# Programowanie Gier Komputerowych

Wykład 12: AI przeciwników i NPC

mgr inż. Hubert Staniszewski

# Czym jest AI przeciwnika lub NPC?

- W grach AI to system, który steruje zachowaniem postaci niekontrolowanych bezpośrednio przez gracza.
- Ważne aby sztuczna inteligencja tworzyła, **czytelne, przewidywalne i interesujące zachowanie**.

# AI w grach a sztuczna inteligencja jako dziedzina

## AI w grach

- ma być czytelne*,
- działa w czasie rzeczywistym,
- jest ograniczone budżetem CPU,
- często jest projektowane ręcznie,
- ma tworzyć ciekawe sytuacje gameplayowe.

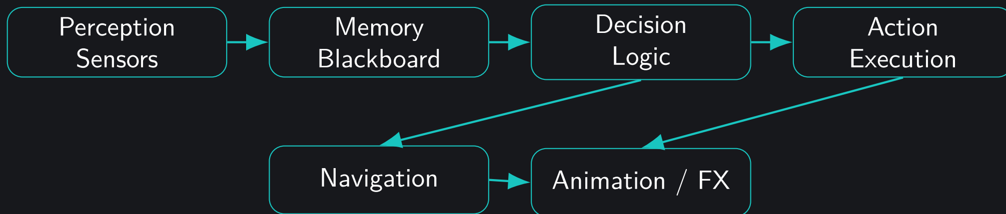
## AI akademickie / ML

- może optymalizować wynik,
- może uczyć się z danych,
- nie zawsze musi być czytelne,
- często wymaga innych narzędzi,
- nie zawsze pasuje do gameplayu.

# Cele projektowe AI przeciwników

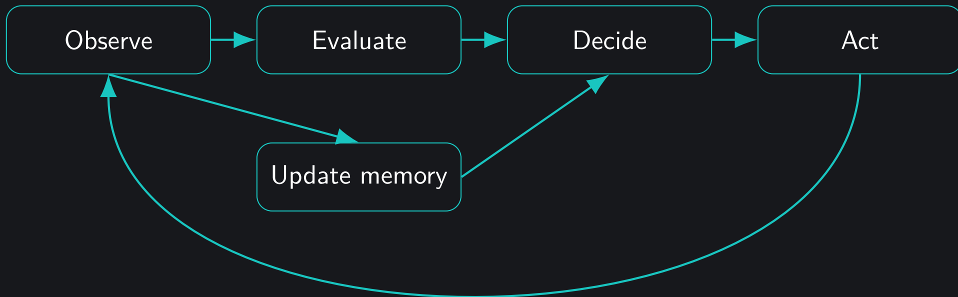
- AI powinno tworzyć sytuacje, w których gracz podejmuje decyzje.
- Przeciwnik powinien być wystarczająco sprawny, ale nie powinien sprawiać wrażenia nieuczciwego.
- Właściwe AI przeciwnika zwykle jest:
  - czytelne - gracz rozumie, co się dzieje,
  - reaktywne - odpowiada na działania gracza,
  - ograniczone - ma zasady i słabości,
  - testowalne - można odtworzyć i zdebugować zachowanie.

# Architektura agenta AI




- Agent AI zbiera informacje, zapisuje stan, wybiera decyzję i wykonuje akcję.
- Błędy AI często wynikają z pomieszania tych warstw w jednym skrypcie.

# Pętla decyzyjna AI




- Nawet proste AI powinno mieć czytelny cykl: obserwacja, ocena, decyzja, akcja.
- Dzięki temu łatwiej rozdzielić sensory, pamięć i logikę zachowania.

# Percepcja: co AI „widzi” i „słyszy”?

- AI nie powinno zwykle znać całego stanu gry bez ograniczeń.
- Typowe sensory to:
  - pole widzenia,
  - słuch lub hałas,
  - wykrywanie obrażeń,
  - bliskość obiektów,
  - pamięć ostatniej pozycji gracza.
- Ograniczona percepcja sprawia, że AI jest bardziej sprawiedliwe i czytelne.

# Percepcja: przykład z gry

# Przykład: prosta detekcja gracza

```
1  bool CanSeePlayer(const Enemy& enemy, const Player& player) {
2      Vector3 toPlayer = player.position - enemy.position;
3  
4      if (Length(toPlayer) > enemy.viewDistance)
5          return false;
6  
7      float angle = AngleBetween(enemy.forward, Normalize(toPlayer));
8      if (angle > enemy.viewAngle * 0.5f)
9          return false;
10 
11     return !Physics::RaycastBlocked(enemy.position, player.position);
12 }
```

- Detekcja zwykle łączy dystans, kąt widzenia i przeszkody w świecie.
- Dzięki temu AI nie ma „magicznej wiedzy” o graczu.

# Pamięć i blackboard

- Blackboard to współdzielony zestaw danych używany przez logikę AI.
- Może przechowywać:
  - ostatnią znaną pozycję gracza,
  - aktualny cel,
  - poziom zagrożenia,
  - stan alarmu,
  - wybraną akcję lub intencję.
- Pozwala rozdzielić sensory od podejmowania decyzji.

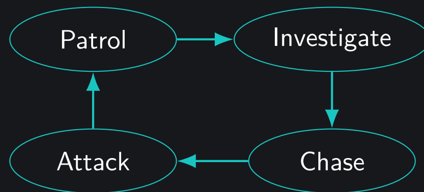
# Blackboard: przykład struktury

```
1  struct AIBlackboard {
2      bool canSeePlayer = false;
3      Vector3 lastKnownPlayerPosition;
4      float timeSincePlayerSeen = 0.0f;
5
6      int currentTargetId = -1;
7      float threatLevel = 0.0f;
8      bool isAlerted = false;
9  };
```

- Blackboard nie powinien zawierać całej logiki AI.
- To raczej miejsce przechowywania danych, na podstawie których inne moduły podejmują decyzje.

# Finite State Machine w AI

- FSM jest dobrym rozwiązaniem dla prostych i średnio złożonych zachowań.
- Stan określa, co agent robi teraz.
- Przejścia określają, kiedy agent może zmienić zachowanie.



# FSM: przykład kodu

```cpp
enum class AIState { Patrol, Investigate, Chase, Attack };

void EnemyAI::Update(float dt) {
    switch (state) {
        case AIState::Patrol:
            UpdatePatrol(dt);
            if (blackboard.canSeePlayer) state = AIState::Chase;
            break;

        case AIState::Chase:
            MoveTo(player.position);
            if (IsInAttackRange()) state = AIState::Attack;
            break;

        case AIState::Attack:
            PerformAttack();
            if (!IsInAttackRange()) state = AIState::Chase;
            break;
    }
}
```

# Kiedy FSM zaczyna przeszkadzać?

- FSM jest czytelny, dopóki liczba stanów i przejść pozostaje mała.
- Problemy pojawiają się, gdy:
  - każdy wyjątek staje się nowym stanem,
  - przejścia zaczynają zależeć od wielu warunków naraz,
  - jeden stan zaczyna wykonywać zbyt wiele logiki,
  - debugowanie wymaga śledzenia kilkunastu możliwych przejść.
- Wtedy warto rozważyć behavior tree, utility AI albo podział logiki na mniejsze warstwy.

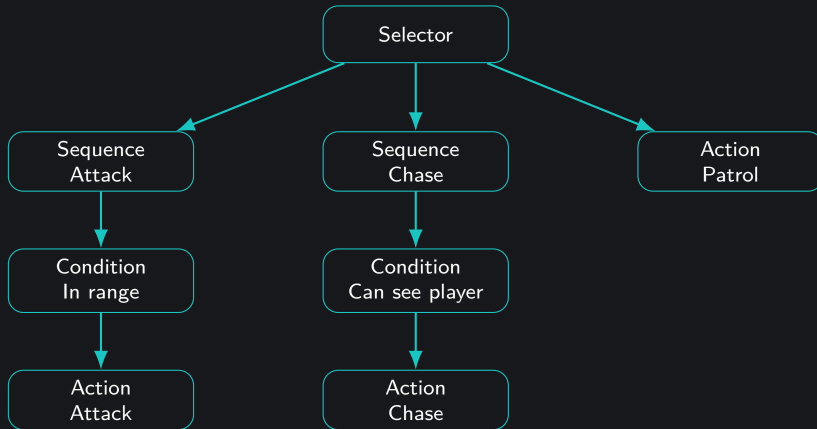
**Problem skali**

Jeżeli dodanie jednego zachowania wymaga zmiany wielu istniejących przejść, FSM przestaje skalować się dobrze.

# Behavior Tree: idea

- Behavior Tree opisuje zachowanie jako drzewo decyzji i akcji.
- Zamiast jednego dużego switcha mamy kompozycję mniejszych węzłów.
- Typowe typy węzłów:
  - **Selector** - wybiera pierwsze działające zachowanie,
  - **Sequence** - wykonuje kroki po kolei,
  - **Condition** - sprawdza warunek,
  - **Action** - wykonuje czynność.

# Behavior Tree: przykład struktury




- Drzewo najpierw próbuje atakować, potem gonić, a na końcu patrolować.
- Kolejność węzłów wpływa na priorytet zachowań.

# Selector i Sequence

## Selector

- próbuje gałąź po gałęzi,
- kończy sukcesem, gdy jedna opcja się powiedzie,
- dobrze nadaje się do priorytetów zachowań.

## Sequence

- wykonuje gałąź po gałęzi,
- kończy porażką, gdy jeden krok się nie powiedzie,
- dobrze nadaje się do procedur i planów.

- Behavior tree jest czytelne tylko wtedy, gdy projektujemy proste, małe węzły.

# Behavior Tree: uproszczony pseudokod

```
Status Selector::Tick() {
    for (Node* child : children) {
        Status result = child->Tick();

        if (result == Status::Success)
            return Status::Success;

        if (result == Status::Running)
            return Status::Running;
    }

    return Status::Failure;
}
```

- Węzły zwracają zwykle: `Success`, `Failure` albo `Running`.
- To pozwala zachowaniom trwać przez wiele klatek.

# Utility AI

- Utility AI wybiera akcję na podstawie wyniku punktowego.
- Każda możliwa akcja dostaje score zależny od kontekstu.
- Przykładowe czynniki:
  - zdrowie agenta,
  - dystans do gracza,
  - zagrożenie,
  - dostępność osłony,
  - liczba sojuszników,
  - cooldowny akcji.
- AI wybiera akcję o najwyższej użyteczności, zamiast przechodzić po sztywnych stanach.

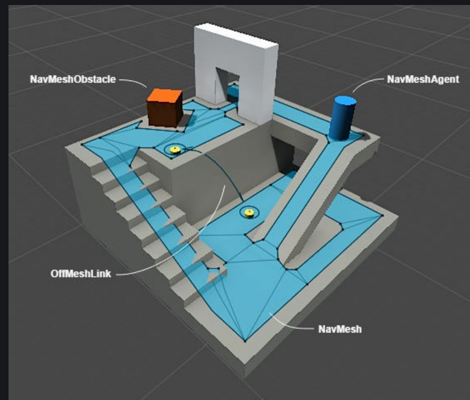
# Utility AI: przykład oceny akcji

| Akcja | Kiedy rośnie score? | Ryzyko |
|---|---|---|
| Atak | gracz blisko, agent ma HP | spam ataku bez przerwy |
| Ucieczka | niskie HP, brak wsparcia | AI może wyglądać tchórzliwie |
| Szukaj osłony | silny ostrzał, osłona blisko | AI może zbyt często przerywać walkę |
| Wezwij pomoc | przewaga gracza, sojusznicy w pobliżu | zbyt trudne starcia |

- Utility AI dobrze sprawdza się, gdy decyzja zależy od wielu miękkich czynników.
- Wymaga jednak dobrego debugowania score'ów.

# Nawigacja i pathfinding

- AI musi nie tylko zdecydować, co zrobić, ale też dotrzeć do miejsca wykonania akcji.
- Najczęściej spotykane rozwiązania:
  - graf waypointów,
  - siatka i A*,
  - navmesh,
  - steering behaviours,
  - lokalne unikanie przeszkód.
- Pathfinding odpowiada za trasę, a steering za lokalny ruch i korekty.



# NavMesh vs grid

## Grid

- łatwy do zrozumienia,
- dobry w grach taktycznych,
- naturalny dla tilemap,
- może być kosztowny przy dużej rozdzielczości.

## NavMesh

- dobry dla światów 3D,
- opisuje powierzchnie dostępne dla ruchu,
- ułatwia naturalne ścieżki,
- wymaga poprawnego bake'u i testów.

# AI w walce

- AI przeciwnika w walce powinno tworzyć decyzje dla gracza, a nie tylko zadawać obrażenia.
- Typowe elementy combat AI:
  - wybór dystansu,
  - decyzja: atak, obrona, unik, cofnięcie,
  - cooldowny i okna akcji,
  - telegraphing zamiaru,
  - reakcje na zachowanie gracza,
  - ograniczanie naraz aktywnych przeciwników.
- Właściwe combat AI często celowo nie wykorzystuje wszystkich informacji i możliwości.

# Fair AI: inteligentne, ale uczciwe

- AI może być technicznie skuteczne, ale jednocześnie frustrujące.
- Gracz zwykle akceptuje porażkę, jeśli rozumie:
  - co przeciwnik zrobił,
  - kiedy pojawił się sygnał ostrzegawczy,
  - jak można było odpowiedzieć,
  - czy przeciwnik działał według reguł gry.
- AI powinno mieć ograniczenia: czas reakcji, błąd, zasięg percepcji i przewidywalne wzorce.

# Telegraphing i czytelność zamiaru

- Telegraphing to komunikowanie graczowi, że przeciwnik zaraz wykona akcję.
- Może przyjmować formę:
  - animacji przygotowania,
  - dźwięku,
  - zmiany postawy,
  - efektu wizualnego,
  - krótkiego opóźnienia przed atakiem.
- Bez telegraphingu nawet poprawne AI może wydawać się nieuczciwe.

# AI bossów: fazy i wzorce

- Boss AI często łączy state machine, behavior tree i skrypty faz.
- Elementy AI Bossów:
  - fazy zależne od HP,
  - zestawy ataków dla każdej fazy,
  - ograniczenia cooldownów,
  - sekwencje specjalne,
  - momenty odpoczynku i podatności na kontrę.
- Fazy bossów są narzędziem kontroli rytmu walki.

# AI grupowe i koordynacja

- Gdy wielu przeciwników atakuje naraz, problemem nie jest tylko AI pojedynczego agenta.
- Potrzebna jest koordynacja pomiędzy agentami wewnątrz grupy.
- Bez koordynacji walka może stać się chaotyczna lub niesprawiedliwa.

# Director AI i kontrola intensywności

- Niektóre gry używają systemu sterującego daną sceną sceną.
- Director może decydować o:
  - liczbie przeciwników,
  - momencie pojawienia się fali,
  - intensywności presji,
  - nagłych przerwach,
  - rozmieszczeniu zasobów lub zagrożeń.
- Taki system nie zastępuje AI agentów, ale zarządza kontekstem, w którym działają.

# Update AI i wydajność

- AI bywa kosztowne, zwłaszcza gdy wielu agentów podejmuje decyzje jednocześnie.
- Ważne optymalizacje:
  - różna częstotliwość update'u dla bliskich i dalekich agentów,
  - LOD AI,
  - cache wyników percepcji,
  - ograniczenie kosztownych raycastów,
  - rozłożenie obliczeń na wiele klatek.
- AI nie musi podejmować pełnej decyzji w każdej klatce.

# Przykład: rzadszy update decyzji

```
1  void EnemyAI::Update(float dt) {
2      perception.UpdateFastSensors(dt);
3
4      decisionTimer -= dt;
5      if (decisionTimer <= 0.0f) {
6          blackboard = perception.BuildBlackboard();
7          currentAction = decisionSystem.ChooseAction(blackboard);
8          decisionTimer = decisionInterval;
9      }
10
11     currentAction.Update(dt);
12 }
```

- Percepcja i wykonanie akcji mogą działać częściej niż pełna decyzja.
- To zmniejsza koszt CPU bez dużej utraty jakości zachowania.

# Data-driven AI

- AI często warto konfigurować danymi, a nie twardo kodować wszystkie parametry.
- Do danych można przenieść:
  - zasięg widzenia,
  - reakcję na hałas,
  - priorytety zachowań,
  - cooldowny akcji,
  - zestawy ataków,
  - parametry agresji i ostrożności.
- Kod powinien wykonywać logikę, a dane powinny opisywać warianty zachowania.

# Data-driven AI: przykład danych

```cpp
struct EnemyAIConfig {
    float viewDistance;
    float viewAngle;
    float hearingRadius;

    float attackRange;
    float attackCooldown;

    float aggression;
    float caution;
    float decisionInterval;
};
```

- Ta sama logika AI może działać inaczej dla strażnika, potwora i bossa.
- Różnice wynikają z konfiguracji, a nie z kopiowania kodu.

# Debugowanie AI

- AI musi być obserwowalne, bo wiele błędów wynika z ukrytego stanu.
- Przydatne narzędzia:
  - podgląd aktualnego stanu lub węzła BT,
  - debug draw pola widzenia,
  - log decyzji i score'ów,
  - wizualizacja ścieżki,
  - podgląd blackboarda.
- Debugowanie AI polega na sprawdzeniu, co agent **wiedział**, co **wybrał** i dlaczego.

# Debugowanie AI

# Testowanie AI

- AI warto testować nie tylko w pełnej rozgrywce, ale też w izolowanych scenariuszach.
- Przykładowe testy:
  - agent widzi gracza tylko w polu widzenia,
  - agent traci gracza za przeszkodą,
  - agent wybiera atak tylko w zasięgu,
  - agent nie blokuje się na przeszkodzie,
  - grupa przeciwników nie atakuje jednocześnie ponad limit.
- Testy AI powinny obejmować zarówno logikę decyzji, jak i zachowanie w świecie gry.

# Typowe błędy AI przeciwników

## Błędy techniczne

- AI widzi przez ściany,
- agent blokuje się na ścieżce,
- decyzje zmieniają się zbyt często,
- wiele agentów wybiera tę samą pozycję,
- stan nie wraca do neutralnego zachowania.

## Błędy projektowe

- AI jest zbyt idealne,
- brak telegraphingu,
- przeciwnik nie ma słabości,
- trudność wynika z oszustwa,
- zachowanie nie wspiera głównej pętli gry.

# Podsumowanie

- AI przeciwników i NPC powinno być projektowane jako system: percepcja, pamięć, decyzja i wykonanie.
- FSM dobrze działa dla prostych zachowań, behavior tree dla kompozycji działań, a utility AI dla decyzji opartych na wielu czynnikach.
- Nawigacja, telegraphing, ograniczenia i debugowanie są równie ważne jak sama logika decyzji.
- Najważniejsza zasada: AI ma tworzyć dobre sytuacje dla gracza, a nie tylko grać optymalnie.