Лекція 23

ECS: Сутність-Компонент-Система

Архітектурний патерн для масштабних симуляцій

Entity • Component • System • World • Query

🤖 Ідеальна архітектура для симуляції тисяч агентів

Частина 1: Концепції та базова реалізація

О. С. Бичков • 2025

План лекції (Частина 1)

1. Що таке ECS?
2. Історія та мотивація
3. Проблеми OOP для симуляцій
4. Entity (Сутність)
5. Component (Компонент)
6. System (Система)
7. World (Світ)
8. Архітектура даних
9. Data-Oriented Design
10. Cache-friendly layout
11. Порівняння ECS vs OOP
12. Порівняння ECS vs Actor
13. Переваги ECS
14. Недоліки ECS
15. 🤖 ECS для MAC
16. Підсумок

Частина 2: bevy_ecs, запити, ресурси, повна симуляція рою

Що таке ECS?

ECS (Entity-Component-System) — архітектурний патерн,
де дані відокремлені від поведінки:

- Entity (Сутність) — унікальний ідентифікатор
- Component (Компонент) — дані без логіки
- System (Система) — логіка без стану

Ключова ідея:
"Composition over Inheritance"

```
┌──────────────────────────────────────────────┐
│  Entity 1:  [Position] [Velocity] [Health]   │
│  Entity 2:  [Position] [Velocity]  [Sprite]  │
│  Entity 3:  [Position]  [Health] [AI]        │
└──────────────────────────────────────────────┘

  System: MovementSystem — обробляє всі (Position, Velocity)
```

Історія ECS

1998 — Thief: The Dark Project
Перші ідеї композиції компонентів

2002 — Dungeon Siege
Scott Bilas: "A Data-Driven Game Object System"
Формалізація підходу

2007 — Unity 3D
GameObject + Components
Популяризація в геймдеві

2017 — Unity ECS (DOTS)
"Чистий" ECS з Data-Oriented Design

2018+ — Rust ECS бібліотеки
specs, legion, hecs, bevy_ecs
Оптимальне поєднання з ownership моделлю

Проблеми OOP для симуляцій

Класична OOP ієрархія:

```
GameObject
  ├── Enemy
  │     ├── FlyingEnemy
  │     └── GroundEnemy
  └── Player
        └── FlyingPlayer  ← Дублювання коду!
```

Проблеми:

❌ Diamond Problem — множинне наслідування
❌ Rigid Hierarchy — важко змінити структуру
❌ Code Duplication — FlyingEnemy та FlyingPlayer
❌ Cache Misses — об'єкти розкидані в пам'яті
❌ Tight Coupling — логіка прив'язана до класу

```
// ООР підхід
class FlyingEnemy extends Enemy {
    position: Vec3;
    velocity: Vec3;
    health: i32;
    fly_height: f32;

    fn update(&mut self) {
        self.position += self.velocity;
        self.maintain_altitude();
        self.attack_player();
    }
}

// ECS підхід
// Компоненти (тільки дані)
struct Position(Vec3);
struct Velocity(Vec3);
struct Health(i32);
struct Flying { height: f32 }
struct Enemy;

// Сутність = набір компонентів
let flying_enemy = world.spawn((
    Position(Vec3::ZERO),
    Velocity(Vec3::new(1.0, 0.0, 0.0)),
    Health(100),
    Flying { height: 10.0 },
    Enemy,
));
```

```rust
// Entity зазвичай просто число
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
pub struct Entity {
    id: u32,
    generation: u32,  // Для переіспользання ID
}

// Створення сутностей
let player = world.spawn_empty();           // Порожня сутність
let enemy = world.spawn((Position::default(), Health(100)));  // З компонентами

// Знищення
world.despawn(enemy);

// Перевірка існування
if world.contains(player) { ... }
```

```rust
use bevy_ecs::component::Component;

// Компоненти — просто структури з даними
#[derive(Component)]
struct Position {
    x: f32,
    y: f32,
    z: f32,
}

#[derive(Component)]
struct Velocity(Vec3);  // Tuple struct

#[derive(Component)]
struct Health(i32);

#[derive(Component)]
struct Name(String);

// Marker components (без даних)
#[derive(Component)]
struct Player;  // Тег "це гравець"

#[derive(Component)]
struct Enemy;   // Тег "це ворог"

#[derive(Component)]
struct Flying;  // Тег "може літати"
```

```rust
use bevy_ecs::system::Query;

// Система руху — обробляє всі сутності з Position та Velocity
fn movement_system(mut query: Query<(&mut Position, &Velocity)>) {
    for (mut pos, vel) in query.iter_mut() {
        pos.x += vel.0.x;
        pos.y += vel.0.y;
        pos.z += vel.0.z;
    }
}

// Система здоров'я — тільки сутності з Health
fn health_system(query: Query<(Entity, &Health)>, mut commands: Commands) {
    for (entity, health) in query.iter() {
        if health.0 <= 0 {
            commands.entity(entity).despawn();  // Видалити мертвих
        }
    }
}

// Система з фільтром — тільки Flying Enemy
fn flying_enemy_system(
    query: Query<&mut Position, (With<Flying>, With<Enemy>)>
) { ... }
```

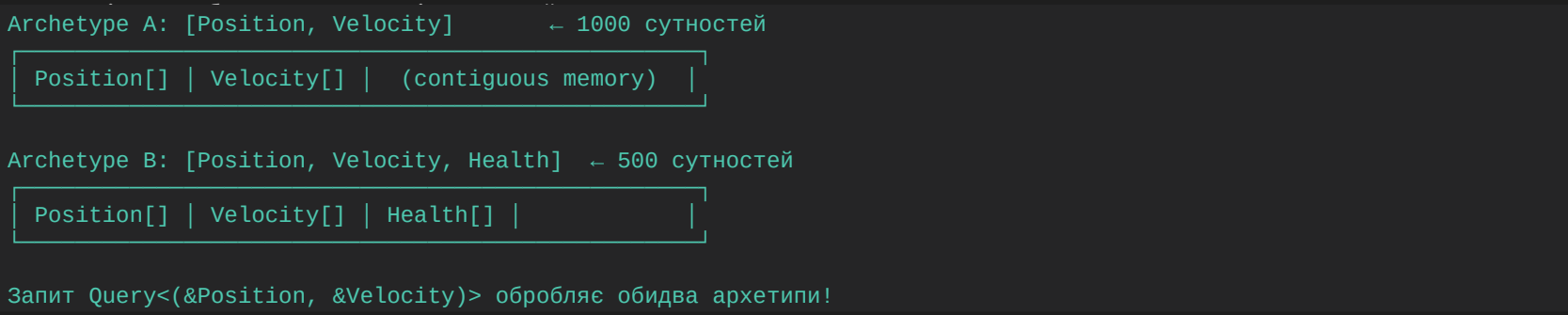
```rust
use bevy_ecs::world::World;

// Створення світу
let mut world = World::new();

// Додавання ресурсів (глобальних даних)
world.insert_resource(GameTime { delta: 0.016 });
world.insert_resource(GameConfig::default());

// Створення сутностей
let player = world.spawn((
    Position { x: 0.0, y: 0.0, z: 0.0 },
    Velocity(Vec3::ZERO),
    Health(100),
    Player,
)).id();

// Доступ до компонента
if let Some(health) = world.get::<Health>(player) {
    println!("Player health: {}", health.0);
}

// Модифікація компонента
if let Some(mut health) = world.get_mut::<Health>(player) {
    health.0 -= 10;
}
```

```rust
use bevy_ecs::system::Query;
use bevy_ecs::query::{With, Without, Changed, Added};

// Базовий запит — всі сутності з Position та Velocity
fn system1(query: Query<(&Position, &Velocity)>) {
    for (pos, vel) in query.iter() { ... }
}

// Мутабельний запит
fn system2(mut query: Query<&mut Position>) {
    for mut pos in query.iter_mut() {
        pos.x += 1.0;
    }
}

// Фільтри
fn system3(
    // Тільки з Player, без Enemy
    query: Query<&Position, (With<Player>, Without<Enemy>)>
) { ... }

// Change detection
fn system4(
    query: Query<&Position, Changed<Position>>  // Тільки змінені
) { ... }

// Optional components
fn system5(
    query: Query<(&Position, Option<&Velocity>)>  // Velocity може бути відсутній
) {
```

Archetype (Архетип)

Archetype — група сутностей з однаковим набором компонентів

Концепція:
- Сутності з однаковими компонентами зберігаються разом
- Оптимізація для cache-friendly доступу



Data-Oriented Design (DOD)

DOD — проектування навколо даних, а не абстракцій

Принципи:
- Дані визначають структуру програми
- Оптимізація для hardware (cache, SIMD)
- Масові операції замість поодиноких

```
OOP: Array of Structs (AoS)

[Obj1: pos,vel,hp]
[Obj2: pos,vel,hp]
[Obj3: pos,vel,hp]

Cache miss при ітерації!
```

```
ECS: Struct of Arrays (SoA)

[pos1, pos2, pos3, ...]
[vel1, vel2, vel3, ...]
[hp1,  hp2,  hp3,  ...]

Cache-friendly!
```

Cache Performance

Чому SoA швидший?

CPU Cache працює з cache lines (~64 байти)
При читанні однієї змінної — завантажується вся лінія

AoS: struct Enemy { pos: Vec3, vel: Vec3, hp: i32, name: String, ... }
[Enemy1: 128 bytes][Enemy2: 128 bytes][Enemy3: 128 bytes]
При ітерації тільки по pos — 75% cache waste!

Бенчмарки (10,000 сутностей):
- AoS ітерація: ~2.5ms
- SoA ітерація: ~0.3ms
- Прискорення: 8x!

ECS vs OOP — порівняння

Аспект	OOP	ECS
Структура	Ієрархія класів	Композиція компонентів
Дані	В об'єктах	Окремо (компоненти)
Логіка	Методи класів	Системи
Зв'язки	Наслідування	Запити (Query)
Пам'ять	AoS (розкидані)	SoA (згруповані)
Розширення	Нові класи	Нові компоненти
Паралелізм	Складний	Природній
Продуктивність	Помірна	Висока

ECS виграє при великій кількості однотипних сутностей

ECS vs Actor Model

Аспект	Actor Model	ECS
Ізоляція	Повна (mailbox)	Часткова (shared world)
Комунікація	Повідомлення	Через компоненти
Паралелізм	По акторах	По системах
Стан	В акторі	В компонентах
Масштаб	~10K акторів	~1M+ сутностей
Latency	Нижча (async)	Вища (batch)
Throughput	Помірний	Високий
Use case	Distributed	Simulation

Actor: кожен агент ізольований | ECS: всі агенти обробляються разом

Переваги ECS

✅ Продуктивність
- Cache-friendly data layout
- Ефективні batch операції
- Легка паралелізація систем

✅ Гнучкість
- Композиція замість наслідування
- Динамічне додавання/видалення компонентів
- Легко створювати нові комбінації

✅ Maintainability
- Системи незалежні одна від одної
- Легко тестувати окремо
- Чіткий поділ відповідальностей

✅ Масштабованість
- Мільйони сутностей
- Легко додавати нові системи
- Hot reload компонентів

Недоліки ECS

❌ Складність
- Інша парадигма мислення
- Крива навчання
- Не інтуїтивно для простих задач

❌ Boilerplate
- Багато дрібних компонентів
- Системи для кожної операції
- Query типи можуть бути складними

❌ Debugging
- Важко відстежити конкретну сутність
- Стан розкиданий по компонентах
- Порядок виконання систем важливий

❌ Overhead для малих систем
- Для <100 об'єктів OOP може бути швидшим
- Archetype management має overhead

🤖 МАС: Чому ECS для МАС?

Мультиагентні системи ідеально підходять для ECS:

1. Багато однотипних сутностей
   - Сотні/тисячі БПЛА
   - Однакова базова структура

2. Спільні операції
   - Рух всіх агентів
   - Оновлення сенсорів
   - Перевірка батареї

3. Комбінаторність
   - Розвідник = Position + Velocity + Camera
   - Штурмовик = Position + Velocity + Weapon
   - Ретранслятор = Position + Antenna

4. Паралелізм
   - Системи можуть працювати паралельно
   - Масова обробка телеметрії

```
// Фізичні компоненти
#[derive(Component)]
struct Position(Vec3);

#[derive(Component)]
struct Velocity(Vec3);

#[derive(Component)]
struct Orientation(Quat);

// Стан
#[derive(Component)]
struct Battery(u8);

#[derive(Component)]
struct Health(i32);

// Ідентифікація
#[derive(Component)]
struct DroneId(u64);

#[derive(Component)]
struct TeamId(u8);

// Спеціалізація (маркери)
#[derive(Component)]
struct Scout;       // Розвідник

#[derive(Component)]
struct Combat;      // Штурмовик
```

```
// Система руху — для ВСІХ сутностей з Position та Velocity
fn movement_system(
    time: Res<Time>,
    mut query: Query<(&mut Position, &Velocity)>,
) {
    let dt = time.delta_seconds();
    for (mut pos, vel) in query.iter_mut() {
        pos.0 += vel.0 * dt;
    }
}

// Система батареї — тільки для дронів
fn battery_system(
    mut query: Query<(&mut Battery, &Velocity), With<DroneId>>,
) {
    for (mut battery, vel) in query.iter_mut() {
        // Швидший рух = більше витрата
        let drain = (vel.0.length() * 0.01) as u8;
        battery.0 = battery.0.saturating_sub(drain);
    }
}

// Система сканування — тільки розвідники з камерою
fn scout_scan_system(
    scouts: Query<(&Position, &Camera), With<Scout>>,
    targets: Query<&Position, With<Target>>,
    mut events: EventWriter<TargetDetected>,
) {
    for (scout_pos, camera) in scouts.iter() {
        for target_pos in targets.iter() {
```

```rust
// Spawning різних типів дронів
fn spawn_drone_fleet(mut commands: Commands) {
    // Базові компоненти для всіх дронів
    let base_bundle = (
        Position(Vec3::ZERO),
        Velocity(Vec3::ZERO),
        Orientation(Quat::IDENTITY),
        Battery(100),
        Health(100),
    );

    // 10 розвідників
    for i in 0..10 {
        commands.spawn((
            base_bundle.clone(),
            DroneId(i),
            TeamId(1),
            Scout,
            Camera { resolution: (1920, 1080), fov: 60.0 },
        ));
    }

    // 5 штурмовиків
    for i in 10..15 {
        commands.spawn((
            base_bundle.clone(),
            DroneId(i),
            TeamId(1),
            Combat,
            Weapon { damage: 50, range: 100.0, cooldown: 1.0 },
```

Підсумок: Частина 1

ECS — Entity-Component-System:

- Entity — ідентифікатор (просто число)
- Component — дані без логіки
- System — логіка без стану
- World — контейнер всього

Переваги:
- Cache-friendly (SoA layout)
- Композиція замість наслідування
- Легка паралелізація
- Масштабованість до мільйонів сутностей

🤖 Ідеально для MAC:
- Багато однотипних агентів
- Спільні операції
- Комбінаторність типів дронів

→ Частина 2: bevy_ecs, ресурси, події, повна симуляція

Лекція 23 (продовження)

ECS: bevy_ecs та практика

Ресурси, події, команди, повна симуляція

Resource • Event • Commands • Schedule • Plugin

🤖 Повна симуляція рою БПЛА на bevy_ecs

Частина 2: Практична реалізація

О. С. Бичков • 2025

План (Частина 2)

1. bevy_ecs огляд
2. Cargo.toml
3. Resource (Ресурс)
4. Event (Подія)
5. Commands (Команди)
6. Schedule (Розклад)
7. SystemSet
8. Plugin (Плагін)
9. Change Detection
10. 🤖 Повна симуляція рою
11. 🤖 Система координації
12. 🤖 Виявлення цілей
13. 🤖 Формації
14. Testing ECS
15. Performance tips
16. Підсумок

```
// Cargo.toml
[dependencies]
bevy_ecs = "0.12"  # Можна без повного Bevy
```

- specs — класичний, менш активний

```
[package]
name = "drone_swarm_ecs"
version = "1.0.0"
edition = "2021"

[dependencies]
# ECS
bevy_ecs = "0.12"

# Math
glam = "0.25"  # Vec3, Quat

# Random
rand = "0.8"

# Serialization
serde = { version = "1", features = ["derive"] }

# Logging
tracing = "0.1"
tracing-subscriber = "0.3"

# Utils
uuid = { version = "1", features = ["v4"] }

[dev-dependencies]
criterion = "0.5"  # Benchmarks

[[bench]]
name = "simulation"
```

```rust
use bevy_ecs::system::Resource;

#[derive(Resource)]
struct SimulationTime {
    elapsed: f32,
    delta: f32,
    tick: u64,
}

#[derive(Resource)]
struct SwarmConfig {
    drone_count: usize,
    area_size: f32,
    tick_rate: f32,
}

#[derive(Resource, Default)]
struct SwarmStats {
    active_drones: usize,
    targets_detected: u64,
    missions_completed: u64,
}

// Доступ в системі
fn update_time(
    mut time: ResMut<SimulationTime>,  // Мутабельний доступ
    config: Res<SwarmConfig>,          // Read-only доступ
) {
    time.elapsed += time.delta;
    time.tick += 1;
```

```rust
use bevy_ecs::event::{Event, EventReader, EventWriter};

#[derive(Event)]
struct TargetDetected {
    detector_id: Entity,
    target_position: Vec3,
    threat_level: ThreatLevel,
}

#[derive(Event)]
struct DroneDestroyed {
    drone_id: Entity,
    position: Vec3,
}

#[derive(Event)]
struct MissionAssigned {
    drone_id: Entity,
    mission_type: MissionType,
    target: Vec3,
}

// Надсилання подій
fn detection_system(
    query: Query<(Entity, &Position, &Camera), With<Scout>>,
    targets: Query<&Position, With<Target>>,
    mut events: EventWriter<TargetDetected>,  // Writer
) {
    for (entity, pos, camera) in query.iter() {
        for target_pos in targets.iter() {
```

```rust
// Читання подій
fn handle_detections(
    mut events: EventReader<TargetDetected>,  // Reader
    mut stats: ResMut<SwarmStats>,
) {
    for event in events.read() {
        tracing::warn!(
            detector = ?event.detector_id,
            position = ?event.target_position,
            "Target detected!"
        );
        stats.targets_detected += 1;
    }
}

// Кілька читачів можуть читати ті самі події
fn log_detections(mut events: EventReader<TargetDetected>) {
    for event in events.read() {
        println!("Logging: target at {:?}", event.target_position);
    }
}

fn alert_system(mut events: EventReader<TargetDetected>) {
    for event in events.read() {
        if event.threat_level == ThreatLevel::High {
            // Trigger alert
        }
    }
}
```

```rust
use bevy_ecs::system::Commands;

fn spawn_system(mut commands: Commands) {
    // Створення сутності
    let drone = commands.spawn((
        Position(Vec3::ZERO),
        Velocity(Vec3::ZERO),
        Battery(100),
    )).id();

    // Додавання компонента до існуючої сутності
    commands.entity(drone).insert(Scout);

    // Видалення компонента
    commands.entity(drone).remove::<Combat>();

    // Видалення сутності
    commands.entity(drone).despawn();
}

fn cleanup_system(
    mut commands: Commands,
    query: Query<(Entity, &Health)>,
) {
    for (entity, health) in query.iter() {
        if health.0 <= 0 {
            commands.entity(entity).despawn();
        }
    }
}
```

```rust
use bevy_ecs::schedule::{Schedule, SystemSet, IntoSystemConfigs};

#[derive(SystemSet, Debug, Clone, PartialEq, Eq, Hash)]
enum SimulationSet {
    Input,      // Читання вводу
    Movement,   // Оновлення позицій
    Physics,    // Фізика та колізії
    AI,         // AI рішення
    Cleanup,    // Очищення
}

fn build_schedule() -> Schedule {
    let mut schedule = Schedule::default();

    schedule
        // Конфігурація порядку sets
        .configure_sets((
            SimulationSet::Input,
            SimulationSet::Movement.after(SimulationSet::Input),
            SimulationSet::Physics.after(SimulationSet::Movement),
            SimulationSet::AI.after(SimulationSet::Physics),
            SimulationSet::Cleanup.after(SimulationSet::AI),
        ))
        // Додавання систем до sets
        .add_systems((
            input_system.in_set(SimulationSet::Input),
            movement_system.in_set(SimulationSet::Movement),
            collision_system.in_set(SimulationSet::Physics),
            ai_system.in_set(SimulationSet::AI),
            cleanup_system.in_set(SimulationSet::Cleanup),
```

```
// Явний порядок систем
schedule.add_systems((
    system_a,
    system_b.after(system_a),
    system_c.after(system_b),
));

// Паралельне виконання (за замовчуванням)
schedule.add_systems((
    independent_system_1,
    independent_system_2,  // Можуть виконуватись паралельно
    independent_system_3,
));

// Chain — строга послідовність
use bevy_ecs::schedule::chain;
schedule.add_systems(
    chain((first, second, third))
);

// Run conditions
fn is_simulation_running(state: Res<SimulationState>) -> bool {
    state.running
}

schedule.add_systems(
    movement_system.run_if(is_simulation_running)
);

// Ambiguity detection
```

```rust
use bevy_ecs::prelude::*;

pub struct DronePlugin;

impl Plugin for DronePlugin {
    fn build(&self, app: &mut App) {
        app
            // Ресурси
            .init_resource::<DroneConfig>()

            // Події
            .add_event::<DroneSpawned>()
            .add_event::<DroneDespawned>()

            // Системи
            .add_systems(Startup, spawn_initial_drones)
            .add_systems(Update, (
                drone_movement_system,
                drone_battery_system,
                drone_ai_system,
            ));
    }
}

pub struct CoordinatorPlugin;

impl Plugin for CoordinatorPlugin {
    fn build(&self, app: &mut App) {
        app
            .add_event::<MissionAssigned>()
```

```rust
use bevy_ecs::query::{Changed, Added, Ref};

// Реагування на зміни
fn on_position_changed(
    query: Query<(Entity, &Position), Changed<Position>>,
) {
    for (entity, pos) in query.iter() {
        println!("Entity {:?} moved to {:?}", entity, pos.0);
    }
}

// Реагування на додавання компонента
fn on_drone_added(
    query: Query<(Entity, &DroneId), Added<DroneId>>,
) {
    for (entity, id) in query.iter() {
        println!("New drone spawned: {:?}", id);
    }
}

// Доступ до інформації про зміни
fn detailed_change_detection(
    query: Query<(Entity, Ref<Position>)>,
) {
    for (entity, pos_ref) in query.iter() {
        if pos_ref.is_changed() {
            println!("Position changed this frame");
        }
        if pos_ref.is_added() {
            println!("Position was just added");
```

```rust
// src/main.rs
use bevy_ecs::prelude::*;

mod components;
mod resources;
mod systems;
mod events;
mod plugins;

fn main() {
    tracing_subscriber::fmt::init();

    let mut app = App::new();

    // Ресурси
    app.insert_resource(SimulationTime::default())
       .insert_resource(SwarmConfig {
           drone_count: 100,
           area_size: 1000.0,
           tick_rate: 60.0,
       })
       .insert_resource(SwarmStats::default());

    // Плагіни
    app.add_plugins((
        DronePlugin,
        CoordinatorPlugin,
        DetectionPlugin,
        FormationPlugin,
    ));
```

```rust
// src/components.rs
use bevy_ecs::prelude::*;
use glam::{Vec3, Quat};

// Фізика
#[derive(Component, Clone, Copy)]
pub struct Position(pub Vec3);

#[derive(Component, Clone, Copy)]
pub struct Velocity(pub Vec3);

#[derive(Component, Clone, Copy)]
pub struct Acceleration(pub Vec3);

// Стан
#[derive(Component)]
pub struct Battery { pub level: u8, pub drain_rate: f32 }

#[derive(Component)]
pub struct Health { pub current: i32, pub max: i32 }

// Ідентифікація
#[derive(Component)]
pub struct DroneId(pub u64);

#[derive(Component)]
pub struct TeamId(pub u8);

// AI стан
#[derive(Component)]
```

```rust
// src/systems/movement.rs

pub fn movement_system(
    time: Res<SimulationTime>,
    mut query: Query<(&mut Position, &mut Velocity, &Acceleration)>,
) {
    let dt = time.delta;

    // Паралельна обробка!
    query.par_iter_mut().for_each(|(mut pos, mut vel, acc)| {
        // Оновлення швидкості
        vel.0 += acc.0 * dt;

        // Обмеження швидкості
        let max_speed = 50.0;
        if vel.0.length() > max_speed {
            vel.0 = vel.0.normalize() * max_speed;
        }

        // Оновлення позиції
        pos.0 += vel.0 * dt;
    });
}

pub fn boundary_system(
    config: Res<SwarmConfig>,
    mut query: Query<(&mut Position, &mut Velocity)>,
) {
    let half_size = config.area_size / 2.0;
```

```rust
pub fn drone_ai_system(
    time: Res<SimulationTime>,
    mut query: Query<(
        Entity,
        &Position,
        &mut Velocity,
        &mut DroneState,
        &Battery,
    ), With<DroneId>>,
    targets: Query<(Entity, &Position), With<Target>>,
) {
    for (entity, pos, mut vel, mut state, battery) in query.iter_mut() {
        // Критична батарея — повертаємось
        if battery.level < 10 {
            *state = DroneState::Returning { base: Vec3::ZERO };
        }

        match &*state {
            DroneState::Idle => {
                // Шукаємо цілі поблизу
                for (target_entity, target_pos) in targets.iter() {
                    if pos.0.distance(target_pos.0) < 100.0 {
                        *state = DroneState::Attacking { target: target_entity };
                        break;
                    }
                }
            }
            DroneState::Patrolling { waypoints, current } => {
                let target = waypoints[*current];
                let direction = (target - pos.0).normalize_or_zero();
```

```rust
pub fn coordination_system(
    mut commands: Commands,
    time: Res<SimulationTime>,
    mut events: EventReader<TargetDetected>,
    mut mission_events: EventWriter<MissionAssigned>,
    idle_drones: Query<
        (Entity, &Position),
        (With<DroneId>, With<Combat>, Without<CurrentMission>)
    >,
) {
    for event in events.read() {
        // Знаходимо найближчого вільного бойового дрона
        let nearest = idle_drones.iter()
            .min_by_key(|(_, pos)| {
                pos.0.distance(event.target_position) as i32
            });

        if let Some((drone_entity, _)) = nearest {
            // Призначаємо місію
            commands.entity(drone_entity).insert(CurrentMission {
                mission_type: MissionType::Attack,
                target: Some(event.target_position),
                started_at: time.elapsed,
            });

            mission_events.send(MissionAssigned {
                drone_id: drone_entity,
                mission_type: MissionType::Attack,
                target: event.target_position,
            });
```

```rust
#[derive(Component)]
pub struct FormationMember {
    pub formation_id: Entity,
    pub offset: Vec3,
}

#[derive(Component)]
pub struct FormationLeader {
    pub formation_type: FormationType,
    pub members: Vec<Entity>,
}

pub enum FormationType {
    Line { spacing: f32 },
    Circle { radius: f32 },
    Wedge { angle: f32, spacing: f32 },
}

pub fn formation_system(
    leaders: Query<(Entity, &Position, &FormationLeader)>,
    mut members: Query<(&FormationMember, &mut Position), Without<FormationLeader>>,
) {
    for (leader_entity, leader_pos, formation) in leaders.iter() {
        for (i, &member_entity) in formation.members.iter().enumerate() {
            if let Ok((member, mut member_pos)) = members.get_mut(member_entity) {
                let offset = match &formation.formation_type {
                    FormationType::Line { spacing } => {
                        Vec3::new(0.0, -(i as f32 + 1.0) * spacing, 0.0)
                    }
                    FormationType::Circle { radius } => {
```

```rust
pub fn spawn_drone_fleet(
    mut commands: Commands,
    config: Res<SwarmConfig>,
) {
    let mut rng = rand::thread_rng();

    for i in 0..config.drone_count {
        let pos = Vec3::new(
            rng.gen_range(-config.area_size/2.0..config.area_size/2.0),
            rng.gen_range(-config.area_size/2.0..config.area_size/2.0),
            rng.gen_range(10.0..100.0),
        );

        let base_components = (
            Position(pos),
            Velocity(Vec3::ZERO),
            Acceleration(Vec3::ZERO),
            Battery { level: 100, drain_rate: 0.1 },
            Health { current: 100, max: 100 },
            DroneId(i as u64),
            TeamId(1),
            DroneState::Idle,
        );

        // Розподіл типів: 60% scout, 30% combat, 10% relay
        match i % 10 {
            0..=5 => {
                commands.spawn((base_components, Scout, Camera { range: 150.0 }));
            }
            6..=8 => {
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_movement_system() {
        let mut world = World::new();

        // Створюємо тестову сутність
        world.spawn((
            Position(Vec3::ZERO),
            Velocity(Vec3::new(10.0, 0.0, 0.0)),
            Acceleration(Vec3::ZERO),
        ));

        world.insert_resource(SimulationTime { delta: 1.0, ..default() });

        // Запускаємо систему
        let mut schedule = Schedule::default();
        schedule.add_systems(movement_system);
        schedule.run(&mut world);

        // Перевіряємо результат
        let pos = world.query::<&Position>().single(&world);
        assert_eq!(pos.0, Vec3::new(10.0, 0.0, 0.0));
    }

    #[test]
    fn test_battery_drain() {
        let mut world = World::new();
```

Performance Tips

Оптимізація ECS:

✅ Використовуйте par_iter_mut() для паралелізації
query.par_iter_mut().for_each(|...| { ... });

✅ Уникайте частих spawn/despawn
Використовуйте object pooling або Enable/Disable компоненти

✅ Batch operations
commands.spawn_batch(entities.into_iter());

✅ Query caching
Bevy автоматично кешує запити

✅ Правильний порядок компонентів
Часто використовувані — першими в запиті

❌ Уникайте:
- Changed<T> на великих запитах
- Глибоких ієрархій Parent/Child
- Багатьох дрібних систем з однаковими запитами

Benchmarks

Продуктивність bevy_ecs (Intel i7-12700):

10,000 сутностей:
• Spawn: 0.5ms
• Query iteration: 0.02ms
• Parallel iteration: 0.005ms

100,000 сутностей:
• Spawn: 5ms
• Query iteration: 0.2ms
• Parallel iteration: 0.03ms

1,000,000 сутностей:
• Spawn: 50ms
• Query iteration: 2ms
• Parallel iteration: 0.3ms

Порівняння з Actor (1000 агентів):
• ECS movement tick: 0.01ms
• Actor messages round-trip: 0.5ms
• ECS виграє в 50x для batch операцій!

Підсумок лекції

bevy_ecs основи:
• Resource — глобальні дані
• Event — комунікація між системами
• Commands — відкладені операції
• Schedule — порядок виконання
• Plugin — модульна організація

🤖 ECS для MAC:
• Компоненти: Position, Velocity, Battery, DroneState
• Маркери: Scout, Combat, Relay
• Системи: movement, battery, AI, coordination
• Події: TargetDetected, MissionAssigned

Переваги:
• Мільйони сутностей
• Паралельна обробка
• Cache-friendly
• Легке тестування

Завдання

1. Базове (2 години):
   Створіть симуляцію з 1000 дронами
   Movement + Battery системи

2. Середнє (4 години):
   Додайте систему виявлення цілей
   Events для комунікації
   Статистика в ресурсах

3. Складне (8 годин):
   Повна симуляція рою:
   • Різні типи дронів
   • AI з патрулюванням
   • Система місій
   • Формації

4. Challenge (16 годин):
   Візуалізація через Bevy (повний)
   або egui для GUI

ECS опановано!

Entity • Component • System • World

Питання?