпечуючи, щоб члени зграї залишалися в межах визначеної зони симуляції. Його робота ґрунтується на перевірці відстані кожного агента до країв ігрового простору (bottomLeft та topRight), яка була попередньо обчислена методом setWallCorners (рис. 3.21). При виявленні наближення до меж, змінюється напрямок руху, запобігаючи виходу за межі світу. Це досягається за допомогою перевірок позиції об’єкту по кожній осі (X та Y). Якщо агент знаходиться занадто близько до однієї з меж, використовується Quaternion.RotateTowards.

Такий підхід дозволяє плавно коригувати рух, зберігаючи природність анімації. Важливою деталлю є використання параметра boidRayLength, який визначає відстань, на якій система починає реагувати на наближення до стін.

```
Ссылок: 3
public class BorderSystem : System {
    readonly List<Boid> boids;
    readonly Predator predator;
    readonly Vector3 bottomLeft;
    readonly Vector3 topRight;
    readonly float boidOffset;
    readonly float predatorOffset;

    Ссылок: 1
    public BorderSystem(List<Boid> boids, Predator predator, GameSettings gameSettings) {
        this.boids = boids;
        this.predator = predator;
        bottomLeft = gameSettings.worldRect.min;
        topRight = gameSettings.worldRect.max;
        boidOffset = gameSettings.boidSettings.size / 2;
        predatorOffset = predator.transform.localScale.x / 2;
    }
```

Рисунок 3.23 – Клас BorderSystem

Клас BorderSystem (рис. 3.23) у Unity відповідає за обмеження руху членів зграї та в межах визначеного ігрового простору, де об'єкти, що виходять за межі сцени, з'являються з протилежного боку екрану.

Основними полями класу є список об’єктів зграї (boids), об'єкт та координати меж ігрового простору bottomLeft і topRight, які визначаються з параметрів сцени gameSettings. Значення boidOffset і predatorOffset використовуються для врахування розміру об'єктів під час перевірок перетину меж, щоб агенти не зникали частково за екраном.

Метод update(float deltaTime) викликається на кожному кадрі симуляції та проходить через список боїдів, а також перевіряє положення хижака, застосовуючи метод teleportToOppositeBorderIfOutside (рис. 3.24) для кожного об'єкта. Метод teleportToOppositeBorderIfOutside виконує перевірку координат об'єкта відносно меж сцени: якщо агент виходить за лівий або правий край, він телепортується на протилежний бік; аналогічно, перевірка відбувається для верхньої та нижньої меж. Така механіка забезпечує безперервність симуляції, дозволяючи агентам вільно переміщуватися у просторі без різких обмежень або блокування руху.

```
Ссылок: 2
public void update(float deltaTime) {
    foreach (var boid in boids) {
        teleportToOppositeBorderIfOutside(boid.transform, boidOffset);
    }
    teleportToOppositeBorderIfOutside(predator.transform, predatorOffset);
}

Ссылок: 2
void teleportToOppositeBorderIfOutside(Transform transform, float offset) {
    var position = transform.position;
    if (position.x + offset < bottomLeft.x) {
        transform.position = new Vector3(topRight.x, position.y, position.z);
    } else if (position.x - offset > topRight.x) {
        transform.position = new Vector3(bottomLeft.x, position.y, position.z);
    }
    if (position.y + offset < bottomLeft.y) {
        transform.position = new Vector3(position.x, topRight.y, position.z);
    } else if (position.y - offset > topRight.y) {
        transform.position = new Vector3(position.x, bottomLeft.y, position.z);
    }
}
```

Рисунок 3.24 – Методи update та teleportToOppositeBorderIfOutside

Це особливо корисно для великих зграйних систем, де агенти можуть покидати межі екрану, але мають залишатися у візуальному полі для спостереження за їхньою поведінкою. Клас BorderSystem чітко займається виключно контролем за межами сцени, не впливаючи на рух або взаємодію боїдів. Завдяки цьому він легко інтегрується з іншими системами симуляції, такими як MovementSystem та EvasionSystem.

Далі клас DragSystem відповідає за моделювання ефекту гальмування (drag) для членів зграї, що імітує поступове зменшення швидкості об'єктів у

середовищі. Його основними компонентами є список об’єктів - boids, передані через конструктор для доступу до параметра гальмування (dragDeceleration). Метод update(float deltaTime) викликається кожен кадр та відповідає за зменшення швидкості кожного агенту відповідно до параметра уповільнення (deceleration), який розраховується як добуток значення dragDeceleration на deltaTime.

```
using System.Collections.Generic;
using GameScene.Settings;

namespace GameScene.Systems {
Ссылок: 3
public class DragSystem : System {
    readonly List<Boid> boids;
    readonly Predator predator;
    readonly GameSettings settings;

    Ссылок: 1
    public DragSystem(List<Boid> boids, Predator predator, GameSettings settings) {
        this.boids = boids;
        this.predator = predator;
        this.settings = settings;
    }

    Ссылок: 2
    public void update(float deltaTime) {
        var deceleration = settings.dragDeceleration * deltaTime;
        foreach (var boid in boids) {
            boid.speed -= deceleration;
            if (boid.speed < 0) boid.speed = 0;
        }
        predator.speed -= deceleration;
        if (predator.speed < 0) predator.speed = 0;
    }
}
}
```

Рисунок 3.24 – Клас DragSystem

Для кожного агента перевіряється, чи його швидкість не опустилася нижче нуля, щоб запобігти руху у зворотному напрямку або некоректним значенням швидкості. Якщо швидкість стає меншою за нуль, вона примусово встановлюється на нуль (boid.speed = 0)

Даний клас зосереджений виключно на зниженні швидкості об'єктів і не втручається у їхній рух або взаємодію з іншими агентами. Використання параметра deltaTime дозволяє зробити симуляцію незалежною від частоти кадрів, забезпечуючи стабільне уповільнення як при високій, так і при низькій частоті оновлення сцени.

Для зберігання налаштувань які присутні у вище розглянутих класах, використовується наступний клас GameSettings. Він використовується для

централізованого управління параметрами сцени, що робить його ключовим елементом у системі контролю симуляції. Клас є серіалізованим ([Serializable]), що дозволяє Unity зберігати його значення у файлах збережень або конфігураціях сцен, а також відображати їх у редакторі Unity для зручного налаштування.

```
namespace GameScene.Settings {
[Serializable]
Ссылок: 20
public class GameSettings {
    public bool showMousePosition;
    public bool showViewAreas;
    public bool showBoidForces;

    //todo: do not save if zero
    public float gameSpeed;

    public float dragDeceleration;

    public BoidSettings boidSettings = new();
    public PredatorSettings predatorSettings = new();
    public BoidControlSettings boidControlSettings = new();

    [NonSerialized] public GameSettings defaultSettings;
    [NonSerialized] public bool settingsPanelEnabled;
    [NonSerialized] public Rect worldRect;
```

Рисунок 3.25 – Клас GameSettings

Серед основних полів класу можна виділити showMousePosition, showViewAreas і showBoidForces — це булеві прапорці, які контролюють елементи налагодження сцени. Вони дозволяють візуалізувати такі елементи, як поточне положення миші, області огляду боїдів та сили, що впливають на їхню поведінку. Це особливо корисно для перевірки роботи симуляції під час розробки.

Поле gameSpeed контролює швидкість симуляції, дозволяючи користувачеві регулювати, наскільки швидко оновлюється фізична модель. Параметр dragDeceleration визначає рівень уповільнення руху агентів через опір середовища, що застосовується у класі DragSystem. Для зберігання більш специфічних параметрів використовуються вкладені об'єкти boidSettings, predatorSettings та boidControlSettings, кожен із яких відповідає за налаштування відповідних компонентів симуляції: боїдів, хижаків та взаємодії з користувачем.

Поле defaultSettings позначене атрибутом [NonSerialized], що означає, що воно не буде збережене у файлах конфігурації Unity. Воно використовується для зберігання стандартних значень налаштувань, які можуть бути відновлені за допомогою методу reset(). Це поле забезпечує зручний механізм повернення до початкових значень параметрів, зберігаючи оригінальні конфігурації. Аналогічно, settingsPanelEnabled також не серіалізується і використовується для управління станом панелі налаштувань під час роботи симуляції.

```
Ссылок: 2
public void reset() {
    if (defaultSettings == null) {
        Log.warn("GameSettings", "cannot reset settings: defaultSettings is null");
        return;
    }

    showMousePosition = defaultSettings.showMousePosition;
    showViewAreas = defaultSettings.showViewAreas;
    showBoidForces = defaultSettings.showBoidForces;

    gameSpeed = defaultSettings.gameSpeed;

    dragDeceleration = defaultSettings.dragDeceleration;

    boidSettings.reset(defaultSettings.boidSettings);
    predatorSettings.reset(defaultSettings.predatorSettings);
    boidControlSettings.reset(defaultSettings.boidControlSettings);
}
```

Рисунок 3.26 – Метод reset

Метод reset () виконує скидання всіх параметрів GameSettings до значень, збережених у defaultSettings. Якщо значення defaultSettings не встановлено (null), виводиться попередження у лог (Log.warn). В іншому випадку, всі параметри, включаючи вкладені об'єкти boidSettings, predatorSettings та boidControlSettings, скидаються до стандартних значень. Це робить метод особливо корисним для тестування та повернення сцени до початкового стану після змін у налаштуваннях.

Таким чином, GameSettings є зручним елементом для управління системними параметрами, їх збереження та відновлення налаштувань в інших структура проекту.

Наступним етапом розробки є налаштування сцени в Unity. Для роботи алгоритму використовується так званий об’єкт контролер, що має всі основні скрипти для створення та відображення зграї (рис. 3.27):

Рисунок 3.27 – Об’єкт контролер сцени

Крім об’єкту контролера також потрібно створити шаблон для відображення окремого об’єкту зграї в Unity для цього використовується Prefab. Prefab (префаб) у Unity — це шаблонний об'єкт, який дозволяє зберігати, відтворювати та використовувати готові об'єкти сцени з усіма їхніми компонентами, налаштуваннями та вкладеними об'єктами. Prefab використовуються для створення повторюваних об'єктів у сценах.

Для відображення стаї використовується шаблон Boid:

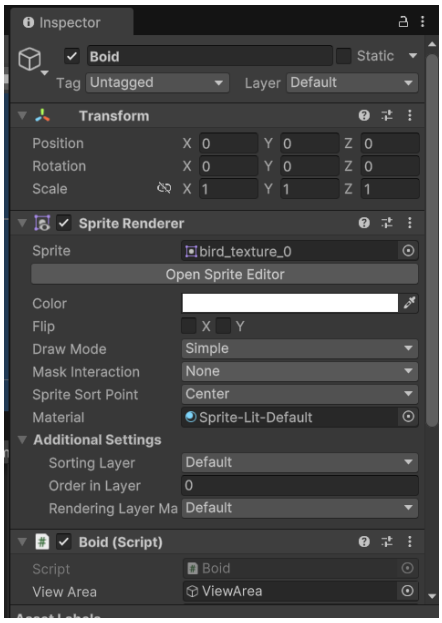


Рисунок 3.28 – Шаблон Boid

3.4 Тестування проекту

Етап тестування алгоритму є критично важливою фазою розробки, яка спрямована на перевірку коректності роботи системи, виявлення можливих помилок та оцінку відповідності алгоритму визначеним вимогам.

Для початку потрібно перевірити всі об’єкти сцени, чи мають воні потрібні прикріплені об’єкти та параметри на основі яких працює алгоритм. Далі потрібно завантажити сцену в Unity:

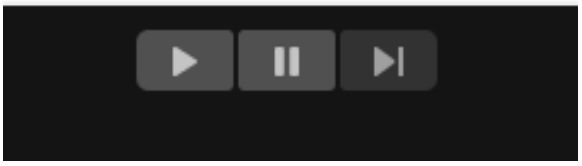


Рисунок 3.29 – Кнопка завантаження сцени

Далі користувач після завантаження повинен спостерігати, створення зграї птахів на сцені:

Рисунок 3.30 – Робота алгоритму симуляції зграї

Користувач спостерігає як коректно відтворюється зграя, за рахунок принципів утримання в стаї об’єктів, та оцінка в реальному часі відстань між сусідніми членами зграї, що дозволяє об’єктам не наштовхуватися один на один.

Під час роботи алгоритму, користувач може приблизити камеру, щоб спостерігати лише окрему ділянку простору (рис. 3.31):

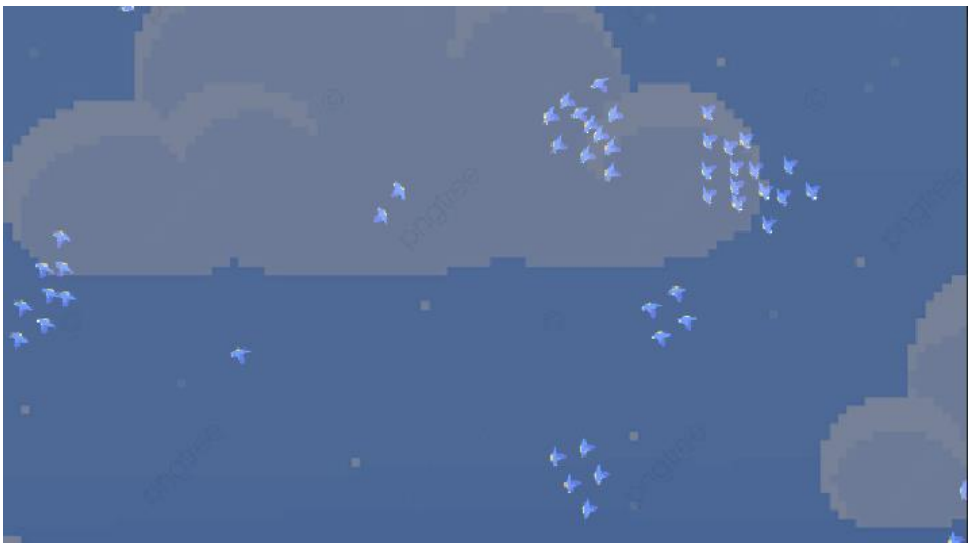


Рисунок 3.31 – Наближення камери сцени

Якщо потрібно, користувач також може віддалити сцену, для більшого огляду зграї:

Рисунок 3.32 – Віддалення камери від сцени

Далі користувач може збільшити кількість зграї за рахунок збільшення параметру BoidAreaFactor:

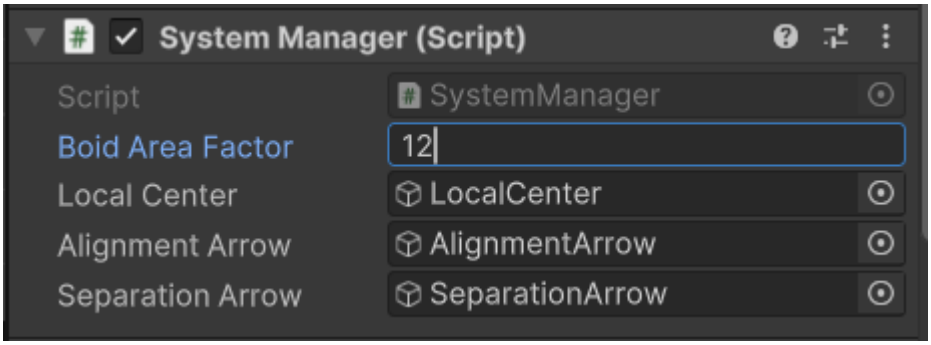


Рисунок 3.33 – Збільшення кількості зграй на сцені

Після завантаження користувач повинен бачити більшу кількість елементів:

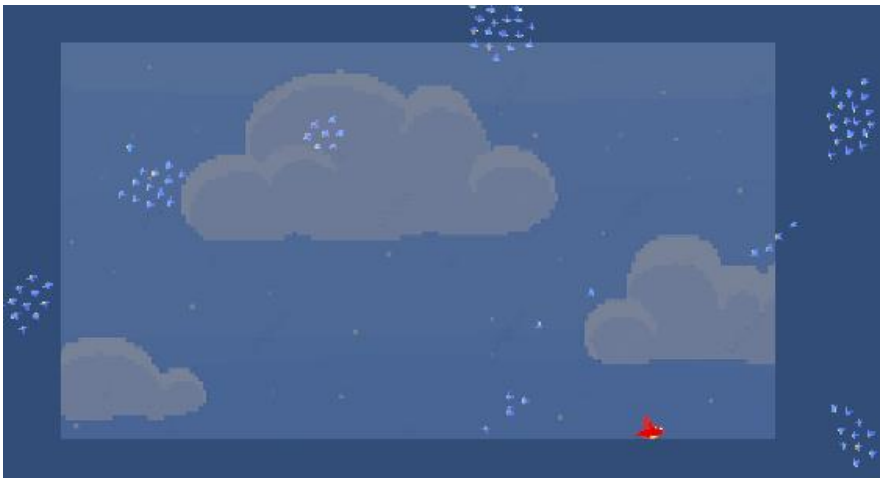


Рисунок 3.34 – Збільшена кількість стай

Завдяки цим діям можна більш детально і наочно спостерігати, яким чином збільшується чисельність зграй у процесі симуляції, як окремі групи агентів поступово зливаються одна з одною, утворюючи більш масштабні структури. Це дозволяє простежити закономірності зграйної поведінки, зокрема ефект самоорганізації, коли агенти слідують заданим правилам вирівнювання, утримання разом та уникнення зіткнень. Важливим елементом спостереження є те, як зграйні структури реагують на зовнішні подразники, зокрема червоного птаха, який виконує роль хижака. Агресивна взаємодія цього об'єкта змушує агентів змінювати свої траєкторії, демонструючи механізми уникнення загрози та перебудову поведінки зграї у відповідь на потенційну небезпеку. Це сприяє глибшому розумінню того, як алгоритм обробляє складні сценарії взаємодії в реальному часі, і наочно демонструє ефективність застосованих правил симуляції.

3.5 Висновок до третього розділу

У третьому розділі було розроблено та детально описано алгоритм симуляції зграйної поведінки, що ґрунтується на класичних принципах вирівнювання, утримання разом та уникнення зіткнень. Було використано архітектурний підхід Entity Component System (ECS), який забезпечив

модульність, гнучкість і високу продуктивність при обробці великої кількості агентів у реальному часі.

Алгоритм було реалізовано в середовищі Unity з використанням компонентної архітектури, яка включає підсистеми Systems, Controller та Settings, кожна з яких відповідає за окремі аспекти симуляції: обробку логіки поведінки агентів, координацію симуляції та управління параметрами сцени. Особливу увагу було приділено оптимізації алгоритму за рахунок використання багатопотокової обробки та зниження навантаження на процесор, що дозволяє моделювати поведінку великих зграй без суттєвих втрат продуктивності.

Було також реалізовано функціональність масштабування кількості агентів, інтерактивну зміну параметрів симуляції та підтримку візуалізації в реальному часі. Крім базових правил зграйної поведінки, алгоритм підтримує симуляцію реакцій агентів на загрози, таких як хижак, що додає реалістичності та дозволяє моделювати складніші сценарії поведінки.

Серед ключових досягнень можна відзначити:

- Реалізацію модульної архітектури, що спрощує розширення та обслуговування коду.
- Можливість динамічного налаштування параметрів через конфігураційний файл.
- Високу продуктивність алгоритму завдяки використанню ECS та оптимізованих обчислень.
- Візуальне представлення ключових параметрів симуляції, таких як області огляду, напрямки руху та відстані між агентами.

Тестування алгоритму показало стабільність роботи, коректне відтворення принципів зграйної поведінки та можливість візуалізації поведінки великих груп агентів у реальному часі.

ВИСНОВОК

У процесі виконання кваліфікаційної роботи на тему «Розробка та програмна реалізація алгоритму симуляції зграйної поведінки в Unity» було проведено комплексне дослідження методів моделювання колективної поведінки агентів та розроблено програмний алгоритм для реалізації симуляції зграйної поведінки у середовищі Unity. Дослідження включало теоретичний аналіз підходів до симуляції зграйної поведінки, обґрунтування вибору методів, практичну розробку програмного забезпечення та його тестування.

Основними завданнями, вирішеними в рамках дослідження, стали:

1. Аналіз існуючих алгоритмів моделювання зграйної поведінки, зокрема алгоритму Boids, рою частинок (PSO) та клітинних автоматів, із детальним описом принципів вирівнювання, когезії та уникнення зіткнень.

2. Обґрунтування вибору алгоритму Boids як основи для моделювання зграйної поведінки з огляду на його простоту, ефективність та поширене використання у комп'ютерних симуляціях.

3. Розробка алгоритму симуляції зграйної поведінки з урахуванням базових правил Boids та компонентно-орієнтованої архітектури Unity (ECS) для підвищення продуктивності та зручності управління великою кількістю агентів.

4. Програмна реалізація алгоритму у середовищі Unity з використанням мови програмування C#, що передбачала розробку системи управління агентами, контролера та системи налаштувань для управління параметрами симуляції.

5. Тестування програмного продукту для перевірки відповідності функціональним та нефункціональним вимогам, включаючи стабільність роботи, продуктивність оброблення великої кількості агентів у реальному часі та візуальну реалістичність.

У процесі дослідження було здійснено порівняльний аналіз програмних середовищ Unity та Godot для розробки симуляцій. Вибір Unity обґрунтовано

його розширеними можливостями для створення інтерактивних симуляцій, підтримкою C#, компонентно-орієнтованою архітектурою та здатністю обробляти великі обсяги даних у реальному часі.

Особливу увагу було приділено структурі проекту, яка реалізована у вигляді трьох основних компонентів: Systems, Controller, Settings. Такий підхід забезпечує модульність, легкість у підтримці коду, можливість масштабування та ефективне управління параметрами симуляції.

У процесі виконання роботи було сформульовано такі вимоги до програмного забезпечення:

- Функціональні вимоги – підтримка симуляції зграйної поведінки в реальному часі, дотримання принципів вирівнювання, когезії та уникнення зіткнень, можливість налаштування параметрів симуляції через конфігураційний файл.
- Нефункціональні вимоги – стабільна продуктивність при обробці великої кількості агентів, оптимізація використання апаратних ресурсів (CPU, GPU), сумісність із великими сценами та стабільність роботи програмного продукту.

Розроблений алгоритм успішно пройшов тестування у середовищі Unity, продемонструвавши здатність до ефективного моделювання складних сценаріїв зграйної поведінки агентів. Результати дослідження можуть бути застосовані для створення відеоігор, симуляцій у сфері комп'ютерної графіки, анімації, робототехніки та наукових експериментів із моделюванням колективної поведінки.

Таким чином, поставлена мета щодо розробки алгоритму симуляції зграйної поведінки у середовищі Unity була досягнута, а отримані результати можуть бути використані як основа для подальших досліджень у сфері моделювання складних систем та колективної поведінки агентів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hildmann, Hanno, et al. Termite algorithms to control collaborative swarms of satellites. In:Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation,2018.P. 109–124.

2. Brooks, Rodney. A robust layered control system for a mobile robot.IEEE journal on robotics and automation, 1986.P. 14–23

3. Міллер П.Д. Ройовий інтелект: Мурахи, бджоли і птахи здатні багато чому нас навчити. М .: Фактор, 2007. С. 88–107.

4. Reynolds C.W. Flocks, Herds, and Schools: A Distributed Behavioral Model. London, 1987. P. 25–34.

5. Lee D., Seo H., Jung M. W. Neural basis of reinforcement learning and decision making. Annual review of neuroscience. 2012. Vol. 35, no. 1. P. 287–308. https://doi.org/10.1146/annurev-neuro-062111-150512.

6. A Cellular automaton model for crowd movement and egress simulation – [Електронний ресурс]. – Режим доступу: http://surl.li/grxufe

7. Imitation Learning with Graph Neural Networks for Improving Swarm Robustness under Restricted Communications Imitation Learning with Graph Neural Networks for Improving Swarm Robustness under Restricted Communications – [Електронний ресурс]. – Режим доступу: http://surl.li/stubci

8. Документація Unity3D – [Електронний ресурс]. – Режим доступу: https://docs.unity.com

9. Документація C# – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/csharp/

10. Godot Engine 4.3 documentation – [Електронний ресурс]. – Режим доступу: https://docs.godotengine.org/en/stable/

11. Yuen, D., Wang, L., Chi X., Johnsson, L., Ge, W., Shi, Y. (2013). GPU Solutions to Multi-scale Problems in Science and Engineering, Springer Science & Business Media. DOI: 10.1007/978-3-642-16405-7_2

12. Lammers, K. (2013). Unity Shaders and Effects Cookbook. Packt Publishing. 268 p.

13. Simple Boids is a simulation of the behavior of flocks of birds and fish. – [Електронний ресурс]. – Режим доступу: http://surl.li/pewndo

14. Boid Behavior and Optimization for a Pikmin-Like – [Електронний ресурс]. – Режим доступу: http://surl.li/nqmbfz

15. Controlling thousands of fish with Particles Like – [Електронний ресурс]. – Режим доступу: http://surl.li/ddxucb