Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

# Obliczanie najkrótszych ścieżek z eliminacją kolizji

Adam Bobowski
Nr indeksu: 208786

Praca inżynierska napisana
pod kierunkiem
dr. Macieja Gębali

Politechnika
Wrocławska
Wrocław 2016

# Spis treści

# Wstęp

W niniejszej pracy rozważany jest problem wyznaczania najkrótszych ścieżek w przestrzeni trójwymiarowej z eliminacją kolizji pomiędzy poruszającymi się jednostkami. Dokładniej, skupiono się na obecnie najpopularniejszym rozwiązaniu w przemyśle gier komputerowych i symulacji, strukturze *Navigation Mesh*. Do rozwiązania problemu eliminacji kolizji postanowiono przetestować skuteczność algorytmów podejmujących decyzję na podstawie aktualnego, lokalnego otoczenia.

Celem pracy jest zaprojektowanie i implementacja systemu o następującej funkcjonalności:

- Wyliczanie najkrótszej ścieżki pomiędzy dwoma punktami w przestrzeni.
- Dynamiczne modyfikowanie ścieżki ruchu na podstawie zmieniającego się otoczenia.

System jest tworzony w oparciu o technologię dostępną w silniku *Unity*, oraz zakłada jego integrację z innymi projektami tworzonymi w tej technologii.

Istnieje kilka dostępnych rozwiązań o zbliżonej funkcjonalności: *NavMesh* dostępny bezpośrednio w silniku *Unity*, *Recast & Detour*, oraz wiele rozwiązań własnych stworzonych na potrzeby innych silników do gier komputerowych takich jak *Cry Engine*, *Unreal Engine* czy *Source Engine*. Cechą odróżniającą projektowany system od podanych rozwiązań jest eksperymentalne podejście do problemu rozwiązywania kolizji oraz lokalność ich rozwiązań. Warto zaznaczyć, że aktualnie architektura systemu *Recast & Detour* stała się standardem rozwiązań w generowaniu i operowaniu na strukturze *Navigation Mesh*.

Praca składa się z czterech rozdziałów. Pierwszy tworzą dwie główne części, w tym pierwsza skupia się na problemie najkrótszej ścieżki tylko dla przypadku, gdy po terenie porusza się jedna jednostka, czyli nie występuje problem kolizji poruszających się jednostek. Omówiono tu także proces budowania struktury *Navigation Mesh* na podstawie istniejącej siatki terenu, oraz sam algorytm wyznaczania ścieżek. W drugiej części omówiono ogólną zasadę działania algorytmów podejmujących decyzję na podstawie lokalnego otoczenia. Następnie pokazano ich zastosowanie w eliminowaniu kolizji. Przybliżono także algorytmy stadne oraz omówiono inne strategie działania.

Rozdział drugi składa się ze szkicu systemu oraz szczegółów dotyczących zastosowanej technologii. Przedstawiono również pewne wskazówki optymalizacyjne oraz uwagi, o których warto pamiętać przy implementacji i użytkowaniu systemu.

W trzecim rozdziale zawarto wnioski z analizy przykładowych scenariuszy oraz ogólne propozycje usprawnień w systemie.

W ostatnim rozdziale przedstawiono proces integracji systemu z własnym projektem w silniku *Unity* oraz instrukcję instalacji i obsługi przykładowego projektu zamieszczonego jako załącznik do pracy.

# Analiza problemu

Omawiane w tej pracy zagadnienie można podzielić na, dwie praktycznie niezależne części: wyznaczanie najkrótszej ścieżki oraz zastosowanie algorytmów lokalnych w celu eliminacji i rozwiązywania konfliktów. Ich niezależność polega na tym, że żadna z wymienionych części nie wpływa na obliczenia przeprowadzone przez drugą, jedynie modyfikuje aktualną prędkość i kierunek ruchu jednostki.

Proces obliczania ścieżki i eliminacji kolizji zostanie przedstawiony jako złożenie następujących problemów:

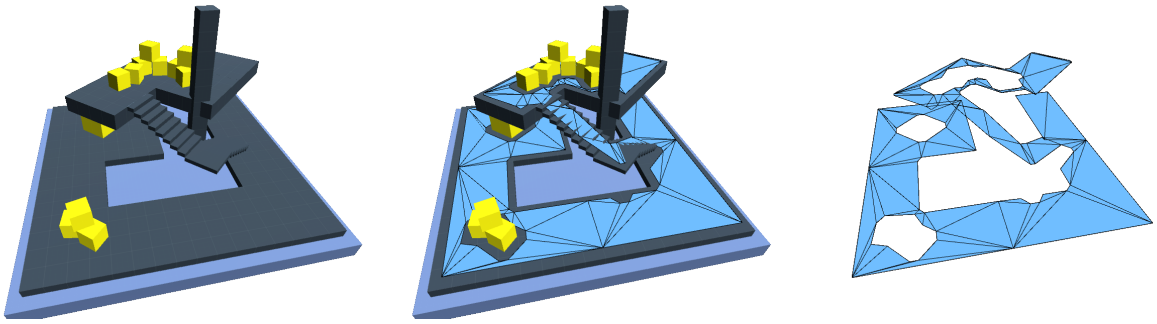
- Budowa grafu na postawie siatki terenu.
- Znalezienie trójkąta, w środku którego leży dany punkt.
- Wyznaczenie najkrótszej ścieżki w grafie.
- Utworzenie listy portali, pomiędzy którymi będzie przebiegała ścieżka.
- Wyznaczenie najkrótszej ścieżki prowadzącej przez portale.
- Modyfikacja aktualnej prędkości i kierunku ruchu na podstawie lokalnego otoczenia.

Część pierwszą w całości poświęcono budowie struktury *Navigation Mesh*. Przedstawiono strukturę siatki w grafice trójwymiarowej, która jest kluczową strukturą i na której opierają się obliczenia najkrótszej ścieżki. Omówiono ogólną ideę rozwiązania i na jej podstawie wyodrębniono zbiór podproblemów. Zawarto dokładny opis i analizę algorytmów, które rozwiązują poruszone wcześniej problemy.

W drugiej części przedstawiono ogólną ideę algorytmów podejmujących decyzję na podstawie lokalnego otoczenia. Omówiono podstawowe algorytmy stadne oraz inne strategie, które umożliwiają eliminowanie kolizji i redukcję konfliktów.

W części trzeciej zaprezentowano ideę integracji obu poprzednich modułów. Zaznaczono niezależność obu części i przeprowadzanych obliczeń. Zakończono przedstawieniem potencjalnych wad i zalet proponowanego rozwiązania.

## 2.1 Navigation Mesh

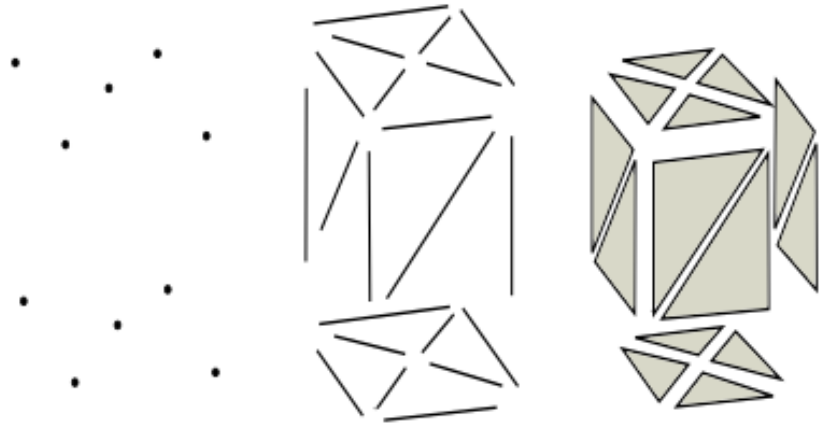


Rysunek 2.1: Przykładowy poziom z gry, wygenerowana siatka Navigation Mesh, sama siatka

Ideą struktury Naviagtion Mesh jest przedstawienie dostępnego terenu za pomocą siatki i na jej podstawie wykonywanie działań. Oczywiście w praktyce siatka nie jest widoczna dla graczy, tutaj jednak została narysowana do celów demonstracyjnych, by zaprezentować jej konstrukcję.

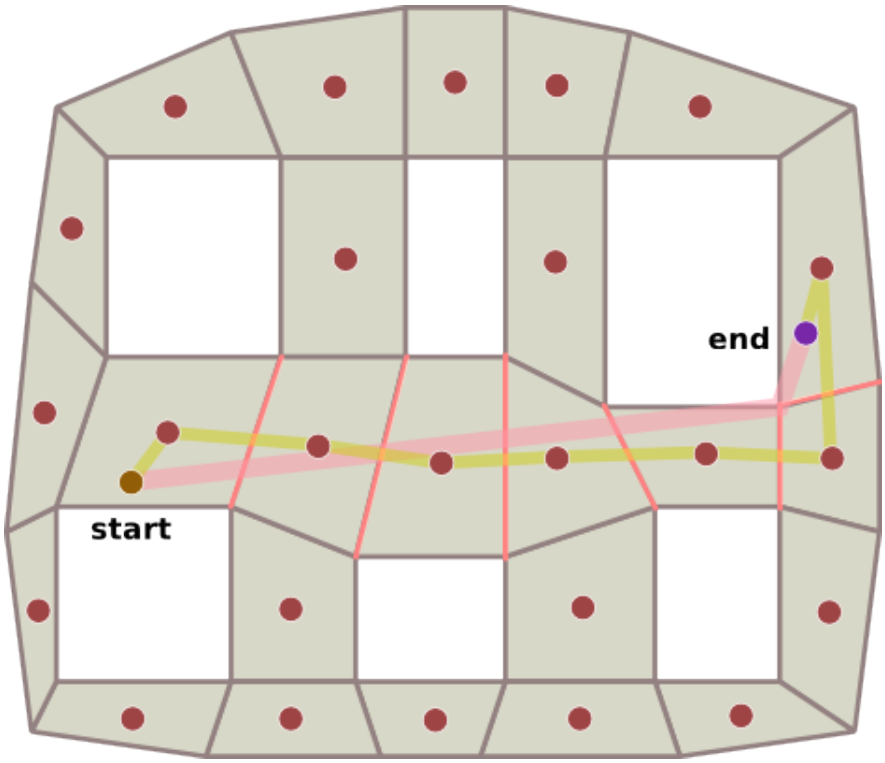
### 2.1.1 Siatka

Siatka jest jedną z podstawowych struktur w grafice trójwymiarowej. W najprostszym przypadku składa się ona z listy wierzchołków oraz listy ścian. Wierzchołek jest reprezentowany za pomocą wektora o trzech składowych, a lista ścian to najczęściej indeksy z listy wierzchołków. Sąsiednimi ściankami nazywamy te, które posiadają wspólną krawędź. Na rysunku 2.2 przedstawiono przykładową budowę prostopadłościanu. Najpierw zdefiniowano położenie wierzchołków, a później połączono je w odpowiedniej kolejności w ścianki.



Rysunek 2.2: Wierzchołki, krawędzie, ściany w strukturze siatki

Należy zauważyć, że w najprostszym przypadku ściankę tworzą trzy wierzchołki, dlatego pojęcie trójkąta należy traktować tutaj jako po prostu ściankę.




Rysunek 2.3: Wierzchołki, krawędzie, ściany w strukturze siatki

### 2.1.2 Ogólny algorytm najkrótszej ścieżki

Idea *Navigation Mesh* polega na przedstawieniu dostępnego terenu dla jednostki za pomocą siatki. Następnie, znając punkt początkowy i końcowy, należy wyszukać ścianki, w których te punkty się znajdują. Traktując siatkę jako graf, wyszukuje się najkrótszą ścieżkę (np. za pomocą algorytmu A*). Na podstawie ścieżki w grafie należy wyznaczyć listę krawędzi wspólnych dla każdych dwóch kolejnych ścianek. Ostatnim krokiem jest wyznaczenie ścieżki pomiędzy kolejnymi portalami.

Tak więc problem wyznaczenia ścieżki w przestrzeni został zredukowany do serii prostych i dobrze znanych problemów.

**Pseudokod 2.1:** Algorytm najkrótszej ścieżki

**Input:** Points $s$, $t$, mesh $m$
**Output:** *path*
1 $face\_s \leftarrow$ `GetFace`($s$, $m$);
2 $face\_t \leftarrow$ `GetFace`($t$, $m$);
3 $graph \leftarrow$ `BuildGraph`($m$);
4 $graph\_path \leftarrow$ `AStarSearch`($face\_s$, $face\_t$, $graph$);
5 $portals \leftarrow$ `CalculatePortals`($graph\_path$, $m$);
6 $path \leftarrow$ `Funnel`($portals$);

### 2.1.3 Wyznaczenie ściany na postawie punktu

Do wyznaczenia ściany na podstawie zadanego punktu można użyć kilku metod. Jeżeli siatka jest skomplikowana najlepszym rozwiązaniem może okazać się wykorzystanie fizyki danego silnika (np. Raycasting). Przy bardzo prostych i regularnych siatkach można zastosować algorytm heurystyczny *Walk & Check*, który próbuje przejść przez siatkę z jednej ścianki na drugą kierując się w stronę zadanego punktu. Jednak to

rozwiązanie w praktyce okazuje się zbyt ograniczone i bardzo rzadko stosowane.

**Pseudokod 2.2:** Walk & Check jako *GetFace*

```
Input: Point p, mesh m
Output: F
F ← any face from m ;
while p not in F do
    F ← neighbour of F closest to p;
```

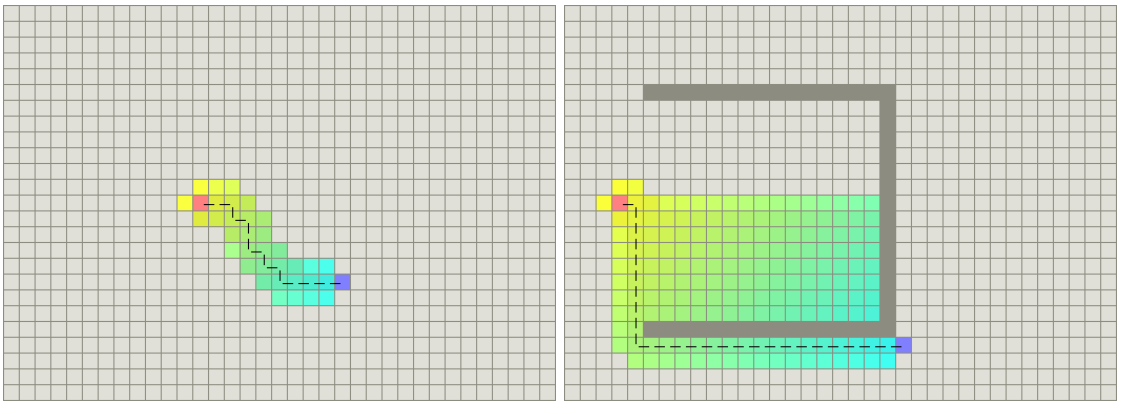
### 2.1.4 Budowa grafu na podstawie siatki

W budowanym grafie każdy wierzchołek odpowiada jednej ściance, a krawędzie grafu łączą te ścianki, które maja wspólną krawędź w siatce. To czy graf jest grafem ważonym i z jaką wagą na krawędziach, w dużej mierze zależy od samej struktury siatki. Dla większości standardowych zastosowań można za wagę krawędzi przyjąć odległość pomiędzy środkami sąsiednich ścian stosując podstawową metrykę euklidesową.

Zbudowanie grafu polega na wyznaczeniu sąsiadów dla każdej ze ścian. Sam proces jest zwykłym przeszukiwaniem i ogólna złożoność wynosi $O(n^2)$. Następnie, aby móc stosować heurystyki do przeszukiwania grafu, należy obliczyć współrzędne wierzchołków oraz odległości pomiędzy sąsiednimi wierzchołkami. Dla wierzchołków można przyjąć średnią ze współrzędnych wierzchołków tworzących ścianę, a odległości to standardowa odległość euklidesowa.

### 2.1.5 Najkrótsza ścieżka w grafie - algorytm A*

Do obliczania najkrótszej ścieżki w grafie można użyć wielu algorytmów. Dla niektórych rodzajów siatek nawet przeszukiwanie grafu wszerz daje bardzo dobre wyniki. Tutaj jednak można wykorzystać zmodyfikowany algorytm Dijkstry, ponieważ przy budowie grafu obliczone zostały odległości pomiędzy sąsiednimi wierzchołkami oraz ich współrzędne. Rozszerzenie tego algorytmu o heurystykę na podstawie odległości umożliwia szybsze przeszukiwanie grafu. Należy jednak pamiętać, że jest to rozwiązanie heurystyczne. Oznacza to, że przy zastosowaniu złej metryki wyniki mogą daleko odbiegać od optymalnych.



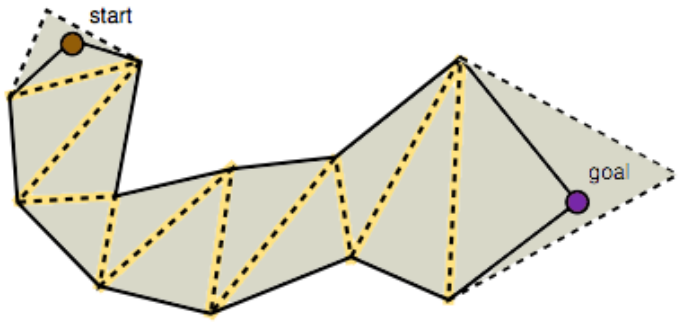
Rysunek 2.4: Przykład działania algorytmu A*

**Pseudokod 2.3:** Algorytm A*

```
Input: s, t, point, graph
Output: path
foreach node v in graph do
    prev[v] ← None;
    dist[v] ← ∞;
Q.put(s, 0);
prev[s] ← s;
dist[s] ← 0;
while Q ≠ ∅ do
    v ← Q.get_min()
    if v = t then
        break;
    foreach neighbour u of v do
        if dist[v] + graph.cost(v, u) < dist[u] then
            dist[u] ← dist[v] + graph.cost(v, u);
            prev[u] ← v;
            Q.put(u, dist[u] + heuristic(point, u));
return reverse prev from t
```

### 2.1.6 Portale

Na podstawie ścieżki w grafie można wyznaczyć listę portali, czyli krawędzi które są wspólne dla kolejnych dwóch ścian, przez które końcowa ścieżka będzie przechodziła. Portalem jest więc para punktów w przestrzeni, pomiędzy którymi będzie przechodzić ścieżka.



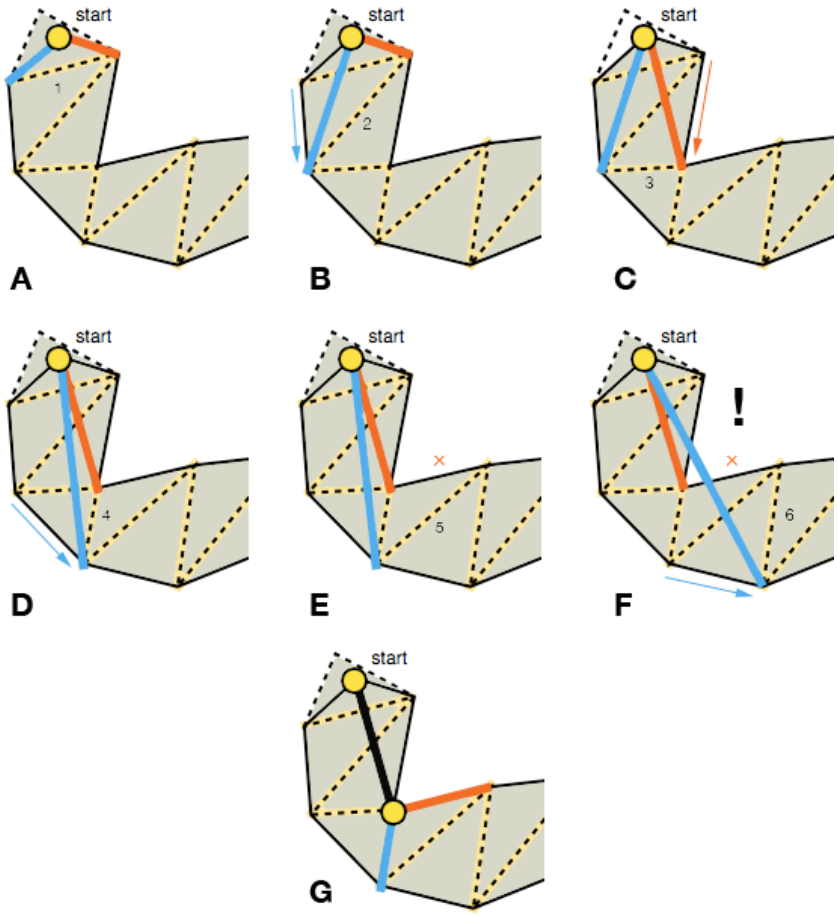

Rysunek 2.5: Wyznaczanie portali na podstawie ścieżki

**Pseudokod 2.4:** Obliczanie portali

```
Input: s, t, Path, mesh
Output: portals
portals[0] ← (s, s);
portals[−1] ← (t, t);
for i from 1 to |Path| do
    a ← Path[i − 1];
    b ← Path[i];
    portals[i] ← GetSharedEdge(a, b);
return portals;
```

### 2.1.7 Wygładzanie ścieżki pomiędzy portalami

Posiadając listę portali ostatnim krokiem jest poprowadzenie możliwie prostej ścieżki pomiędzy nimi. Z faktu, że najkrótsza droga pomiędzy dwoma punktami jest linią prostą, należy zminimalizować ilość zagięć ścieżki pomiędzy portalami. Bardzo prostym i zarazem szybkim algorytmem rozwiązującym ten problem jest algorytm lejka (*Funnel algorithm*). Zasada opiera się na stopniowym zawężaniu pola widzenia z danego punktu na każdym z portali. Gdy kolejny portal jest już całkowicie zasłonięty dodaje się punkt do ścieżki i od tego miejsca rozpoczyna ponownie procedurę, aż do uzyskania ścieżki do punktu docelowego.




Rysunek 2.6: Przykład działania algorytmu lejka

**Pseudokod 2.5:** Funnel algorithm

```
   Input: Portals
   Output: path
 1 apex ← portals[0];
 2 left ← right ← apex;
 3 path.add(apex);
 4 foreach l, r in Portals do
 5     if r is between left and right then
 6         right ← r;
 7     else
 8         path.add(left);
 9         apex ← left;
10         left ← right ← apex;
11         continue foreach loop from last l, r pair;
12     if l is between left and right then
13         left ← l;
14     else
15         path.add(right);
16         apex ← right;
17         left ← right ← apex;
18         continue foreach loop from last l, r pair;
19 return path;
```

## 2.2 Algorytmy lokalnego otoczenia

Jednym z głównych problemów w nawigacji jest eliminacja kolizji pomiędzy uczestnikami ruchu. Aby optymalnie rozwiązać ten problem i zapewnić możliwie najkrótszą ścieżkę wszystkim jednostkom, każda z nich musiałaby posiadać informacje na temat każdego innego uczestnika ruchu i ich aktualnych ścieżek. Takie rozwiązanie jest bardzo kosztowne, dlatego można skupić się na lokalnej eliminacji kolizji, a wymóg optymalnej ścieżki każdej z jednostek zastąpić "możliwie optymalną"ścieżką. Zaletą rozwiązania, gdzie każda z jednostek podejmuje decyzję samodzielnie jest fakt, że niepotrzebny jest jakikolwiek nadrzędny nadzorca, albo synchronizowanie jednostek. Takie rozwiązanie wiązałoby się z dużą stratą wydajności.

### 2.2.1 Ogólna zasada działania algorytmów

Szkielet każdego z omawianych algorytmów jest praktycznie taki sam. Dokładne różnice pomiędzy konkretnymi algorytmami wynikają tylko z różnych celów jakim mają służyć modyfikacje zachowań. Podstawowymi zmiennymi w tych algorytmach są: promień lokalnego otoczenia - *Radius* oraz kąt widzenia - *Angle* jednostki. Na podstawie tych wartości jednostka *(boid)* niejako decyduje czy ma w ogóle modyfikować swoje zachowanie względem otaczających ją jednostek.

Można bardzo łatwo zauważyć tutaj analogię do zachowania ludzkiego. Idąc przez korytarz człowiek będzie reagował tylko na osoby nadchodzące z naprzeciwka, podczas gdy osoby idące za nim niejako w ogóle nie wpływają na jego kierunek i prędkość ruchu. Właśnie takie analogie są bardzo pomocne w zrozumieniu i projektowaniu zachowań na podstawie lokalnego otoczenia.

**Pseudokod 2.6:** Ogólny algorytm

**Input:** *Radius*, *Angle*, *Boids*
**Output:**
1 **foreach** $b \in Boids$ **do**
2  **if** `Distance`$(self, b) \leqslant Radius$ and `Angle`$(self, b) \leqslant Angle$ **then**
3  `ModifyBehaviour`$(self, b)$;

### 2.2.2 Algorytmy stada

Jednymi z najczęściej występujących algorytmów wykorzystujących zasadę podejmowania decyzji na podstawie lokalnego otoczenia są algorytmy stada. Są to trzy niezależne zachowania modyfikujące aktualną prędkość i kierunek ruchu jednostki. Są to:

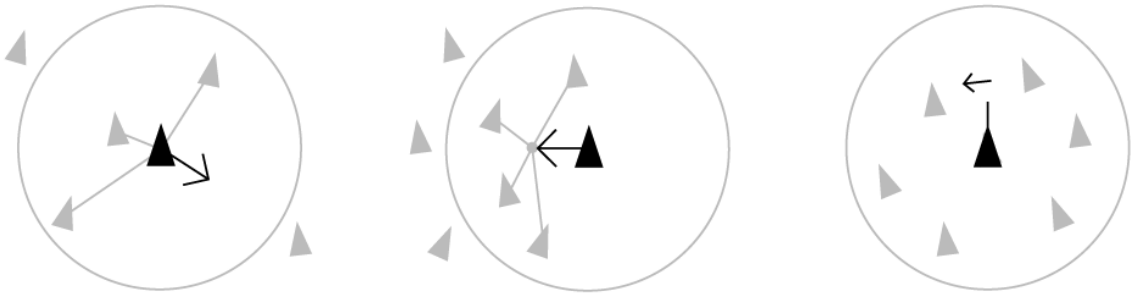
- Rozdzielność
- Spójność
- Wyrównanie

Każde z tych zachowań działa z określoną siłą, od której zależy faktyczne zachowanie jednostek. Współczynniki przy konkretnych siłach zależą od faktycznych potrzeb i konkretnego celu. Najczęściej również ich dobór jest przeprowadzany drogą eksperymentalną ponieważ dokładna analiza teoretyczna dla przypadków z tysiącami poruszających się jednostek może okazać się po prostu nieproporcjonalnie nieopłacalna.

Warto zaznaczyć, że dokładna implementacja algorytmów jest w dużym stopniu zależna od faktycznych potrzeb programisty. Oznacza to, że najbardziej podstawowa implementacja może korzystać z jednakowych współczynników modyfikujących siłę. Natomiast dla przypadków, gdzie potrzebne są bardziej złożone zachowania, każda z trzech sił może posiadać niezależne współczynniki takie jak promień i kąt. Współczynniki nie muszą być stałe, ich wartość może zależeć od bezpośredniej odległości pomiędzy jednostkami.

W ogólności algorytmy stada modyfikują wektor prędkości w następujący sposób:

$$V = V + f_r V_r + f_s V_s + f_w V_w$$



Rysunek 2.7: Rozdzielność, spójność i wyrównanie

**Rozdzielność** modyfikuje aktualną prędkość jednostki tak, aby nie zbliżała się za bardzo do innych. Dzięki takiemu zachowaniu można w prosty sposób osiągnąć regularne i bezpieczne odległości pomiędzy poruszającymi się jednostkami, jak również zapobiec tworzeniu lokalnych zgromadzeń.

**Pseudokod 2.7:** Rozdzielność

```
Input: Radius, Angle, Boids
Output: V
1 V ← (0, 0, 0);
2 foreach b ∈ Boids do
3     if Distance(self, b) ⩽ Radius and Angle(self, b) ⩽ Angle then
4         rel ← Position(b) − Position(self);
5         V = V − Normalized(rel);
6 return Normalized(V);
```

**Spójność** modyfikuje aktualną prędkość tak, aby jednostki dążyły do pozostania przy sobie (w stadzie). Dzięki takiemu zachowaniu, dla pewnych przypadków, grupę jednostek można traktować jako pojedynczą, większą, jednostkę. Nie będzie potrzebne wyznaczanie ścieżki dla każdej jednostki z grupy osobno, ale wystarczy wyznaczyć ścieżkę od środka masy grupy do punktu docelowego i każdej z jednostek przekazać dodatkowo wektor wyrównania *(offset)*.

**Pseudokod 2.8:** Spójność

```
Input: Radius, Angle, Boids
Output: V
1 V ← (0, 0, 0);
2 foreach b ∈ Boids do
3     if Distance(self, b) ⩽ Radius and Angle(self, b) ⩽ Angle then
4         V = V + Position(b);
5 V = V/Count(Boids);
6 V = V − Position(self);
7 return Normalized(V);
```

**Wyrównanie** modyfikuje aktualną prędkość tak, aby jednostki kierowały się w tym samym kierunku. Dzięki takiemu zachowaniu można w prosty sposób kierować po krętych ścieżkach grupy jednostek, które

będą dążyły do tego, aby wykonywać zakręty możliwie łagodnie i bez ostrych zmian prędkości.

**Pseudokod 2.9:** Wyrównanie

```
Input: Radius, Angle, Boids
Output: V
V ← (0, 0, 0);
foreach b ∈ Boids do
    if Distance(self, b) ⩽ Radius and Angle(self, b) ⩽ Angle then
        V = V + Velocity(b);
V = Normalized(V) − Velocity(self);
return V;
```

### 2.2.3 Inne strategie

**Przepuść po prawej.** Zasad i reguł jakie można użyć w algorytmach lokalnych jest wiele. Jednym z bardzo prostych, ale też bardzo efektywnych zachowań jest np. przepuszczanie na skrzyżowaniu jednostek po prawej stronie.

**Pseudokod 2.10:** Przepuść po prawej

```
Input: Radius, Angle, Boids
Output:
foreach b ∈ Boids do
    if Distance(self, b) ⩽ Radius and Angle(self, b) ⩽ Angle then
        if OnRightSide(self, b) then
            Stop(self)
```

**Kodeks ruchu drogowego.** Łatwo zauważyć analogię omawianych tutaj algorytmów do kodeksu ruchu drogowego. O ile sam kodeks jest dość obszerny i niekiedy skomplikowany, to same zasady sprowadzają się właściwie do bardzo prostych reguł. Najczęściej są to ograniczenia prędkości, z których wynikają takie wartości jak minimalna odległość od pojazdu. Drogi podporządkowane i główne to nic innego jak priorytet jednostki, który działa identycznie jak zasada przepuszczania po prawej stronie.

Należy jednak pamiętać, że istnieją przypadki, których występowanie jest tak rzadkie, że często się o nim zapomina. Przykładem może być scenariusz, gdy do skrzyżowania z czterech różnych stron, w tym samym momencie podjeżdżają cztery samochody. Kodeks drogowy jasno opisuje rozwiązanie takiego przypadku, natomiast podany wcześniej algorytm doprowadziłby do zaklinowania jednostek.

## 2.3 Integracja

Integracja systemu obliczania najkrótszej ścieżki z systemem eliminacji kolizji na podstawie lokalnego otoczenia jest stosunkowo prosta. Opiera się ona na założeniu, że każda z jednostek wie dokąd chce się przemieścić, czyli na podstawie *Navigation Mesh* oblicza ogólny kierunek do kolejnego węzła ścieżki, gdzie algorytmy lokalne dodają lekkie odchylenia.

Warto pamiętać o niezależności obu systemów. Oznacza to tyle, że najkrótsza ścieżka w żaden sposób nie uwzględnia innych poruszających się obiektów, a jedyne co faktycznie oblicza to kierunek ruchu. Kierunek ten może równie dobrze być stałą wartością, albo w przypadku symulacji może to być pozycja kursora myszki. Natomiast zachowania lokalne opierają się tylko na informacji o sąsiednich jednostkach i są agnostyczne względem informacji o samym terenie. W ten sposób oba systemy można stosować niezależnie, chociaż ich połączenie pozwala na rozwiązanie problemów, które nie są adresowane przed drugi z modułów.

Podstawową zaletą sprzężenia obu modułów w jeden jest rozwiązanie dwóch problemów na raz. Oblicza się najkrótszą ścieżkę w sensie globalnym oraz omija kolizję lokalnie, żaden z systemów osobno nie rozwiązuje obu problemów dostatecznie dobrze i dokładnie.

Podstawową wadą takiego rozwiązania są niedeterministyczne rozwiązania na podstawie lokalnego otoczenia. Bardzo często wprowadzają one odchylenia, które mogę zmienić tor ruchu jednostki na kolizyjny z istniejącymi statycznymi obiektami. Rozwiązaniem tego jest wprowadzenie siły odpychającej jednostki od ścian i przeszkód, ale jak łatwo zauważyć dodaje to kolejny poziom skomplikowania i konieczności synchronizacji większej ilości współczynników.

# Szkic systemu

W tym rozdziale przedstawiony jest szkic systemu w ujęciu programistycznym. Zamieszczono tutaj również pewne wskazówki optymalizacyjne oraz uwagi, o których warto pamiętać przy implementacji i korzystaniu z systemu.

Podobnie jak w części teoretycznej, system składa się z dwóch, praktycznie niezależnych części. Oznacza to tylko tyle, że nie jest konieczne korzystanie z obu modułów na raz, czyli w zależności od potrzeb można z któregoś modułu zrezygnować.

## 3.1 Silnik Unity

Do implementacji postanowiono wykorzystać technologię udostępnioną przez silnik *Unity*. Jest to zintegrowane środowisko do tworzenia trójwymiarowych lub dwuwymiarowych gier komputerowych oraz symulacji. Dostępne na systemach *Windows*, *Mac OS* oraz w wersji beta na systemach *Linux*. Pozwala na budowanie aplikacji na szeroką gamę platform, w tym *PC*, konsole *Xbox* i *PlayStation*, platformy mobilne jak *Android* czy *IOS*. *Unity* pozwala korzystać z języków *C#* oraz *Javascript*. Warto jednak zaznaczyć, że silnik konwertuje skrypty z tych języków do kodu *C++* wykorzystując *IL2CPP*.

Głównym argumentem, aby korzystać z technologii *Unity* jest jej wszechstronność oraz bardzo dobra dokumentacja wraz z materiałami szkoleniowymi. Pozwalają one na zrozumienie mechanizmów działania dzisiejszych silników do gier. Szeroka gama narzędzi umożliwia swobodne tworzenie i szybkie testowanie pomysłów.

Omawiane w tej pracy *Navigation Mesh* oraz system *Boid'ów* były projektowane z myślą o późniejszym wykorzystaniu w innych projektach. Dlatego starano się zachować możliwie dużą niezależność od przykładowych poziomów i scen.

## 3.2 Navigation Mesh

Moduł *Navigation Mesh* składa się z czterech głównych klas. Najważniejszą oraz łączącą całą funkcjonalność jest *NavMesh*, która udostępnia metodę *CalculatePath*. Metoda ta oblicza ścieżkę na podstawie dwóch promieni (*Ray*) wyznaczających punkty. Pozostałe klasy odpowiadają za poszczególne obliczenia, wcześniej opisane w pierwszym rozdziale.

Przy każdej z omawianych tutaj klas zamieszczono kilka uwag oraz sugestii, które mogą okazać się przydatne przy samodzielnej implementacji podobnego systemu. Opierają się one na spostrzeżeniach nabytych podczas budowy oraz na uwagach typowo programistycznych.

### 3.2.1 Mesh

Struktura siatki w grafice 3D jest w najprostszym przypadku zbiorem dwóch list. Pominięto tutaj aspekty takie jak tekstury czy subsiatki. W pracy wykorzystuje się klasę *Mesh* dostępną w *Unity*.

```
struct Mesh {
    Vector3[] vertices;
    int[] triangles;
    ...
}
```

Lista *vertices* to nic innego jak zbiór wierzchołków siatki. Natomiast lista *triangles* kolejno przedstawia ścianki siatki. N-ty trójkąt składa się więc z wierzchołków pod indeksami $3n$, $3n+1$, $3n+2$ z listy.

Warto pamiętać o tym, że lista *triangles* jest dokładnie trzy razy dłuższa od dokładnej ilości trójkątów, a w grafice komputerowej każdy trójkąt jest ścianą. Dlatego można zauważyć, że przedstawiona budowa jest przypadkiem szczególnym siatki omawianej w pierwszej części pracy. Po prostu obowiązuje założenie, że ściankę tworzą dokładnie trzy wierzchołki, nie więcej.

### 3.2.2 NavMesh

```
public class NavMesh : MonoBehaviour
{
    Mesh _mesh;
    NavGraph graph;
    public Mesh mesh {...};
    public Vector3[] CalculatePath (Ray sourceRay, Ray targetRay, ...) {..};
    public bool Raycast (Ray ray, out RaycastHit hitInfo, ...) {...};
}
```

Głównym zadaniem klasy *NavMesh* jest obliczenie ścieżki dla dwóch zadanych promieni *(Ray)*. Poszczególne obliczenia są delegowane do odpowiednich klas, opisanych w dalszej części pracy. *CalculatePath* przedstawia jedynie wysokopoziomowe spojrzenie na ten proces, podobnie jak w części z analizą problemu.

### 3.2.3 NavGraph

```
public class NavGraph
{
    public readonly Mesh mesh;
    public readonly int[] nodes;
    public readonly float[] costs;
    public readonly Vector3[] centers;

    public NavGraph (Mesh mesh) {...};

    public static int[] CalculateNodes (Mesh mesh) {...};
    public static Vector3[] CalculateMiddles (Mesh mesh) {...};
    public static float[] CalculateCosts (int[] nodes, Vector3[] middles) {...};
    public int[] AStarSearch (int source, int target, Vector3 targetPoint) {...};
    float Heuritic (Vector3 center, Vector3 target) {...};
}
```

Zadaniem klasy *NavGraph* jest zbudowanie grafu na podstawie istniejącej siatki oraz wykonywanie operacji przeszukiwania grafu za pomocą algorytmu $A*$. Listy *costs* i *centers* są równoliczne z listą trójkątów w siatce. Wierzchołek o indeksie $i$ jest trójkątem opisanym przez trzy wierzchołki siatki o indeksach $3i$, $3i+1$, $3i+2$. Lista *nodes* to lista sąsiadów dla każdego z wierzchołków.

Pożyteczną obserwacją jest fakt, że ponieważ każda ścianka jest trójkątem to może posiadać co najwyżej trzech sąsiadów. Dlatego listę *nodes* można z góry ograniczyć przez $3n$.

### 3.2.4 NavPortals

```
public static class NavPortals
{
    public static Vector3[] Calculate (Mesh mesh, ...) {...};
}
```

Klasa *NavPortals* wykonuje tylko jedno zadanie. Na podstawie siatki, punktu startowego, końcowego oraz ścieżki w grafie wyznacza listę współrzędnych wierzchołków portali (wspólnych krawędzi).

### 3.2.5 NavPath

```
public static class NavPath
{
    public static Vector3[] Smooth (Vector3[] portals) {...};
}
```

Zadaniem klasy *NavPath* jest wygładzenie ścieżki. Oznacza to tyle, że korzystając z algorytmu lejka obliczane są kolejne punkty ścieżki.

Warto zauważyć kilka ciekawych właściwości tego algorytmu. Obliczenia można poprowadzić równolegle od obu stron jednocześnie. Jest to jedno z możliwych usprawnień jednak w praktyce może okazać się po prostu nieprzydatne. W dynamicznie zmieniającym się środowisku i przy częstych obliczeniach ścieżek może się okazać, że jednostka nigdy nie dochodzi do dalszych punktów ścieżki niż drugi lub trzeci. Taki scenariusz występuje w większości gier strategicznych, gdzie bardzo często wydaje się rozkaz jednostkom i bardzo często są one zmieniane. Dlatego nie ma potrzeby obliczania wszystkich punktów ścieżki, ale np. tylko pierwsze $n$. Rozwiązanie to pozwala na zastosowanie tablic o stałej długości, eliminując w ten sposób konieczność dynamicznej alokacji pamięci.

## 3.3 Boids

Moduł *Boids* składa się z dwóch głównych klas. Podstawową jest *Boid* która opisuje stan jednostki i jej aktualną prędkość i kierunek ruchu. Na podstawie tych informacji inne jednostki modyfikują swoje zachowania. Drugą klasą jest *BoidGroup* która opisuje ruch grupy jednostek.

### 3.3.1 Boid

```
public class Boid : MonoBehaviour
{
    public Vector3 direction {...}
    public float speed;

    public float nearbyRadius;
    public float visibleAngle;

    public Vector3 velocity;
    public float affectDistance;
    public float separation;
    public float alignment;
    public float cohesion;

    public bool passOnRight;
    public float passAngle;


    void Move () {...}
    void Rotate () {...}
    void UpdateNearby () {...}

    Vector3 Separation () {...}
    Vector3 Alignment () {...}
    Vector3 Cohesion () {...}
    void PassOnRight () {...}
}
```

Klasa *Boid* opisuje ruch jednostki oraz wprowadza modyfikacje w zachowaniu, w zależności od otaczającej ją jednostek. Szereg parametrów pozwala na dość swobodne testowanie zachowań dla różnych wartości. Parametry *nearbyRadius* oraz *visibleAngle* to dwie wartości, na podstawie których jednostka skanuje otocznie w poszukiwaniu jednostek. W praktyce do wyszukiwania jednostek zastosowano funkcję *Physics.OverlapSphere*. Po czym sprawdza się czy spełnione jest ograniczenie kąta widoczności. Operacja aktualizacji otoczenia jest przeprowadzana w funkcji *UpdateNearby*.

Omówione wcześniej algorytmy stadne zostały bezpośrednio zaimplementowane w klasie *Boid*. Są to metody: *Separation*, *Alignment*, *Cohesion* oraz *PassOnRight*. Jedyna różnica pomiędzy wymienionymi funkcjami polega na tym, że pierwsze trzy generują dodatkową prędkość jednostki, gdzie ostatnia modyfikuje ją bezpośrednio.

### 3.3.2 BoidGroup

```
public class BoidGroup : MonoBehaviour
{
    NavigationMesh.NavMesh navMesh;
    Boid[] boids;
    Vector3[] path;
    Vector3 center;

    public Vector3[] SetDestination (Ray targetRay) {...}
}
```

Klasa *BoidGroup* służy do opisu ruchu kilku jednostek o wspólnym celu. Dzięki tej klasie, proces wyznaczania ścieżki wykonywany jest tylko raz, po czym jest propagowany do każdej z jednostek z osobna. Takie rozwiązanie pozwala na bardzo proste testowanie zachowań stadnych oraz w makroskali pozwala traktować grupę jednostek jak jedną.

Parametr *center* to środek ciężkości grupy. Przy wyznaczaniu ścieżki jest on traktowany jako punkt początkowy.

# Scenariusze testowe

Na potrzeby tej pracy przygotowano sześć scenariuszy testowych. Każdy z nich stara się rozwiązać jeden z przypadków spotykanych w grach i symulacjach. Celem jest przetestowanie stworzonego systemu w sytuacjach jak najbardziej praktycznych.

W celach demonstracyjnych zaprezentowane są ścieżki każdej z jednostek za pomocą białej linii. W zbudowanej wersji aplikacji ten widok nie jest dostępny. Dla scenariuszy interaktywnych możliwe jest wyznaczanie punktów docelowych dla jednostek. Dokonuje się tego za pomocą kliknięcia na dowolne miejsce w terenie za pomocą lewego przycisku myszy. Obracanie kamery dookoła poziomu możliwe jest za pomocą jednoczesnego przytrzymania prawego przycisku i przesuwania myszy. Jeżeli nie istnieje ścieżka do wyznaczonego punktu, zniknie pomarańczowy wskaźnik celu oraz postać zatrzyma się w miejscu.

## 4.1 Staircase

Interaktywny scenariusz może być przykładem prostej gry. Testowany jest tutaj tylko moduł *Navigation Mesh*, czyli wyszukiwanie najkrótszej ścieżki tylko dla jednej jednostki. Na rysunku 4.1 zaprezentowano przykładową ścieżkę.

## 4.2 Boxes

Interaktywny scenariusz reprezentuje standardowy poziom, z wieloma nieregularnymi przeszkodami statycznymi. Tutaj również testowany jest tylko moduł obliczania najkrótszej ścieżki dla pojedynczej jednostki.

Można zauważyć, że wyznaczana ścieżka faktycznie odpowiada intuicji i nie występują nienaturalne zagięcia ścieżki 4.2.

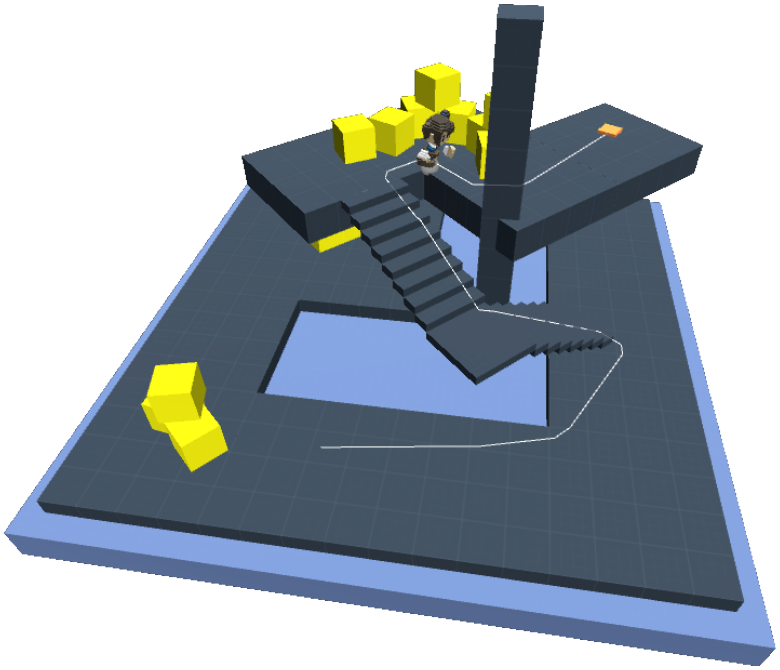
## 4.3 Warehouse

Interaktywny scenariusz reprezentuje standardowy poziom, z wieloma regularnie ułożonymi, statycznymi przeszkodami. Ten przypadek bardzo dobrze prezentuje, że obliczanie ścieżki za pomocą *Navigation Mesh* nie zawsze daje optymalne wyniki. Jest to bezpośrednio związane z budową siatki. Na rysunkach 4.3 ścieżka przyjmuje nienaturalny kształt. Należy jednak pamiętać, że strata dokładności na rzecz prędkości jest w tym przypadku niewielka. Dlatego dla takich szczególnych przypadków, należy po prostu dostosować siatkę terenu oraz heurystykę do przeszukiwania grafu.

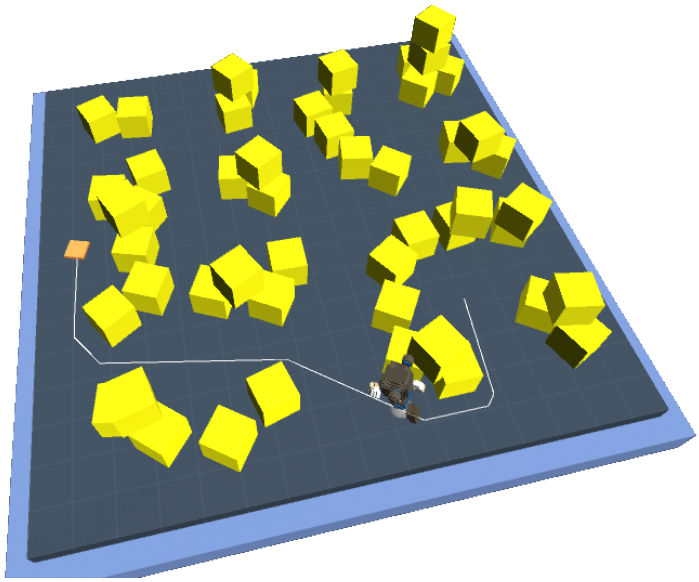
Rysunek 4.4 jest bardzo dobrym przykładem, obliczeń zwracających niedokładne wyniki. Ścieżka widoczna na rysunku początkowo wręcz oddala się od wyznaczonego punktu. Jak już zaznaczono, takie obliczenia wynikają z nieprzystosowania heurystyki oraz struktury wygenerowanej siatki.

## 4.4 Passing

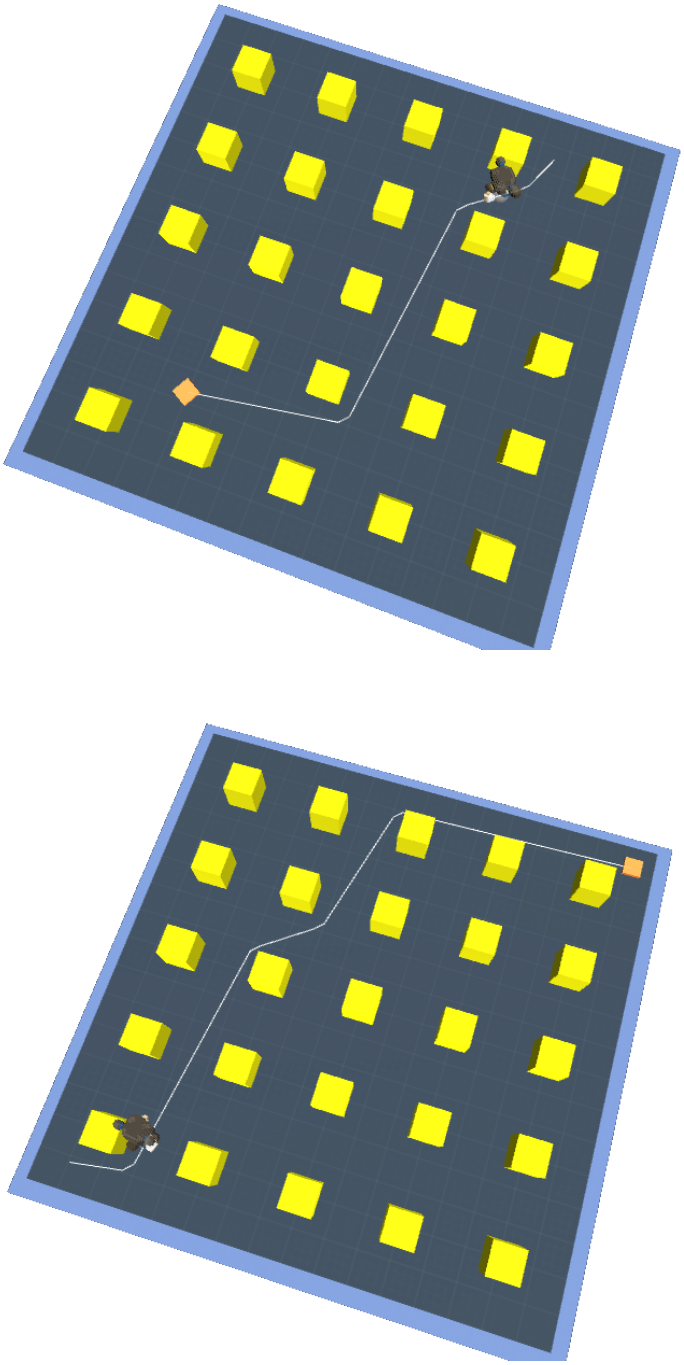
Jest to scenariusz mający na celu przetestowanie algorytmów lokalnego otoczenia, a dokładniej zasady przepuszczania jednostek po prawej stronie. Celem jednostek jest przedostanie się do punktów na drugim



Rysunek 4.1: Staircase - przykładowy poziom z gry



Rysunek 4.2: Boxes - przykładowy poziom z gry



Rysunek 4.3: Warehouse - dostatecznie dobre ścieżki

krańcu poziomu możliwie unikając kolizji z innymi jednostkami.

Na rysunku 4.5 widać, że dwie jednostki zatrzymały się, aby przepuścić inne. Dokładne działanie algorytmu jest jednak widoczne najlepiej w aplikacji, gdzie widać które jednostki, kiedy i gdzie się zatrzymują. Bardzo prosta modyfikacja ruchu, polegająca na przepuszczaniu innych jednostek po prawej stronie, pozwoliła uniknąć zaklinowania jednostek na zwężeniach. Na rysunkach 4.6 można zauważyć, że jednostki utworzyły odstępy umożliwiające swobodne dotarcie do celu.

## 4.5 Switching

Jest to scenariusz, w którym praktycznie wszystkie jednostki muszą przejść przez centrum poziomu. Tutaj również celem było przetestowanie algorytmów lokalnego otoczenia.

Na rysunkach 4.7 widać jak jednostki przepuszczają siebie nawzajem w centrum.

## 4.6 Square

Ostatni scenariusz testowy ma na celu przetestowanie zachowania stadnego. Grupie jednostek wyznaczany jest wspólny cel na podstawie aktualnego położenia centrum masy grupy (centrum jest zaznaczone za pomocą szarej sfery).

Bardzo proste cele, które nie wymagają omijania przeszkód są osiągane bez większych problemów. Widać to na rysunku 4.8, gdzie jednostki po prostu przemieszczają się w linii prostej do wyznaczonego punku.

W przypadku, gdy pomiędzy grupą a celem znajduje się przeszkoda, ścieżka jest poprawnie wyznaczana za pomocą *Navigation Mesh*. Jednak samo zachowanie grupy na podstawie algorytmów stada doprowadziło do lekkiego rozproszenia grupy. Początkowo, zielona jednostka zaklinowała się na przeszkodzie, a później, z powodu zmiany rozłożenia masy w grupie, nie dołączyła z powrotem do reszty. O ile większość grupy dotarła do wyznaczonego celu, to widoczne są tutaj pierwsze problemy związane z traktowaniem grup jednostek jako jednej większej jednostki. Rysunki 4.10.

Rysunek 4.11 prezentuje przypadek, gdy przy omijaniu przeszkody przez grupę jednostek centrum masy przesunęło się na tyle, że ominięto punkt docelowy.

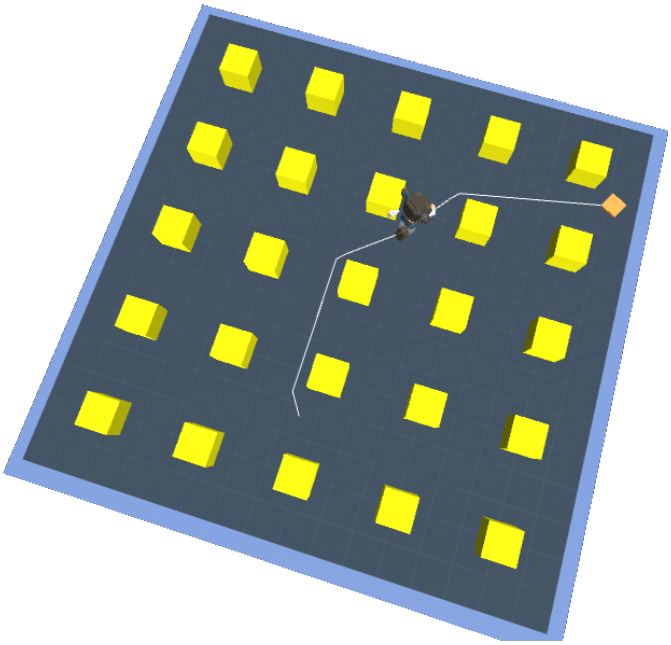
## 4.7 Wnioski i spostrzeżenia

Tak jak było to miejscami zaznaczane we wcześniejszych częściach pracy, jakość obliczonej ścieżki zależy od dwóch głównych czynników: jakości siatki *Navigation Mesh* oraz użytej heurystyki w zależności od charakterystyki siatki

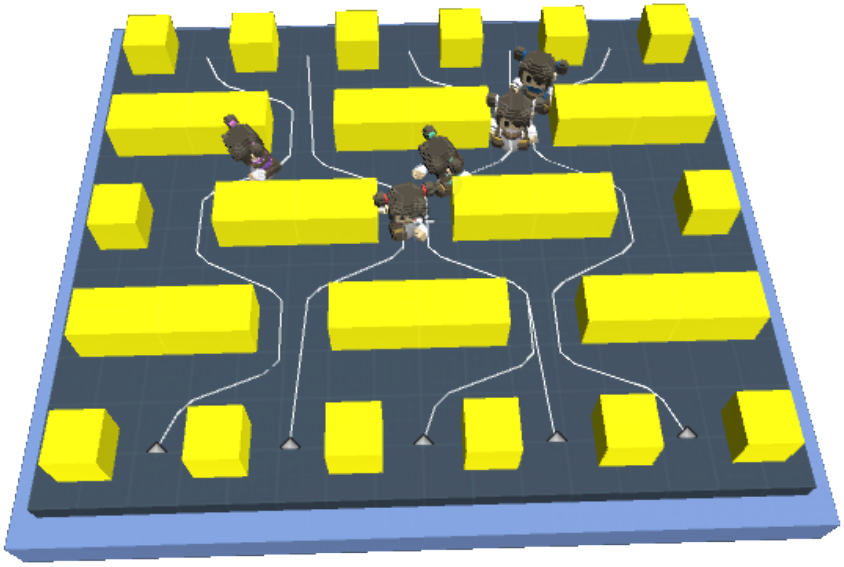
Scenariusze *Staircase* i *Boxes* są przykładami takich poziomów, gdzie standardowo wygenerowana siatka *Navigation Mesh* przez silnik *Unity* daje poprawne rozwiązania. Jednak dla poziomu o regularnej budowie (*Warehouse*) wygenerowana siatka oraz ogólna funkcja heurystyczna powodują pewne nienaturalne zachowania przy obliczaniu najkrótszych ścieżek. Aby ten problem rozwiązać, najprawdopodobniej należałoby w pierwszej kolejności spróbować dostosować heurystykę, a gdy to okaże się nieskuteczne, rozparzyć ręczne generowanie siatki dla *Navigation Mesh*.

Algorytmy podejmujące decyzje na podstawie lokalnego otoczenia okazały się częściowo skuteczne. Dla scenariuszy *Passing* i *Switching*, gdzie głównym problemem było klinowanie się na wąskich przejściach, algorytmy te okazały się bardzo skuteczne. Proste przepuszczanie jednostek po swojej prawej stronie skutecznie rozwiązuje problem pierwszeństwa przy przejściu przez zwężenia. Należy oczywiście pamiętać, że istnieją scenariusze tutaj nie rozpatrywane, dla których zastosowanie bez modyfikacji tej zasady może doprowadzić do zatrzymania wszystkich jednostek. Przykładem może być hipotetyczne spotkanie się czterech jednostek, każdej poruszającej się z innej strony, na skrzyżowaniu. Rozwiązaniem tego typu problemów może być również wprowadzenie czynników losowych.

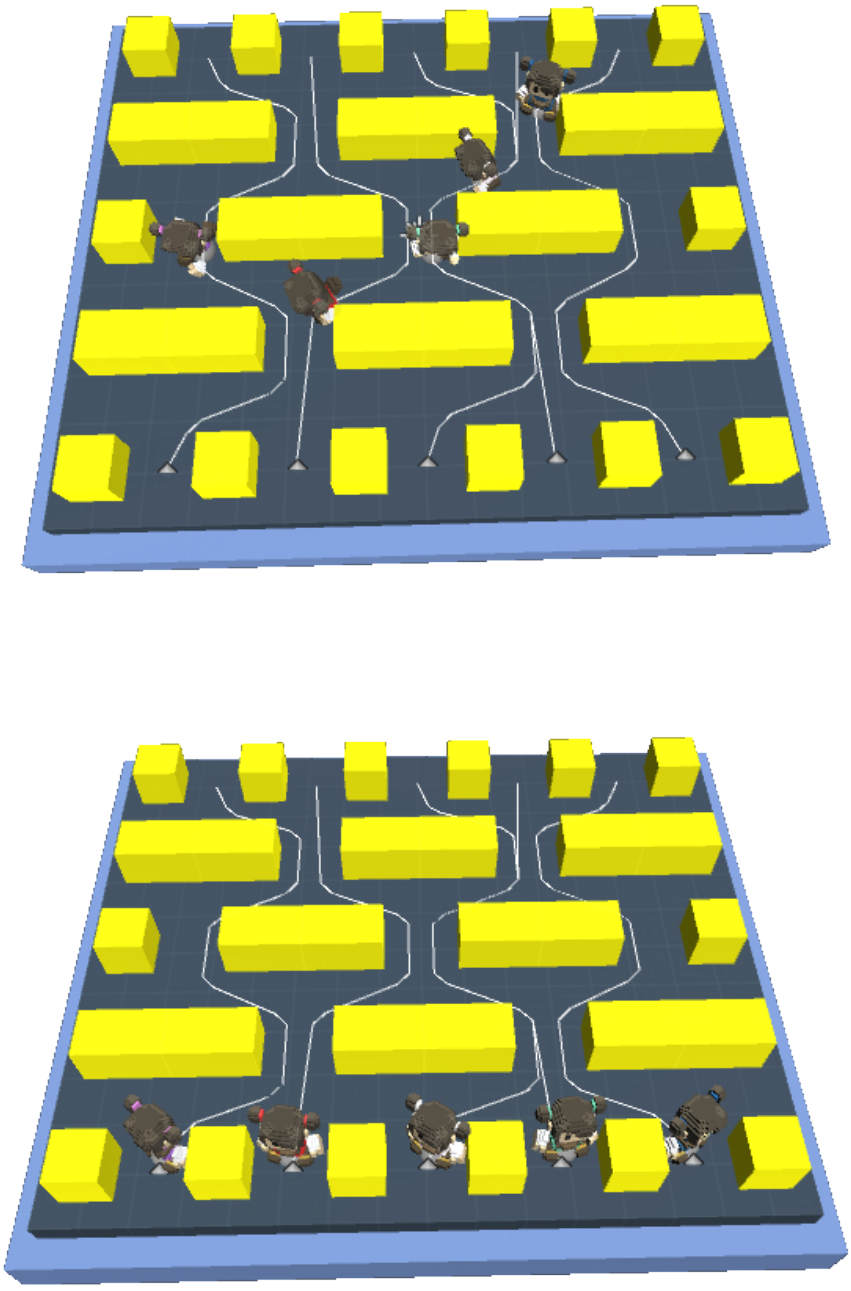
Scenariusz *Square* pokazał dużą słabość próby traktowania grupy jednostek jako jednej na podstawie ich środka ciężkości. Bardzo często dochodziło do sytuacji, gdzie cel był mijany i grupa nie zatrzymywała się



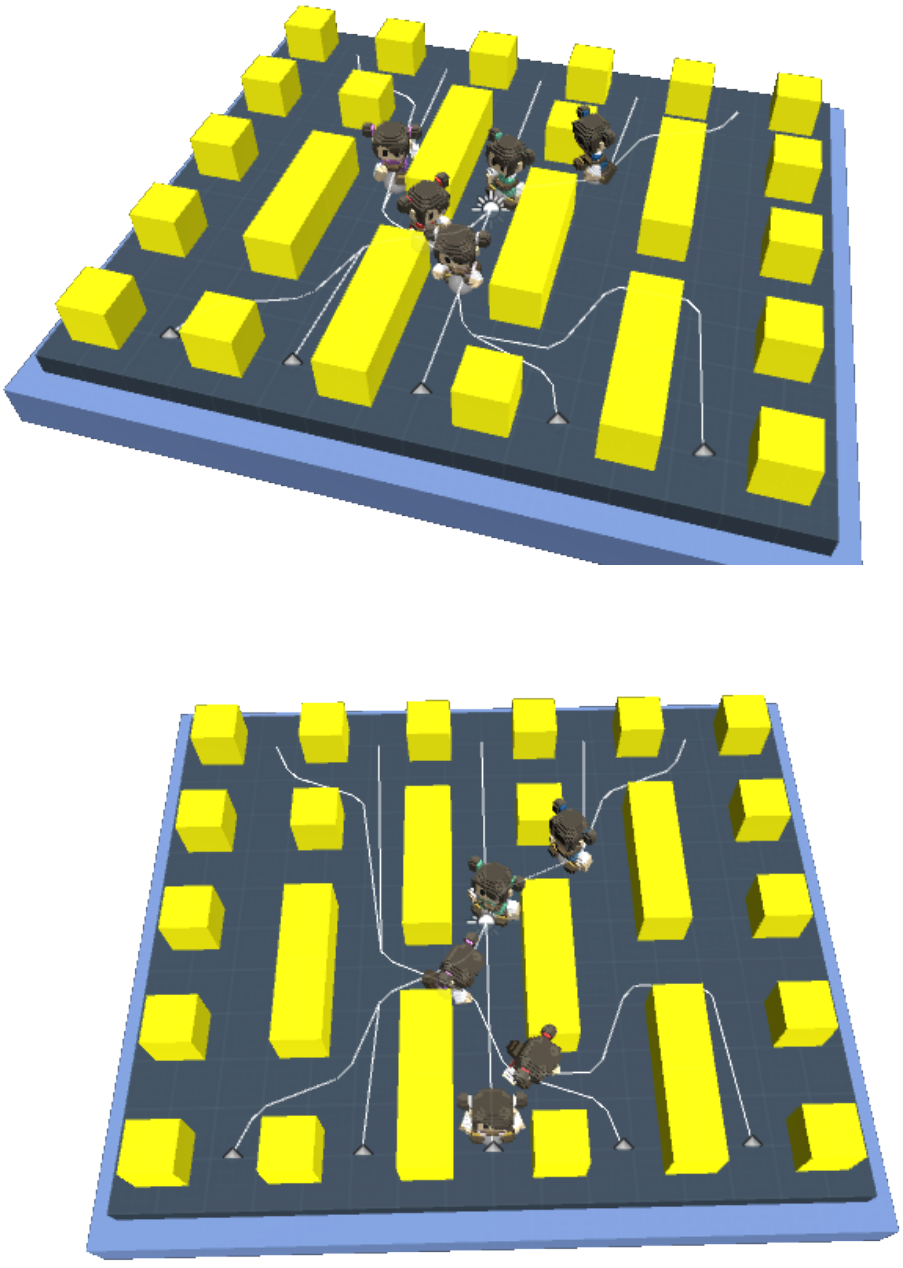
Rysunek 4.4: Warehuse - przypadek gdy otrzymana ścieżka jest nienaturalna



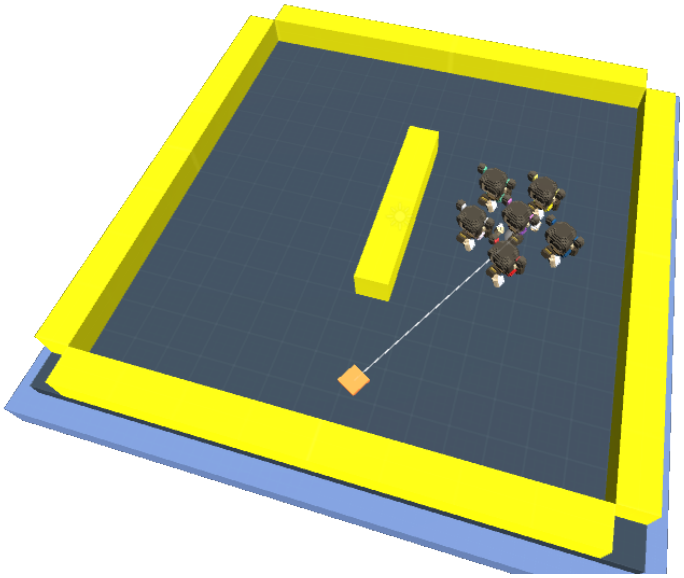
Rysunek 4.5: Passing - przepuszczanie jednostek po prawej stronie



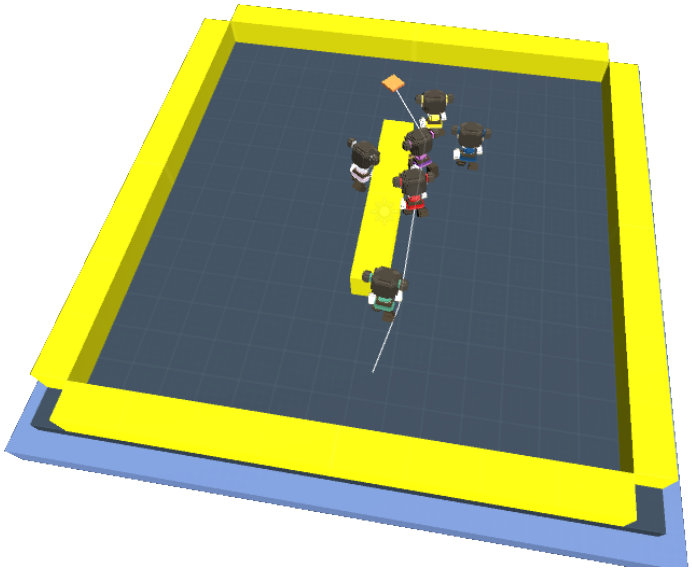
Rysunek 4.6: Passing - utworzenie odstępów oraz dotarcie do celów



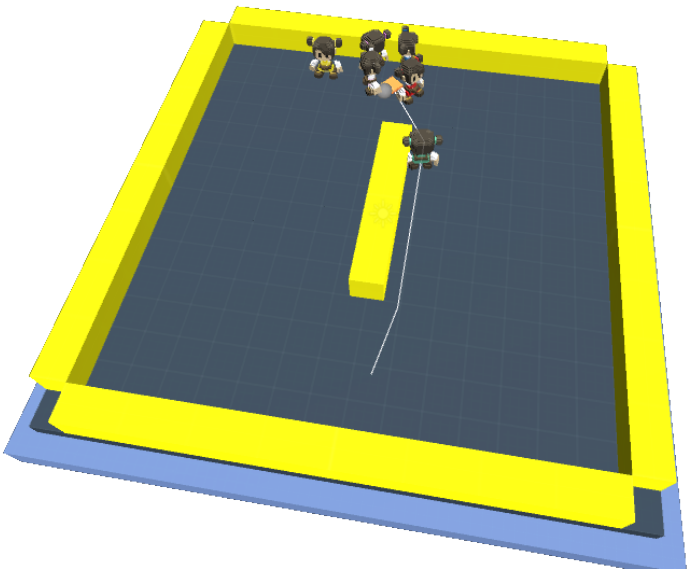
Rysunek 4.7: Switching - przepuszczanie jednostek po prawej stronie



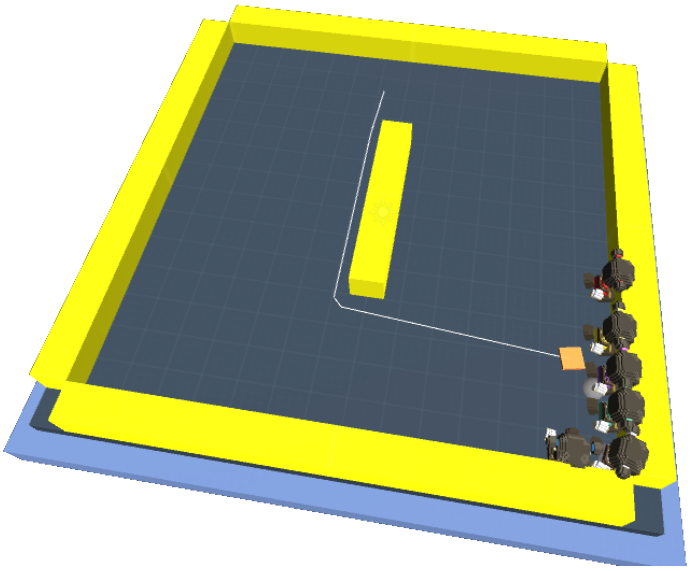
Rysunek 4.8: Square - proste ścieżki



Rysunek 4.9: Square - omijanie statycznych przeszkód



Rysunek 4.10: Square - omijanie statycznych przeszkód



Rysunek 4.11: Square - przesunięcie środka masy

przy nim. Jednak pomimo tych niepowodzeń można zastanowić się nad ewentualnymi poprawkami w tym rozwiązaniu. Pierwszym usprawnieniem może okazać się hierarchizacja grupek w taki sposób, by każdą wyższą tworzyły np. dwie o stopniu o jeden niższym. Dla tak pogrupowanych jednostek obliczanie najkrótszych ścieżek będzie wspólne tylko dla tych grupek, które są odpowiednio blisko siebie.

Jak można zauważyć algorytmy lokalnego otoczenia nie rozwiązały wszystkich problemów, ale ich analiza oraz przedstawione wnioski sugerują, że odpowiednia modyfikacja może owe problemy rozwiązać. Jednak nie będą one już tak proste jak te tutaj poruszane.

# Instalacja i wdrożenie

Projekt z przykładowymi poziomami jest zbudowany za pomocą *Unity*. Na załączonej płycie obok plików i kodu projektu w folderze *Builds* zamieszczono wersje na trzy systemy operacyjne: *Windows*, *Linux* i *Max OS*. W celu otworzenia aplikacji wystarczy kliknąć na ikonę *Navigation*.

Ponieważ system z założenia miał być łatwo integrowalny z innymi projektami w technologii *Unity* oraz już na samym początku zaznaczono niezależność obu części (*Navigation Mesh* oraz *Boids*) oba całe moduły umieszczono w folderze *Assets/Scripts*. Także do wykorzystania funkcjonalności oferowanej przez system wystarczy zaimportować oba foldery do swojego projektu.

Aby przeprowadzić obliczenia na *Navigation Mesh* należy w *Unity* utworzyć nowy obiekt i dodać do niego skrypt *NavMesh*. Obiekt ten musi posiadać komponenty *Mesh Filter*, *Mesh Collider* oraz *Mesh Renderer*. Domyślnie do obliczeń wykorzystywana jest siatka z komponentu *Mesh Filter*.

Skrypt do zachowań lokalnych *Boid* zakłada, że dany obiekt będzie posiadał komponent *Character Controller*, wykorzystując go do nadawania prędkości oraz zmiany kierunku ruchu.

# Podsumowanie

W ramach pracy zaprojektowano i zaimplementowano system wyznaczający najkrótsze ścieżki, z eliminacją kolizji pomiędzy poruszającymi się jednostkami. Struktura *Navigation Mesh* została zaimplementowana w całości. Istnieją jeszcze dodatkowe rozwiązania rozszerzające tę strukturę, zostaną zaproponowane na końcu tego rozdziału. Przetestowano również kilka algorytmów podejmujących decyzje na podstawie lokalnego otocznia. Wyniki eksperymentów wskazały wiele problemów, które wiążą się z takimi rozwiązaniami Jednak przy dokładniejszej analizie problemów można je efektywnie rozwiązać. Nie jest to jednak częścią tej pracy. Okazało się, że najlepiej działającą strategią jest przepuszczanie jednostek po prawej stronie.

Opisany w tej pracy system spełnia podstawowe zadania wyznaczania najkrótszej ścieżki oraz dla pewnych przypadków rozwiązuje problem kolizji. Podczas dalszego rozwoju systemu należy skupić się na następujących funkcjach:

- Optymalizacja kodu pod konkretne zastosowania (unikanie dynamicznej alokacji pamięci).
- *NavMesh Link*, czyli umożliwienie np. zeskoczenia z przeszkód w określonych miejscach.
- Dodanie kosztu związanego z trudnością poruszania się przez dane części terenu.
- Zwiększenie ilości strategii korzystających z lokalnego otoczenia.
- Szersza analiza algorytmów lokalnego otoczenia w wykorzystaniu do nawigacji bardzo dużej ilości jednostek.

# Bibliografia


[1] Boids. Web pages:
http://www.kfish.org/boids/pseudocode.html.

[2] Funnel algorithm. Web pages:
http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html.

[3] Introduction to a*. Web pages:
http://www.redblobgames.com/pathfinding/a-star/introduction.html.

[4] Map representations. Web pages:
http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html.

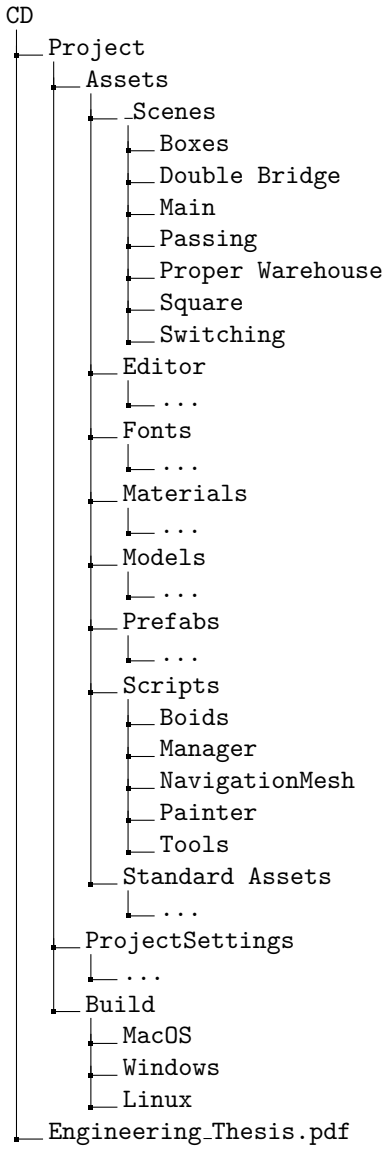
[5] Unity game engine. Web pages:
https://unity3d.com/.

[6] Unity il2cpp. Web pages:
https://docs.unity3d.com/Manual/IL2CPP.html.

[7] Unity manual. Web pages:
https://docs.unity3d.com/Manual/index.html.

[8] Unity scripting reference. Web pages:
https://docs.unity3d.com/ScriptReference/index.html.

# Zawartość płyty CD




Rysunek A.1: Zawartość płyty.

Na rysunku A.1 przedstawiona jest ogólna struktura plików znajdujących się na płycie. W folderze *Project* znajdują się pliki projektu aplikacji *Unity*. W folderze *Assets/_Scenes* znajdują się sceny z wcześniej omawianymi scenariuszami, a wszystkie skrypty i implementacja omawianego systemu w odpowiednich podfolderach *Assets/Scripts*. W folderze *Project/Build* znajdują się zbudowane wersje aplikacji na trzy najpopularniejsze systemy operacyjne. Na płycie znajduje się również kopia niniejszej pracy *Engineering_Thesis.pdf*.