UNIWERSYTET JAGIELLOŃSKI
WYDZIAŁ FIZYKI, ASTRONOMII I INFORMATYKI
STOSOWANEJ

TOMASZ BOROWSKI

SZTUCZNA INTELIGENCJA W SYMULATORZE DZIAŁAŃ ANTYTERRORYSTYCZNYCH

PRACA MAGISTERSKA NAPISANA POD KIERUNKIEM

DR HAB. PIOTRA BIAŁASA

KRAKÓW 2012

Streszczenie

Niniejsza praca dyplomowa omawia projekt gry symulacyjnej, w której gracz ma możliwość planowania i przeprowadzania działań antyterrorystycznych. Zastosowane w projekcie algorytmy sztucznej inteligencji, typowe dla gier wideo, zostały uzupełnione algorytmami realizującymi charakterystyczne dla strony konfliktu taktyki. Dokumentacja projektu jest uzupełniona opisem technologi HTML5 Canvas oraz bibliotek JavaScript wykorzystanych podczas implementacji.

Spis treści

Oświadczenie

Świadomy odpowiedzialności prawnej oświadczam, że złożona praca magisterska pt.: „Sztuczna inteligencja w symulatorze działań antyterrorystycznych” została napisana przeze mnie samodzielnie.

Równocześnie oświadczam, że praca ta nie narusza prawa autorskiego w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U.1994 nr 24 poz. 83) oraz dóbr osobistych chronionych prawem cywilnym.

Ponadto praca nie zawiera informacji i danych uzyskanych w sposób nielegalny i nie była wcześniej przedmiotem innych procedur urzędowych związanych z uzyskaniem dyplomów lub tytułów zawodowych uczelni wyższej.

Wprowadzenie

Gry wideo, które dotychczas kojarzone były niemal wyłącznie z pojęciem interaktywnej formy dostarczania rozrywki, od wielu lat zdobywają coraz to nowsze pola zastosowań. Przykładem tutaj mogą być gry oparte o zasadę tzw. *edutainment* (w tłum. *edurozrywka*)[1]. Mają one na celu efektywne przekazywanie wiedzy dzięki swojemu atrakcyjnemu i rozrywkowemu charakterowi, w takich dyscyplinach naukowych jak biologia, fizyka, informatyka lub języki obce. Innym polem zastosowań elementów gier jest biznes. Coraz częściej można spotkać się z pojęciem *gamefication* (w tłum. *grywalizacji*) miejsca pracy. Określa ono zestaw technik i narzędzi związanych z grami, które pomagają motywować pracowników do lepszego wykonywania powierzonej im pracy. Dzieje się to poprzez nagradzanie najlepszych pracowników wirtualnymi punktami doświadczenia, osiągnięciami oraz umieszczaniem ich wizerunku na szczytach rankingów[2]. Wreszcie, możemy mieć również do czynienia z grami symulacyjnymi. Ich celem jest umożliwienie graczom doznawania wrażeń znanych z rzeczywistości, a których oni bezpośrednio mogą na co dzień nie doświadczać. Wśród takich gier można wyróżnić gry, których celem jest szkolenie użytkowników - np. symulatory lotu - oraz te, których głównym celem jest dostarczenie użytkownikom rozrywki - np. symulator prowadzenia sieci pizzerii.

Niniejsza praca dyplomowa skupia się na projekcie gry symulacyjnej, która odwzorowuje, w dużym uproszczeniu, działania oddziałów antyterrorystycznych pod-

[1]przykładowy serwis z grami edukacyjnymi - http://www.edugames.pl/
[2]przykładowa aplikacja bazująca na idei grywalizacji - https://dueprops.com/

Rysunek 1: Fligt Simulator 2004 - przykład gry symulacyjnej

czas szturmu na budynek, zajęty przez wrogie jednostki. Użytkownik grający w tą grę ma możliwość stworzenia schematu budynku, parametryzacji liczby jednostek po obu stronach konfliktu oraz określenia planu działania antyterrorystów. Na podstawie tej konfiguracji gra przeprowadza symulację szturmu na budynek, którą gracz może obserwować.

Realizacja tego projektu obejmuje zaprojektowanie i zaimplementowanie gry oraz omówienie taktyk stosowanych przez strony konfliktu. Zwrócona jest szczególna uwaga na twórcze wykorzystanie algorytmów sztucznej inteligencji, charakterystycznych dla gier wideo. Uzupełnieniem dokumentu jest przedstawienie technologii i bibliotek, które zostały wykorzystane podczas implementacji.

Rozdział 1

State of Art

1.1 Planowanie operacji antyterrorystycznych w rzeczywistości

Problem terroryzmu i skutków, jakie może on wyrządzać ludności, jest dla instytucji państwowych podstawą do przygotowywania długoterminowych strategii jego zapobiegania. Strategie te ujęte są w dokumentach[1] przygotowywanych przez instrumenty państwowe. Opisują one środki i metody zabezpieczania obywateli przed aktami terroryzmu. Niestety, w zetknięciu z rzeczywistością bywają one nie zawsze skuteczne.

Mając do czynienia z aktem terroryzmu, polegającym na przejęciu kontroli przez terrorystów nad pewną przestrzenią (np. nad budynkiem), służby odpowiadające za bezpieczeństwo podejmują szereg działań, które mają na celu zminimalizować ryzyko utraty zdrowia lub życia przez osoby postronne (w tym ew. zakładników). Prócz zabezpieczenia okolicznego terenu (odizolowaniu go od cywili oraz mediów) oraz prowadzenia negocjacji z terrorystami, bardzo ważnym elementem jest przygotowa-

[1] polskim przykładem jest dokument "Narodowy Program Antyterrorystyczny RP na lata 2012-2016"

nie planu przejęcia zakładników oraz ew. eliminacji terrorystów z użyciem siły. Do takiej czynności może dojść w przypadku, gdy terroryści odmówią negocjacji, bądź gdy zaczynają zabijać zakładników.

Proces planowania akcji antyterrorystycznych jest często charakterystyczny dla przeprowadzającej go jednostki specjalnej i zawsze jest strzeżony tajemnicą. Jednakże na przełomie kwietnia i maja 1980 roku, gdy grupa sześciu terrorystów przejęła kontrolę nad Ambasadą Irańską w Londynie, biorąc za zakładników 26 osób, to brytyjskie jednostki specjalne przeprowadziły skuteczną eliminację terrorystów na oczach całego świata[2]. Dzisiaj Operacja Nimrod jest szczegółowo udokumentowana licznymi artykułami[3], książkami oraz dokumentami wideo. Dzięki tej wiedzy jesteśmy w stanie odtworzyć proces planowania takiej akcji antyterrorystycznej, co zostało ukazane w tabeli 1.1. Spełnienie wszystkich wymienionych czynności znacznie zwiększa szanse na powodzenie operacji: uratowanie zakładników, eliminacja terrorystów i nieodniesienie strat własnych przez jednostkę przeprowadzającą atak.

W grze symulacyjnej, będącej przedmiotem tej pracy dyplomowej, gracz może zaplanować podstawowe elementy operacji antyterrorystycznej:

1. zdefiniować liczbę terrorystów i antyterrorystów
2. zaplanować jednopoziomową architekturę budynku
3. oznaczyć punkty kluczowe wokół których można spodziewać się obecności terrorystów
4. zdefiniować punkt wejścia oraz punkt ewakuacji

[2]świadkami operacji byli dziennikarze wielu stacji telewizyjnych, a wśród zakładników byli m. in. reporterzy BBC

[3]przy przygotowywaniu tej pracy został wykorzystany artykuł ze strony Elite UK Forces[1]

Planowana czynność	Realizacja (Nimrod)
1. Przygotowanie IA Plan[4] 2. Zbieranie danych 3. Rozpoznanie wroga 4. Rozpoznanie wyposażenia wroga 5. Rozpoznanie terenu 6. Określenie niezbędnych środków 7. Określenie punktów wejścia 8. Określenie punktów ewakuacji	1. Szturm ambasady od głównego wejścia i zabezpieczanie budynku piętro po piętrze 2. Zainstalowane podsłuchy w ścianach, snajperzy jako obserwatorzy, sprawdzanie punktów wejścia pod osłoną nocy 3. Wywiad dostarcza dane osobowe terrorystów, którzy starali się o wizy w ambasadzie Wielkiej Brytanii w Belgradzie 4. Jeden z uwolnionych zakładników informuje policję o liczbie i uzbrojeniu terrorystów 5. Analizowane są plany architektoniczne budynku i prowadzona jest konsultacja z woźnym ambasady 6. Cztery drużyny (24 żołnierzy), pistolety maszynowe MP5, ładunki wybuchowe, granaty ogłuszające, liny itp. 7. Wejście przez dach, wejście przez balkony na pierwszym piętrze, wejście tylnymi drzwiami na parterze 8. Ewakuacja zakładników do ogrodu za budynkiem ambasady

Tabela 1.1: Czynności dokonywane podczas planowania operacji antyterrorystycznej

1.2 Gry symulacyjne

Gatunek gier symulacyjnych charakteryzuje się wiernym odzwierciedlaniem realiów świata rzeczywistego lub fikcyjnego. Prócz zastosowania rozrywkowego, gry symulacyjne wykorzystuje się do celów szkoleniowych (np. wirtualna nauka jazdy) lub badawczych (np. analiza bezpieczeństwa terytorialnego). Wśród symulacyjnych gier wideo należy wymieć kilka podgatunków[5]:

Symulatory budowania i zarządzania cechują się brakiem obecności wroga, którego gracz musi pokonać. Są to gry o pewnych procesach (ekonomicznych, politycznych, wytwórczych itp.), w ramach których gracz odgrywa rolę architekta i zarządcy. Obiektami budowanymi mogą być parki rozrywki, porty lotnicze, szpitale, zoo czy też miasta. Im lepiej gracz rozumie zachodzące procesy, tym skuteczniejszy jest w wykonywaniu powierzonych mu zadań. Pierwszym symulatorem tego typu była gra **SimCity** [Maxis 1989].

Symulatory życia pozwalają na kontrolowanie istnień i rozwijaniu relacji między nimi. Mechanizmy są tu podobne do symulatorów budowania i zarządzania, często nie ma określonego kryterium zwycięstwa. Gry symulacyjne, gdzie gracz hoduje zwierzę lub jakiś antropomorficzny twór, skupiają się na tworzeniu i rozwijaniu relacji tej formy życia z graczem. Przykładami takich gier jest **The Sims** [Maxis 2000] oraz **Spore** [Maxis 2008].

Symulatory sportowe pozwalają graczowi na wirtualne uprawianie dyscyplin sportowych, których zasady i kryteria zwycięstwa są zgodne z rzeczywistymi odpowiednikami[6]. Często takie symulatory wymagają od swoich twórców modelowania rzeczywistych postaci ze świata sportu, wraz z uwzględnieniem ich

[5]przedstawiona lista wywodzi się z podziału przedstawionego w książce A. Rollingsa i E. Adamsa[2] i dopełniona jest podgatunkami omawianymi w różnych publikacjach internetowych

[6]choć część zasad może być wyłączana, np. czas trwania meczu piłkarskiego lub błąd kroków w koszykówce

umiejętności, charakterystycznych ruchów czy ubioru. Przykładami takich gier są gry z serii **Pro Evolution Soccer** [Konami] oraz **NBA Live** [EA Sports].

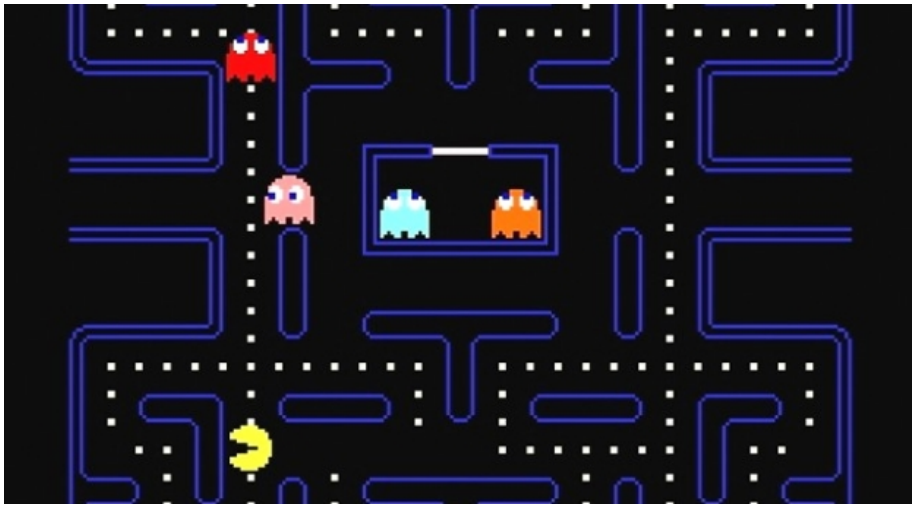
Symulatory pojazdów mają na celu dostarczyć graczom wrażeń, jakie mogliby odczuć podczas kierowania rzeczywistymi pojazdami, w określonych warunkach. Tego typu gry najczęściej charakteryzują się bardzo wysoką wiernością odzwierciedlenia pojazdów, do której należy zaliczyć takie czynniki jak wygląd, parametry jazdy lub lotu, wyposażenie oraz sterowanie. Przykładami takich gier jest seria **Colin McRae Rally** [Codemasters] oraz seria **Microsoft Flight Simulator** [Microsoft].

Symulatory czynności i zawodów to dość popularny w ostatnim czasie typ gier. Mają one na celu umożliwienie graczom na wirtualne wykonywanie prac związanych z zawodami, którymi na co dzień się nie zajmują. Przykładami takich gier jest **Symulator Farmy 2011** [Atari / Infogrames 2011] czy **Symulator Koparki 2011** [astragon Software 2011]. Realizm nie jest tutaj najważniejszym kryterium.

Grę symulacyjną, będącą przedmiotem tej pracy dyplomowej, można sklasyfikować w podgatunku symulatorów czynności i zawodów.

1.3 Sztuczna inteligencja w grach

Sztuczna inteligencja, jako dział informatyki, zajmuje się analizą zachowań człowieka oraz formalizowaniem (np. w postaci algorytmów) zaobserwowanych procesów m.in. myślowych i decyzyjnych. Dzięki takiej analizie jest możliwe przygotowywanie programów, pozwalających na rozwiązywanie problemów, które do tej pory były domeną ludzką. Przykładami mogą tu być wyszukiwanie danych, rozpoznawanie obiektów, syntezacja mowy lub podejmowanie decyzji. W tym celu algorytmy sztucznej



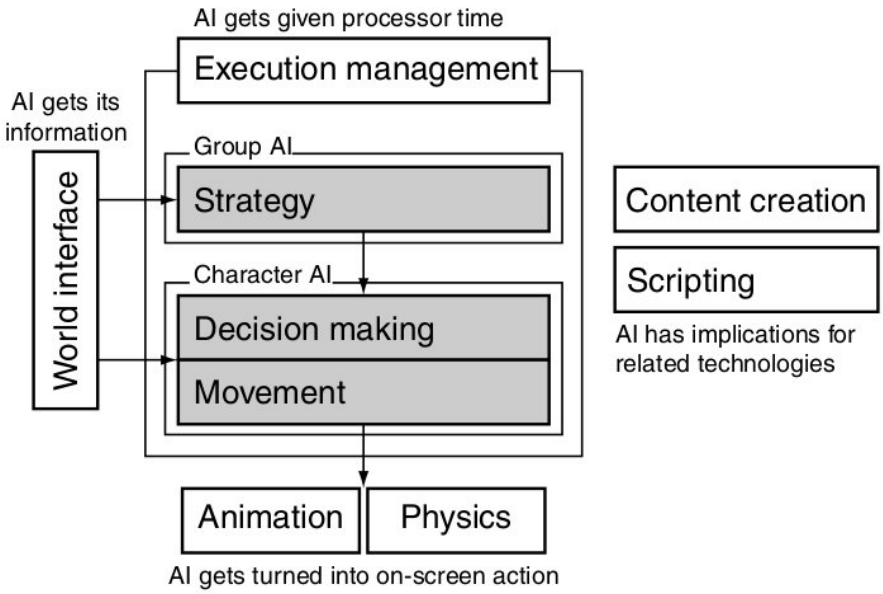
Rysunek 1.1: Pac-Man - przykład prostych technik sztucznej inteligencji w grach

inteligencji mogą wykorzystywać implementacje takich zagadnień jak sieci neuronowe, algorytmy genetyczne czy logika rozmyta.

W grach wideo sztuczna inteligencja najczęściej sprowadza się do zastosowania prostych technik sztucznej inteligencji, które mają na celu zaspokoić trzy podstawowe potrzeby bohaterów gry[3]:

- zdolność poruszania się
- zdolność do podejmowania decyzji gdzie należy się poruszyć
- zdolność taktycznego i strategicznego myślenia

Pac-Man [Namco, 1980] była jedną z pierwszych gier, która posiadała zauważalne dla odbiorców elementy sztucznej inteligencji. Gracz, poruszając się po dwuwymiarowym labiryncie, zdobywał punkty zjadając kropki (rysunek 1.1). W tej czynności aktywnie przeszkadzały mu cztery duchy, które starały się podążać korytarzami labiryntu w kierunku gracza. Od strony implementacyjnej gra opierała się o bardzo prostą maszynę stanową, która dla duchów definiowała dwa stany: podążaj za graczem i uciekaj od gracza. Na każdym skrzyżowaniu dróg labiryntu podejmowana



Rysunek 1.2: AI Model - zdefiniowany przez I. Millingtona i J. Funge

była decyzja[7] o następnym kierunku.

W późniejszych grach elementy myślenia i podejmowania decyzji stawały się coraz bardziej rozbudowane. Przykładem jest gra **Goldeneye 007** [Rare Ltd. 1997], gdzie postaci zostały wyposażone w system symulowanych zmysłów. Jedna postać analizowała pulę informacji ze świata gry, co pozwalało np. na dostrzeżenie martwego towarzysza i wykonanie odpowiedniej reakcji na ten fakt, czyli zmiany własnego stanu.

Analizując elementy składowe sztucznej inteligencji w grach wideo, należy odwołać się do modelu AI (rysunek 1.2). Postaci z gry posiadają wiedzę (całościową lub cząstkową) o świecie w którym funkcjonują (*World interface*). Na podstawie tej wiedzy każda postać, za pomocą odpowiednich algorytmów, podejmuje jakieś decyzje (*Decision making*) oraz porusza się (*Movement*). Element strategii (*Strategy*) jest przetwarzany na poziomie grupy postaci i może wpływać na podejmowane przez jednostki decyzje lub wykonywane ruchy. Rezultatem bezpośrednim tych obliczeń

[7]decyzja była losowa lub poparta prostymi obliczeniami

są wykonywane animacje (*Animation*) oraz wyliczenia fizyki ruchu postaci (*Physics*). Efektem ubocznym mogą tu być zmiany stanu gry, polegające na modyfikacji elementów świata (*Content creation*) gry oraz wykonywaniu oskryptowanych akcji (*Scripting*).

W grze symulacyjnej, będącą przedmiotem tej pracy dyplomowej, będziemy mogli wyróżnić każdy z trzech elementów modelu sztucznej inteligencji:

Podejmowanie decyzji np. otwarcie ognia do wroga

Poruszanie się np. poruszanie po ścieżce, wędrowanie

Strategia np. role i ich przejmowanie w grupie antyterrorystów

1.4 Istniejące rozwiązania: Tom Clancy's Rainbow Six

Gra **Rainbow Six** [Red Storm 1998] bazuje na powieści Toma Clancy'ego o tym samym tytule, która opisuje działania tajnego, międzynarodowego oddziału antyterrorystycznego Rainbow. Gra łączy w sobie elementy FPP[8] oraz strategii. Przeprowadzane operacje antyterrorystyczne są każdorazowo poprzedzane planowaniem szturmu na podstawie mapy lokacji (rysunek 1.3).

Przed misją gracz otrzymuje dane wywiadowcze, które posiadają zbliżony charakter do tych, które są wykorzystywane w planowaniu w rzeczywistości (szczegóły rozdziale 1.1). Podczas planowania misji w Rainbow Six gracz może:

1. zdefiniować liczbę drużyn antyterrorystów

2. zdefiniować liczbę antyterrorystów w drużynie oraz wskazać ich wyposażenie

[8]First Person Perspective - gra akcji z pierwszoosobową perspektywą

Rysunek 1.3: Tom Clancy's Rainbow Six - planowanie operacji antyterrorystycznej

3. oznaczyć punkty kluczowe, wzdłuż których będzie poruszać się drużyna antyterrorystów

4. zdefiniować lokacje, w których drużyna będzie czekała na polecenia innej drużyny

Po zaplanowaniu operacji antyterrorystycznej, gracz może uczestniczyć aktywnie w rozgrywce (będąc dowódcą jednej z drużyn) lub przyglądać się jej w roli obserwatora. Celem misji jest najczęściej odbicie zakładników, ale może też nim być rozbrojenie ładunków wybuchowych, zdobycie danych lub eliminacja konkretnej postaci. Element planowania misji wyróżnia Rainbow Six spośród innych gier o podobnej tematyce. Gra okazała się na tyle popularna, że doczekała się kolejnych części.

Różnice pomiędzy planowaniem operacji antyterrorystycznej w Rainbow Six a grą symulacyjną, będącą przedmiotem tej pracy dyplomowej, wymieniono w tabeli 1.2

1. Brak definiowania lokacji	1. Posiada prosty edytor lokacji
2. Posida podział antyterrorystów na drużyny	2. Dostępna jest tylko jedna drużyna antyterrorystów
3. Szczegółowe definiowanie wyposażenia	3. Brak możliwości definiowania wyposażenia
4. Możliwość podglądu planu w 3D	4. Podgląd planu wyłącznie w 2D

Tabela 1.2: Różnice pomiędzy planowaniem w Rainbow Six a przygotowaną grą symulacyjną

1.5 HTML5 Canvas i Kinetic.js

Canvas to część języka **HTML5**, której początki sięgają 2004 roku. Pozwala ona na dynamiczne renderowanie kształtów oraz obrazów w obrębie dokumentu HTML. Dzięki temu tworzenie animacji 2D i 3D nie wymaga instalowania dodatkowego oprogramowania, ponieważ całość jest obsługiwana przez środowisko współczesnych przeglądarek internetowych[9].

Istnieje duża ilość bibliotek javascript'owych, które ułatwiają pracę z HTML5 Canvas. Jedną z nich jest **Kinetic.js**, która dodatkowo pozwala na animowanie obiektów na scenie, przetwarzanie ich (translacje, rotacje, skalowanie itp.) oraz obsługę zdarzeń. Scena w Kinetic.js jest złożona z warstw zdefiniowanych przez użytkownika. Każda warstwa składa się z dwóch kontekstów: kontekst sceny i kontekst bufora. Podczas gdy kontekst sceny reprezentuje to, co jest renderowane na ekranie, to kontekst bufora odpowiada za wydajną obsługę zdarzeń. Każda warstwa może zawierać kształty lub grupy kształtów, które mogą być indywidualnie lub grupowo przetwarzane.

W grze symulacyjnej, będącej przedmiotem tej pracy dyplomowej, zostały zasto-

[9]zgodność danej przeglądarki internetowej ze standardami HTML5 można sprawdzić pod adresem http://html5test.com/

sowane następujące biblioteki javascript'owe:

- Kinetic.js[5] - pozwala na renderowanie obiektów oraz ich przetwarzanie
- Sylvester[6] - pozwala na obliczenia na wektorach
- jQuery[7] - pozwala na przetwarzanie elementów HTML
- javascript-astar[8] - pozwala na wyszukiwanie ścieżek algorytmem A*

Rozdział 2

Założenia projektu

2.1 Wymagania funkcjonalne

Realizacja projektu opierała się w całości o stosowanie tzw. technik zwinnych[1]. Proces tworzenia gry symulacyjnej został podzielony na etapy. Przed implementacją każdego etapu przygotowywany był zestaw scenariuszy opisujący funkcjonalności, jakie powinny zostać zaimplementowane w danym etapie. Natomiast po implementacji każdego z etapów gra symulacyjna była udostępniania kilku testerom, którzy w ramach informacji zwrotnej wskazywali, jakie funkcjonalności lub zachowania jednostek chcieliby zaobserwować w grze. Przy czytaniu scenariuszy przydatna jest znajomość słownika pojęć projektu (rozdział 2.3).

Pierwszy etap implementacji projektu zakładał zbudowanie architektury kodu aplikacji - utworzenie podstawowych klas, metod odpowiedzialnych za zarządzanie obiektami na scenie oraz metody wyznaczania bezkolizyjnej ścieżki do zadanego punktu. Dodatkowo jednostki miały mieć możliwość poruszania się do określonego punktu docelowego. Szczegóły zostały przedstawione w tabeli 2.1.

Podczas drugiego etapu miały zostać zaimplementowane kluczowe elementy in-

[1]Agile development - punktem wyjścia do tego podejścia jest Manifest Zwinnego Tworzenia Oprogramowania z 2001 roku http://agilemanifesto.org/iso/pl/

Etap 1 - Setup aplikacji
• aplikacja może tworzyć obiekty i renderować je na scenie • aplikacja może tworzyć terrorystów (obiekty ruchome) • aplikacja może tworzyć antyterrorystów (obiekty ruchome) • aplikacja może tworzyć ściany (obiekty statyczne) • aplikacja może wyliczać ścieżki dla obiektów ruchomych • jednostki mogą się poruszać do zadanego punktu

Tabela 2.1: Zestaw scenariuszy dla funkcjonalności pierwszego etapu

terfejsu użytkownika. Miał on pozwalać na skonfigurowanie symulacji poprzez zbudowanie ścian, określenie punktów kluczowych oraz zdefiniowanie liczby uczestniczących jednostek. Ponadto należało przygotować odpowiednie kontrolki, które sterują symulacją. Szczegóły zostały przedstawione w tabeli 2.2.

Implementacja trzeciego etapu zakładała wdrożenie podstawowych elementów taktyk dla jednostek. Domyślnym zachowaniem terrorystów jest wędrowanie, które może być losowo wstrzymywane na kilka sekund. Domyślnym zachowaniem antyterrorystów jest podążanie w małych odstępach jeden za drugim, za wyjątkiem lidera, który podąża wytyczoną ścieżką do kolejnych punktów kluczowych. Ponadto implementacja zakładała wdrożenie systemu logów - generowanie wiadomości dotyczących kluczowych momentów w symulacji. Szczegóły zostały przedstawione w tabeli 2.3.

Czwarty etap implementacji projektu zakładał wprowadzenie elementu walki między jednostkami. Jednostka może zaatakować wrogą jednostkę wystrzeliwując pociski. Pociski trafiające w jednostkę zmniejszają jej liczbę punktów życia przeciw proporcjonalnie do odległości, jaką pokonał wystrzelony pocisk (symulacja utraty energii). Gdy liczba punktów życia danej jednostki spada poniżej zera, wtedy ta jednostka ginie. Ponadto postrzelona jednostka próbuje podążać do lokacji, z której padł strzał. Jeżeli w walce polegnie lider antyterrorystów, to jego funkcję (prowadze-

ETAP 2 - Interfejs
• interfejs pozwala na definiowanie ścian (dodawanie nowej, usuwanie ostatniej, usuwanie wszystkich) • interfejs pozwala na definiowanie punktu startowego / końcowego antyterrorystów (dodawanie, usuwanie) • interfejs pozwala na definiowanie punktów kluczowych (dodawanie nowego, usuwanie ostatniego, usuwanie wszystkich) • interfejs pozwala na definiowanie liczby antyterrorystów oraz liczby terrorystów • interfejs pozwala na rozpoczęcie symulacji • interfejs pozwala na zakończenie symulacji • interfejs pozwala na wstrzymanie symulacji

Tabela 2.2: Zestaw scenariuszy dla funkcjonalności drugiego etapu

ETAP 3 - Poruszanie się
• interfejs może wyświetlać logi dotyczące aktualnej symulacji • antyterrorysta będący liderem może poruszać się ścieżką po punktach kluczowych • antyterrorysta nie będący liderem może poruszać się w linii za poprzedzającym go antyterrorystą • terrorysta może wędrować • terrorysta może stać

Tabela 2.3: Zestaw scenariuszy dla funkcjonalności trzeciego etapu

ETAP 4 - Odczyt / zapis oraz walka
• interfejs pozwala na zapisanie bieżącej konfiguracji • interfejs pozwala na usunięcie konfiguracji • interfejs pozwala na wczytanie konfiguracji • jednostka może zaatakować wrogą jednostkę • jednostka może zginąć • zaatakowana jednostka sprawdza lokację, z której padł strzał • antyterrorysta może zostać liderem, jeśli ten zginie

Tabela 2.4: Zestaw scenariuszy dla funkcjonalności czwartego etapu

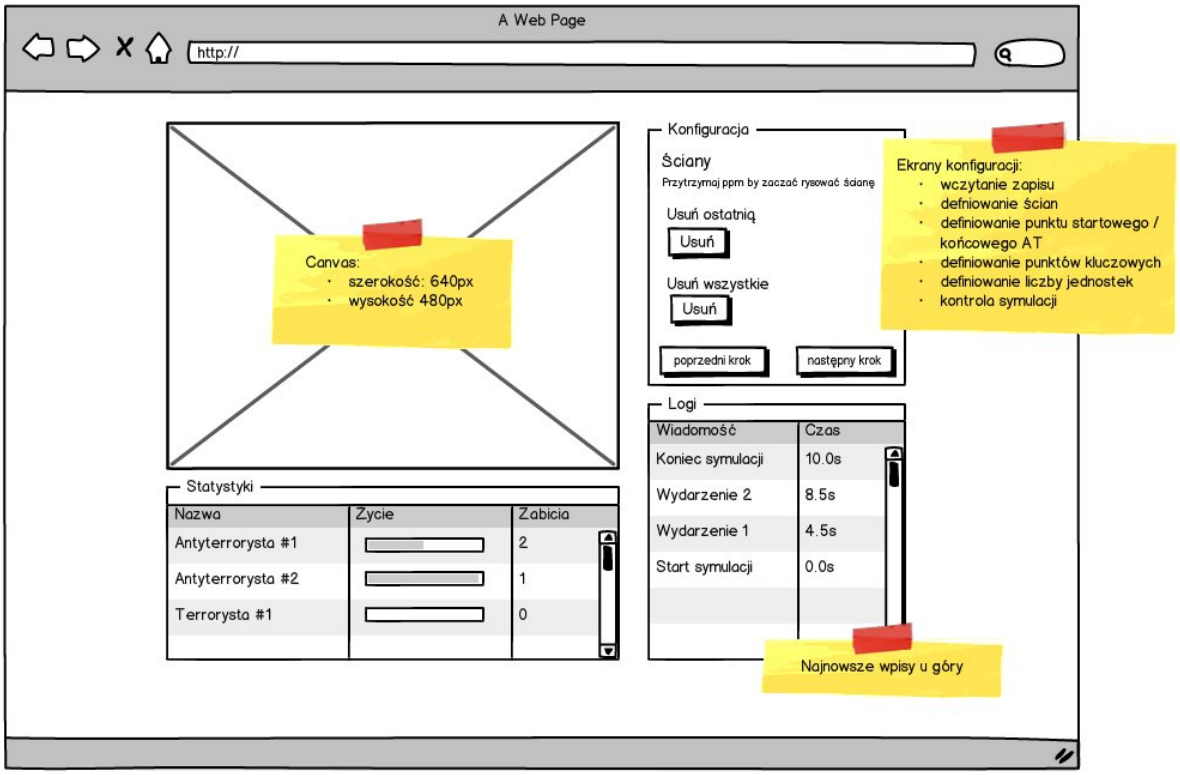
nie grupy) przejmuje następny antyterrorysta. Dodatkowo w tym etapie miały zostać zaimplementowane nowe funkcjonalności interfejsu, które pozwalają użytkownikowi na zapisywanie, usuwanie oraz wczytywanie wcześniej przygotowanej konfiguracji. Szczegóły zostały przedstawione w tabeli 2.4.

Ostatni etap implementacji składał się z mniejszych funkcjonalności, które miały swoje źródła w informacji zwrotnej uzyskanej podczas testów. Antyterroryści podążający za liderem wyposażeni są w detekcję kolizji ze ścianami, co pozwala na ich bezpieczne omijanie. Terroryści natomiast reagują na dźwięk wystrzału, kierując się do jego źródła. Dodatkowo interfejs użytkownika jest wzbogacony o statystyki jednostek, a gra symulacyjna posiada dźwięki odgrywane podczas startu symulacji oraz przy oddawaniu strzałów. Szczegóły zostały przedstawione w tabeli 2.5.

Prócz scenariuszy, użytecznym elementem specyfikacji był szkic interfejsu użytkownika. Podczas implementacji nanoszone były na niego nieznaczne zmiany. Ostateczna wersja szkicu jest zaprezentowana na rysunku 2.1.

ETAP 5 - Pożądane funkcjonalności
• antyterrorysta, nie będący liderem, może aktywnie omijać ściany • terrorysta reaguje na dźwięk wystrzału i trafienia (w określonym promieniu) podążając do jego źródła • interfejs może wyświetlać statystyki dla jednostek (pozostałe życie, liczba zabić) • aplikacja może odtwarzać dźwięki

Tabela 2.5: Zestaw scenariuszy dla funkcjonalności piątego etapu



Rysunek 2.1: Końcowy szkic interfejsu użytkownika

- System operacyjny: Windows, Linux lub MacOS
- Przeglądarka internetowa:
  - Chrome w wersji 15.0 lub wyższej
  - Firefox w wersji 4.0 lub wyższej
  - Internet Explorer w wersji 9.0 lub wyższej
  - Safari w wersji 5.1 lub wyższej

Tabela 2.6: Lista wymagań niefunkcjonalnych

2.2 Wymagania niefunkcjonalne

Zestaw wymagań niefunkcjonalnych dla gry symulacyjnej, będącej przedmiotem tej pracy dyplomowej, został przedstawiony w tabeli 2.6. Po zaimplementowaniu aplikacji, została pozytywnie zweryfikowana zgodność z przywołanymi przeglądarkami[2].

2.3 Słownik pojęć

Podczas sporządzania specyfikacji gry symulacyjnej, która jest przedmiotem tej pracy dyplomowej, niezbędne było dokładne zdefiniowanie niektórych wykorzystywanych pojęć. Poniżej znajduje się lista pojęć, uporządkowana alfabetycznie.

Antyterrorysta jest to jednostka, która w grze symulacyjnej oznaczona jest kolorem niebieskim. Celem antyterrorysty jest eliminacja wszystkich terrorystów

Antyterrorysta lider jest to antyterrorysta, który prowadzi oddział antyterrorystyczny. Reszta antyterrorystów podąża za liderem. Liderem jest wybierana pierwsza żyjąca jednostka na liście antyterrorystów

[2]do weryfikacji została użyta usługa http://www.browserstack.com/

Interfejs jest to część aplikacji, która służy do przygotowania konfiguracji, sterowania symulacją oraz prezentacji logów i statystyk jednostek

Jednostka jest to obiekt ruchomy, wykazujący pewne działanie taktyczne. W grze symulacyjnej jednostkami są terroryści i antyterroryści

Konfiguracja są to dane o położeniu ścian, punktu startowego / końcowego antyterrorystów oraz punktów kluczowych. Konfiguracja może być zapisana, wczytana lub usunięta z poziomu interfejsu

Obiekt ruchomy jest nim każda jednostka oraz każdy pocisk

Punkt kluczowy jest to punkt należący do uporządkowanego zbioru, na podstawie którego budowane są ścieżki dla antyterrorystów. Wokół punktów kluczowych tworzeni są terroryści na początku rozgrywki

Punkt startowy / końcowy jest to punkt, w którym są tworzeni i do którego wracają antyterroryści po przejściu przez wszystkie punkty kluczowe

Scena jest to część interfejsu ukazująca mapę lokacji oraz ruchome obiekty

Statystyki jest to część interfejsu ukazująca aktualny stan punktów życia oraz ilość zabić dla poszczególnych jednostek.

Symulacja jest to stan gry, w którym na scenie znajdują się jakiekolwiek jednostki

Terrorysta jest to jednostka, która w grze symulacyjnej oznaczona jest kolorem czerwonym. Celem terrorysty jest obrona terytorium przed antyterrorystami

Warstwa sceny jest to część sceny, do której aplikacja może przypisać obiekty w celu późniejszego renderowania.

Zamknięcie konfliktu jest to sytuacja, w której nie żyją wszyscy antyterroryści lub nie żyją wszyscy terroryści.

Rozdział 3

Projekt symulatora operacji antyterrorystycznych

3.1 Model obiektowy

Wykorzystywany podczas implementacji Javascript, jako skryptowy język programowania, nie jest językiem ściśle obiektowym (jak np. Java), lecz mimo wszystko umożliwia on pisanie aplikacji technikami programowania obiektowego. Do tego celu służy m. in. prototypowanie, które pozwala także na dziedziczenie przygotowywanych klas.

W grze symulacyjnej, będącej przedmiotem tej pracy dyplomowej, możemy wyróżnić trzy obiekty, które wywodzą się bezpośrednio z klasy obiektu javascriptowego oraz dziesięć klas, które dziedziczą atrybuty i metody z różnych klas kształtów, zawartych w bibliotece Kinetic.js. Nazwy metod, które są postrzegane jako prywatne dla danej klasy, rozpoczynają się od znaku podkreślenia "_"[1]. W prezentowanych tabelach zostały pominięte atrybuty i metody odziedziczone z innych klas. Definicje klas wchodzących w skład biblioteki Kinetic.js można znaleźć na stronie projektu[5].

[1] w Javascript'cie nie ma dedykowanego mechanizmu rozróżniania metod prywatnych i publicznych

Obiekt **Game** zawiera informacje dotyczące świata gry, tj. jego wymiarów, aktualnej konfiguracji oraz funkcjonujących jednostek. Stanowi on interfejs dla obiektów innych klas, przez który mogą one dostrzegać zmiany zachodzące w świecie gry. Metody zawarte w obiekcie Game pozwalają ma kontrolowanie symulacji. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.1.

Obiekt **GameControl** zawiera przede wszystkim metody, które wiążą interfejs z obiektem Game. Są tutaj zdefiniowane wszystkie metody wywoływane poprzez kliknięcia użytkownika w przyciski znajdujące się na interfejsie. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.2.

Obiekt **Sounds** zawiera definicję ścieżek do plików dźwiękowych wykorzystywanych w grze oraz metody pozwalające je odtwarzać i zatrzymywać. W grze symulacyjnej zdefiniowane są trzy dźwięki: dźwięk rozpoczynający symulację, wystrzały terrorystów oraz wystrzały antyterrorystów. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.3.

Obiekt klasy **Game.Map** rozszerza klasę Kinetic.Rect. Zawiera on referencje do elementów konfiguracji symulacji - ściany, punkty kluczowe. Ponadto obiekt mapy zawiera graf, którego węzły odpowiadają poszczególnym polom na mapie. Polami są kwadratowe kafelki, które są widoczne dzięki narzuceniu siatki na mapę. Wezeł w grafie może mieć jeden z dwóch stanów: otwarty lub zamknięty. Dzięki zastosowaniu grafu możliwe jest generowanie ścieżek dla poruszających się jednostek algorytmem A* (rozdział 3.2.2). Obiekt mapy posiada także metody pozwalające na serializowanie konfiguracji do formatu JSON[2] oraz do importu konfiguracji dostarczonej w takim formacie. Obsługa zdarzeń nad sceną, których źródłem jest urządzenie wskazujące, jest również zaimplementowana na obiekcie mapy. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.4.

Klasa **Game.Line** rozszerza klasę Kinetic.Line. Sama stanowi podstawę dla klas

[2]JavaScript Object Notation - tekstowy format wymiany danych, alternatywa dla XML

Game	Opis
width	szerokość sceny wyrażona w pikselach
height	wysokość sceny wyrażona w pikselach
mapDensity	wymiar jednego kafelka na mapie, na potrzeby reprezentacji grafowej
uiState	aktualny stan interfejsu graficznego, określa która strona konfiguracji jest aktualnie otwarta
stage	obiekt sceny zawierający poszczególne warstwy
map	obiekt mapy zawierający część informacji o konfiguracji
entities	warstwa sceny zawierająca istniejące w symulacji jednostki (terroryści i antyterroryści)
configObjects	warstwa sceny zawierająca obiekty wspomagające konfigurację (szkic punktu kluczowego, szkic punktu startowego itp.)
mapObjects	warstwa sceny zawierająca obiekty należące do mapy (ściany, punkty kluczowe itp.)
paused	zmienna logiczna informująca o włączeniu/wyłączeniu pauzy
antiterroristsCount	liczba antyterrorystów wynikająca z konfiguracji
terroristsCount	liczba terrorystów wynikająca z konfiguracji
keypointIndex	numer aktualnie realizowanego punktu kluczowego przez antyterrorystów
init	inicjalizuje aplikację tworząc scenę oraz warstwy
initMap	tworzy obiekt mapy
togglePause	przyłącza stan pauzy
startGame	rozpoczyna symulację
endGame	kończy symulację
getEntities	zwraca listę wszystkich jednostek istniejących w bieżącej symulacji
getAliveTerrorists	zwraca listę niezabitych terrorystów w bieżącej symulacji
getAliveAntiterrorists	zwraca listę niezabitych antyterrorystów w bieżącej symulacji
checkAliveEntities	sprawdza stan jednostek, a w razie zaistnienia zamknięcia konfliktu, tworzy odpowiedni wpis w logach
getNodeByPosition	zwraca węzeł w grafie na podstawie zadanych współrzędnych
_spawnTerrorists	tworzy obiekty terrorystów podczas startu symulacji
_spawnAntiterrorists	tworzy obiekty antyterrorystów podczas startu symulacji

Tabela 3.1: Obiekt gry - Game

GameControl	Opis
storagePrefix	stała zawierająca informację o prefiksie dla nazw zapisywanych konfiguracji
simStartTime	czas rozpoczęcia symulacji wyrażony w milisekundach
winMessage	wiadomość o ew. zwycięstwie jednej ze stron konfliktu
init	inicjalizuje interfejs, tworzy powiązania z obiektem Game
log	umieszcza wpis o zadanym tekście w logach
setWinMessage	tworzy wiadomość dotyczącą zwycięstwa jednej ze stron konfliktu
configs	zwraca listę wcześniej zapisanych konfiguracji
loadConfig	wczytuje wybraną konfigurację
saveConfig	zapisuje bieżącą konfigurację
removeConfig	usuwa wybraną konfigurację
startSim	powiązana z przyciskiem rozpoczynającym symulację
pauseSim	powiązana z przyciskiem wstrzymującym symulację
stopSim	powiązana z przyciskiem zatrzymującym symulację
removeLastWall	powiązana z przyciskiem usuwającym ostatnio utworzoną ścianę
clearWalls	powiązana z przyciskiem usuwającym wszystkie ściany
removeSpawnZone	powiązana z przyciskiem usuwającym punkt startowy / końcowy dla antyterrorystów
removeLastKeypoint	powiązana z przyciskiem usuwającym ostatnio utworzony punkt kluczowy
clearKeypoints	powiązana z przyciskiem usuwającym wszystkie punkty kluczowe
nextConfig	otwiera następną stronę konfiguracji
previousConfig	otwiera poprzednią stronę konfiguracji
changeUiState	otwiera zadaną stronę konfiguracji
clearEntitiesList	usuwa dane jednostek ze statystyk
createEntitiesList	dodaje dane jednostek do statystyk
updateStat	aktualizuje statystyki dla danej jednostki
_updateCursor	zmienia styl kursora myszy nad sceną
_updateNumberData	zmienia dane liczbowe o jednostkach w obiekcie Game
_updateConfigStatus	zwraca informację o ew. niekompletnej konfiguracji

Tabela 3.2: Obiekt kontroli gry - GameControl

Sounds	Opis
list	tablica asocjacyjna zawierająca ścieżki do plików dźwiękowych
instances	tablica asocjacyjna zwierająca instancje odgrywanych plików dźwiękowych
init	tworzy powiązanie z obiektem Game
play	rozpoczyna odtwarzanie danego dźwięku
stop	zatrzymuje odtwarzanie danego dźwięku

Tabela 3.3: Obiekt dźwięków gry - Sounds

Game.GridLine oraz Game.Wall. Klasa Game.Line zawiera metody sprawdzające przecięcia linii z inną linią oraz linii z okręgiem. Metody te są wykorzystywane w wielu metodach należących do obiektów ruchomych (klasa Game.Entity). Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.5.

Klasa **Game.Wall** rozszerza klasę Game.Line. Instancje tej klasy reprezentują ściany w grze symulacyjnej. Klasa zawiera dodatkowo metodę sprawdzającą poprawność budowanej ściany, która musi być linią poziomą lub pionową, a nie może być linią skośną. Ściany w grze stanowią dla jednostek jedyną przeszkodę, którą jednostki muszą omijać. Utworzone ściany mają swoje odzwierciedlenie na grafie w postaci niedostępnych dla jednostek węzłów. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.6.

Klasa **Game.Keypoint** rozszerza klasę Kinetic.Text. Instancje tej klasy reprezentują punkty kluczowe w grze symulacyjnej. Klasa zawiera metodę sprawdzającą poprawność tworzonego punktu kluczowego, który nie może leżeć w miejscu gdzie jest ściana. Podczas rozpoczynania symulacji, terroryści są tworzeni w losowo wybranych punktach kluczowych. Punkty te jednocześnie wytyczają trasę jaką muszą pokonać antyterroryści podczas przeprowadzanego szturmu. Antyterrorysta lider, po dotarciu do danego punktu kluczowego, wytycza bezkolizyjną ścieżkę do kolejnego punktu. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.7.

Game.Map	Opis
newWall	obiekt szkicu tworzonej ściany
graph	obiekt grafu, niezbędnego do wytyczania ścieżek
zone	obiekt punktu startowego / końcowego antyterrorystów
zoneDraft	obiekt szkicu punktu startowego / końcowego antyterrorystów
newKeypoint	obiekt szkicu punktu kluczowego
walls	tablica zawierająca istniejące ściany
keypoints	tablica zawierająca istniejące punkty kluczowe
init	inicjalizuje obiekt mapy
removeLastWall	usuwa ostatnio utworzoną ścianę
clearWalls	usuwa wszystkie ściany
removeLastKeypoint	usuwa ostatnio utworzony punkt kluczowy
clearKeypoints	usuwa wszystkie punkty kluczowe
removeZone	usuwa punkt startowy / końcowy antyterrorystów
serializeConfig	serializuje bieżącą konfigurację
importConfig	deserializuje dostarczoną konfigurację i tworzy nowe obiekty na jej podstawie
_bindEvents	inicjalizuje obsługę zdarzeń nad sceną
_buildGraph	inicjalizuje graf
_buildGrid	buduje siatkę nad sceną
_initWall	inicjalizuje nową ścianę
_updateWall	uaktualnia obiekt szkicu ściany
_addWall	dodaje utworzoną ścianę do listy ścian
_updateWallOnGraph	oznacza węzły grafu, które pokrywa zadana ściana
_buildZoneDraft	inicjalizuje punkt startowy / końcowy antyterrorystów
_buildKeypoint	inicjalizuje punkt kluczowy
_showDraftZone	pokazuje szkic punktu startowego / końcowego antyterrorystów gdy ukryty
_hideDraftZone	ukrywa szkic punktu startowego / końcowego antyterrorystów gdy widoczny
_setZone	ustanawia punkt startowy / końcowy antyterrorystów
_updateDraftZone	uaktualnia położenie szkicu punktu startowego / końcowego antyterrorystów
_addKeypoint	dodaje utworzony punkt kluczowy do listy punktów kluczowych
_showNewKeypoint	pokazuje szkic punktu kluczowego gdy ukryty
_hideNewKeypoint	ukrywa szkic punktu kluczowego gdy widoczny
_updateNewKeypoint	uaktualnia położenie szkicu punktu kluczowego

Tabela 3.4: Klasa mapy - Game.Map

Game.Line	Opis
	klasa ta zawiera wyłącznie atrybuty dziedziczone
init	inicjalizuje obiekt linii
getStartPoint	zwraca współrzędne punktu początkowego linii
getEndPoint	zwraca współrzędne punktu końcowego linii
getVecStartPoint	zwraca punkt początkowy linii w postaci wektora
getVecEndPoint	zwraca punkt końcowy linii w postaci wektora
setStartPoint	ustawia punkt początkowy linii
setEndPoint	ustawia punkt końcowy linii
getIntersectionPointWithLine	zwraca współrzędne punktu przecięcia dwóch linii
getVecIntersectionPointWithSphere	zwraca punkt przecięcia linii i okręgu w postaci wektora
getVecIntersectionPoint	zwraca punkt przecięcia dwóch linii w postaci wektora
getNormals	zwraca wektory normalne dla danej linii
_getClosestPointOnLine	zwraca najbliższy punkt na linii do zadanego punktu

Tabela 3.5: Klasa linii - Game.Line

Game.Wall	Opis
valid	zawiera informację czy ściana jest poprawna
init	inicjalizuje obiekt ściany
setEndPoint	nadpisana metoda klasy Game.Line, dodatkowo wywołuje metodę _validate
isVertical	zwraca informację czy linia jest pionowa
isHorizontal	zwraca informację czy linia jest pozioma
_validate	sprawdza poprawność zbudowanej linii

Tabela 3.6: Klasa ściany - Game.Line

Game.Keypoint	Opis
valid	zawiera informację czy punkt kluczowy jest poprawny
init	inicjalizuje obiekt punktu kluczowego
updatePosition	uaktualnia położenie punktu kluczowego
_validate	sprawdza poprawność tworzonego punktu kluczowego

Tabela 3.7: Klasa punktu kluczowego - Game.Keypoint

Klasa **Game.Entity** rozszerza klasę Kinetic.Image. Sama stanowi podstawę dla klas Game.Terrorist, Game.Antiterrorist oraz Game.Bullet. Klasa ta posiada metody pozwalające wyliczać wektor prędkości dla algorytmów poruszania się oraz sprawdzać kolizję z innymi obiektami. Część atrybutów i metod jest tutaj odpowiedzialna za realizację wspólnych dla terrorystów i antyterrorystów taktyk. Dzięki implementacji metody *update*, położenie obiektów na scenie jest stale aktualizowane. Opis atrybutów znajduje się w tabeli 3.8, natomiast opis zaimplementowanych metod znajduje się w tabeli 3.9.

Klasa **Game.Antiterrorist** rozszerza klasę Game.Entity. Instancje tej klasy reprezentują antyterrorystów w grze symulacyjnej. Najważniejszą metodą zawartą w tej klasie jest *think*. Decyduje ona o działaniach antyterrorysty poprzez umożliwienie zmiany stanów. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.10.

Klasa **Game.Terrorist** rozszerza klasę Game.Entity. Instancje tej klasy reprezentują terrorystów w grze symulacyjnej. Podobnie jak w klasie Game.Antiterrorist, kluczową rolę w tej klasie odgrywa metoda *think*. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.11. Dodatkowo w rozdziale 4 jest przedstawiony diagram przejść międzystanowych, który w ilustruje taktykę charakterystyczną dla działań antyterrorystów i terrorystów.

Klasa **Game.Bullet** rozszerza klasę Game.Entity. Instancje tej klasy reprezentują pociski w grze symulacyjnej. Podczas przemieszczania się pocisku, sprawdzane jest czy nie trafił on w ścianę lub jednostkę. Symulowany odgłos wystrzału oraz trafienia może przyciągać uwagę terrorystów znajdujących się w określonej odległości od pocisku. Im dłużej pocisk się porusza, tym mniejszą posiada energię, która decyduje o ew. ranach zadanych jednostce. Opis atrybutów oraz zaimplementowanych metod znajduje się w tabeli 3.12.

Game.Entity	Opis atrybutów
imageSrc	ścieżka do pliku z reprezentacją graficzną
maxSpeed	maksymalna prędkość obiektu
velX	współrzędna X wektora prędkości
velY	współrzędna Y wektora prędkości
tarX	współrzędna X celu
tarY	współrzędna Y celu
rayLine	obiekt linii, która sprawdza możliwość wystąpienia kolizji
groupIndex	numer obiektu w danej stronie konfliktu
isAlive	zawiera informację czy obiekt żyje
dieAlpha	stopień przezroczystości, jaka jest stosowana, gdy obiekt ginie
speed	aktualna prędkość
avoidDistance	dystans, jaki obiekt powinien zachowywać od kolidujących obiektów
lookAhead	długość linii, sprawdzającej możliwość wystąpienia kolizji
arrivePrecision	dokładność, z jaką się określa czy obiekt dotarł do celu
targetEntity	obiekt ruchomy, który jest aktualnym celem
watchedEntity	obiekt ruchomy, który jest obserwowany, ale nie atakowany
healthPoints	liczba punktów życia
healthPointsMax	liczba punktów życia, jakie obiekt posiada po utworzeniu
collisionRadius	promień okręgu wytyczającego strefę kolizyjną obiektu
kills	liczba zabić dokonanych przez obiekt
nodeIndex	indeks aktualnie odwiedzanego węzła na ścieżce
path	ścieżka reprezentowana przez tablicę węzłów
currentState	nazwa aktualnego stanu obiektu
checkLocationTimeMax	maksymalny czas, jaki może zostać poświęcony na sprawdzenie lokacji
checkLocationTime	aktualny czas, jaki pozostał na sprawdzenie lokacji
sightDistance	zasięg wzroku obiektu
name	nazwa typu obiektu
enemyName	nazwa wrogiego typu obiektu

Tabela 3.8: Atrybuty klasy obiektu ruchomego - Game.Entity

Game.Entity	Opis metod
init	inicjalizuje obiekt ruchomy
setTarget	ustawia cel wg zadanych współrzędnych
setTargetEntity	ustawia inny obiekt jako swój cel
currentTargetEntity	zwraca aktualny obiekt będący celem
unsetTargetEntity	usuwa przypisanie celu, który jest obiektem
updateTargetEntity	uaktualnia wiedzę o położeniu celu
setVelocity	ustawia wektor prędkości
hasVelocity	zwraca informację czy obiekt się porusza
getVecPosition	zwraca aktualną pozycję w postaci wektora
getVecVelocity	zwraca aktualną prędkość w postaci wektora
getVecTarget	zwraca aktualną pozycję celu w postaci wektora
update	przesuwa obiekt o zadany wektor prędkości oraz aktualizuje orientację
changeState	zmienia stan obiektu na zadany
die	uśmierca obiekt
setRandomPositionInCircle	wybiera losową pozycję obiektu wokół określonego okręgu
isInCollision	sprawdza czy obiekt nie koliduje ze ścianą lub inną istniejącą jednostką
seek	aktualizuje wektor prędkości w kierunku do celu
flee	aktualizuje wektor prędkości w kierunku przeciwnym do celu
stop	zatrzymuje obiekt
arrived	sprawdza czy obiekt dotarł do celu
checkForCollision	sprawdza czy obiekt nie koliduje ze ścianą
closestSeenOpponent	zwraca najbliższego przeciwnika w zasięgu wzroku
takeDamage	zadaje rany obiektowi poprzez odbiór punktów życia
watchForEnemy	metoda pozwalająca na obserwowanie przeciwnika i po określonym czasie przejście do ataku
attack	atakowanie przeciwnika poprzez oddawanie strzałów co określony interwał czasowy
calculatePath	wyliczanie ścieżki do zadanego celu
checkLocation	metoda pozwalająca na przejście po ścieżce do wcześniej wytyczonego celu
setCheckLocation	metoda określająca cel, do którego należy dotrzeć wg wytyczonej ścieżki
_logDeath	tworzenie wpisu w logach o śmierci obiektu
_updateCollisionRay	uaktualnia położenie linii, która sprawdza możliwość wystąpienia kolizji
_calculateVelocity	wyliczanie wektora prędkości

Tabela 3.9: Metody klasy obiektu ruchomego - Game.Entity

Game.Antiterrorist	Opis
reactionTimeMax	czas reakcji przejścia z obserwowania do ataku
reactionTime	aktualny czas pozostały do przejścia do ataku
shootInterval	czas pomiędzy kolejnymi wystrzałami
shootTime	aktualny czas pozostały do wystrzału
followDistance	odległość, w jakiej antyterrorysta podąża za poprzednikiem
isLeader	zawiera informację, czy antyterrorysta jest liderem
keypointIndex	numer aktualnie realizowanego punktu kluczowego przez lidera antyterrorystów
think	metoda odpowiedzialna za realizację taktyk
followEntity	podążanie w linii za poprzednikiem
followPath	podążanie do następnego punktu kluczowego po ścieżce
followExtraction	podążanie do punktu startowego / końcowego antyterrorystów
avoid	omijanie napotkanej ściany poprzez wytyczenie ścieżki
changeToDefaultState	zmiana stanu do domyślnego (dla antyterrorystów jest to *followEntity*)
_reactOnDamage	reakcja na postrzał

Tabela 3.10: Klasa antyterrorysty - Game.Antiterrorist

Game.Terrorist	Opis
reactionTimeMax	czas reakcji przejścia z obserwowania do ataku
reactionTime	aktualny czas pozostały do przejścia do ataku
shootInterval	czas pomiędzy kolejnymi wystrzałami
shootTime	aktualny czas pozostały do wystrzału
wanderCircleDistance	odległość środka okręgu wyznaczającego kurs wędrówki od terrorysty
wanderRadius	promień okręgu wyznaczającego kurs wędrówki
wanderRate	zakres wahania kierunku wędrówki
wanderOrientation	orientacja kierunku wędrówki
standingProbability	prawdopodobieństwo przejścia do stanu postoju
standingTimeMax	maksymalny czas, jaki może trwać pojedynczy postój
standingTime	czas pozostały do zakończenia aktualnego postoju
think	metoda odpowiedzialna za realizację taktyk
stand	metoda odpowiedzialna za sprawdzanie czy postój ma nadal trwać
wander	poruszanie się zgodnie z kierunkiem wędrówki
avoid	omijanie napotkanej ściany poprzez skierowanie terrorysty do punktu wyznaczanego przez normalną ściany
changeToDefaultState	zmiana stanu do domyślnego (dla terrorystów jest to *wander*)
_wantToStand	sprawdzenie czy terrorysta na wykonać postój
_reactOnDamage	reakcja na postrzał

Tabela 3.11: Klasa terrorysty - Game.Terrorist

Game.Bullet	Opis
shooter	referencja do obiektu jednostki, która wystrzeliła pocisk
energy	energia, jaką aktualnie posiada pocisk
bulletRange	zasięg pocisku
attentionRange	promień w jakim wystrzelony pocisk jest słyszalny
move	metoda odpowiedzialna za ruch pocisku i sprawdzenie ew. trafienia
_drawTerroristsAttention	zmiana stanu terrorystów będących w zasięgu słyszalności wystrzału lub trafienia
_playSound	odtwarzanie dźwięku wystrzału

Tabela 3.12: Klasa pocisku - Game.Bullet

Poza wymienionymi klasami, w grze symulacyjnej zdefiniowane są jeszcze klasy, które wyłącznie nadpisują konfigurację wyświetlania kształtu. Należą do nich:

- Game.Zone - dziedziczy z klasy Kinetic.Circle. Instancje tej klasy reprezentują punkt startowy / początkowy antyterrorystów
- Game.GridLine - dziedziczy z klasy Kinetic.Line. Instancje tej klasy reprezentują siatkę narzuconą na scenę

3.2 Opis algorytmów

W tym rozdziale zostaną przedstawione wybrane algorytmy, jakie są wykorzystywane w grze symulacyjnej, będącej przedmiotem tej pracy dyplomowej. Słowny opis jest uzupełniony pseudokodami lub implementacją w języku Javascript.

3.2.1 Myślenie i poruszanie się jednostek

Antyterroryści i terroryści w grze symulacyjnej posiadają zaimplementowaną metodę *think* pozwalającą na realizację działań zapisanych w konkretnych stanach (listing 3.13 oraz 3.14). Prócz wywołania metod odpowiadającym konkretnym stanom, wywoływane są także metody *watchForEnemy* oraz *checkForCollision*. Ta pierwsza pozwala jednostce na obserwowanie otoczenia w poszukiwaniu przeciwników, natomiast druga na bezpieczne omijanie ścian. Metody odpowiadające za realizację poszczególnych stanów podejmują najczęściej decyzję, gdzie leży następny cel jednostki.

Gdy cel jest już zdefiniowany, to wykonywane są algorytmy niższego poziomu, odpowiedzialne za obliczenie wektora prędkości oraz prędkości, z jaką ma poruszać się jednostka. Takimi algorytmami są *seek* (ruch w kierunku celu) oraz *flee* (ruch w przeciwnym kierunku do celu). Metoda think jest wywoływana wewnątrz metody

```
1     think: function(){
2         this.watchForEnemy();
3         switch(this.currentState) {
4             case 'idle': break;
5             case 'init': this.setup(); break;
6             case 'follow entity': this.followEntity(); break;
7             case 'follow path': this.followPath(); break;
8             case 'follow extraction': this.followExtraction(); break;
9             case 'check location': this.checkLocation(); break;
10            case 'attack': this.attack(); break;
11            default: this.changeToDefaultState(); break;
12        }
13        if (this.avoiding) this.wanderOrientation = this.getRotation();
14        this.avoiding = this.checkForCollision();
15    }
```

Tabela 3.13: Metoda think w klasie Game.Antiterrorist

```
1     think: function(){
2         this.watchForEnemy();
3         switch(this.currentState) {
4             case 'idle': break;
5             case 'init': this.setup(); break;
6             case 'stand': this.stand(); break;
7             case 'wander': this.wander(); break;
8             case 'check location': this.checkLocation(); break;
9             case 'attack': this.attack(); break;
10            default: this.changeToDefaultState(); break;
11        }
12        if (this.avoiding) this.wanderOrientation = this.getRotation();
13        this.avoiding = this.checkForCollision();
14    }
```

Tabela 3.14: Metoda think w klasie Game.Terrorist

```
1     update: function(frame) {
2         if (!Game.paused && this.isAlive) {
3             this.think();
4             if (this.hasVelocity()) {
5                 this._updateCollisionRay();
6                 var pos = this.getVecPosition().add(this.getVecVelocity
                      ().multiply(frame.timeDiff * this.speed));
7                 this.setPosition(pos.e(1),pos.e(2));
8                 var rot = Math.atan2(-this.getVecVelocity().e(1), this.
                      getVecVelocity().e(2));
9                 this.setRotation(rot);
10            }
11        }
12    }
```

Tabela 3.15: Metoda update w klasie Game.Entity

update (listing 3.15), która służy do zmiany pozycji obiektu na scenie. Po wykonaniu metody *think* zmieniana jest aktualna pozycja obiektu o wyliczony wektor prędkości, natomiast orientacja obiektu jest uaktualniana, by była zgodna z wektorem prędkości. Aktualizacja pozycji i orientacji nie następuje, gdy gra jest w trybie pauzy lub dany obiekt ruchomy już nie żyje.

Metoda update jest wywoływana w anonimowej funkcji, przypisanej do zdarzenia zmiany klatki animacji[3]. Obsługa tego zdarzenia stanowi główną pętlę aplikacji. Co klatkę następuje iteracja po wszystkich warstwach przypisanych do sceny (listing 3.16). Dla każdej warstwy interujemy po wszystkich obiektach, jakie zostały dodane do danej warstwy. Jeżeli obiekt posiada zdefiniowaną metodę update (co oznacza, że jego klasa dziedziczy z Game.Entity), to jest ona wykonywana. Po przetworzeniu wszystkich obiektów w danej warstwie, wykonywana jest metoda draw, która renderuje ponownie warstwę na scenie.

[3]definicja obsługi tego zdarzenia znajduje się w metodzie Game.init()

```
1     this.stage.onFrame(function(frame) {
2           for(var layerIndex in self.stage.getChildren()) {
3               var layer = self.stage.getChildren()[layerIndex];
4               for (var objectIndex in layer.getChildren()) {
5                   var object = layer.getChildren()[objectIndex];
6                   if(object.update) object.update(frame);
7               }
8               layer.draw();
9           }
10    });
```

Tabela 3.16: Rysowanie klatek animacji

3.2.2 Wyznaczanie ścieżki - A*

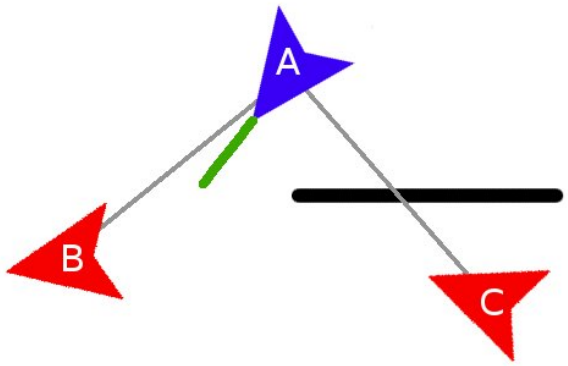
Wyznaczanie ścieżek bezkolizyjnych (uwzględniających położenie ścian na mapie) jest realizowane poprzez algorytm A*, który na bazie grafu jest w stanie odnaleźć drogę między dwoma zadanymi węzłami. W grze symulacyjnej graf zawiera węzły, które mogą mieć przypisany jeden z dwóch stanów: otwarty lub zamknięty. Wyliczona ścieżka nigdy nie prowadzi przez węzły zamknięte.

Algorytm został opisany przez Petera Harta, Nilsa Nilssona oraz Bertrama Raphaela w 1968 roku i początkowo nosił nazwę *A*. Jednakże ze względu na zastosowanie heurystyki, polegającej na przeszukiwaniu grafu z pierwszeństwem analizy węzłów obiecujących (tj. tych, które znajdują się bliżej węzła docelowego), nadano algorytmowi nazwę *A**.

W grze symulacyjnej wykorzystywana jest javascript'owa implementacja algorytmu A*, przygotowana przez Briana Grinsteada w formie biblioteki javascript-astar[8]. Inicjalizacja grafu oraz przykład wyszukiwania ścieżki zostały przedstawione w listingu 3.17. W definicji grafu *1* oznacza węzeł otwarty, a *0* węzeł zamknięty. Podczas definiowania przez użytkownika konfiguracji symulacji, w miejscu gdzie jest postawiona ściana węzły grafu zmieniają swój typ na zamknięty. Wynikiem wyszukiwania jest uporządkowana lista węzłów, jakie jednostka musi odwiedzić w drodze

```
1   var graph = new Graph([
2     [1,1,1,1,1,1,1,1,1,1],
3     [1,1,1,1,0,0,1,1,1,1],
4     [1,1,1,1,0,0,1,1,1,1],
5     [1,1,1,1,0,0,1,1,1,1],
6     [1,1,1,1,1,1,1,1,1,1],
7   ]);
8   var start = graph.nodes[0][0];
9   var end = graph.nodes[9][4];
10  var result = astar.search(graph.nodes, start, end);
```

Tabela 3.17: Inicjalizacja grafu 10x5 oraz wyszukiwanie ścieżki między węzłami



Rysunek 3.1: Zauważanie przeciwnika: Najbliższym przeciwnikiem jednostki A jest jednostka B. Jednostka C nie jest analizowana, ponieważ znajduje się za ścianą. Zielona linia to linia sprawdzająca ew. kolizje dla jednostki A

do celu. Algorytm wyznaczania scieżki jest zaimplementowany w metodzie *calculatePath* klasy Game.Entity.

3.2.3 Zauważanie przeciwnika

Algorytm zauważania przeciwnika jest zaimplementowany w metodzie *closestSeenOpponent* klasy Game.Entity. Zwraca on referencję do najbliższego przeciwnika, będącego w zasięgu wzroku. Algorytm iteruje po liście przeciwników, dokonując szeregu sprawdzeń tylko dla tych jednostek, które jeszcze żyją. Tzw. *dłuższy dystans*

```
1   cel = null
2   dystans_do_celu = ja.zasieg_wzroku
3   DLA KAZDEGO przeciwnik z lista_przeciwnikow WYKONUJ
4     JEZELI przeciwnik.nie_zyje TO wykonaj_nastepna_iteracje
5     dluzszy_dystans = oblicz_dystans (ja.pozycja, przeciwnik.pozycja)
6     krotszy_dystans = oblicz_dystans (ja.koniec_promienia_kolizji,
          przeciwnik.pozycja)
7     JEZELI krotszy_dystans < dluzszy_dystans ORAZ krotszy_dystans <
          dystans_do_celu TO
8       JEZELI lista_scian_na_drodze (ja.pozycja, przeciwnik.pozycja)
            JEST PUSTA TO
9         cel = przeciwnik
10        dystans_do_celu = krotszy_dystans
11  ZWROC cel
```

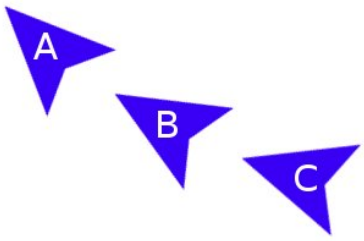
Tabela 3.18: Pseudokod algorytmu zauważania przeciwnika

obliczany jest między pozycją obserwującej jednostki a pozycją potencjalnego przeciwnika. *Krótszy dystans* obliczany jest między końcem linii sprawdzającej ew. kolizje dla jednostki obserwującej a pozycją potencjalnego przeciwnika (rysunek 3.1). Jeżeli *krótszy dystans* jest rzeczywiście mniejszy od *dłuższego dystansu*, to oznacza to, że przeciwnik jest przed jednostką obserwującą[4]. Jeżeli dystans pomiędzy jednostkami jest większy od zasięgu wzroku lub jest większy niż zapamiętany dystans do aktualnego celu, to taki przeciwnik jest ignorowany. Jeżeli jednak odległości są mniejsze, a przeciwnik nie znajduje się za jakąkolwiek ścianą, to oznaczamy go za cel i zapamiętujemy nowy, aktualny dystans do celu. Pseudokod algorytmu jest zapisany w listingu 3.18.

3.2.4 Podążanie jednostek w linii

Antyterroryści w grze symulacyjnej poruszają się w szyku liniowym (rysunek 3.2). Taka kolumna zaatakowana bezpośrednio od przodu lub od tyłu ma najmniejszą siłę ogniową. Linia zaatakowana od boku posiada bardzo dużą siłę ogniową, bowiem

[4]jednostki będące za plecami jednostki obserwującej są ignorowane



Rysunek 3.2: Podążanie jednostek w linii: Jednostka A jest liderem. Z jednostką A, w określonym odstępie porusza się jednostka B, natomiast za jednostką B porusza się jednostka C

antyterroryści nie zasłaniają sobie na wzajem celu. Algorytm podążania w linii jest zaimplementowany w metodzie *followEntity* klasy Game.Antiterrorist. Próbuje on dla danej jednostki znaleźć przyjazną jednostkę, która ją poprzedza i nie zginęła. Jeżeli taka jednostka nie zostanie znaleziona, to oznacza to, że dana jednostka jest liderem i należy zmienić jej stan na *follow path*. W przeciwnym wypadku dana jednostka wylicza i podąża do współrzędnych celu, które są iloczynem odległości podążania oraz różnicy aktualnej pozycji znalezionego sprzymierzeńca i jego wektora prędkości. Pseudokod algorytmu jest zapisany w listingu 3.19.

3.2.5 Atakowanie jednostki

Atak na wrogą jednostkę jest poprzedzany obserwacją, która przeprowadzana jest niezależnie od stanu, w jakim znajduje się aktualnie jednostka. Z tego algorytmu korzystają zarówno antyterroryści, jak i terroryści. Jest on zaimplementowany w metodzie *watchForEnemy* klasy Game.Entity. Na początku metoda korzysta z algorytmu zauważania przeciwnika (rozdział 3.2.3). Jeżeli dana jednostka nie widzi w danym momencie potencjalnego przeciwnika, a obecnie znajduje się w stanie *attack*, to następuje przejście do stanu domyślnego tej jednostki. Jednak gdy przeciwnik został

```
1   indeks = ja.indeks_w_grupie
2   POWTARZAJ
3     indeks = indeks - 1
4     sprzymierzeniec = sprzymierzency[indeks]
5   DOPOKI (ISTNIEJE(sprzymierzeniec) ORAZ sprzymierzeniec.nie_zyje)
6   JEZELI (NIE_ISTNIEJE(sprzymierzeniec)) TO
7     ja.jestLiderem = PRAWDA
8     ja.zmien_stan('follow path')
9   WPP
10    ja.jednostka_cel = sprzymierzeniec
11
12  ja.pozycja_celu = (sprzymierzeniec.wektor_pozycji - sprzymierzeniec.
        wektor_predkosci) * ja.odleglosc_podazania
13  ja.seek()
```

Tabela 3.19: Pseudokod algorytmu podążania za jednostką

znaleziony, ale nie był wcześniej obserwowany, to dana jednostka rozpoczyna jego obserwację ustawiając czas do ataku na zdefiniowany w parametrach czas reakcji. Gdy jednak znaleziony przeciwnik jest już obserwowany i upłynie czas do ataku, to dana jednostka przechodzi do stanu *attack*, jeżeli tylko nie ma przed sobą żadnych sprzymierzeńców stojących na linii ognia. Pseudokod algorytmu jest zapisany w listingu 3.20.

Realizacja ataku polega na wystrzeliwaniu pocisku co określony interwał czasowy. Jeżeli upływa czas do kolejnego strzału, to tworzona jest nowa instancja pocisku, którego pozycja i orientacja są zgodne z odpowiednikami u strzelca. Obiekt pocisku wykonuje metodę *move*, która sprawdza, czy pocisk nie trafił w ścianę lub jednostkę. W tym drugim przypadku zadawane są obrażenia zgodnie z energią, jaką posiadał pocisk w momencie trafienia. Pseudokod algorytmu ataku jest zapisany w listingu 3.23.

```
1   najblizszy_przeciwnik = znajdz_najblizszego_przeciwnika();
2   JEZELI (NIE_ISTNIEJE(najblizszy_przeciwnik)) TO
3     JEZELI (ja.aktualny_stan == 'attack')
4       ja.pozycja_celu = null
5       ja.zmien_stan_na_domyslny()
6   WPP
7     JEZELI (najblizszy_przeciwnik != ja.obserwowany_przeciwnik) TO
8       ja.obserwowany_przeciwnik = najblizszy_przeciwnik
9       ja.czas_do_ataku = ja.czas_reakcji
10    WPP
11      ja.czas_do_ataku = ja.czas_do_ataku - 1
12      JEZELI (ja.czas_do_ataku < 0) TO
13        JEZELI lista_sprzymierzencow_na_drodze (ja.pozycja,
              najblizszy_przeciwnik.pozycja) JEST PUSTA TO
14          ja.jednostka_cel = najblizszy_przeciwnik
15          ja.zmien_stan('attack')
```

Tabela 3.20: Pseudokod algorytmu obserwowania wroga

```
1   ja.uaktualnij_pozycje_cel()
2   ja.seek()
3   JEZELI (ja.czas_do_strzalu < 0) TO
4     ja.czas_do_strzalu = ja.czas_miedzy_strzalami
5     strzelec = ja
6     UTWROZ('pocisk', strzelec)
7   ja.czas_do_strzalu = ja.czas_do_strzalu - 1;
```

Tabela 3.21: Pseudokod algorytmu atakowania wroga

3.2.6 Sprawdzanie lokacji

W szczególnych przypadkach jednostki mogą przejść do stanu *check location*, którego definicja nakazuje przejście jednostki do zadanego miejsca na mapie. Przed przejściem do tego stanu jest ustawiany cel i obliczana bezkolizyjna ścieżka do niego (listing 3.22). Dla antyterrorystów przejście do stanu *check location* jest wywoływane w następujących sytuacjach:

- antyterrorysta nie będący liderem natrafi na ścianę, która odgradza go od antyterrorysty poprzednika; wyznaczanym celem jest pozycja poprzednika na mapie
- antyterrorysta otrzyma trafienie; wyznaczanym celem jest pozycja strzelca na mapie

Terroryści również korzystają z implementacji stanu *check location*. W ich przypadku stan ten jest wywoływany gdy:

- terrorysta usłyszy odgłos wystrzału lub trafienia; wyznaczanym celem jest pozycja pocisku w momencie wystrzału lub trafienia
- antyterrorysta otrzyma trafienie; wyznaczanym celem jest pozycja strzelca na mapie

Jednostki posiadają ograniczony czas na przejście do zadanej lokacji (listing 3.23). Antyterroryści posiadają bardzo niską wartość maksymalnego czasu na sprawdzenie lokacji, gdyż priorytetem dla nich jest wykonanie planu poruszając się w grupie, gdzie posiadają większą siłę ognia. Warunkiem wyścia ze stanu jest upłynięcie czasu na sprawdzenie lokacji lub dotarcie do niej. Jednostka opuszczająca stan *check location* przechodzi do własnego domyślnego stanu.

```
1   ja.czas_na_sprawdzenie_lokacji = ja.
        maksymalny_czas_na_sprawdzenie_lokacji
2   ja.wytycz_sciezke_do(cel)
3   ja.zmien_stan('check location')
```

Tabela 3.22: Pseudokod algorytmu sprawdzania lokacji (metoda setCheckLocation)

```
1   wezel = sciezka[ja.indeks_wezla]
2   JEZELI (ja.czas_na_sprawdzenie_lokacji < 0 LUB NIE_ISTNIEJE(wezel))
        TO
3     ja.zmien_stan_na_domyslny()
4   WPP
5     ja.pozycja_celu = wspolrzedne_wezla(wezel)
6     JEZELI (ja.dotarl_do_celu) TO
7       ja.indeks_wezla = ja.indeks_wezla + 1
8     ja.czas_na_sprawdzenie_lokacji = ja.czas_na_sprawdzenie_lokacji - 1
9     ja.seek()
```

Tabela 3.23: Pseudokod algorytmu sprawdzania lokacji (metoda checkLocation)

3.3 Parametryzacja

Antyterroryści i terroryści w grze symulacyjnej współdzielą część metod oraz parametrów, które dziedziczą z klasy Game.Entity. Jednakże część parametrów wpływających na przebieg symulacji ma u nich różne wartości. Stroną faworyzowaną są tutaj antyterroryści, u których zakłada się, że są lepiej wyposażeni (np. większa szybkostrzelność broni, kamizelki kuloodporne) oraz są lepiej wyszkoleni (np. szybciej potrafią zareagować na pojawienie się przeciwnika). Dzięki takiej parametryzacji mniej liczebny odział antyterrorystów ma wyrównane szanse walki z liczniejszymi terrorystami. Tabela 3.24 przedstawia porównanie wartości wybranych parametrów jednostek.

Parametr	Antyterrorysta	Terrorysta
healthPoints (punkty życia)	150	100
reactionTimeMax (czas reakcji)	20	25
shootInterval (czas między strzałami)	10	15
checkLocationTimeMax (czas na sprawdzenie lokacji)	100	1000
maxSpeed (maksymalna prędkość)	0.025	0.020

Tabela 3.24: Porównanie parametrów antyterrorysty i terrorysty

Rozdział 4

Sztuczna inteligencja - taktyki

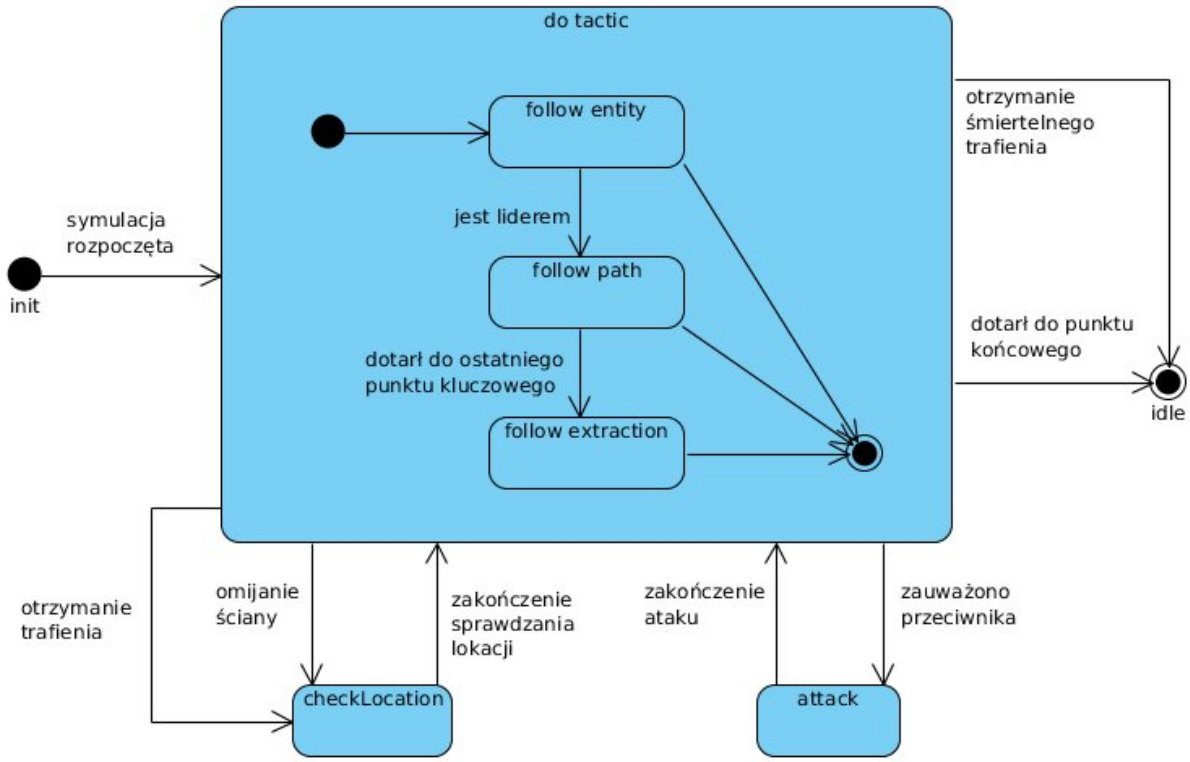
W grze symulacyjnej, będącej przedmiotem niniejszej pracy dyplomowej, terroryści oraz antyterroryści posiadają sztuczną inteligencję[1], która pozwala im na podejmowanie decyzji oraz poruszanie się. Interfejsem, z którego jednostki uczestniczące w symulacji czerpią wiedzę o świecie gry, jest obiekt Game.

4.1 Taktyka antyterrorystów

Antyterroryści posiadają strategię, która nakazuje im posiadanie lidera przez cały czas trwania symulacji. Lider antyterrorystów jest jednostką, za którą w szyku poruszają się pozostali antyterroryści. Jeżeli lider zginie, to natychmiastowo wybierany jest nowy lider, a działania grupy są kontynuowane.

Antyterrorysta posiada skończony zbiór stanów (rysunek 4.1), które odzwierciedlają podjęte przez niego decyzje i definiują jego działania. Po zainicjalizowaniu obiektu antyterrorysty przechodzi on do stanu *follow entity*, który pozwala mu podążać za swoimi poprzednikami. Jest to domyślny stan, do którego antyterrorysta może wrócić ze stanów, do których przeszedł na podstawie zdarzenia. Jeżeli anty-

[1] charakterystyka modelu SI została przedstawiona w rozdziale 1.3



Rysunek 4.1: Diagram przejść międzystanowych antyterrorysty

terrorysta jest liderem, to następuje natychmiastowe przejście do stanu *follow path*, które definiuje konieczność poruszania się po wyznaczonej ścieżce do następnego punktu kluczowego. Wraz z dotarciem do danego punktu kluczowego, wyznaczana jest ściseżka do następnego punktu kluczowego. Jeżeli antyterrorysta lider dotarł do ostatniego punktu kluczowego, to zmienia on swój stan na *follow extraction*, który nakazuje jednostce poruszanie się po ścieżce do punktu startowego / końcowego antyterrorystów. Po dotarciu do tego punktu antyterrorysta zatrzymuje się i przechodzi do stanu bezczynności - *idle*.

Każdy antyterrorysta może zmienić swój stan na podstawie zaistniałego zdarzenia. Postrzelenie antyterrorysty lub natrafienie przez niego na ścianę, wiąże się z natychmiastowym wywołaniem stanu *check location*. Stan ten definiuje poruszanie się jednostki do zadanego punktu, z wykorzystaniem wyliczonej ścieżki bezkolizyj-

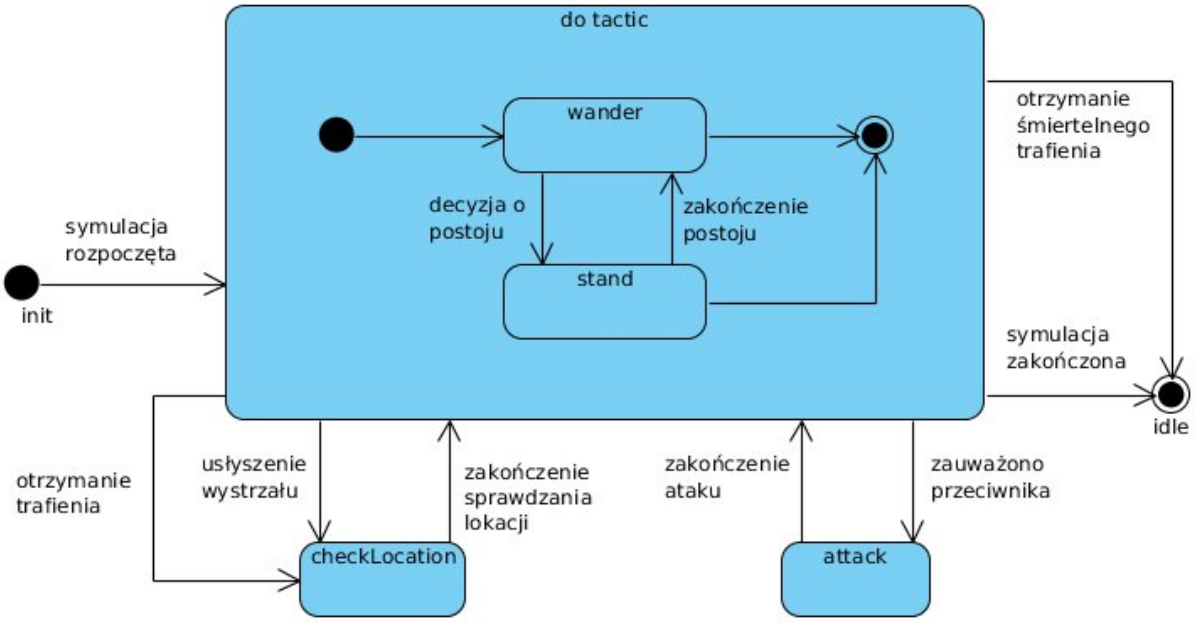
nej. Antyterrorysta opuszcza ten stan po dotarciu do celu lub po upłynięciu limitu czasowego na wykonanie tej czynności. Zdarzenie polegające na zauważeniu przeciwnika, wywołuje stan *attack*. Stan ten pozwala antyterroryście na oddawanie strzałów w kierunku zauważonego terrorysty, jeżeli na linii strzału nie znajduje się żaden antyterrorysta. Wyjście z tego stanu następuje po zabiciu terrorysty lub po straceniu celu z pola widzenia.

4.2 Taktyka terrorystów

Terroryści nie posiadają grupowej strategii działania, kierują się wyłącznie indywidualnie podejmowanymi decyzjami. Terrorysta posiada skończony zbiór stanów (rysunek 4.2). Po zainicjalizowaniu obiektu terrorysty przechodzi on do stanu *wander*, który pozwala mu na wędrowanie po świecie gry. Podczas wędrowania terrorysta może podjąć decyzję o wykonaniu postoju, co wiąże się z przejściem do stanu *stand*. Stan ten zatrzymuje jednostkę oraz odlicza czas pozostały do końca postoju, a po jego upłynięciu zmienia stan terrorysty ponownie na *wander*.

Każdy terrorysta także może zmienić swój stan na podstawie zaistniałego zdarzenia. Postrzelenie terrorysty lub usłyszenie przez niego odgłosu wystrzału skutkuje wywołaniem stanu *check location*, który definiuje identyczne zachowanie i warunek wyjścia ze stanu, jak w przypadku antyterrorysty. Podobnie jest ze zdarzeniem polegającym na zauważeniu przeciwnika. Wywołuje ono stan *attack*, który nakazuje terroryście strzelać do wrogiej jednostki. Warunek wyjścia z tego stanu jest taki sam, jak u antyterrorysty.

Zdarzeniem wspólnym dla antyterrorystów oraz dla terrorystów jest otrzymanie śmiertelnego trafienia. W tym przypadku natychmiastowo zmieniany jest stan jednostki na bezczynność - *idle*. Jednostka pozostająca w stanie bezczynności nie reaguje na zdarzenia, co skutkuje brakiem możliwości zmiany swego stanu.



Rysunek 4.2: Diagram przejść międzystanowych terrorysty

Zakończenie

Realizacja projektu gry symulacyjnej, która jest przedmiotem tej pracy dyplomowej, rozpoczęła się od pomysłu wizualizacji odpowiednio uproszczonego procesu planowania operacji antyterrorystycznej. Po przygotowaniu pierwszej wersji szkicu interfejsu i rozpisaniu scenariuszy, jakie powinny zostać zaimplementowane w pierwszym etapie, należało wybrać technologię, w jakiej gra będzie implementowana. Pierwszy prototyp powstał w języku Ruby z wykorzystaniem biblioteki Rubygame[2]. Jednakże końcowy wybór padł na Javascript i HTML5 Canvas, które w tym wypadku są wspierane przez bibliotekę Kinetic.js. Na korzyść tych technologi przemawiało kilka faktów:

- Javascript posiada dużo mniej wymagające środowisko uruchomieniowe od Ruby'iego
- Kinetic.js jest stale rozwijaną biblioteką w przeciwieństwie to Rubygame
- implementacja interfejsu w HTML jest bardzo prosta

Implementacja kolejnych etapów następowała sprawnie. Informacja zwrotna uzyskiwana podczas testów przyczyniała się do dodania nowych funkcjonalności oraz motywowała do dalszej pracy. Każdy etap dostarczał nową, kompletną funkcjonalność. Gra symulacyjna może być nadal rozwijana, a najbliższe działania powinny się skupić na następujących obszarach:

[2]strona projektu Rubygame - http://rubygame.org/

- Urozmaicenie taktyk terrorystów: terroryści mogą z określonym prawdopodobieństwem uciekać na widok antyterrorystów
- Urozmaicenie taktyk antyterrorystów: podczas ataku antyterroryści zmieniają szyk na tyralierę, by dysponować większą siłą ogniową w kierunku celu
- Refaktoring: wydzielenie części kodu Game.Entity do nowej klasy Game.Person
- Interfejs: przygotowanie atrakcyjniejszej oprawy graficznej
- Testy: sporządzenie testów automatycznych, weryfikujących poprawność działania interfejsu oraz ważniejszych metod publicznych

Wykorzystane technologie w zupełności wystarczyły do zaimplementowania projektu. Javascript i HTML5 Canvas można z powodzeniem wykorzystywać do wykonywania prostych, nie pochłaniających dużej ilości zasobów (CPU, Ram) aplikacji. Z drugiej strony, wraz z obserwowanym, bardzo szybkim tempem rozwoju technologi internetowych (np. gry 3D w przeglądarce z użyciem WebGL), możemy wkrótce się doczekać bardzo atrakcyjnych i wydajnych rozwiązań, które będą kreować nowe standardy w dziedzinie tworzenia gier wideo.

Bibliografia

[1] Elite UK Forces: Operation Nimrod - the iranian embassy siege. [online], 2012. [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://www.eliteukforces.info/special-air-service/sas-operations/iranian-embassy/`.

[2] E. Adams A. Rollings, editor. *Andrew Rollings and Ernest Adams on Game Design*. New Riders Publishing, New Jersey, 2003.

[3] J. Funge I. Millington, editor. *Artificail Intelligence for Games - second edition*. Morgan Kaufmann Publishers, Burlington, 2009.

[4] Gra tom clancy's rainbow six - encyklopedia gier — gry-online.pl. [online], 2012. [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://www.gry-online.pl/gra.asp?ID=3102`.

[5] Eric Drowell. *Kinetic.js Home Page*, 2012. werjsa 3.10.5, [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://www.kineticjs.com/`.

[6] James Coglan. *Sylvester Home Page*, 2012. werjsa 0.1.3, [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://sylvester.jcoglan.com/`.

[7] jQuery Team. *jQuery Home Page*, 2012. werjsa 1.8.1, [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://www.jquery.com/`.

[8] Brian Grinstead. *A* Search Algorithm in JavaScript*, 2012. werjsa 0.1.3, [dostęp: 2012-09-10 12:00], Dostępny w Internecie: `http://www.briangrinstead.com/blog/astar-search-algorithm-in-javascript-updated`.

Spis tabel

Spis rysunków