**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Sławomir Tomaszewski**

Numer albumu: 1118787

**Natalia Zmysłowska**

Numer albumu:

**PROTOTYP GRY STRATEGICZNEJ CZASU RZECZYWISTEGO. BADANIA SZTUCZNEJ INTELIGENCJI PRZECIWNIKÓW**

Praca magisterska

na kierunku: Informatyka Stosowana

Praca wykonana pod kierunkiem

dr Jan K. Argasiński

Zakład Technologii Gier

**Oświadczenie autora pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

**Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

# Spis treści

[Spis treści 3](#_Toc462364063)

[Wstęp 6](#_Toc462364064)

[Wprowadzenie 7](#_Toc462364065)

[1. Reguły i założenia prototypu 10](#_Toc462364066)

[1.1. Konwencja 10](#_Toc462364067)

[1.2. Podstawowa mechanika 10](#_Toc462364068)

[1.3. Typy jednostek 12](#_Toc462364069)

[1.4. Typy budynków 13](#_Toc462364070)

[1.5. Technologie i drzewko technologiczne 13](#_Toc462364071)

[1.6. Interfejs użytkownika 15](#_Toc462364072)

[2. Sposób zaprogramowania prototypu 16](#_Toc462364073)

[2.1. Opis najważniejszych elementów biblioteki *Unity* 19](#_Toc462364074)

[2.2. Obiekty globalne 21](#_Toc462364075)

[2.2.1. Singleton Globals 21](#_Toc462364076)

[2.2.2. Obiekty gracza i armii 22](#_Toc462364077)

[2.2.3. Obiekty środowiska 23](#_Toc462364078)

[2.2.4. Obiekty konfiguracyjne 24](#_Toc462364079)

[2.2.5. Klasy ze stałymi 25](#_Toc462364080)

[2.3. Podsystem elementów mapy 25](#_Toc462364081)

[2.3.1. Implementacja elementu mapy 25](#_Toc462364082)

[2.3.2. Statystyki 32](#_Toc462364083)

[2.3.3. Technologie 33](#_Toc462364084)

[2.3.4. Rozkazy 33](#_Toc462364085)

[2.3.5. Akcje rozkazów 38](#_Toc462364086)

[2.3.6. Sąsiedztwo murów 39](#_Toc462364087)

[2.4. Podsystem mapy 40](#_Toc462364088)

[2.4.1. Klasy mapy 40](#_Toc462364089)

[2.4.2. Klasy drzewa czwórkowego 43](#_Toc462364090)

[2.5. Podsystem mgły wojny 46](#_Toc462364091)

[2.6. Podsystem poszukiwania ścieżek 48](#_Toc462364092)

[2.7. Podsystem sterowania 49](#_Toc462364093)

[2.8. Podsystem interfejsu gracza 52](#_Toc462364094)

[2.8.1. Skróty klawiszowe 52](#_Toc462364095)

[2.8.2. Opisy guzików 56](#_Toc462364096)

[2.8.3. Wygląd minimapy 56](#_Toc462364097)

[3. Kreacja graficzna prototypu 60](#_Toc462364098)

[3.1. Grafika dwuwymiarowa 60](#_Toc462364100)

[3.1.1. Kursor myszy 60](#_Toc462364101)

[3.1.2. Budowa interfejsów gracza 61](#_Toc462364102)

[3.2. Grafika trójwymiarowa 64](#_Toc462364103)

[3.2.1. Modele 64](#_Toc462364104)

[3.2.2. System cząsteczkowy dymu 72](#_Toc462364105)

[4. Zaprogramowanie sztucznej inteligencji 74](#_Toc462364106)

[4.1. Mechanizmy pomocnicze 76](#_Toc462364108)

[4.1.1. Regiony 76](#_Toc462364109)

[4.1.2. Rodzaje elementów mapy 79](#_Toc462364110)

[4.2. System wieloagentowy 80](#_Toc462364111)

[4.2.1. Gracz AI 81](#_Toc462364112)

[4.2.2. Klasa agenta 83](#_Toc462364113)

[4.2.3. Wiadomości i żądania 85](#_Toc462364114)

[4.2.4. Klasa zadania 85](#_Toc462364115)

[4.2.5. Agent jednostki 86](#_Toc462364116)

[4.2.6. Agent gromadzący wiedzę 88](#_Toc462364117)

[4.2.7. Agent odpowiedzialny za zwiady 94](#_Toc462364118)

[4.2.8. Zadanie zgrubnego rekonesansu 101](#_Toc462364119)

[4.2.9. Agent rozbudowujący bazę 104](#_Toc462364120)

[4.2.10. Agent produkcyjny 108](#_Toc462364121)

[4.2.11. Agent zbierający zasoby 110](#_Toc462364122)

[4.2.12. Zadanie zbierania zasobów 114](#_Toc462364123)

[5. Badania skuteczności sztucznej inteligencji 116](#_Toc462364124)

[5.1. Badanie przebiegu domyślnej rozgrywki 116](#_Toc462364126)

[5.1.1. Założenia eksperymentu 116](#_Toc462364127)

[5.1.2. Przebieg i analiza symulacji 117](#_Toc462364128)

[5.1.3. Analiza przyrostu zasobów i odkrytego terenu 121](#_Toc462364129)

[5.2. Badanie skuteczności rekonesansu 126](#_Toc462364130)

[5.2.1. Założenia eksperymentów 127](#_Toc462364131)

[5.2.2. Sortowanie według odległości do bazy 127](#_Toc462364132)

[5.2.3. Sortowanie według sumy odległości do bazy i zwiadowcy 130](#_Toc462364133)

[5.2.4. Sortowanie według iloczynu odległości do bazy i zwiadowcy 133](#_Toc462364134)

[5.2.5. Sortowanie według iloczynu oraz odsetka zbadanych pól 135](#_Toc462364135)

[6. Zakończenie 143](#_Toc462364136)

[7. Streszczenie 145](#_Toc462364137)

[8. Bibliografia 146](#_Toc462364138)

[9. Aneks/Spis ilustracji 147](#_Toc462364139)

# Wstęp

Tematem niniejszej pracy magisterskiej jest przeprowadznie badań nad zachowaniem sztucznej inteligencji (ang. *artificial intelligence*, *AI*). Zostaną one przedstawione na przykładzie stworzonego przez nas specjalnie do tego celu prototypu gry strategicznej czasu rzeczywistego (ang. *real-time strategy*, skr. *RTS*), którą nazwaliśmy **MechWars**.

Praca składa się z pięciu rozdziałów. Pierwszy z nich opisuje założenia i konwencje, na jakich będziemy się opierać. Zostanie tam przedstawiona podstawowa mechanika, obowiązująca w prototypie. Dodatkowo wypiszemy typy jednostek, budynków oraz drzewko technologiczne, które zamierzamy zaprojektować. Na końcu znajdzie się opis interfejsu użytkownika.

W drugim rozdziale przedstawimy sposób, w jaki prototyp został zaprogramowany. Na początku znajdzie się tam pobieżny przegląd podsystemów projektu oraz opis najważniejszych elementów silnika *Unity*. Następnie przejdziemy do konkretów. Omówimy w szczegółach moduły obiektów globalnych, elementów mapy, samej mapy, mgły wojny wyszukiwania ścieżek oraz interfejsu gracza. Podsystem sterowania będzie opisany tylko pod kątem jednej funkcjonalności.

Trzeci rozdział poświęcimy podziałowi pomiędzy grafiką dwuwymiarową a trójwymiarową, która znajdzie się w prototypie **MechWars**. W pierwszej części opiszemy kursor myszy oraz dwa dostępne w *Unity* sposoby zaprojektowania interfejsów użytkownika. W podrozdziale o grafice trójwymiarowej zostaną zaprezentujemy zaś utworzone przez nas modele wykorzystane w projekcie oraz sposób użycia systemu cząsteczkowego *Unity* do generowania dymu.

Czwarty rozdział będzie dotyczył rozwiązania problemu, jakim jest zaprogramowanie sztucznej inteligencji dla gry *RTS*. Najpierw krótko przeprowadzimy teoretyczne rozważania, jak należałoby stworzyć *AI*, które mogłoby symulować zachowanie gracza ludzkiego. Przedstawimy wyzwania, jakie są sztucznej inteligencji stawiane. Na końcu przejdziemy do omówienia zaprogramowanych przez nas rozwiązań dla tych wyzwań.

W piątym rozdziale wykonamy badania stworzonej przez nas sztucznej inteligencji. Zostanie uruchomiana i zanalizowana rozgrywka dla domyślnych ustawień paramterów i planszy gry. Następnie przeprowadzimy szereg owocnych eksperymentów związanych z zadaniem rekonesansu.

# Wprowadzenie

*RTS*, czyli tłumacząc dosłownie z języka angielskiego „strategia czasu rzeczywistego” jest odmianą gry strategicznej, w której gracze skupiają się na rozbudowie gospodarki ekonomicznej oraz siły militarnej w celu pokonania przeciwnika (zniszczenia jego armii i bazy).[[1]](#footnote-1) Jej główna cecha polega na tym, iż dzieje się w czasie rzeczywistym, co oznacza, że cała rozgrywka nie jest podzielona na tury lub kolejki.[[2]](#footnote-2)

Strategie czasu rzeczywistego są zróżnicowane pod względem tematyki oraz sposobu prezentacji świata, mimo to większość bazuje na podobnym schemacie działania. Gracz kontroluje poczynania jednej z konkurujących frakcji. Jego zadaniem jest pozyskiwanie surowców, budowa zaplecza gospodarczego, stworzenie silnej armii i zajęcie terenów przeciwnika. W 1992 roku twórcy gry *Dune II* wprowadzili wzajemną zależność struktur od siebie, a także ideę różnych stron, które posiadają różne jednostki i bronie.[[3]](#footnote-3) Cechą wspólną gier strategicznych są warunki zwycięstwa, czyli pokonanie przeciwnika poprzez zniszczenie jego głównej struktury dowodzącej lub pozbawienie go zasobów, dzięki czemu nie ma on możliwości odbudowy swoich jednostek.[[4]](#footnote-4)

Główną mechaniką gry jest wybór ścieżki rozwoju, którą gracz zamierza podążać podczas trwania rozgrywki. Do wyboru zazwyczaj ma dwie podstawowe gospodarki — ekonomiczną oraz militarną, które później może krzyżować. Pierwsza z nich sprawia, że na początku gracz ma bardzo słabe jednostki militarne (lub nie ma ich wcale), gdyż skupiamy się na rozwoju związanym z pozyskiwaniem surowców budulcowych. W ten sposób gromadzi dużą liczbę zasobów w krótkim czasie, co w konsekwencji pozwala mu na szybszą rozbudowę oraz masowe zrekrutowanie jednostek w późniejszym etapie gry. Druga droga jest bardziej militarna i skupia się na inwestowaniu w oddziały zbrojne oraz badania z nimi związane. Umożliwia to zbudowanie silnych jednostek we wczesnej fazie gry. Gracz korzystający z takiej strategii nie może pozwolić sobie na szybki rozwój technologiczny, ale zazwyczaj jest w stanie odeprzeć ataki wroga. Podczas podejmowania decyzji odnośnie wyboru ścieżki (którą w każdej chwili można zmienić), gracz musi również zrozumieć konstrukcję mapy oraz rozmieszczenie poszczególnych surowców, w stopniu wystarczającym, by jak najbardziej optymalnie jego zdaniem rozlokować budynki oraz jednostki.[[5]](#footnote-5)

Sztuczna inteligencja w grach typu *RTS* ma między innymi za zadanie postępować jak prawdziwy gracz. W skład tego wchodzą takie zachowania jak:[[6]](#footnote-6)

* znalezienie jak najbardziej optymalnej drogi (pathfinding), który pozwoli na szybsze dotarcie jednostek do wskazanego celu,
* posiadanie bazowej wiedzy na temat gry,
* planowanie swoich działań,
* rozbudowa jednostek,
* wieczne uczenie się na błędach oraz sukcesach,
* wyciąganie wniosków z podjętych działań,
* dostosowywanie szybkości nauki oraz wykonywania do wybranego poziomu trudności rozgrywki,
* wysyłanie jednostek zwiadowczych w celu zebrania informacji o poziomie zaawansowania gracza,
* przewidywanie ewentualnej strategii gracza,

Głównym problemem dla sztucznej inteligencji, jaki pojawia się podczas rozgrywek, jest podejmowanie decyzji opartych o zbyt małą liczbę informacji, a co za tym idzie — optymalnie zaplanowanie taktyk oraz rozlokowanie sił. Po zdobyciu informacji konieczne jest wyselekcjonowanie, która z nich ma najwyższy priorytet w danej sytuacji oraz zrozumienie jej wagi w odniesieniu do całości. Widać, że napisanie dobrej *AI* w grze *RTS* jest niezwykle trudnym zadaniem. Większości obecnych tytułów brakuje wysublimowanej sztucznej inteligencji — trudność gry skaluje się sprawiając, że *AI* oszukuje (ma zwiększoną siłę liczebną, lub zmniejszony koszt produkcji jednostek), podczas gdy można by było poprawić jej zdolności myślenia strategicznego.

Celem naszej pracy magisterskiej jest próba stworzenia sztucznej inteligencji dla gracza w grze *RTS*. Aby móc ją na czymś testować, zostanie zaimplementowany prototyp takiej gry, który będzie oparty o proste reguły. Głównym założeniem jest zaprogramowanie *AI*, która będzie posiadała zdolność do pozyskiwania informacji, przetwarzania ich oraz postępowania będącego ich konsekwencją. Ma to być zatem coś w rodzaju fundamentu pod bardziej złożoną sztuczną inteligencję, udającą z powodzeniem człowieka. Nie przewidujemy pełnej *AI* pozwalającej na jednoosobową rozgrywkę przeciwko niej, ale stworzymy pojedyncze mechanizmy rozwiązujące zadania stawiane przed graczem w *RTS*, które później zostaną przebadane co do ich skuteczności.

# Reguły i założenia prototypu

Celem niniejszej pracy magisterskiej jest zaprogramowanie prototypu gry, stworzenie sztucznej inteligencji w ograniczonym stopniu sterującej graczem i przebadanie jej skuteczności. Prototyp dostał roboczą nazwę **MechWars**. Postanowiliśmy oprzeć go o proste zasady, gdyż nie chcemy skupiać się zbyt mocno na rozbudowie mechaniki rozgrywki. Mimo to nie mogą one też być zbyt proste, gdyż za mocno ograniczyłoby to możliwości oraz decyzje sztucznej inteligencji. Prototyp zostanie umieszczony w konkretnej konwencji określającej zasób, a także rodzaje jednostek, budynków i badań technologicznych.

## Konwencja

Program, którego będziemy używać do stworzenia prototypu gry *RTS* to *Unity*. Modele budynków, jednostek jak i otoczenia będą trójwymiarowe, a perspektywiczna kamera będzie ustawiona pod kątem 45° do planszy. Teren, na którym będzie miała miejsce rozgrywka będzie płaską i symetryczną względem środka mapą tak, aby obie strony miały równe szanse przy rozpoczęciu partii.

Przy tworzeniu konwencji prototypu zamierzamy inspirować się wizją świata po apokalipsie, w której jedynymi ocalałymi są maszyny. Dodatkowo wygląd otoczenia inspirowany będzie stylem retrofuturystycznym, czyli wyobrażeniem przyszłości zazwyczaj stylizowanym na erę wiktoriańską.[[7]](#footnote-7) W związku z tym tematem przewodnim naszego prototypu będą walki maszyn w futurystycznym świecie, który będzie wielkim złomowiskiem. Oprócz budynków produkcyjnych będą w nim występować jednostki zmechanizowane, czyli mechy. Głównym zasobem świata będzie złom występujący rozsypany po powierzchni. Dodatko będzie można go zbierać z wraków jednostek oraz zniszczonych budynków.

Budynki oraz jednostki, które zostaną wykorzystane w prototypie będą trójwymiarowymi modelami stworzonymi w całości na potrzeby niniejszej pracy. Dym z kominów niektórych budynków zostanie zaprojektowany dzięki systemowi cząsteczkowemu w *Unity*.

## Podstawowa mechanika

W podstawowej mechanice przewidujemy jeden funkcjonalny tryb gry — gracz ludzki przeciwko sztucznej inteligencji. Dodatkowo, jeśli uda się zaprogramować dość zachowań *AI* by miało to sens, zostanie wprowadzony drugi tryb gry: rozgrywka pomiędzy dwoma sztucznymi inteligencjami. Do żadnego z tych trybów gry nie zamierzamy tworzyć ani fabuły, ani kampanii, ponieważ chcemy skupić się na programowaniu sztucznej inteligencji.

Plansza (zwana również mapą), na której toczyć się będzie rozgrywka ma być podzielona na kwadratowe pola. Jednostka, budynek czy zasób stanowić mają elementy mapy. Jednostki i zasoby zajmować będą dokładnie jedno pole mapy, budynki będą mogły zajmować kilka (będą mieć określony kształt). Każdy element mapy znajduje się zawsze na określonych polach, z wyjątkiem jednostki — ta może znajdować się pomiędzy polami tylko w trakcie ruchu na sąsiednie pole.

Zamierzamy zaimplementować podstawowe zachowania jednostek kierowanych zarówno przez fizycznego gracza, jak i przez sztuczną inteligencję. Zaliczają się do nich:

* wykonywanie rozkazów gracza,
* przemieszczanie jednostek po mapie,
* atakowanie oddziałów przeciwnika,
* zbieranie zasobów z planszy.

Jednostki i budynki będą posiadać własną, ograniczoną sztuczną inteligencję. Będzie ona zrealizowana w postaci rozkazów wydawanych przez gracza. Rozkazy będą pewnymi procesami wykonującymi szereg czynności. W skład tego będą wchodziły nie tylko pojedyncze rozkazy odnośnie przemieszczania się, ale także automatyczne atakowanie, gdy przeciwnik pojawi się w zasięgu widzenia danej jednostki. Planowane rozkazy to:

* *Idle* — bezczynność; jednostka lub budynek stoi w miejscu, obraca się raz na jakiś czas (jeśli potrafi) i atakuje wrogów, którzy znajdą się w zasięgu ataku (też: jeśli potrafi),
* *Move* — jednostka przemieszcza się do określonej pozycji docelowej,
* *FollowAttack* — jednostka atakuje wroga i goni go,
* *StandAttack* — jednostka atakuje wroga w miejscu i przerywa atak, gdy wróg wyjdzie z zasięgu,
* *AttackMove* — jednostka przemieszcza się do określonej pozycji docelowej, ale po drodze angażuje się w walkę z wrogami, jeśli takich napotka,
* *Escort* — jednostka podąża za wskazanymi przyjaznymi jednostkami i atakuje wszystkich wrogów w zasięgu,
* *Harvest* — jednostka kursuje między wskazanym zasobem a najbliższą do niego rafinerią, zbierając zasób i odkładając go do rafinerii, gdy się zapełni,
* *Stop* — jednostka przerywa obecny rozkaz,
* *UnitProduction* — budynek produkuje określony rodzaj jednostki,
* *BuildingConstruction* — budynek konstruuje inny, określony budynek we wzkazanym miejscu,
* *TechnologyDevelopment* — budynek opracowuje określoną technologię.

Idle będzie rokazem domyślnym — jeśli jednostka lub budynek nie będzie mieć wydanych żadnych rozkazów, będzie wykonywać Idle. Jednostki i budynki różnić się będą co do rozkazów, które będą mogły wykonywać. Przykładowo zwiadowca nie będzie mógł atakować ani odkrywać technologii, a wieżyczka obronna nie będzie mogła się poruszać.

Zamierzamy zastosować mechanizm tzw. mgły wojny. Polega on na tym, że tylko pewna część mapy znajduje się w polu widzenia jednostek, a zatem gracza. W związku z tym w pozostałych miejscach lokalizacja jednostek przeciwnika pozostaje nieznana. Mgła wojny jest odpowiednikiem poziomu niepewności gracza co do sytuacji, w której znajduje się przeciwnik w danym momencie.[[8]](#footnote-8) Dzięki jednostkom takim jak zwiadowca będziemy mogli odkryć fragment terytorium i zebrać informacje na temat stanu zaawansowania przeciwnika oraz położenia surowców.

## Typy jednostek

W związku z tym, iż chcemy zachować optymalny balans rozgrywki, stworzone zostaną jedynie jednostki naziemne. Będą się one różniły między sobą nie tylko wyglądem, ale również statystykami, takimi jak:

* szybkość obrotu i poruszania się,
* siła i szybkość ataku,
* zasięg ataku i widzenia,
* liczba punktów życia.

Dwoma podstawowymi niemilitarnymi jednostkami będą jednośladowy zwiadowca — *Scout* — oraz zbieracz zasobów, czyli *Harvester*. Zwiadowca charakteryzować się będzie wysokim zasięgiem widzenia, i szybkością, ale jednocześnie nie będzie miał możliwości ataku i bardzo łatwo go będzie zniszczyć. Jego zadaniem będzie zatem rekonensans: sprawdzenie terenu, odnalezienie zasobów, a także szpiegowanie postępów przeciwnika. *Harvester* natomiast będzie jednostką, która będzie potrafiła wydobywać zasoby (złom) i transportować go do rafinerii. On również nie będzie potrafił walczyć.

Podstawowymi oddziałami militarnymi będą dwa typy zmechanizowanych jednostek kroczących — tzw. mechów. Dalekozasięgowy mech z karabinami maszynowymi (*MechMachinegun*) będzie ostrzeliwał wrogów niską siłą ataku, podczas gdy wytrzymalszy i silniejszy mech z miotaczem ognia (*MechFlamethrower*) będzie musiał podejść bezpośrednio do wroga. Obie jednostki będą poruszały się dość szybko, jednak nie będą zadawały zbyt wysokich obrażeń jednostkom i budynkom.

Oprócz powyżej opisanych jednostek powstaną jeszcze dwa rodzaje pojazdów militarnych. Jednym z nich będzie czołg (*Tank*), które będzie powolną maszynami ze stosunkowo dużym zasięgiem ataku i siłą ognia. Drugim będzie mobilna wyrzutnia rakiet (*RocketLauncher*), która cechować się będzie olbrzymim zasięgiem i siłą rażenia, ale z bliska jej rakiety będą niezwykle niecelne — tak, że mogą przypadkiem trafić we własne jednostki i struktury. Oba typy pojazdów będą posiadały na tyle duże obrażenia, by w bardzo szybkim czasie niszczyć budynki i całe oddziały wroga.

## Typy budynków

Budynki będą służyły konstrukcji nowych budynków, produkcji jednostek, opracowywaniu technologii, składowaniu zasobów oraz obronie bazy. Kominy przy niektórych budynkach będą generować dym utworzony za pomocą systemu cząsteczkowego w programie *Unity*.

Typy planowanych przez nas budynków to:

* *ConstructionYard* — budynek służący do konstrukcji nowych budynków; nie można go wybudować, ale każda armia zaczyna z jednym,
* *Refinery* — rafineria złomu, będąca punktem, do którego Harvestery znoszą zasoby (fabularnie służy do przetwarzania złomu na użyteczny metal),
* *Factory* — fabryka produkująca jednostki,
* *Laboratory* — laboratorium naukowe służące do odkrywania nowych technologii, dających bonusy do statystyk oraz odblokowujących nowe jednostki i budynki,
* *Turret* — zmechanizowana wieża obronna, która strzela do wrogich jednostek,
* *Wall* — mur który nie pozwala jednostkom przejechać.

## Technologie i drzewko technologiczne

Głównym zadaniem technologii jest spełnianie wymagań do konstrukcji nowych budynków, produkcji nowych jednostek i odkrywania następnych technologii. Istnienie pewnego budynku w bazie również może być wymaganiem. Gdy wziąć wszystkie zależności wymagań pomiędzy technologiami, budynkami i jednostkami, tworzą one tak zwane drzewko technologii pokazujące drogi rozwoju ekonomicznego i militarnego (Rysunek 1).



**Rysunek 1.** Drzewko technologiczne prototypu **MechWars**

Czerwone prostokąty oznaczają jednostki, zielone budynki, a niebieskie technologie. Wszystkie jednostki są produkowane w *Factory*, a budynki konstruowane w *ConstructionYard*. Technologie odkrywane są głównie w *Laboratory*, z wyjątkiem dwóch — te w nawiasie mają podany budynek, który służy do ich odkrycia. Strzałka wychodzi od obiektu który wymaga i wskazuje na obiekt wymagany. Oczywiście wszystkie jednostki wymagają *Factory* a wszystkie technologie odkrywane w laboratorium wymagają *Laboratory*. Nie ma jednak potrzeby wymuszać takiego wymagania, gdyż po prostu jest to zrealizowane poprzez konfigurację możliwości budynków. Tylko fabryka potrafi produkować jednostki, a tylko laboratorium odkrywać większość technologii. Dlatego istnieją jednostki, budynki i technologie, które (pozornie) nie wymagają niczego. Na rysunku można zauważyć też, że jednostki nie są do niczego wymagane. Oczywiście jest to celowe, gdyż taki zabieg nie miałby sensu.

Przykładem wymagań może być to sytuacja, w której aby stworzyć wieżyczkę obronną (*Turret*), trzeba najpierw wybudować fabrykę (*Factory*) i odkryć ciężkie działa (*HeavyCannons*). Z kolei do produkcji dowolnej jednostki militarnej fabryka musi opracować dział zbrojeniowy (*ArmsDivision*).

Niektóre technologie poza spełnianiem wymagań zapewniają również bonusy do statystyk. Poniżej znajduje się ich lista:

* *PortableScrapGrinder*: szybkość zbierania zasobów przez jednostki *Harvester* ×1.5,
* *ImprovedLenses*: zasięg widzenia jednostki *Scout* +2,
* *MachinegunRotationBooster*: szybkość ataku jednostki *MechMachinegun* ×1.2,
* *HighOctaneFuels*: szybkość ruchu jednostek *Tank*, *RocketLauncher* i *Scout* ×1.2,
* *PlatedTracks*: punkty życia jednostek *Tank* i *RocketLauncher* +100,
* *RefinedSteelPlating*: punkty życia jednostki *Tank* ×1.3,
* *HighExplosives*: siła ataku jednostki *RocketLauncher* ×1.2.

## Interfejs użytkownika

Na interfejs użytkownika będzie składać się menu główne gry pozwalające wybrać tryb gry oraz *GUI* wewnątrz samej rozgrywki. Głównym elementem tego drugiego ma być panel dolny zawierający przyciski rozkazów oraz minimapę (czyli miniaturę całej planszy, na której widać jej stan — położenie jednostek, budynków oraz kształt mgły wojny). Poza tym w górnej części ekranu wyświetlana będzie obecna liczba jednostek zasobów, a nad przyciskami rozkazów pokazywać się będą dymki z opisem rozkazu po najechaniu na niego kursorem myszy.

Sama myszka ma zachowywać się w sposób zbliżony do znanego ze współczesnych *RTS*-ów. Lewy przycisk myszy ma służyć do zaznaczania elementów mapy. Przy przytrzymaniu go i przeciągnięciu myszy ma powstać ramka zaznaczenia, która wpływa na zaznaczenie wszystkich elementów mapy znajdującej się wewnątrz niej. Z kolei prawy przycisk myszy ma służyć wydawaniu rozkazów kontekstowo — na przykład przy kliknięciu na teren jednostka otrzyma rozkaz *Move*, na zasób — *Harvest*, a na wroga — *FollowAttack*. Za to po wybraniu rozkazu przyciskiem z panelu dolnego, to lewy przycisk myszy będzie służył do jego wydania, a prawy — anulował wydawanie rozkazu.

Wokół zaznaczonych jednostek i budynków będą pojawiać się ramki z tzw. paskami życia. Paski życia będą pokazywać ile punktów życia zostało elementowi mapy, a ich kolor będzie wskazywał na to, z której armii pochodzi.

# Sposób zaprogramowania prototypu

Gra *RTS* w ogólności jest bardzo złożonym projektem informatycznym. Napisanie nawet jej uproszczonego prototypu było skomplikowanym zadaniem. Wymagało dogłębnego przeanalizowania wszystkich koniecznych funkcjonalności oraz zaprojektowania zależności między klasami i obiektami od podstaw. Zaprogramowany prototyp można zatem podzielić na kilka sporych, choć różniących się wielkościami podsystemów (w nawiasie podana została nazwa jednej lub kilku najważniejszych klas z danego podsystemu — wyróżniających ten podsystem):

* Obiekty globalne (Globals),
* Elementy mapy (MapElement, Unit, Building, Resource),
  + Ataki (Attack),
  + Rozkazy (Order),
    - Akcje rozkazów (OrderAction),
    - Produkty (Product),
  + Statystyki (Stat),
* Obiekty mapy (Map),
* Mgła wojny (VisibilityTable, VisualFog),
* Poszukiwanie ścieżek (AStarPathfinder),
* Sterowanie (InputController),
* Sztuczna Inteligencja (AIBrain),
  + Agenci (Agent),
    - Cele (Goal),
  + Regiony (Region),
* GUI (CanvasScript),
* Narzędzia (folder *Utils* – brak konkretnej klasy wyróżniającej).

Obiekty globalne to podsystem, którego klasy najczęściej mają tylko po jednej instancji, a pobranie referencji do tych instancji jest możliwe z dowolnego miejsca kodu. Klasy z tego modułu służą przede wszystkim ogólnej konfiguracji gry oraz dostępowi do danych o stanie gry takich jak mapa, jednostki danej armii czy jej obecny rozwój technologiczny.

Elementy mapy to zdecydowanie największy i najbardziej złożony moduł. Najważniejszą klasą jest tu MapElement — będący dowolną rzeczą która może znajdować się na polu (lub polach) mapy. Może być to jednostka, budynek albo zasób. W skład tego podsystemu wchodzą też wszystkie klasy związane z obsługą elementów mapy (więc np. technologie), dotyczące tzw. duchów (*snapshotów* pokazujących ostatni stan obiektu MapElement, zanim został skryty przez mgłę wojny), efektów cząsteczkowych, markerów do minimapy, poruszania się jednostek, obrotu głowic elementu mapy (np. lufy czołgu) tudzież konstrukcji nowych budynków.

Można tu wydzielić jeszcze 3 podmoduły. Podmoduł ataków zawiera klasy obsługujące wyprowadzania ataków przez jednostki. Podmoduł statystyk dotyczy utrzymywania statystyk elementu mapy (takich jak punkty życia, siła ataku, szybkość poruszania) i stosowania bonusów do nich. Podmoduł rozkazów natomiast sam w sobie jest obszerny i można w nim wydzielić jeszcze pomniejsze części.

Przede wszystkim podmoduł rozkazów zawiera klasy obsługujące wszelakie rozkazy wydawane jednostkom: poruszanie, atakowanie, zbieranie zasobów, eskortowanie, produkcja jednostek, konstrukcja budynków, odkrywanie technologii. Istnieje tu do tego część związana z tzw. akcjami rozkazów, które stanowią o możliwości wykonania danego rozkazu. Na przykład jednostka może wykonać rozkaz FollowAttackOrder tylko, jeśli ma przypisaną akcję FollowAttackOrderAction. Dla każdego rozkazu, który gracz może wydać jednostce/budynkowi istnieje akcja tego rozkazu.

Drugą częścią do wydzielenia z podmodułu rozkazów jest część związana z produktami. Produkty to efekty działania rozkazów produkcji (produkcji jednostek, konstrukcji budynków, odkrywania technologii). Dopóki dana rzecz (jednostka, budynek, technologia) jest w produkcji, istnieje dla niej stworzony produkt (obiekt klasy Product), który zawiera informacje o postępach produkcji.

Obiekty mapy zawierają informacje o planszy: jej rozmiar, dwuwymiarową tablicę elementów mapy, listę graczy i armii biorących udział w rozgrywce oraz obiekt „widza” (Spectator). W module tym znajduje się również implementacja struktury danych drzewa czwórkowego w postaci klasy QuadTree.

Mgła wojny to niewielki podsystem zarządzający obecnie widocznym przez armię terenem i jednostkami. Aktualizowana przez MapElement tablica VisibilityTable zawiera informacje o tym, które pola są widoczne, które poza zasięgiem widzenia, a które nieodkryte. VisualFog i MinimapFog to obiekty zajmujące się wizualizacją mgły wojny na scenie (przyciemnianie terenu poza polem widzenia).

W module poszukiwania ścieżek znajduje się kilka klas, które współpracując stanowią implementację algorytmu *A\**. Znajdujący się tu interfejs IPathfinder pozwala na stworzenie innego algorytmu, jednak istnieje tylko jedna klasa implementująca go: AStarPathfinder. Jako wynik algorytmu zwracany jest obiekt typu Path, składający się z obiektów WayPoint.

Sterowanie nie jest bardzo dużym podsystemem, lecz dość skomplikowanym. W takiej postaci w jakiej jest teraz, został on stworzony po solidnym refactoringu. Znajdują się tu klasy związane z przemieszczaniem kamery, obsługą myszki gracza (PlayerMouse), jej stanów (MouseStateController, myszka ma różne tryby działania w zależności od jej stanu — np. kliknanie wydaje taki albo inny rozkaz), podświetlania i zaznaczania elementów mapy (HoverBox i SelectionMonitor), wyboru miejsca konstrukcji budynku, decydowania o kolorach ramek zaznaczonych jednostek i HoverBoxa.

Zaprogramowana sztuczna inteligencja jest drugim co do obszerności modułem.[[9]](#footnote-9) W skrócie opiera się ona na systemie wieloagentowym (klasa Agent) zastosowanym do obsługi całej strony konfliktu. Każdy agent spełnia inne zadanie: jest osobny agent od wiedzy, od rekonesansu, od zbierania zasobów i od konstrukcji budynków. Dla każdej jednostki w grze również istnieje osobny agent — niższego poziomu. Agenty symulują równoczesne działanie (obiekt AIBrain — będącym odpowiednikiem obiektu Spectator po stronie sztucznej inteligencji — uruchamia aktualizację każdego istniejącego agenta raz na cykl pętli gry). Komunikują się między sobą za pomocą niezbyt rozbudowanego systemu wiadomości. Agenty jednostek mogą zostać „wzięte” przez inne agenty — agent, który taką jednostkę „wziął” chwilowo nią zarządza i żaden inny nie ma do niej dostępu, dopóki nie zostanie ona „zwolniona”. Agenty (głównie agenty jednostek) mogą posiadać kolejkę celów (Goal), które po kolei wykonują. Obok agentów istnieje kilka sposobów w jaki AI organizuje sobie wiedzę o stanie gry. FilteringMapProxy pośredniczy w pobieraniu informacji z Map. MapElementKind opisuje zastosowania rodzajów jednostek. CreationMethod mówi w jaki sposób dany element mapy może zostać stworzony. Wreszcie Regiony pozwalają AI „rozumieć” wycinki mapy mające pewien sens strategiczny (np. obszary posiadające dużą liczbę zasobów).

Podsystem GUI zawiera wszystkie klasy związane z interfejsem użytkownika. Znajdują się tu zatem m.in. MainMenuScript (używany w scenie menu głównego), CanvasScript obsługujący tzw. *Canvas* (mechanizm *Unity* do układania GUI), licznik zasobów (ResourceCounter), wizualizacje statusów zaznaczonych elementów mapy (StatusDisplayDrawer), przyciski akcji rozkazów (OrderActionButtton), obsługa ich skrótów klawiszowych, wyświetlanie dymków pomocy.

Ostatnim modułem jest moduł narzędziowy. Zawiera on bardzo różne narzędzia pomocnicze: często używane typy które mogłyby istnieć w oderwaniu od projektu (np. IVector2, SquareBounds, BinaryHeap) albo statyczne klasy z metodami rozszerzającymi (tzw. *extension methods*) do innych typów (np. UnityExtensions, EnumerableExtensions, DotNetExtensions).

Rozdział ten nie opisuje wszystkich powyższych podsystemów w szczegółach — zajęłoby to zbyt obszerną część pracy. Dlatego niektóre z nich pominięto, by móc skupić się na najważniejszych elementach projektu.

## Opis najważniejszych elementów biblioteki *Unity*

Prototyp stworzony został przy pomocy silnika *Unity*, co wiązało się z niejednokrotnym korzystaniem z jego *API* w programie. Aby więc móc zrozumieć kod prototypu, trzeba wpierw rozumieć działanie *Unity*.

*Unity* w wygodny sposób zarządza tworzoną grą. Silnik oddaje nam do dyspozycji edytor, w którym programista może zarządzać tzw. sceną gry. Scena jest kontenerem na obecnie znajdujące się w grze obiekty. Tylko jedna scena naraz może być wczytana.

Należy podkreślić rozróżnienie dwóch podobnych pojęć: obiektu oraz obiektu gry. Za każdym razem gdy poniżej napisane jest „obiekt”, oznacza to instancję klasy *C#*. Natomiast „obiekt gry” wyjaśniony jest w następnym akapicie.

W najczystszej postaci obiekt gry jest punktem materialnym z niewielką funkcjonalnością. Może on zostać stworzony (do czego służy funkcja GameObject.Instantiate()), zniszczony (GameObject.Destroy()), a do tego posiada aktualną transformację (obiekt klasy Transform): translację, rotację i skalę przedstawione w panelu *Inspector* jako trójwymiarowe wektory. Dla rotacji pokazane są jej kąty Eulera (i to nimi można zarządzać z poziomu edytora), ale *Unity* przechowuje ją w postaci kwaternionu. Oprócz tego obiekty gry mogą zostać ułożone wobec siebie w relacji rodzic-dziecko (o czym informację również trzyma Transform) — co sprawia, że transformacja rodzica staje się bazą dla dziecka. W ten sposób można tworzyć całe hierarchie obiektów gry, których strukturę pokazuje panel *Hierarchy*. Dla tych obiektów gry *Unity* automatycznie zarządza pętlą gry, jednak aby miało to znaczenie, należy rozszerzyć im funkcjonalność.

Funkcjonalność obiektów gry rozszerza się za pomocą komponentów (obiekt klasy Component). Mają one bardzo różnorodne zastosowania — służą między innymi do: przypisania siatki do obiektu gry, nałożenia na nią materiału i tekstury, detekcji kolizji, symulowania fizyki bryły sztywnej, rzucania światła, pełnienia funkcji kamery, odtwarzania lub nasłuchiwania dźwięków, emitowania efektów cząsteczkowych, wykonywania własnej obsługi obiektu gry w postaci skryptu. Dla każdej funkcji istnieje osobny komponent. Wszystkie przypisane do obiektu gry komponenty wyświetlają się w panelu *Inspector*, poniżej części Transform (notabene: Transform również jest komponentem, jednak wpisanym w każdy obiekt gry na stałe). Tam to możliwa jest konfiguracja komponentów. Najważniejszym i najbardziej wykorzystywanym przez prototyp **MechWars** komponentem jest skrypt *C#*.

Skrypty pozwalają w rozległy sposób modyfikować zachowanie obiektu gry. By to było możliwe muszą one spełniać pewną konwencję. Każdy skrypt jest klasą dziedziczącą po klasie MonoBehaviour. Może być klasą abstrakcyjną, lecz wtedy nie da się go przypisać do obiektu gry, ponieważ komponenty-skrypty są instancjami klas tych skryptów. *Unity* tworząc obiekt gry woła bezparametrowe konstruktory we wszystkich przypisanych do niego skryptach, by utworzyć ich komponenty.

Publiczne pola skryptu są rozpoznawane przez edytor — dla każdego z nich w panelu *Inspector* *Unity* tworzy odpowiednie elementy interfejsu użytkownika, pozwalające edytować wartości tych pól (nawet w trakcie działania gry). Oprócz tego istnieje kilka nazw metod, które *API* rozpoznaje w skrypcie. Nie są to metody wirtualne; *Unity* dostaje się do nich za pomocą refleksji. W związku z tym najczęściej tworzy się te metody w skrypcie jako prywatne. Najważniejszymi trzema metodami są: Start() (wołana raz na początku życia obiektu gry), Update() (wołana dla każdego obiektu gry raz na cykl aktualizacji pętli gry) oraz OnDestroy() (wołana tuż przed zniszczeniem obiektu gry). W tych metodach (zwłaszcza Update()) umieszcza się większą część kodu obsługującego obiekt gry. Za pomocą panelu konfiguracyjnego *Script Execution Order* w edytorze można wymusić kolejność, w jakiej uruchamiane są powyższe metody na różnych klasach (np. Start() klasy A zostanie zawołany wcześniej, niż Start() klasy B).

Wewnątrz metod skryptu można uzyskać dostęp m.in. do: kroku czasowego obecnego cyklu (Time.deltaTime), obiektu reprezentującego obiekt gry (this.gameObject), obiektu jego transformacji (this.transform), innych komponentów znajdujących się na tym obiekcie gry (this.GetComponent<T>()). Można również w wypadku nieprawidłowego przebiegu kodu bezpiecznie generować wyjątki — *Unity* przechwytuje je i wyświetla w konsoli nie przerywając działania gry.

Ostatnią kwestią do poruszenia w związku z *Unity* są tzw. prefaby. Prefab jest zapisanym w projekcie *Unity* obiektem gry — skopiowanym wprost ze sceny (ang. *prefabricated*). Prefab zawiera wszystkie informacje (położenie, hierarchia, komponenty i wartości ich parametrów), jakie zawierał obiekt gry w momencie jego zapisywania. Funkcjonalność ta daje rozległe możliwości. Podstawowym przypadkiem użycia prefabu jest stworzenie szablonu do obiektu gry, który może być potem powielany na scenie. Można jednak wykorzystać je również do zapisania parametrów konfigurujących różne aspekty gry. Wiedząc, że *Unity* tworzy pole interfejsu w panelu *Inspector* dla każdego publicznego pola w skrypcie, można stworzyć publiczne pole typu *GameObject* — a następnie za pomocą *drag&drop* przypisać mu w interfejsie obiekt gry. Ów obiekt gry może być również prefabem, który z kolei może mieć własny skrypt z publicznymi polami wypełnionymi danymi, albo nawet przypisanymi jeszcze innymi obiektami gry (lub prefabami). Możliwości takiego zagnieżdżenia są nieograniczone. Sposób ten jest kompleksowo wykorzystywany w prototypie m.in. przy konfiguracji drzewka technologicznego oraz akcji rozkazów.

## Obiekty globalne

Moduł ten składa się z klas ogólnego zastosowania, których obiekty w większości przypadków istnieją w tylko jednej kopii, lub nawet nie — w przypadku klas statycznych. Można tu wydzielić 5 kategorii:

* Singleton Globals,
* Obiekty gracza i armii,
* Obiekty środowiska,
* Obiekty konfiguracyjne,
* Klasy ze stałymi.

### Singleton Globals

Globals jest jedną z ważniejszych klas w projekcie. Klasa dziedziczy po MonoBehaviour, by można było jej skrypt umieścić na obiekcie gry — dzięki temu *Unity* automatycznie utworzy obiekt Globals. W grze może istnieć tylko jedna instancja tego skryptu (jako konwencja — nie jest to wymuszone). Globals luźno korzysta ze wzorca *singleton*: posiada statyczną właściwość Instance zwracającą obiekt tej klasy. **[źródło do wzorców projektowych]** Właściwość ta co prawda nie konstruuje obiektu, lecz za pierwszym do niej odwołaniem wyszukuje go na scenie i zapisuje do prywatnego pola statycznego. Dzięki temu mechanizmowi dostęp do obiektu Globals jest zapewniony z dowolnego miejsca w kodzie.

Dostęp ten jest istotny, ponieważ w klasie Globals znajduje się kilka publicznych pól z parametrami (możliwymi do ustawienia w *Inspectorze*), a na klasie — duża liczba statycznych właściwości do pobierania innych komponentów-skryptów z obiektu gry *Globals*. Są to m.in. takie skrypty, jak: MapSettings, Map, Textures, Prefabs, WallNeighbourhoodDictionary. Można się też dostać do instancji ShapeDatabase i LOSShapeDatabase — które nie są skryptami Unity, ale istnieją na obiekcie Globals.

Klasa Globals ułatwia dostęp do obiektów Spectator oraz — jeśli Spectator ma je przypisane — Player oraz Army. Udostępnia także listę armii biorących udział w rozgrywce. Posiada właściwości do pobrania obiektu gry posiadającego główną kamerę i skrypt GLRenderer (do którego delegowane są zadania renderowania linii — wykorzystane w wizualizacji zaznaczonych jednostek i HoverBoxa).

Wreszcie, Globals posiada metody Start() oraz OnDestroy() — przede wszystkim do obsługi informacji o tym, czy obiekt gry *Globals* w danej chwili istnieje (statyczna właściwość Destroyed). Wewnątrz Start() znajduje się też kilka instrukcji związanych z debugowaniem (m.in. utworzenie pliku będącego logiem wiadomości przesyłanych między agentami AI).

Należy zaznaczyć, że sporo komponentów obiektu gry *Globals* nie należy do podsystemu obiektów globalnych, gdyż mają pewne wyspecjalizowane funkcje.

### Obiekty gracza i armii

W skład tej kategorii wchodzą 3 klasy: Spectator, Player oraz Army. Obiekt klasy Spectator istnieje tylko w pojedynczym egzemplarzu, podczas gdy obiektów Player i Army może być kilka — obecnie jednak prototyp jest zaprogramowany na pracę z maksymalnie dwoma. Każda z tych klas dziedziczy po MonoBehaviour — skrypty będące ich instancjami są przypisane do obiektów gry odpowiedzialnych za armie, graczy oraz „widza” (Spectator).

Obiekt Army reprezentuje armię (stronę konfliktu w rozgrywce) i zawiera wszystkie niezbędne informacje z nią związane: zbiór jej jednostek, zbiór jej budynków, obiekt TechnologyController (zarządzający rozwojem jej technologii), liczbę aktualnie posiadanych zasobów oraz obiekt VisibilityTable (gromadzący dane o widzialności pól mapy). Oprócz tego w trzech obiektach QuadTree[[10]](#footnote-10) przetrzymuje widoczne przez nią na mapie zasoby, własne elementy mapy i wrogie elementy mapy. Jedyne publiczne metody tej klasy: AddMapElement() oraz RemoveMapElement() pozwalają na zarządzanie zbiorami jednostek i budynków. W prototypie są umieszczone dwa obiekty gry posiadające skrypt Army — dwie przeciwne sobie strony konfliktu.

Obiekt Player reprezentuje w prototypie gracza. Mowa tu o graczu w podejściu ogólnym — zarówno sterowanego przez człowieka, jak i przez sztuczną inteligencję. Klasa Player jest bardzo niewielka — jej jedyną składową jest publiczne pole Army. W ten sposób armia może zostać poprzez Inspector przypisana do gracza — w efekcie staje się ona jego armią.

Na podobnej zasadzie (dzięki kompozycji) obiekt Player przypisany jest do jego sterowania. Spectator — „widz” — jest jednym z obiektów, które mogą sterować graczem. Klasa Spectator jest również prosta (choć nie tak prosta jak Player). Posiada publiczne pole typu Player, oraz publiczną właściwość InputController — obiekt ten wykonuje niemal wszystkie zadania potrzebne do sterowania graczem i armią przez gracza-człowieka. W metodzie Update() Spectator wywołuje metodę InputController.Update() (ponieważ InputController nie jest skryptem, więc *Unity* samo nie wywoła aktualizacji). Należy podkreślić, że Spectator nie musi mieć wcale przypisanego gracza i wciąż będzie zapewniał ograniczone sterowanie grą. W tym trybie nadal można obsługiwać kamerę, obserwować poczynania armii, zaznaczać jej jednostki i podglądać ich status. Nie można natomiast wydawać żadnych rozkazów — a zatem wpływać na poczynania którejkolwiek z armii.

### Obiekty środowiska

Docelowo w tej kategorii miało się znaleźć więcej klas jednak jest tu tylko jedna — DayAndNight. Jest skryptem MonoBehaviour, gdyż potrzebuje skorzystać z metody Update(). Wewnątrz znajdują się:

* lista GameObject zawierającą dwa obiekty gry — światła imitujące księżyc oraz słońce przemieszczające się dookoła sceny,
* pole cycleTime, które przechowuje informację ile czasu trwa jeden cykl obrotu świateł,
* metoda Update()*,* która aktualizuje obroty świateł.

Wewnątrz metody znajduje się warunek obsługujący sytuację, w której wartość cycleTime jest równa 0 (wtedy 0 jest traktowane jak 1). Gdyby nie to, program wygenerowałby wyjątek dzielenia przez 0, gdyż prędkość obrotu jest obliczana poprzez odwrotność okresu. Na końcu metody Update() znajduje się pętla, która odpowiada za ustawienie każdemu światłu kierunku padania oraz obrotu względem środka układu współrzędnych.

public class DayAndNight : MonoBehaviour

{

public List<GameObject> lights;

public float cycleTime;

void Update()

{

var minutes = cycleTime;

if (minutes == 0) minutes = 1;

float seconds = minutes \* 360;

float speed = 360 / seconds;

foreach (var light in lights)

{

var transform = light.transform;

transform.RotateAround(

Vector3.zero, Vector3.right, speed \* Time.deltaTime);

transform.LookAt(Vector3.zero);

}

}

}

**Fragment kodu 1**. Klasa DayAndNight

### Obiekty konfiguracyjne

W tej kategorii mieszczą się dwie klasy mające tylko po jednej instancji — skrypty MonoBehaviour przypisane do obiektu gry *Globals*.

Klasa Prefabs zawiera publiczne pola typu GameObject, do których przypisane są prefaby do instancjonowania na scenie (zasoby, marker do minimapy oraz zasięg budowania). Dzięki nałożonemu na te pola atrybutowi PrefabTypeAttribute można przy pomocy metody GetPrefabByType() otrzymać listę prefabów o danym typie (wyrażonym jako enum PrefabType). Następnie z tej listy można pobrać losowy prefab danego typu — np. losowy zasób. Wykorzystywane jest to w momencie gdy niszczona jest jednostka lub budynek i zamieniane są one na zasoby.

Drugą klasą o charakterze konfiguracyjnym jest Textures. Znajdują się na niej jedynie publiczne pola typów Texture oraz Sprite – tekstury wykorzystywane z poziomu skryptów przy tworzeniu *GUI* (pasków życia elementów mapy oraz tzw. markerów na minimapie).

Obie klasy zapewniają konfigurację jedynie na bardzo ogólnym poziomie. Są jednak potrzebne, ponieważ obiekty zapewniane przez ich skrypty muszą być dostępne z poziomu kodu. Dlatego, jako komponenty *Unity*, są przypisane do obiektu gry *Globals*.

### Klasy ze stałymi

Do tej kategorii należą dwie klasy statyczne: Tag oraz Layer. Obie zawierają zestaw publicznych stałych typu string z nazwami tagów oraz warstw (dwóch mechanizmów kategoryzowania obiektów gry przez *Unity*).

## Podsystem elementów mapy

Elementy mapy to na tyle rozległy podsystem, że nie sposób go tu opisać dokładnie w całości. Szczegółowo omówiono więc tylko wybrane fragmenty modułu.

Klasa MapElement będąca skryptem MonoBehaviour jest tu głównym typem. Po tej klasie dziedziczą trzy następne uzupełniające jej funkcjonalność: Unit, Building oraz Resource(które poprzez dziedziczenie również są skryptami). Każdy obiekt gry znajdujący się na planszy jako jednostka, zasób, budynek bądź przeszkoda ma przypisany odpowiedni z tych skryptów. Ponieważ zarówno jednostka jak i budynek potrafią wykonywać rozkazy, obsługę rozkazów finalnie zdecydowano się umieścić w klasie MapElement*.* To, czy określony rodzaj elementu mapy potrafi wykonać dany rozkaz, nie zależy zatem od jego klasy (*ergo* od dziedziczenia), tylko od konfiguracji — czyli tego, jakie ma przypisane akcje rozkazów. Rozdział ten opisuje więc też obie klasy abstrakcyjne za to odpowiedzialne — Order i OrderAction — oraz ich przykładowe implementacje. Dodatkowo omówione są pobieżnie statystyki, technologie i zarządzanie nimi oraz mechanizm ustalania sąsiedztwa murów.

### Implementacja elementu mapy

Klasa MapElement ma niezwykle rozległą funkcjonalność. Musi ona wykonywać wszystkie zadania związane z obsługą elementów mapy, oraz zawierać wszelkie potrzebne do tego informacje. MapElement zawiera więc sporo publicznych pól (ustawialnych w panelu *Inspector*).

Pole mapElementName typu string jest nazwą określającą rodzaj elementu mapy (jest wspólna dla elementów jednego rodzaju i różna dla każdego z rodzajów — np. każdy czołg ma tu ustawione "Tank"). Pole id to generowany automatycznie int unikalny dla każdego elementu mapy znajdującego się na scenie.

Armia, do której należy element jest trzymana we właściwości Army. Przy jej zmianie MapElement wypisuje się ze zbioru jednostek lub budynków starej armii a wpisuje do zbioru w nowej.

public string mapElementName;

public int id;

public Army Army { get; private set; }

public TextAsset shapeFile;

public TextAsset statsFile;

public List<GameObject> aims;

public AttackHead attackHead;

public List<OrderAction> orderActions;

**Fragment kodu 2.** Wybrane publiczne pola klasy MapElement

Publiczne pola shapeFile i statsFile typu TextAsset służą do przypisania plików tekstowych z informacjami na temat kształtu elementu (np. budynku) oraz jego statystyk. Na ich bazie tworzone są później obiekty MapElementShape oraz Stats.

Lista aims obiektów GameObject to lista celów obieranych przez pociski wystrzeliwywane w kierunku elementu mapy przez jego wrogów. W momencie wykonywania ataku wybierany jest najbliższy z celów i pocisk leci do jego pozycji.

Pole attackHead trzyma referencję na obiekt będący osobno obracającą się głowicą jednostki/budynku, która może atakować. Korzystają z niego czołg, wieżyczka obronna (mają obrotową lufę) oraz mobilna wyrzutnia rakiet (ma obrotową prowadnicę).

public Stats Stats { get; private set; }

public MapElementShape Shape { get { return Globals.ShapeDatabase[this]; } }

public Vector2 Coords

{

get { return new Vector2(transform.position.x, transform.position.z); }

set

{

var pos = transform.position;

pos.x = value.x;

pos.z = value.y;

transform.position = pos;

}

}

public IEnumerable<IVector2> AllCoords

{

get

{

if (Shape == null) yield return Coords.Round();

else

{

var list = new List<IVector2>();

int xFrom = Mathf.RoundToInt(Coords.x + Shape.DeltaXNeg);

int xTo = Mathf.RoundToInt(Coords.x + Shape.DeltaXPos);

int yFrom = Mathf.RoundToInt(Coords.y + Shape.DeltaYNeg);

int yTo = Mathf.RoundToInt(Coords.y + Shape.DeltaYPos);

for (int x = xFrom, i = 0; x <= xTo; x++, i++)

for (int y = yFrom, j = 0; y <= yTo; y++, j++)

if (Shape[i, j])

yield return new IVector2(x, y);

}

}

}

public float Rotation

{

get { return transform.rotation.eulerAngles.y; }

set

{

var ea = transform.rotation.eulerAngles;

ea.y = value;

transform.rotation = Quaternion.Euler(ea);

}

}

**Fragment kodu 3.** Właściwości klasy MapElement związane ze statystykami, kształtem i transformacją

Publiczna lista obiektów OrderAction definiuje jakie rozkazy można wydać elementowi mapy. Dla każdego prefabu elementu mapy jest ona skonfigurowana w panelu *Inspector*. Każda przypisana akcja rozkazu umożliwia wydanie jednego rodzaju rozkazu.

Oprócz publicznych pól MapElement posiada też dużą liczbę właściwości. Stats trzyma statystyki elementu mapy. Shape pobiera jego kształt z ShapeDatabase. Coords pośredniczy w pobieraniu i ustawianiu pozycji — która nie musi być całkowita (np. jeśli jednostka jest w trakcie ruchu). Właściwość AllCoords zwraca kolekcję współrzędnych wszystkich pól zajmowanych przez MapElement. Rotation służy do pobierania i ustawiania obrotu elementu mapy wokół osi Y (pionowej).

Właściwości LifeValue, Dying i Alive służą kontrolowaniu czasu życia elementu mapy. LifeValue zwraca albo wartość statystyki „*Hit points*”, albo pozostałe jednostki zasobu (tylko w obiektach Resource). Gdy LifeValue wyniesie 0, metoda UpdateDying() ustawia Dying na **true** i nakazuje się przerwać wszystkim rozkazom. Ponieważ niektóre rozkazy nie mogą zostać przerwane natychmiast (np. pojedynczy ruch), MapElement może być „umierający” przez kilka cykli pętli gry. Gdy wszystkie rozkazy się zakończą, metoda UpdateAlive() ustawia Alive na **false**. Dying i Alive mają publiczne gettery, więc każdy obiekt może sprawdzać, czy dany element mapy jest „umierający”. Ustawienie Alive powoduje uruchomienie metody OnLifeEnd(), która finalizuje MapElement i niszczy jego obiekt gry. Wszyscy, którzy nasłuchują na zdarzeniu LifeEnding, zostają powiadomieni o tym, że MapElement ulega zniszczeniu i mogą na to zareagować.

Rozkazy MapElement trzyma na obiekcie klasy OrderQueue. Obiekt ten zawiaduje w całości kolejnością wykonywania rozkazów, które się na nim zakolejkuje. Udostępnia metody by rozkaz wydać (zakolejkować) lub anulować (usunąć, także poza kolejką). Można mu również ustawić domyślny rozkaz, który wykonywany jest przez MapElement, jeśli żaden inny nie został wydany. Rozkazem tym okazał się być we wszystkich przypadkach IdleOrder.

public OrderQueue OrderQueue { get; private set; }

public virtual bool Selectable { get { return false; } }

protected virtual bool CanAddToArmy { get { return false; } }

public virtual bool CanHaveGhosts { get { return true; } }

public virtual bool CanBeAttacked { get { return false; } }

public virtual bool CanBeEscorted { get { return false; } }

public virtual bool CanRotateItself { get { return false; } }

public bool CanAttack { get { return orderActions.Any(oa => oa.IsAttack); } }

public bool CanEscort { get { return orderActions.Any(oa => oa.IsEscort); } }

**Fragment kodu 4.** Właściwości klasy MapElement związane z kolejką rozkazów  
oraz definiujące możliwości elementu mapy

MapElement zawiera też szereg wirtualnych właściwości typu bool definiujących jego możliwości, które na różnych obiektach dziedziczących zwracają różne rezultaty. Przykładowo budynek i zasób nie mogą się obracać, więc CanRotateItself w klasach Building i Resource pozostaje takie, jak w bazowym MapElement — zwraca **false**, natomiast Unit nadpisuje tę metodę zwracając tam **true**. Jedynie CanAttack i CanEscort nie są wirtualne — zamiast tego po prostu stanowią skrót do sprawdzenia czy dany MapElement ma akcję rozkazu pozwalającą na atakowanie lub eskortę.

W klasie MapElement znajduje się kilka właściwości związanych z duchami. Należy zatem chociaż pobieżnie wyjaśnić to pojęcie. Sam pomysł duchów zaczerpnięty został z wypowiedzi twórcy systemu mgły wojny dla gry *Age of Empires*.[[11]](#footnote-11) Duch to klon elementu mapy, cechujący się ograniczoną funkcjonalnością. Powstaje dla dowolnego elementu mapy z wyjątkiem jednostki, w momencie gdy wszystkie pola, na których się znajduje, zostaną ukryte przez mgłę wojny. Duch posiada skrypt MapElement i ma skopiowane wszystkie wartości statystyk oraz wygląd — pokazuje zatem ostatni stan elementu mapy, kiedy jeszcze go było widać. W ten sposób ograniczana jest wiedza gracza o sytuacji poza polem widzenia. Na przykład gracz nie może wiedzieć, że zasób ukryty przez mgłę wojny jest właśnie zbierany przez przeciwnika. Duch jest niszczony dopiero, gdy MapElement stanie się na powrót widoczny. Każda armia widzi własne duchy, dlatego MapElement posiada słownik jego duchów, którego kluczem jest Army.

Istnienie duchów powoduje szereg problemów. Przykładem może być sytuacja, gdy zaznaczony budynek zostaje ukryty za mgłą. Należy wtedy odznaczyć oryginalny budynek i zaznaczyć jego ducha. Oprócz tego duch musi mieć możliwość być celem rozkazu (np. ataku), a jednocześnie taki rozkaz musi zmienić cel na oryginalny MapElement, gdy tylko ten zacznie być widoczny. Duchy muszą automatycznie zamieniać się z oryginalnymi elementami mapy w QuadTree trzymanych przez armie. Widać zatem, że konieczne jest, by duchy miały referencję do oryginalnych elementów mapy, choć jednocześnie powinny mieć możliwość istnieć niezależnie od nich (bo element mapy może zostać zniszczony gdy znajduje się poza polem widzenia i gracz nie może o tym wiedzieć). Podsumowując, duch musi być dla gracza nieodróżnialny od oryginalnego elementu mapy i imitować wszelakie jego zachowania. Apsekty te sprawiły, że implementacja mechanizmu duchów była niezwykle skomplikowana.

MapElement udostępnia dwie metody wirtualne chronionego dostępu: OnStart() i OnUdpate(), które są wołane w prywatnych funkcjach Start() i Update(). Dzięki temu inicjalizacja i aktualizacja może działać na dwóch poziomach dziedziczenia. Dla duchów metody te mają inne przebiegi, ale ze względu na brak miejsca nie zostaną one opisane.

protected virtual void OnStart()

{

alive = true;

if (!IsGhost)

{

id = NewId;

OrderQueue = CreateOrderQueue();

ReadStats();

if (nextArmy != null)

UpdateArmy();

UpdateDying();

UpdateAlive();

InitializeInMap();

InitializeMinimapMarker();

VisibleToSpectator = false;

VisibleToArmies = new Dictionary<Army, bool>();

foreach (var a in Globals.Armies)

VisibleToArmies[a] = false;

if (CanHaveGhosts)

{

Ghosts = new Dictionary<Army, MapElement>();

foreach (var a in Globals.Armies)

Ghosts[a] = null;

}

}

else

{

//\*\*\*\* Pominięty kod \*\*\*\*

}

}

**Fragment kodu 5.** Metoda MapElement.OnStart() dla elementu mapy nie będącego duchem

Metoda OnStart() generuje elementowi mapy id, tworzy kolejkę rozkazów, wczytuje mu z pliku statystyki i wstawia go do zbioru jednostek/budynków w armii, aktualizując przy tym tablicę widoczności (VisibilityTable). Może się okazać, że element mapy zostanie zniszczony tuż po stworzeniu, zanim jeszcze zostanie wywołana funkcja Start(), dlatego wołane są tu również UpateDying() i UpdateAlive(). Następnie InitializeInMap() rezerwuje pola w obiekcie Map, na których znajduje się MapElement, a InitializeMinimapMarker()[[12]](#footnote-12) tworzy marker widziany przez kamerę minimapy. Na końcu ustawiana jest widoczność elementu mapy (VisibleToSpectator i VisibleToArmies) oraz tworzony jest słownik duchów. Po wykonaniu tych czynności MapElement jest gotowy do działania.

Funkcja ReadStats() wczytuje statystyki z pliku zapisanego w standardzie *XML*. Wykorzystywana jest do tego .NETowa klasa System.Xml.XmlDocument.[[13]](#footnote-13)

MapElement zawiera szereg metod związanych ze sprawdzaniem wycinka mapy w pewnym zasięgu od elementu. Są one wykorzystywane przez rozkazy i do szybkiego przeszukiwania używają QuadTree. Np. GetClosestAimTo() pobiera jeden z celów MapElementu z listy aims — najbliższy zadanej pozycji. HasMapElementInRange() sprawdza z kolei, czy zadany element mapy znajduje się w zasięgu np. ataku. Natomiast PickClosestResourceInRange() zwraca najbliższy zasób w zasięgu widzenia (używane przez Harvestery w rozkazie HarvestOrder).

bool firstUpdate = true;

protected virtual void OnUpdate()

{

if (!IsGhost)

{

if (Army != nextArmy)

UpdateArmy();

if (CanHaveGhosts) UpdateGhosts();

UpdateArmiesQuadTrees();

if (CanHaveGhosts) AddGhostsToQuadTrees();

VisibleToSpectator = Globals.Armies

.Where(a => a.actionsVisible)

.Any(a => AllCoords

.Any(c => a.VisibilityTable[c.X, c.Y] == Visibility.Visible));

if (CanAttack)

UpdateAttack();

if (OrderQueue.Enabled)

OrderQueue.Update();

if (firstUpdate)

foreach (var kv in VisibleToArmies)

if (kv.Value)

kv.Key.InvokeOnVisibleMapElementCreated(this);

UpdateDying();

UpdateAlive();

}

else

{

//\*\*\*\* Pominięty kod \*\*\*\*

}

firstUpdate = false;

}

**Fragment kodu 6.** Metoda MapElement.OnUpdate() dla elementu mapy nie będącego duchem

Metody związane z atakiem to ReadyAttack() i MakeAttack(). Atak musi zostać najpierw przygotowany — bo przed jego wykonaniem należy wyliczyć różne parametry (jak np. kąt o jaki atakujący musi obrócić siebie bądź głowicę). Następnie może zostać uruchomiony. Naliczane zostaje wtedy opóźnienie ataku (tzw. *cooldown*), dzięki któremu jednostka wykonuje atak raz na określony czas.

W metodzie OnUpdate() wykonywanych jest kilka różnych czynności. Następuje aktualizacja armii (armia z nextArmy trafia do właściwości Army) oraz jej VisibilityTable. Aktualizowane są duchy (tworzone bądź usuwane), oraz pozycja elementu mapy w QuadTree każdej armii. Wartość właściwości VisibleToSpectator zostaje ustalona na bazie VisibilityTable armii którą steruje Spectator. Aktualizowany jest atak, jeśli został przygotowany i uruchomiony. Aktualizowana jest kolejka rozkazów (która wykonuje aktualizację rozkazu, lub uruchamia następny rozkaz, gdy dotychczasowy się zakończył). Metody UpdateDying() i UpdateAlive() wołane są na końcu, by na bieżąco zarządzać czasem życia elementu mapy.

### Statystyki

Za statystyki odpowiadają przede wszystkim klasy Stat i Stats. Klasa Stats jest pośrednikiem (wzorzec projektowy *proxy*) dla słownika statystyk, w których kluczem jest ich nazwa. Obiekt klasy natomiast Stat jest pojedynczą statystyką. Posiada nazwę i referencję na posiadający ją MapElement. Cechują ją trzy właściwości: float Value reprezentująca wartość statystyki, float MaxValue wyznaczająca jej maksymalną wartość oraz bool Limited, które mówi czy statystyka korzysta z tej maksymalnej wartości. Przykładowo statystyka „*Hit points*” ma Limited ustawione na **true**. Posiada więc pewną maksymalną wartość i Value na początku wynosi tyle co MaxValue. Natomiast statystyka „*Firepower*” korzysta jedynie z właściwości Value, więc Limited ustawione ma na **false**. Jeśli Limited jest **true**, to po zmianie Value lub MaxValue, wartość Value jest zawsze poprawiana by być w przedziale od **0** do wartości MaxValue.

<Stats>

<Stat name="Movement speed" value="3" />

<Stat name="Rotation speed" value="1.5" />

<Stat name="Hit points" value="120" max\_value="120" />

<Stat name="Firepower" value="5" />

<Stat name="Attack speed" value="10" />

<Stat name="Attack range" value="3" />

<Stat name="View range" value="4" />

</Stats>

**Przykład 1.** Plik *XML* ze statystykami jednostki MechMachinegun

Dodatkową klasą wspierającą statystyki jest StatBonus, używana przez technologie. StatBonus jest skryptem MonoBehaviour i służy do modyfikowania statystyk bez zmiany ich fizycznych wartości. Posiada publiczne pola do konfiguracji w panelu *Inspector*. Są to m.in. MapElement określający rodzaj elementu mapy, do którego statystyk bonus się aplikuje oraz wartość *float* która mówi jaka liczba jest dodawana do lub mnożona przez wartość statystyki. Metoda ApplyTo() przyjmuje wartość float i zwraca inną, zmodyfikowaną za pomocą bonusu.

### Technologie

Technologie znalazły się w tym module, gdyż mają bezpośredni związek z elementami mapy. Są odkrywane za pomocą budynków, ich odkrywanie spełnia wymagania do produkcji nowych jednostek i konstrukcji nowych budynków oraz odblokowuje bonusy do statystyk elementów mapy. Dwie klasy które odpowiadają za technologie to Technology i TechnologyController.

Technology jest skryptem MonoBehaviour. Jako publiczne pola udostępnia m.in. nazwę (string) i listę bonusów (List<StatBonus>), które można dzięki temu skonfigurować w edytorze *Unity*. Wymagania są konfigurowane w innym miejscu — w akcjach rozkazów.

Każda armia posiada jeden obiekt klasy TechnologyController, służący do składowania technologii już odkrytych oraz tych, które właśnie są w trakcie odkrywania. Zawiera on zarówno metody umożliwiające kontrolowanie tych aspektów, jak i pomocnicze funkcje pozwalające pobrać bonusy dla zadanego elementu mapy.

### Rozkazy

Klasa Order i jej potomne są stworzone jako połączenie wzorców projektowych *template method* oraz *command* **[znowu źródło do wzorców projektowych]**. Zamiast pojedynczej funkcji execute() (znanej ze zwykłego wzorca *command*), jest tu kilka chronionych wirtualnych metod do implementacji w klasach potomnych: OnStart(), OnUpdate(), LateOnUpdate()*,* OnStopping(), OnStopped(), OnFinished(), OnTerminating(). Są one w szablonowy sposób wołane przez publiczne metody Start(), Update(), Stop() i Terminate(). Wszystkie razem stanowią szkielet programu każdego rozkazu, co spełnia założenia wzorca *template method*. Jednocześnie za konstrukcję obiektów Order odpowiadają akcje rozkazów (klasa OrderAction i jej potomne), a wyżej wspomniane publiczne metody uruchamiane są przez kolejkę rozkazów OrderQueue znajdującą się na elemencie mapy. Implementacja operacji do wykonania jako obiektu, który może być tworzony w jednym miejscu, a wykorzystywany w innym oraz oddzielenie tej operacji od obiektu na którym pracuje (MapElement) spełnia więc z kolei założenia wzorca *command*.

Każdy rozkaz znajduje się w pewnym stanie (enum OrderState) — etapie jego życia. Świeżo utworzony obiekt Order ma stan BrandNew. Po wywołaniu przez OrderQueue metody Start() przechodzi do stanu Started, w którym OrderQueue woła jego metodę Update() co cykl pętli gry. Stamtąd może albo przejść do stanu Finished (jeśli rozkaz zakończył się normalnie, z wewnętrznej przyczyny), albo Stopping i następnie Stopped (gdy rozkaz jest zatrzymywany z zewnątrz przy pomocy metody Stop()). Jeśli wystąpił błąd i rozkaz trzeba natychmiast przerwać, wołana jest metoda Terminate() i rozkaz przechodzi w stan Terminated. Stany Finished, Stopped i Terminated są stanami końcowymi pozwalającymi na usunięcie rozkazu z kolejki OrderQueue.



**Rysunek 2.** Diagram stanów rozkazu

Niektóre rozkazy nie mogą zostać zatrzymane przy pomocy metody Stop() — są to tzw. rozkazy atomowe, czyli niewielkie komendy które muszą się wykonać w całości. Przykładem takiego rozkazu jest SingleMoveOrder, który wykonuje pojedyncze przemieszczenie się jednostki z jednej kratki mapy na sąsiednią. Przerwanie tego rozkazu w trakcie jego wykonywania spowodowałoby niedopuszczalną sytuację — jednostka zatrzymałaby się pomiędzy kratkami.

O tym w jaki sposób rozkaz da się przerywać decydują wirtualne właściwości CanStop, CanFinalizeStop oraz CanFinish. Domyślnie każdy rozkaz jest atomowy (czyli CanStop zwraca **false**).

Rozkazy nie mogą samodzielnie zmieniać swojego stanu (z wyjątkiem możliwości użycia metody Stop(), ale nie powinny tego robić — jest ona do użytku zewnętrznego). Obsługa stanu znajduje się wyłącznie w bazowej klasie Order. By sterować stanem, Order udostępnia klasom potomnym mechanizm rezultatu rozkazu. Klasa OrderResult jest niewielka, zawiera tylko informację o sukcesie (bool) oraz komunikat, w przypadku błędu (string). Chronione metody Succeed(), TrySucceed(), TryFail() oraz TryResolve() służą do ustawienia właściwości trzymającej rezultat. Funkcje szkieletowe (Start(), Update()) sprawdzają, czy istnieje już rezultat i zmieniają stan odnośnie do tego. Właściwości Failed, Succeeded i Conclusive zwracają **true** albo **false** na bazie rezultatu.

Klasa ComplexOrder dziedziczy po Order i rozszerza funkcjonalność zwykłego rozkazu o możliwość wykonywania pod rozkazów. Właściwie każdy rozkaz uruchamiany przez gracza jest rozkazem złożonym, a więc dziedziczy po tej klasie. Przykładowo rozkaz HarvestOrder korzysta naprzemiennie z rozkazów MoveOrder (ruch pomiędzy zasobem a rafinerią), CollectOrder (zbieranie zasobu) oraz DepositOrder (odkładanie zebranych jednostek zasobów do rafinerii).

public bool Failed { get { return Conclusive && !Result.Success; } }

public bool Succeeded { get { return Conclusive && Result.Success; } }

public bool Conclusive { get { return Result != null; } }

protected virtual bool CanStop { get { return false; } }

protected virtual bool CanFinalizeStop { get { return true; } }

protected virtual bool CanFinish { get { return true; } }

public Order(MapElement mapElement)

{

MapElement = mapElement;

State = OrderState.BrandNew;

}

public void Start()

{

if (State != OrderState.BrandNew) return;

TryFail(OrderResultAsserts.AssertMapElementIsNotDying(MapElement));

if (!Failed) OnStart();

if (Failed) Terminate();

else if (State != OrderState.Stopping) State = OrderState.Started;

}

public void Update()

{

if (!CanUpdate) return;

OnUpdate();

LateOnUpdate();

if (State == OrderState.Stopping) FinalizeStop();

else if (Conclusive) Finish();

}

public bool Stop()

{

if (!(State == OrderState.BrandNew ||

State == OrderState.Started && CanStop)) return false;

State = OrderState.Stopping;

OnStopping();

return true;

}

void Finish()

{

if (!CanFinish) return;

State = OrderState.Finished;

OnFinished();

}

public void Terminate(string explicitReason = null)

{

OnTerminating();

Debug.LogError(string.Format("Order {0} of {1} terminated: {2}",

Name, MapElement, explicitReason != null ? explicitReason :

Result != null ? Result.Message : "NULL"));

State = OrderState.Terminated;

}

**Fragment kodu 7.** Główne funkcje i właściwości klasy Order

public Order SubOrder { get; private set; }

protected override bool CanStop { get { return true; } }

protected override bool CanFinalizeStop { get { return !HasSubOrder; } }

protected override bool CanFinish { get { return !HasSubOrder; } }

protected void GiveSubOrder(Order subOrder)

{

if (subOrder.State != OrderState.BrandNew)

throw new System.ArgumentException(

"State property of suborder argument must be BrandNew.");

NextSubOrder = subOrder;

if (SubOrder == null)

AdvanceSubOrders();

}

void AdvanceSubOrders()

{

SubOrder = NextSubOrder;

NextSubOrder = null;

}

protected sealed override void LateOnUpdate()

{

if (!HasSubOrder) return;

if (SubOrder.State == OrderState.BrandNew)

{

OnSubOrderStarting();

SubOrder.Start();

if (SubOrder.State == OrderState.Started)

OnSubOrderStarted();

}

if (SubOrder.State == OrderState.Started ||

SubOrder.State == OrderState.Stopping)

{

OnSubOrderUpdating();

SubOrder.Update();

OnSubOrderUpdated();

}

if (SubOrder.State == OrderState.Finished)

{

OnSubOrderFinished();

AdvanceSubOrders();

}

else if (SubOrder.State == OrderState.Stopped)

{

OnSubOrderStopped();

AdvanceSubOrders();

}

else if (SubOrder.State == OrderState.Terminated)

{

OnSubOrderTerminated();

AdvanceSubOrders();

}

}

**Fragment kodu 8.** Główne funkcje i właściwości klasy ComplexOrder

Obiekt ComplexOrder samodzielnie trzyma referencję na aktualny i następny pod rozkaz nie korzystając z kolejki OrderQueue. Nadpisuje i pieczętuje funkcję LateOnUpdate() (słowo kluczowe sealed — następni potomkowie nie mogą nadpisać tej funkcji), w której implementuje automatyczną obsługę pod rozkazów. Udostępnia następne metody wirtualne w których rozkazy potomne mogą zareagować na zmianę stanu pod rozkazu (dalsze użycie wzorca *template method*). Poza tym zmienia też zachowanie właściwości decydujących, czy rozkaz może być zatrzymany (CanStop itd.). Dzięki zwracanym przez nie nowym wartościom,[[14]](#footnote-14) ComplexOrder nigdy nie jest atomowy (można zawołać Stop), ale nie zatrzyma się (będzie trwał w stanie Stopping), dopóki jego pod rozkaz nie zostanie zakończony. Funkcje OnStop()i OnTerminate()zostały zaś zaimplementowane w taki sposób, by powodowały zatrzymanie pod rozkazów.

### Akcje rozkazów

Abstrakcyjna klasa OrderAction służy dwóm celom. Po pierwsze, konstruując obiekty dziedziczące po Order, akcja rozkazu realizuje wzorzec projektowy *abstract factory* **[i jeszcze raz źródło do wzorców projektowych]**. Abstrakcyjna metoda CreateOrder() przyjmuje wykonawcę (MapElement) oraz parametry rozkazu (takie jak cel ataku albo ruchu), a zwraca nowo stworzony rozkaz. Dla każdej klasy rozkazu, który gracz może wydać elementowi mapy istnieje klasa akcji rozkazu, która go tworzy. Metoda CreateOrder() jest jednak chroniona. Upubliczniono za to funkcję GiveOrder(), która woła tę pierwszą, a stworzony rozkaz automatycznie kolejkuje na OrderQueue wykonawcy. Dodatkowo metoda wirtualna CanCreateOrder(), domyślnie zwracająca **true**, służy do sprawdzenia, czy wymagania do stworzenia rozkazu zostały spełnione.

public bool GiveOrder(MapElement orderExecutor,

IOrderActionArgs orderActionArgs)

{

if (orderExecutor.OrderQueue.Enabled && CanCreateOrder(orderActionArgs))

{

orderExecutor.OrderQueue.Give(

CreateOrder(orderExecutor, orderActionArgs));

return true;

}

return false;

}

protected virtual bool CanCreateOrder(IOrderActionArgs orderActionArgs)

{

return true;

}

protected abstract Order CreateOrder(MapElement orderExecutor,

IOrderActionArgs orderActionArgs);

**Fragment kodu 9.** Główne metody klasy OrderAction

Po drugie, klasa OrderAction dziedziczy po MonoBehaviour, może więc służyć do konfiguracji. W projekcie Unity zapisany jest szereg prefabów zawierających jako komponenty skrypty potomne do OrderAction. Prefaby te poustawiane są każdemu prefabowi elementu mapy na jego liście. W ten sposób skonfigurowane jest to, jakie rozkazy potrafi wykonywać każdy MapElement.

Same akcje rozkazów również mogą być konfigurowalne (sic[[15]](#footnote-15)). Dla trzech akcji rozkazów: UnitProductionOrderAction, BuildingConstructionOrderAction, oraz TechnologyDevelopmentOrderAction istnieje szereg prefabów ustawionych w różny sposób. Przykładowo komponentowi UnitProductionOrderAction przypisać można prefab jednostki, która jest tworzona, koszt i czas trwania produkcji oraz wymagania: listę budynków, które muszą być wybudowane i listę technologii, które muszą być zbadane.

### Sąsiedztwo murów

W zależności od tego z czym sąsiaduje mur - jego wygląd zostaje zmieniony. Na przykład jeżeli stawiamy go na otwartej przestrzeni, gdzie z niczym się nie styka - jego model jest podstawowy (w kształcie litery I). Natomiast jeżeli z jednej lub wielu stron będzie otoczony obiektami, wtedy zamiast bazowego wyglądu jego model zostanie zastąpiony innym – „skręcającym” w stronę otoczenia. Do zarządzania wyborem modelu muru, który ma zostać wybrany w konkretnej sytuacji używamy czterech skryptów:

* WallModelReplacer,
* WallNeighbourhood,
* WallNeighbourhoodDefinition,
* WallNeighbourhoodDictionary.

Skrypt WallNeighbourhood tworzy cztery zmienne typu bool, które określają zajętość pola względem modelu muru. WallNeighbourhoodDefinition jest skryptem MonoBehaviour, który został stworzony, aby w panelu Inspector można było skonfigurować sąsiedztwo muru. Dzięki podpięciu go pod prefaby odpowiednich modeli możemy wybrać jak chcemy je podmieniać. Dodatkowo posiada posiada publiczną właściwość Neigbourhood, która odpowiada za tworzenie i zwracanie nowego obiektu klasy WallNeighbourhood.

WallNeighbourhoodDictionary jest kolejnym skryptem MonoBehaviour. Posiada publiczną listę definicji, którą również można konfigurować w panelu *Inspector*. Tworzy on klucz za pomocą skopiowania definicji do słownika, którego kluczem jest WallNeighbourhood*.*

Skrypt WallModelReplacer sprawdza otoczenie, z którym sąsiaduje mur. Na tej podstawie tworzy on nowy WallNeighbourhood, który jest używany w WallNeighbourhoodDictionary, czyli podaje go jako klucz. Dzięki temu zwraca on wartość z odpowiednim modelem, który następnie jest podstawiany wewnątrz prototypu **MechWars**.

public GameObject auxiliaryModel;

WallNeighbourhood neighbourhood;

bool notGenerated = true;

void Update()

{

var mapElement = GetComponent<MapElement>();

var coords = mapElement.Coords.Round();

var up = Globals.Map[coords.X, coords.Y + 1];

var down = Globals.Map[coords.X, coords.Y - 1];

var right = Globals.Map[coords.X + 1, coords.Y];

var left = Globals.Map[coords.X - 1, coords.Y];

var neighbourhood = new WallNeighbourhood(

up is Building,

down is Building,

right is Building,

left is Building);

if (!neighbourhood.Equals(this.neighbourhood) || notGenerated)

{

if (auxiliaryModel != null) Destroy(auxiliaryModel);

var model = Globals.WallNeighbourhoodDictionary

.WallTypesDictionary[neighbourhood].gameObject;

auxiliaryModel = Instantiate(model);

auxiliaryModel.transform.SetParent(

gameObject.transform, false);

this.neighbourhood = neighbourhood;

notGenerated = false;

}

}

**Fragment kodu 10.** Ciało klasy WallModelReplacer

## Podsystem mapy

Podsystem mapy jest stosunkowo niewielki – zawiera 4 klasy umieszczone w przestrzeni nazw Mapping. Są to: MapSettings, Map, QuadTree i QuadTreeMapElement.

### Klasy mapy

MapSettings to niewielki skrypt MonoBehaviour znajdujący się jako komponent na obiekcie gry *Globals*. Posiada kilka publicznych pól: rozmiar mapy (musi być potęgą dwójki), listę graczy, listę obiektów gry — armii, oraz obiekt gry *Spectator*. Pola te należy wypełnić w edytorze — zwłaszcza musi być ustawiony *Spectator*. Jeśli rozmiar mapy nie jest potęgą dwójki lub *Spectator* nie jest ustawiony, skrypt MapSettings w swojej metodzie Start(), generuje stosowny wyjątek.

Najistotniejszym obiektem w tym podsystemie jest obiekt klasy Map. Tak jak MapSettings, jest on skryptem MonoBehaviour i komponentem obiektu gry *Globals*. Ponieważ plansza gry podzielona jest na kratki, jej obiekt przechowuje rezerwacje elementów mapy w postaci dwuwymiarowej tablicy. Obiekty MapElement znajdują się tam na odpowiednich współrzędnych — czasem na kilku, jeśli zajmują więcej niż jedno pole. Obiekt Map posiada też słownik, który ma odwrotną zależność: dla klucza MapElement trzyma listę jego pozycji. Mapa śledzi też położenia duchów w dwóch analogicznych strukturach danych. Poza tym publiczne właściwości umożliwiają pobranie rozmiaru mapy, listy pozycji elementu mapy, albo jego ducha (ze słownika) tudzież elementu mapy, lub ducha znajdujących się na podanej pozycji (z tablicy). Właściwość Size jest ustawiana na bazie MapSettings.Size w metodzie Start(). Mapa wystawia publiczne metody (sic[[16]](#footnote-16))IsInBounds() do sprawdzania czy dana pozycja znajduje się w granicach map,[[17]](#footnote-17) a także funkcje do: sprawdzania czy określone pole jest zajmowane przez zadany element mapy, tworzenia i zwalniania rezerwacji przez elementy mapy (MakeReservation(), ReleaseReservation()) oraz dodawania i usuwania duchów (AddGhost(), RemoveGhost()).

public int Size { get; private set; }

Dictionary<MapElement, List<IVector2>> reservationDictionary;

MapElement[,] reservationTable;

Dictionary<MapElement, List<IVector2>> ghostDictionary;

List<MapElement>[,] ghostsTable;

public void MakeReservation(MapElement mapElement, IVector2 coords)

{

if (mapElement == null)

throw new System.Exception("Cannot make reservation for NULL.");

if (this[coords] != null)

{

throw new System.Exception(string.Format("Reservation conflict. " +

"Coords: {0}, Old reservation: {1}, new reservation: {2}.",

coords.ToString(), this[coords].ToString(),

mapElement.ToString()));

}

this[coords] = mapElement;

List<IVector2> reservations;

reservationDictionary.TryGetValue(mapElement, out reservations);

if (reservations == null)

{

reservations = new List<IVector2>();

reservationDictionary.Add(mapElement, reservations);

}

reservations.Add(coords);

}

public void ReleaseReservation(MapElement mapElement, IVector2 coords)

{

if (mapElement == null)

throw new System.Exception("Cannot release reservation for NULL.");

if (this[coords] != mapElement)

{

var realReservation = this[coords] == null ?

"NULL" : this[coords].ToString();

throw new System.Exception(string.Format(

"Given MapElement doesn't have reservation in given coords. " +

"Coords: {0}, Real reservation: {1}, Given MapElement: {2}",

coords.ToString(), realReservation, mapElement.ToString()));

}

var reservations = reservationDictionary[mapElement];

reservations.Remove(coords);

if (reservations.Count == 0)

reservationDictionary.Remove(mapElement);

this[coords] = null;

}

**Fragment kodu 11.** Najważniejsze fragmenty klasy Map

Nazwa „rezerwacja” wynika z tego, że żaden element mapy nie może zajmować pola już zajętego przez inny element mapy. Jednostka jest ruchomym elementem mapy, musi więc na bieżąco aktualizować swoje położenie. W momencie gdy ma wykonać ruch z jednego pola na inne, sprawdza najpierw, czy jest ono wolne, lub zarezerwowane przez nią samą. Jeśli tak nie jest, ruch jest odwoływany. Jeśli nowe pole jest wolne, zostaje zarezerwowane, a ze starego pola rezerwacja jest zwalniana. Jeśli element mapy podejmie próbę zarezerwowania już zajętego pola, spowoduje to rzucenie wyjątku.

Metoda MakeReservation() przyjmuje MapElement (element mapy) oraz IVector2 (wektor dwóch liczb całkowitych — pozycję do zarezerwowania). Jeśli nie nastąpi konflikt rezerwacji, wstawia MapElement do zadanego miejsca tablicy. Następnie pobiera ze słownika rezerwacji listę pozycji elementu mapy i uzupełnia ją o nowe współrzędne.

Podobnie, choć odwrotnie, zachowuje się funkcja ReleaseReservation(). Jeśli MapElement faktycznie ma rezerwację w podanym miejscu, to zadana pozycja kasowana jest z jego listy współrzędnych w słowniku reservationDictionary. Następnie usuwany jest MapElement z zadanego miejsca w tablicy reservationTable.

Metod AddGhost() i ReleaseGhost() nie ma potrzeby omawiać szczegółowo, gdyż zachowują się one bardzo podobnie do funkcji MakeReservation() i ReleaseReservation(). Warto jedynie nadmienić, że duchy obowiązują luźniejsze zasady — na jednym polu może znajdować się kilka duchów (o ile są to duchy widziane przez różne armie), a także: duch może znajdować się tam, gdzie już znajduje się zwykły element mapy. Dlatego właśnie duchy umieszczone są w odrębnych strukturach danych, a ghostsTable jest tablicą **list** obiektów MapElement (podczas gdy reservationTable jest tablicą **pojedynczych** obiektów MapElement).

### Klasy drzewa czwórkowego

Pozostałe dwie klasy — QuadTree i QuadTreeMapElement — współpracują ściśle ze sobą i stanowią implementację tzw. drzewa czwórkowego. Struktura ta umożliwia bardzo szybkie przeszukiwanie wycinków planszy i znalezienie wszystkich znajdujących się w nich elementów mapy. Zastosowanie drzew czwórkowych było podyktowane problemami wydajnościowymi, które zachodziły, gdy jednostki w stanie spoczynku poszukiwały wrogów w swojej okolicy.

W implementacji drzewa w postaci klasy QuadTree każdy węzeł reprezentuje pewien obszar mapy. Korzeń to cała mapa. Jeśli w danym obszarze znajduje się więcej niż jeden element mapy, obszar dzielony jest na cztery ćwiartki — stają się one węzłami drzewa, których rodzicem jest właśnie podzielony obszar. Procedura jest powtarzana dla każdego z pomniejszych obszarów dopóty, dopóki każdy obszar nie będzie zawierał maksymalnie jednego elementu mapy.

Ponieważ dwuwymiarowa przestrzeń jest dyskretna i za każdym razem dzielona na 4 (w każdym wymiarze na 2), maksymalną głębokością drzewa będzie zawsze , gdzie to rozmiar planszy (liczba całkowita). Na przykład dla zastosowanego w prototypie **MechWars** rozmiaru 64×64 najniższe węzły drzewa będą 6-tymi co do głębokości. Znacznie zmniejsza to złożoność wyszukiwania elementów mapy w prostokątnym obszarze (funkcja QueryRange()). Operacje dodawania i usuwania elementów mapy są logarytmiczne, więc również szybkie, a podział i łączenie ma wręcz złożoność stałą. Wykonywane są one jednak zdecydowanie rzadziej niż przeszukiwanie, dlatego ich wydajność nie jest tu problemem.

QuadTreeMapElement to niewielka klasa trzymająca w jednej całości referencję na MapElement oraz pozycję w QuadTree (IVector2). Ponieważ istnieją budynki które zajmują więcej niż jedno pole, może być kilka obiektów QuadTreeMapElement wskazujących na ten sam MapElement ale mających różne pozycje.

QuadTreeMapElement QuadTreeMapElement;

SquareBounds bounds;

QuadTree x0y0;

QuadTree x0y1;

QuadTree x1y0;

QuadTree x1y1;

**Fragment kodu 12.** Pola klasy QuadTree

Klasa QuadTree nie udostępnia żadnych publicznych właściwości — wszystkie jej pola i właściwości są prywatne. Drzewo jest jednocześnie traktowane jako węzeł: każdy obiekt QuadTree zawiera cztery pola typu QuadTree, reprezentujące węzły potomne. Działa to, gdyż każdy węzeł jednocześnie stanowi pod-drzewo. Oprócz tego w klasie znajdują się jeszcze dwa pola: QuadTreeMapElement w którym (jeśli drzewo jest liściem) może być trzymany element mapy z jego pozycją oraz SquareBounds — klasa pomocnicza do testowania granic kwadratowego obszaru drzewa

Trzy metody: Insert(), InsertCore() oraz Subdivide() współpracują ze sobą. Insert() jest publiczną fasadą dla rekurencyjnej metody InsertCore(). W ogólnym zarysie InsertCore() szuka najniższego węzła drzewa (czyli poddrzewa), który zawiera współrzędną podanego w argumencie elementu mapy, i wstawia go do drzewa. Jeśli liść jest już zajęty, zostaje podzielony metodą Subdivide(). Subdivide() ma stałą złożoność, gdyż wymaga zawsze utworzenia czterech obiektów i przeniesienia elementu mapy z drzewa do jednego z jego dzieci. Przebieg Insert() zależy natomiast od lokalnej głębokości drzewa, więc jej złożoność jest .

Na podobnej zasadzie działa funkcjonalność usuwania obiektów MapElement z QuadTree. Publiczna metoda Remove() jest opakowaniem prywatnej, rekurencyjnej funkcji RemoveCore(), która poszukuje liścia zawierającego MapElement do usunięcia. Czasami gdy MapElement zostanie usunięty, należy jeszcze dostosować drzewo by spełniało reguły QuadTree — czyli połączyć dzieci metodą TryUnsubdivide().Metoda ta sprawdza, czy wszystkie dzieci QuadTree są liśćmi i dokładnie jedno z nich trzyma QuadTreeMapElement. Jeśli tak jest, QuadTreeMapElement z niepustego dziecka przenoszony jest do rodzica, a następnie wszystkie dzieci są usuwane. Widać, że złożoności tych operacji są analogiczne do złożoności przy dodawaniu: Remove() wykonuje się w czasie logarytmicznym, a TryUnsubdivde() — w stałym.

public List<QuadTreeMapElement> QueryRange(IRectangleBounds range)

{

var mapElements = new List<QuadTreeMapElement>();

if (!bounds.IntersectsOther(range))

return mapElements;

if (QuadTreeMapElement != null &&

range.ContainsPoint(QuadTreeMapElement.Coords))

mapElements.Add(QuadTreeMapElement);

if (!HasChildren)

return mapElements;

mapElements.AddRange(x0y0.QueryRange(range));

mapElements.AddRange(x0y1.QueryRange(range));

mapElements.AddRange(x1y0.QueryRange(range));

mapElements.AddRange(x1y1.QueryRange(range));

return mapElements;

}

**Fragment kodu 13.** Metoda QuadTree.QueryRange()

Ostatnią metodą QuadTree do opisania jest QueryRange(). Funkcja ta znajduje wszystkie obiekty MapElement znajdujące się w QuadTree w zadanym prostokątnym obszarze. QueryRange() również działa rekurencyjnie. Schodzi aż do wszystkich liści, których obszar przecina ten podany w argumencie. Niepuste liście zwracają jednoelementowe listy, a węzły na coraz wyższych poziomach łączą je w swoich listach. Na koniec metoda zwraca wypełnioną listę obiektów QuadTreeMapElement z całego wycinka drzewa.

Złożoność czasowa operacji QueryRange() jest bardziej skomplikowana. Zależy ona od wielu czynników, m.in. rozkładu danych w drzewie, a także rozmiaru i położenia prostokątnego obszaru do przeszukania. Trudno jest wyliczyć złożoność średnią, ale można ocenić złożoność maksymalną. Niech będzie liczbą elementów mapy w drzewie, a jego głębokością. Najszersze zapytanie (o obszarze całej mapy) zwróci oczywiście wszystkie elementy, więc jego złożoność to . Najwęższe z kolei będzie musiało przejść od korzenia do dokładnie jednego liścia, stąd . W połączeniu daje nam to złożoność .

## Podsystem mgły wojny

Pomysł na działanie mgły wojny został zaczerpnięty z książki *Perełki programowania gier*.[[18]](#footnote-18) Zarządzaniem informacją o kształcie mgły wojny zajmuje się klasa VisibilityTable. Każda armia ma własną tablicę widzialności[[19]](#footnote-19). Jest to obiekt zawierający dwie dwuwymiarowe tablice odwzorowujące rozmiarem planszę gry. Dla każdego pola gry w tablicy bool[,] fieldsUncovered znajduje się informacja, czy dane pole w ogóle zostało odkryte przez armię. Z kolei tablica int[,] fieldsSeenByUnits trzyma dla każdego pola liczbę jednostek, która je widzi. Odkryte pola, dla których liczba ta jest równa zeru, znajdują się we mgle wojny — nie widać na nich jednostek, a zamiast budynków i zasobów widać ich duchy[[20]](#footnote-20).

W VisibilityTable znajdują się metody IncreaseVisibility() i DecreaseVisibility(). Służą do inkrementacji i dekrementacji tablicy fieldsSeenByUnits przez MapElementy. Przykładowo gdy jednostka się porusza, dekrementuje pola tablicy w swoim zasięgu widzenia względem do starego położenia, a następnie inkrementuje te względem nowego. By uniknąć kosztownego wyliczania pól znajdujących się wewnątrz okręgu pola widzenia wyznaczanego przez statystykę „*View range*” (będącą promieniem okręgu), tablica widzialności korzysta z obiektu LOSShapeDatabase, który w leniwy sposób generuje kształty pola widzenia (w książce zwane „szablonami linii widoczności”) dla różnych promieni (po stworzeniu trzyma kształt i następnym razem po prostu go zwraca). Funkcje te uruchamiają też zdarzenie VisibilityChanged, na którym nasłuchuje sztuczna inteligencja by móc aktualizować swą wiedzę.

Tablica widzialności posiada też właściwość-indeksator zwracającą wygodną w użyciu wartość enuma Visibility — przyjmuje on wartości Unknown, Fogged oraz Visible. Elementy mapy sprawdzają wartość tej właściwości, gdy określają to, czy je widać. Wpływa to zarówno na faktyczne renderowanie elementu mapy, jak i na jego obecność w QuadTree armii. Informacja o widzialności ma również związek z decyzją o tworzeniu lub niszczeniu duchów oraz wiedzą *AI*.

Graficzny efekt mgły wojny jest renderowany na bazie VisibilityTable. Skrypt VisualFog (dziedziczący po MonoBehaviour) generuje w locie teksturę o wymiarach takich jak plansza, gdzie jeden piksel odpowiada jednemu polu. Piksele są czarne, ale różnią się składową alpha. Dla pól nieodkrytych są całkowicie czarne, pola we mgle są półprzezroczyste a widzialne — całkowicie przezroczyste. Tekstura ta nakładana jest następnie na płaszczyznę zasłaniającą planszę, stanowiąc coś w rodzaju filtru koloru dla tego, co widzi kamera. Dzięki użyciu warstw w *Unity* mgła zasłania jedynie podłoże, nie przykrywając żadnych elementów mapy. Klasa MinimapFog kopiuje teksturę z VisualFog i nakłada ją na własną płaszczyznę widoczną przez kamerę minimapy.[[21]](#footnote-21)

bool[,] fieldsUncovered;

int[,] fieldsSeenByUnits;

void IncreaseVisibilityOfTile(int x, int y)

{

if (x < 0 || Size <= x || y < 0 || Size <= y) return;

bool justUncovered = !fieldsUncovered[x, y];

fieldsUncovered[x, y] = true;

fieldsSeenByUnits[x, y]++;

if (VisibilityChanged != null)

if (justUncovered) VisibilityChanged(new IVector2(x, y),

Visibility.Unknown, Visibility.Visible);

else VisibilityChanged(new IVector2(x, y),

Visibility.Fogged, Visibility.Visible);

}

public void IncreaseVisibility(MapElement mapElement, float x, float y)

{

var meShape = mapElement.Shape;

var radiusStat = mapElement.Stats[StatNames.ViewRange];

if (radiusStat == null) return;

var radius = radiusStat.Value;

var losShape = Globals.LOSShapeDatabase[radius, meShape];

IncreaseVisibility(x, y, losShape);

}

void DecreaseVisibilityOfTile(int x, int y)

{

if (x < 0 || Size <= x || y < 0 || Size <= y) return;

fieldsSeenByUnits[x, y]--;

if (VisibilityChanged != null && fieldsSeenByUnits[x, y] == 0)

VisibilityChanged(new IVector2(x, y),

Visibility.Visible, Visibility.Fogged);

}

public void DecreaseVisibility(MapElement mapElement, float x, float y)

{

var meShape = mapElement.Shape;

var radiusStat = mapElement.Stats[StatNames.ViewRange];

if (radiusStat == null) return;

var radius = radiusStat.Value;

var losShape = Globals.LOSShapeDatabase[radius, meShape];

DecreaseVisibility(x, y, losShape);

}

**Fragment kodu 14:** Wybrane składowe klasy VisibilityTable

## Podsystem poszukiwania ścieżek

Podsystem ten został stworzony w ogólny sposób, tak by można było zaimplementować dowolny algorytm na szukanie ścieżek. Interfejs IPathdfinder wystawia fukcję FindPath(), którą każda klasa poszukująca ścieżek musi posiadać. Przyjmuje ona punkt startowy, punkt docelowy oraz MapElement, który szuka ścieżki Jednak istnieje w projekcie tylko jedna klasa implementująca ten interfejs — jest to AStarPathfinder.

AStarPathfinder jest realizacją popularnego algorytmu *A\**, będącego rozszerzeniem algorytmu Edsgera Dijkstry do wyszukiwania ścieżki w grafie. *A\** uzyskuje lepszą wydajność poprzez użycie heurystyki. AStarPathfinder traktuje planszę jako graf, gdzie każde pole jest węzłem posiadającym ośmiu sąsiadów — pola sąsiednie. Odległość wyznaczana jest geometrycznie: między polami sąsiadującymi bokiem wynosi , a po przekątnej — . Graf generowany jest na bieżąco — algorytm tworzy węzeł dla pola, dopiero gdy potrzebuje go odwiedzić. Jeśli docelowe pole jest zajęte lub nieosiągalne, algorytm zwraca ścieżkę do osiągalnego pola najbliższego celowi. Użytą w algorytmie heurystyką jest euklidesowa odległość do punktu docelowego pomnożona przez 5.

Przeszacowanie odległości pozwala poprawić czas wykonywania algorytmu (mniej węzłów zostaje odwiedzonych), choć sprawia, że algorytm nie gwarantuje już najkrótszej ścieżki. Jest to tak zwana ograniczona relaksacja heurystyki. Liczba stanowi ograniczenie relaksacji gwarantujące, że rozwiązanie nie będzie gorsze niż najlepsze rozwiązanie pomnożone przez. Heurystyka taka to heurystyka dopuszczalna. Istnieje kilka dopuszczalnych odmian algorytmu *A\**. Wybraną przez nas jest odmianą jest *A\** statycznie ważony. W tej odmianie stanowi wagę przez którą mnoży się wartość heurystyki.[[22]](#footnote-22) U nas wynosi zatem 5. Wartość tę dobrano eksperymentalnie, jako taką, która wystarczająco poprawia wydajność jednocześnie nie zmniejszając skuteczności algorytmu w widocznym stopniu.

## Podsystem sterowania

Z modułu tego zdecydowano się przedstawić jedynie jeden jego mechanizm. Jest nim sposób decydowania o wyborze miejsca budynku, poprzez wyświetlanie tzw. cienia budynku (pojęcia nie należy mylić z duchem elementu mapy!).

Każdy z prefabów modeli budynków ma podpięty skrypt BuildingShadow. Odpowiada on za stworzenie *GameObjectu* oraz wyświetlanie go w określonych warunkach. Wewnątrz skryptu znajduje się pare funkcji, jednak najważniejsze z nich to:

* UpdateLocation(),
* UpdateLook().

Klasa UpdateLocation() odpowiada za lokalizację oraz przyciągnie cienia budynku. Na początku sczytywane są współrzędne dotyczące położenia kursora oraz przypisywane do poszczególnych zmiennych. Na tej podstawie wyliczane są nabliższe punkty, do których obiekt będzie przyciągany. Dzięki temu uniemożliwione jest budowanie obiektu pomiędzy kratkami planszy. Następnie sprawdzane są warunki dostępności budowania. Cała operacja jest powtarzana przy każdym odświeżeniu klatki. Jeżeli chociaż jedno pole jest poza dozwolonym zasięgiem budowania lub nachodzi na inny obiekt - cień będzie miał kolor czerwony. W przeciwnym wypadku będzie on zielony. Klasa UpdateLook() odpowiada za wizualną stronę cienia budynku – nadaje mu odpowiednie barwy.

void UpdateLocation()

{

var coords = inputController.Mouse.MapRaycast.PreciseCoords;

InsideMap = coords.HasValue;

if (!coords.HasValue) return;

var p = coords.Value;

var shape = prefab.Shape;

var W = shape.Width;

var H = shape.Height;

float xSnap1, xSnap2, ySnap1, ySnap2;

if (W % 2 == 0)

{

xSnap1 = Mathf.Floor(p.x - 0.5f) + 0.5f;

xSnap2 = Mathf.Ceil(p.x - 0.5f) + 0.5f;

}

else

{

xSnap1 = Mathf.Floor(p.x);

xSnap2 = Mathf.Ceil(p.x);

}

if (H % 2 == 0)

{

ySnap1 = Mathf.Floor(p.y - 0.5f) + 0.5f;

ySnap2 = Mathf.Ceil(p.y - 0.5f) + 0.5f;

}

else

{

ySnap1 = Mathf.Floor(p.y);

ySnap2 = Mathf.Ceil(p.y);

}

float x, y;

if (Mathf.Abs(p.x - xSnap1) > Mathf.Abs(p.x - xSnap2))

x = xSnap2;

else x = xSnap1;

if (Mathf.Abs(p.y - ySnap1) > Mathf.Abs(p.y - ySnap2))

y = ySnap2;

else y = ySnap1;

Position = new Vector2(x, y);

shadow.Coords = Position;

var allCoords = shadow.AllCoords.ToList();

bool cannotBuild = false;

foreach (var c in allCoords)

{

if (Globals.Map[c] != null ||

inputController.ConstructionRange.FieldInRange(c))

{

cannotBuild = true;

break;

}

}

CannotBuild = cannotBuild;

shadow.transform.position = new Vector3(Position.x, 0, Position.y);

}

**Fragment kodu 15.** Funkcja BuildingShadow.UpdateLocation()

## Podsystem interfejsu gracza

Jedną z podstawowych rzeczy w grze jest interfejs gracza. Pozwala on na ułatwioną komunikację między użytkownikiem a mechaniką gry. Elementami wspierającymi rozgrywkę są:

* Skróty klawiszowe przycików,
* Opisy guzików,
* Minimapa.

### Skróty klawiszowe

Przydatną funkcjonalnością w grach typu *RTS* są skróty klawiszowe, dzięki którym gracz może przyspieszyć swoją rozgrywkę. W prototypie **MechWars** odpowiadają za to odpowiednie skrypty podpięte do poszczególnych przycisków. Są to:

* OrderActionButton,
* OrderUtilityButton,
* OrderActionButtonController.

OrderActionButton jest klasą dziedziczącą po MonoBehaviour. Znajduje się w niej funkcja OnClick\_ChangeCarriedOrderAction(), która odpowiada za zablokowanie innych akcji na rzecz wywołanej po naciśnięciu przycisku na interfejsie gracza. Następnie w OnClick\_ChangeCarriedBuildiingAction() wywoływane są cień budynku, który ma zostać zbudowany oraz jego zasięg dostępności. Funkcja OnClick\_InvokeOrderAction() dla każdej z zaznaczonych jednostek lub budynku natychmiastowo wywołuje akcję przypisaną do naciśniętego przycisku. Ostatnie dwie funkcje odpowiadają za podświetlenie przycisków. OnPointerEnter() ustawia wybrany przez gracza guzik na podświetlony. Natomiast OnPointerExit() sprawia, że po zakończonej operacji żaden przycisk nie jest podświetlany.

public void OnClick\_ChangeCarriedOrderAction()

{

Globals.Spectator.InputController.CarriedOrderAction = orderAction;

}

public void OnClick\_ChangeCarriedBuildingOrderAction()

{

var inputController = Globals.Spectator.InputController;

var selected = inputController.SelectionMonitor.SelectedMapElements;

inputController.CarriedOrderAction = orderAction;

inputController.CreateShadow(

(BuildingConstructionOrderAction)orderAction);

inputController.CreateConstructionRange((Building)selected.First());

}

public void OnClick\_InvokeOrderAction()

{

var inputController = Globals.Spectator.InputController;

var selected = inputController.SelectionMonitor.SelectedMapElements;

foreach (var mapElement in selected)

{

orderAction.GiveOrder(mapElement,inputController.OrderActionArgs);

}

}

public void OnPointerEnter()

{

canvasScript.SetHoveredButton(GetComponent<Button>());

}

public void OnPointerExit()

{

canvasScript.SetHoveredButton(null);

}

**Fragment kodu 16.** Metody klasy OrderActionButton

Klasa OrderUtilityButton również dziedziczy po MonoBehaviour i jest bardzo podobna do OrderActionButton. Nie posiada funkcji odpowiadających za blokowanie akcji po naciśnięciu przycisku. Zamiast tego zaprogramowano guzik *Cancel*, który anuluje wybrane działanie w funkcji OnClick\_CancelProduction(). Dodatkowo OnClick\_CancelSelection() sprawia, że dany przycisk nie znika z interfejsu gracza.

public void OnClick\_CancelProduction()

{

var buildings = Globals.Spectator.InputController

.SelectionMonitor.SelectedMapElements.OfType<Building>();

var bld = buildings.FirstOrDefault();

if (bld != null)

bld.OrderQueue.CancelCurrent();

}

public void OnClick\_CancelSelection()

{

Globals.Spectator.InputController.SelectionMonitor.ClearSelection();

}

**Fragment kodu 17.** Obsługi zdażeń przycisków *CancelProduction*  
i *CancelSelection* w klasie OrderUtilityButton

Ostatnim, nie mniej ważnym skryptem jest OrderActionButtonController, w którym główna klasa także dziedziczy po MonoBehaviour. Na początku sprawdzane jest czy jakakolwiek budowla została zaznaczona przez gracza. Jeżeli tak to funkcja wyciąga informacje na jej temat. Typ bool sprawia, że na interfejsie pojawia się przycisk anulujący produkcję tylko w przypadku, gdy został wskazany pojedyńczy budynek. W przypadku gdy nic nie zostało wybrane lub wrogie obiekty są oznaczone to wszystkie przyciski dotyczące rozkazów znikają. Jeżeli zaznaczone zostały różne jednostki lub budynki, ale posiadają wspólne funkcjonalności to tylko one pojawią się na panelu interfejsu wewnątrz gry. Na koniec sprawdzane jest czy wymagania dotyczące produkcji zostały spełnione i jeżeli jest to prawda – wtedy przycisk jest aktywny i można go użyć.

void Update()

{

var selected = Globals.Spectator.InputController

.SelectionMonitor.SelectedMapElements.ToList();

Building building = null;

if (selected.Count == 1 && selected.First() is Building)

building = (Building)selected.First();

bool cancelProductionButtonVisible = building != null &&

building.orderActions.Any(oa =>

oa is UnitProductionOrderAction ||

oa is BuildingConstructionOrderAction ||

oa is TechnologyDevelopmentOrderAction);

cancelProductionButton.gameObject.SetActive(

cancelProductionButtonVisible);

if (selected.Empty() || selected.Any(

me => me.Army != Globals.HumanArmy) ||

Globals.Spectator.InputController.CarriesOrderAction ||

building != null && building.UnderConstruction)

{

foreach (var b in buttons) b.gameObject.SetActive(false);

return;

}

var orderActionNames = new HashSet<string>();

bool first = true;

foreach (var mapElement in selected)

{

if (first)

{

orderActionNames.UnionWith(mapElement.orderActions

.Select(oa => oa.gameObject.name));

first = false;

}

else orderActionNames.IntersectWith(mapElement.orderActions

.Select(oa => oa.gameObject.name));

}

foreach (var b in buttons)

{

bool active = orderActionNames.Contains(

b.orderAction.gameObject.name);

if (active)

{

var coa = b.orderAction as BuildingConstructionOrderAction;

var poa = b.orderAction as UnitProductionOrderAction;

var doa = b.orderAction as TechnologyDevelopmentOrderAction;

if (coa != null)

active = coa.CheckRequirements(Globals.HumanArmy);

else if (poa != null)

active = poa.CheckRequirements(Globals.HumanArmy);

else if (doa != null)

active = doa.CheckRequirements(Globals.HumanArmy) &&

Globals.HumanArmy.Technologies.CanDevelop(doa.technology);

}

b.gameObject.SetActive(active);

}

}

**Fragment kodu 18.** Funkcja OrderActionButtonController.Update()

### Opisy guzików

W celu lepszego zrozumienia przez gracza funkcji poszczególnych przycisków na interfejsie gracza został napisany skrypt ButtonsInfoDisplayer, który został podpięty do pola nad guzikami. Odpowiada za wyświetlanie podpowiedzi i zawiera nie tylko nazwę, ale także opis działania oraz skrót klawiszowy. Dodatkowo wykorzystywany jest skrypt InfoDisplayer, w którym znajduje się klasa do wyświetlania tekstu.

public class ButtonsInfoDisplayer : InfoDisplayer

{

Button hoveredButton;

Vector2 anchorPosition;

void Start()

{

anchorPosition = GetComponent<RectTransform>().anchoredPosition;

}

void Update()

{

var rectTransform = GetComponent<RectTransform>();

rectTransform.anchoredPosition = anchorPosition +

new Vector2(-rectTransform.rect.width, rectTransform.rect.height);

}

public void SetHoveredButton(Button button)

{

if (button == hoveredButton) return;

hoveredButton = button;

var text = GetComponent<Text>();

if (button == null)

{

text.text = "";

}

else

{

var oaButton = button.GetComponent<OrderActionButton>();

var ouButton = button.GetComponent<OrderUtilityButton>();

if (oaButton != null)

text.text = oaButton.description;

else if (ouButton != null)

text.text = ouButton.description;

}

}

}

**Fragment kodu 19.** Klasa ButtonsInfoDisplayer

### Wygląd minimapy

Minimapa to miniaturowy obraz, który pokazuje świat gry lub jego część. Jest ona zazwyczaj ortograficznym rzutem z kamery od góry. Dzięki niej gracz może w łatwy sposób zorientować się w położeniu ważnych elementów w przestrzeni świata gry. W związku z ograniczoną wielkością na ekranie gracza pokazuje ona najważniejsze strategiczne elementy, takie jak budynki czy jednostki, które są oznaczone kolorowymi symbolami. Dodatkowo bardzo często na minimapach pokazywana jest mgła wojny.[[23]](#footnote-23)



**Ilustracja 1.** Podgląd minimapy z zasobami oraz  
wybudowanymi sojuszniczymi budynkami i jednostkami

W naszym prototypie minimapa jest teksturą renderowaną w czasie rzeczywistym z ortograficznej kamery, znajdującej się nad sceną. Generowany przez nią co klatkę obraz jest zapisywany w projekcie *Unity* jako tekstura MinimapRenderTexture. Następnie jest ona nakładana na element *GUI*, aby ją móc wyświetić.

Bardzo istotną rzeczą przy jej tworzeniu był fakt, żeby widać na niej było tylko konkretne obiekty gry. W *Unity* można to osiągnąć ustawiając pożądane obiekty gry na odpowiednich warstwach sceny i określając które z tych warstw mają być renderowane przez kamerę. W naszym przypadku chcemy renderować:

* planszę gry,
* symbole strategicznych elementów – tak zwane markery,
* mgłę wojny.

Komponent Camera na obiekcie gry *MinimapCamera* jest zatem skonfigurowany tak, by kamera renderowała jedynie warstwy:

* *Minimap*, na której znajdują się markery,
* *MinimapFogOfWar*, czyli płaszczyznę mgły wojny do minimapy
* *Enviro*, czyli teren.

Do tworzenia markerów w czasie trwania rozgrywki napisana została metoda InitializeMinimapMarker() w klasie MapElement. Korzysta ona z wirtualnej funkcji GetMarkerImage(), by określić wygląd markera. Wersje tych funkcji w klasach potomnych zachowują się inaczej dla każdego z rodzajów elementu mapy. Building zwraca okrągły symbol, a Unit — kwadratowy. Dla obu tych typów tekstura markera ma kolor armii elementu mapy (niebieski albo czerwony). Jeśli armia nie jest przypisana - barwa markera jest biała. Resource natomiast zwraca zawsze żółty kwadrat. Dzięki temu markery jednoznacznie określają rodzaj i przynależność elementu mapy.

void InitializeMinimapMarker()

{

var markerImage = GetMarkerImage();

if (markerImage == null) return;

var markerPrefab = Globals.Prefabs.marker;

var marker = Instantiate(markerPrefab);

var sr = marker.GetComponent<SpriteRenderer>();

sr.sprite = markerImage;

marker.transform.localScale \*= Mathf.Max(Shape.Width, Shape.Height);

var pos = marker.transform.localPosition;

pos.y = GetMarkerHeight();

marker.transform.localPosition = pos;

marker.transform.SetParent(this.gameObject.transform, false);

}

**Fragment kodu 20.** Metoda MapElement.InitializeMinimapMarker()

Aby zdeterminować, odległość markera od kamery, a więc wzajemne zasłanianie się markerów, stosowana jest wirtualna funkcja GetMarkerHeight(). Jest to zaprogramowane w taki sposób, aby jednostki zasłaniały zasoby, które z kolei przykrywać będą budynki. Z reguły nie powinno do tego dochodzić, gdyż nie może być dwóch elementów mapy na tym samym polu. Jednak przy odmiennym ustawieniu markery budynków mogłyby zasłaniać pozostałe markery. Wynika to z faktu, że budynek może zajmować więcej niż jedno pole i mieć niekwadratowe wymiary, natomiast jego marker zawsze będzie okręgiem — większym bądź mniejszym.

Renderowanie mgły wojny na minimapie jest realizowane dzięki skryptowi MinimapFog. Klasa MinimapFog w funkcji Update() kopiuje teksturę z VisualFog i nakłada ją na własną płaszczyznę na warstwie *MinimapFogOfWar*, która jest wyświetlana tylko przez kamerę minimapy. W ten sposób obie płaszczyzny minimapy korzystają z tej samej tekstury.

public class MinimapFog : MonoBehaviour

{

public VisualFog visualFog;

void Update()

{

GetComponent<Renderer>().material.mainTexture =

visualFog.GetComponent<Renderer>().material.mainTexture;

}

}

**Fragment kodu 21.** Klasa MinimapFog

# Kreacja graficzna prototypu

Każda gra oprócz części programistycznej składa się również z kreacji graficznej, która nie tylko ułatwia korzystanie z niej, ale także nadaje walorów estetycznych. Ważną rolę odgrywa zarówno interfejs gracza, jak i trójwymiarowe modele czy efekty specjalne wewnątrz gry.



## Grafika dwuwymiarowa

Na grafikę dwuwymiarową do naszego prototypu składa się wiele powiązanych ze sobą elementów stworzonych w programie *Adobe Photoshop*. Są to:

* własny wygląd kursora myszy wewnątrz prototypu,
* interfejs głównego menu gry,
* interfejs gracza oraz jego elementy widoczne podczas rozgrywki.

### Kursor myszy

Jednym z podstawowych walorów estetycznych jest wygląd kursora. Prawie każda gra posiada zaprojektowany własny kontroler myszy pasujący dobrze do klimatu rozgrywki. W prototypie **MechWars** znajduje się klasa CursorController, która odpowiada głównie za przemieszczanie się kursora. Za wygląd kursora odpowiada obiekt gry za wygląd kursora odpowiada obiekt gry *Cursor*, o ustawionej w panelu *Inspector* teksturze. Przypisany do niego skrypt CursorController zajmuje się przemieszczaniem go po ekranie.

Klasa CursorController zawiera funkcje Start() oraz Update(). Pierwsza z nich wyłącza podstawowy kursor Windows. Druga ustala jego położenie co cykl pętli gry. W tym celu funkcja Update() pobiera aktualne położenie myszki z obiektu InptController i ustawia ją jako pozycję w transformacji obiektu gry *Cursor*.

public class CursorController : MonoBehaviour

{

void Start()

{

Cursor.visible = false;

}

void Update()

{

var pos = Globals.Spectator.InputController.Mouse.Position;

var cursor = gameObject;

var rt = cursor.GetComponent<RectTransform>();

rt.position = pos;

}

}

**Fragment kodu 22.** Klasa CursorController

### Budowa interfejsów gracza

W związku z obowiązującym aktualnie trendem flat designu postanowiliśmy wykorzystać go w naszej pracy. Mimo iż nie jest on popularny w grach *RTS*, wygląda czysto i schludnie, a jego główną cechą jest minimalizm oraz prostota. Interfejs gracza jest możliwy do zaprogramowania na różne sposoby, między innymi:

* *Immediate Mode GUI*, czyli tak zwane *IMGUI*,
* *Canvas GUI*.

Interfejs *IMGUI* wykorzystano do stworzenia głównego menu. W jego skrypcie znajduje się podstawowa klasa MainMenuScript, w której oprócz funkcjonalności została opisana część wytycznych dotyczących wyglądu. Zanim do przycisków zostanie przypisany graficzny wygląd w panelu Inspector trzeba ustalić ich parametry. Na przykład w funkcji FirstMenu() jest tworzony przycisk o określonych wymiarach bazujących częściowo na wysokości rozdzielczości ekranu. Jednak już marginesy pomiędzy elementami a danym guzikiem są wpisane na sztywno. Wewnątrz znajduje się również warunek, który odpowiada za włączanie konkretnego podmenu po przyciśnięciu danego przyciskuJest to problematyczne z dwóch względów:

* Nie od razy widać efekty pracy,
* Elementy nie zawsze dobrze się skalują przy zmianie rozdzielczości ekranu.

void Start()

{

\_gameVolume = PlayerPrefs.GetFloat("Game Volume", \_gameVolume);

if (PlayerPrefs.HasKey("Game Volume"))

AudioListener.volume = PlayerPrefs.GetFloat("Game Volume", \_gameVolume);

else PlayerPrefs.SetFloat("Game Volume", \_gameVolume);

}

void OnGUI()

{

GUI.skin = GameSkin;

GUI.Label(new Rect(Screen.width / 2 - 50, 35, 300, 25),

gameTitle, "Menu Title");

FirstMenu();

PlayMenu();

OptionsMenu();

}

void FirstMenu()

{

if (GUI.Button(new Rect(50, Screen.height / 2 - 100, 181, 48),

"", "Play Button Style"))

{

if (\_isPlayMenu == false)

{

\_isPlayMenu = true;

\_isOptionsMenu = false;

}

else \_isPlayMenu = false;

}

//\*\* Pominięty analogiczny kod, dot. Przycisku "OPTIONS" \*\*//

}

**Fragment kodu 23.** Wybrane składowe klasy MainMenuScript

Niektóre z przycisków w głównym menu po kliknięciu otwierają podmenu, co pozwala na wybór dodatkowych opcji. Guziki stworzone w prototypie **MechWars** to:

* „**PLAY**”: pozwala nam wybrać tryb gry, który chcemy rozpocząć:
  + „**AI vs AI**”: ma na celu wprowadzenie gracza ludzkiego do rozgrywki w trybie obserwatora, co pozwala mu na oglądanie rozgrywki bez aktywnego udziału,
  + „**PLAYER vs AI**”: wczytuje scenę rozgrywki w trybie potyczki gracza ludzkiego przeciwko sztucznej inteligencji,
* „**OPTIONS**”: otwiera podmenu konfiguracji:
  + „**AUDIO**”: włącza i wyłącza dźwięki w grze,
  + „**FULLSCREEN**”: przechodzi między trybem okienkowym a pełnoekranowym,
* „**EXIT**”: wyłącza prototyp **MechWars**.

Pomimo graficznego działania przycisków, nie wszystkie mają zaprogramowaną funkcjonalność. Przykładem jest guzik „**AUDIO**”. Mimo zmiany wyglądu i pojawiania się zaznaczenia i odznaczenia nie włącza on dźwięków w grze, gdyż nie zostały one stworzone.

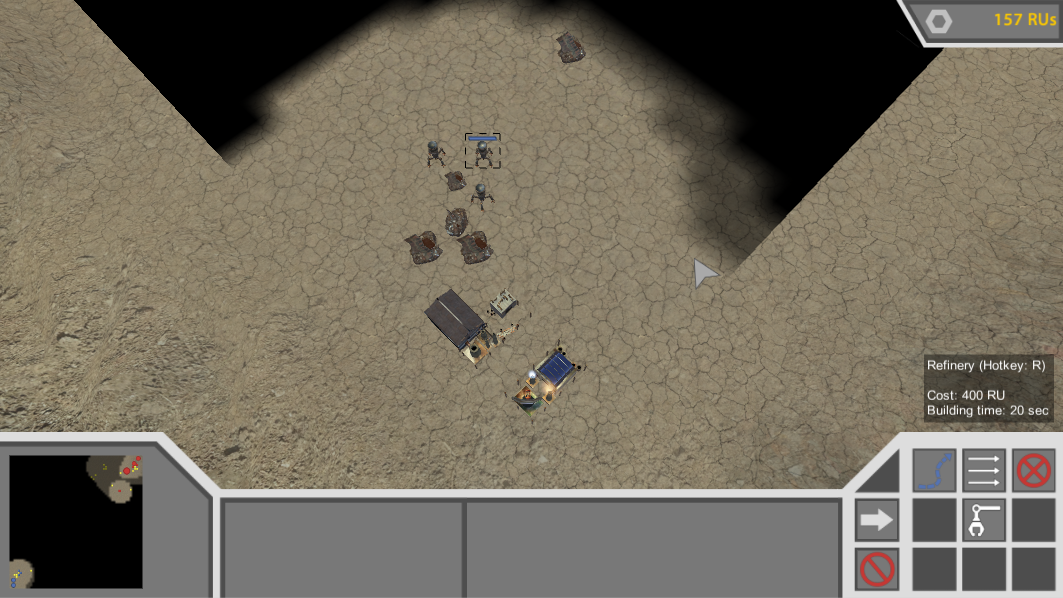


**Ilustracja 2.** Podgląd głównego menu w programie *Unity*

Interfejs graficzny wyświetlany w czasie rozgrywki został stworzony przy pomocy *Canvas GUI*. W tej metodzie umieszcza się elementy interfejsu na specjalnym obiekcie gry o nazwie Canvas. Na jego obszarze umieszczane są dwuwymiarowe obrazki, tak zwane *Sprite’y*. Dużym ułatwieniem w tworzeniu tego typu interfejsu jest możliwość przypięcia poszczególnych obiektów do konkretnych powierzchni, które są jego tak zwanym rodzicem. Dzięki temu podczas skalowania obiektu nadrzędny, podpięte do niego dzieci będą się przeskalowywać i zmieniać pozycję w odpowiednich proporcjach razem z nim. Dodatkowo w panelu *Inspector* możemy wybrać takie rzeczy, jak wybór czcionki oraz jej wielkość.

W górnym prawym rogu ekranu znajduje się licznik zasobów, które aktualnie posiada gracz. Po prawej stronie interfejsu została umieszczona minimapa, natomiast po lewej znajdują się pola, w który zmieniają się wyświetlane przyciski w zależności od zaznaczonego obiektu podczas rozgrywki. Każdy z guzików ma przypisany do siebie konkretny rozkaz. Stałymi klawiszami są anulowanie akcji wybranego przycisku oraz przejście do następnej strony z opcjami, jeżeli jest ich więcej niż dziewięć. Na przykład jeżeli zaznaczymy jednostkę *Harvestera* pokazują nam się cztery przyciski wraz z opisami:

* *Move*, po wskazaniu celu zaznaczona jednostka udaje się do wybranego przez gracza miejsca,
* *Escort*, jednostka eskortuje inną sojuszniczego osobnika,
* *Stop*, zatrzymuje jednostkę,
* *Harvest*, po wskazaniu konkretnego zasobu zaznaczone przez gracza osobnik udaje się w tamto miejsce i wydobywa surowiec.



**Ilustracja 3.** Podgląd interfejsu gracza z przyciskami dostępnymi dla *Harverstera* w programie *Unity*

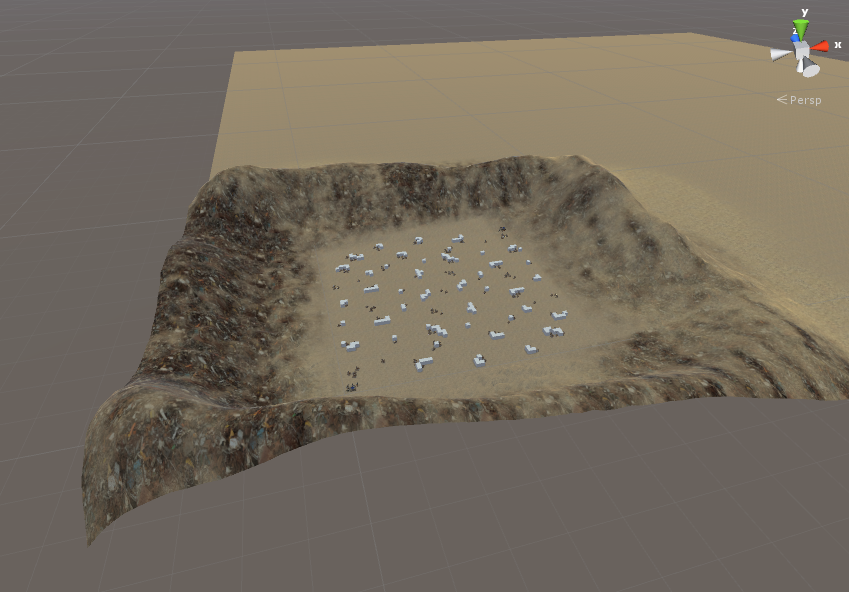
## Grafika trójwymiarowa

Duża liczba gier typu *RTS* wykorzystuje grafikę przestrzenną do przedstawienia świata, w którym toczy się rozgrywka. W naszym prototypie grafika trójwymiarowa została użyta w formie modeli terenu, zasobów, budynków, jednostek i przeszkody oraz systemu cząteczkowego dymu. Do stworzenia wszystkich tych elementów wykorzystaliśmy programy *Autodesk 3ds Max® 2016*, *Adobe Photoshop* oraz *Unity*.

### Modele

Obszar, na którym toczy się rozgrywka został stworzony stworzony wewnątrz *Unity* jako trójwymiarowa płaszczyzna, czyli tak zwany *Plane*. Program posiada narzędzia do modelowania terenu pozwalające dowolnie kształtować wzniesienia i doliny. Jednak jednym z przyjętych uproszczeń prototypu jest płaska plansza — pozbawiona ukształtowania terenu w obszarze gry. Użycie zupełnie płaskiej płaszczyzny spowodowało, że gdy kamerę przesunięto blisko krawędzi, widoczne było czarne tło. By temu zaradzić, stworzono większy teren i wypiętrzono obszar dookoła tworząc kotlinę. Dzięki temu nawet po maksymalnym oddaleniu kamery czarne tło wciąż pozostaje poza polem jej widzenia.

Inną funkcjonalnością udostępnionych narzędzi jest malowanie terenu określonymi przez użytkownika teksturami. W celu uzyskania klimatu prototypu zgodnego z założeniami zdecydowaliśmy się na motywy spękanej gleby dla planszy gry oraz złomowiska dla zewnętrznych gór. W ten sposób uzyskaliśmy efekt pustynnej areny otoczonej ogromnymi hałdami złomu.



**Ilustracja 4.** Podgląd terenu planszy w programie *Unity*

Kolejnymi ważnymi elementami w prototypie **MechWars** są trójwymiarowe budynki, w których skład wchodzą:

* Fabryka jednostek,
* Laboratorium
* Rafineria złomu,
* Warsztat budowlany.

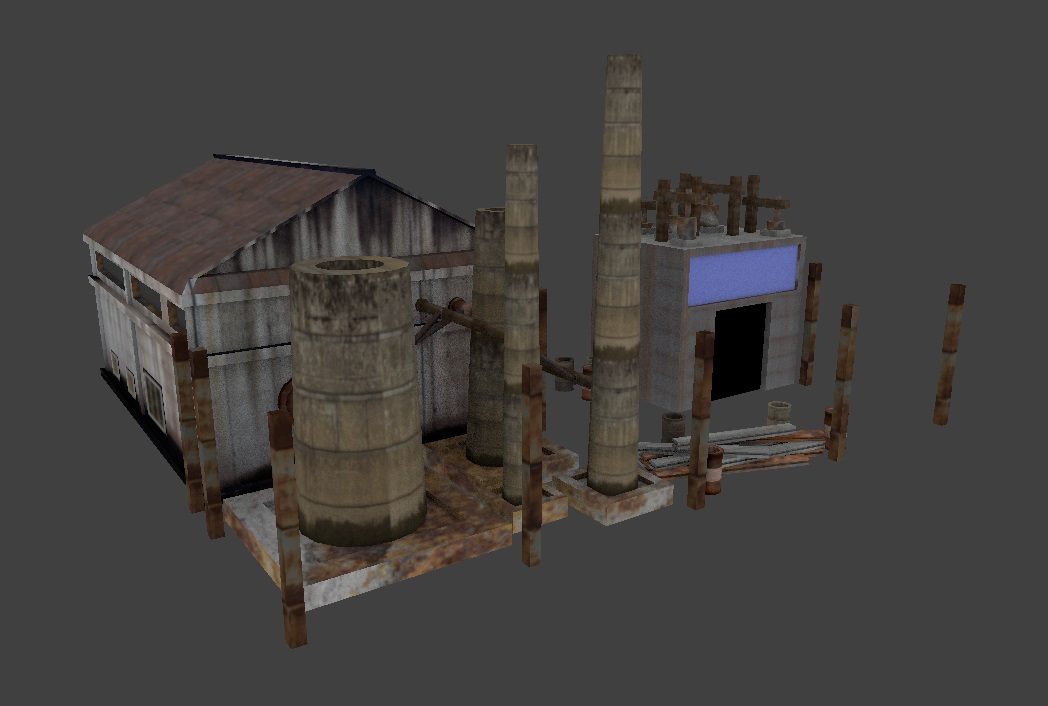
Wszystkie budowle mimo iż posiadają wspólną cechę – otoczone siatką, to są zróżnicowane, dzięki czemu gracz może z łatwością je od siebie odróżnić. Wszystkie obiekty posiadają kominy przemysłowe oprócz laboratorium.



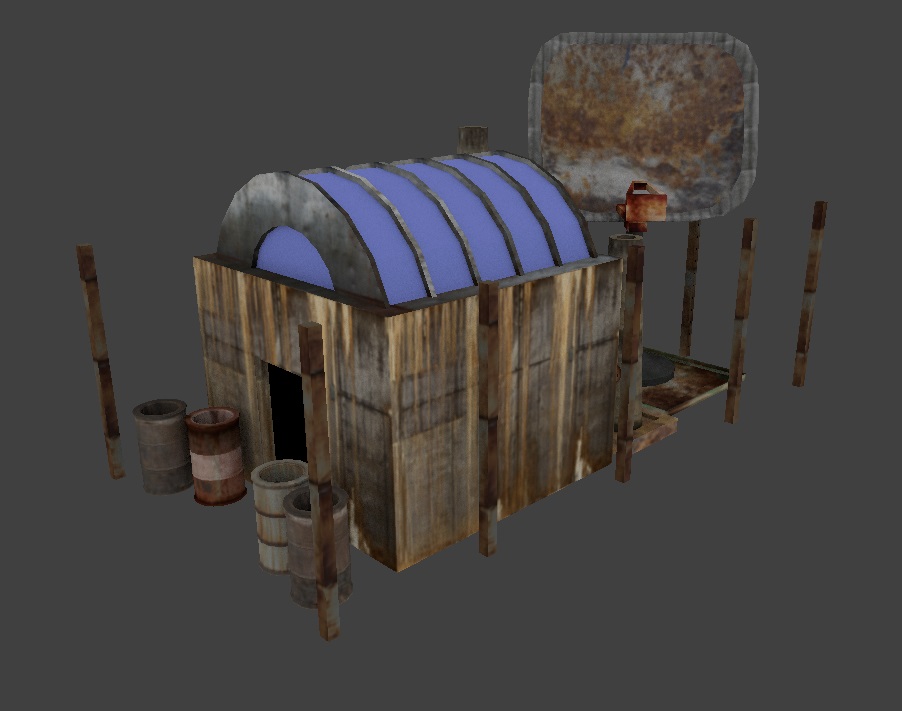
**Ilustracja 5.** Podgląd trójwymiarowego modelu fabryki jednostek



**Ilustracja 6.** Podgląd trójwymiarowego modelu laboratorium



**Ilustracja 7.** Podgląd trójwymiarowego modelu rafinerii złomu



**Ilustracja 8.** Podgląd trójwymiarowego modelu warsztatu budowlanego

Modele jednostek na potrzeby prototypu **MechWars** zostały stworzone w oparciu o to, do czego założeniach służą. Podstawowymi oddziałami militarnymi są blisko- oraz dalekozasięgowy mech. Są bardzo podobne do siebie, jednak różnią się typem posiadanej broni. Bliskozasięgowa jednostka posiada plecakowy miotacz ognia, w związku z czym z tyłu modelu umieszczono dwie butle z materiałem pędnym. Zbiornik zawierający mieszankę zapalającą jest ukryty w przedramionach ze względów bezpieczeństwa, w związku z czym go nie widać. Dalekozasięgowa jednostka natomiast posiada karabin maszynowy typu minigun, w związku z czym z tylu korpusu posiada miejsce na zbiorniki amunicji.

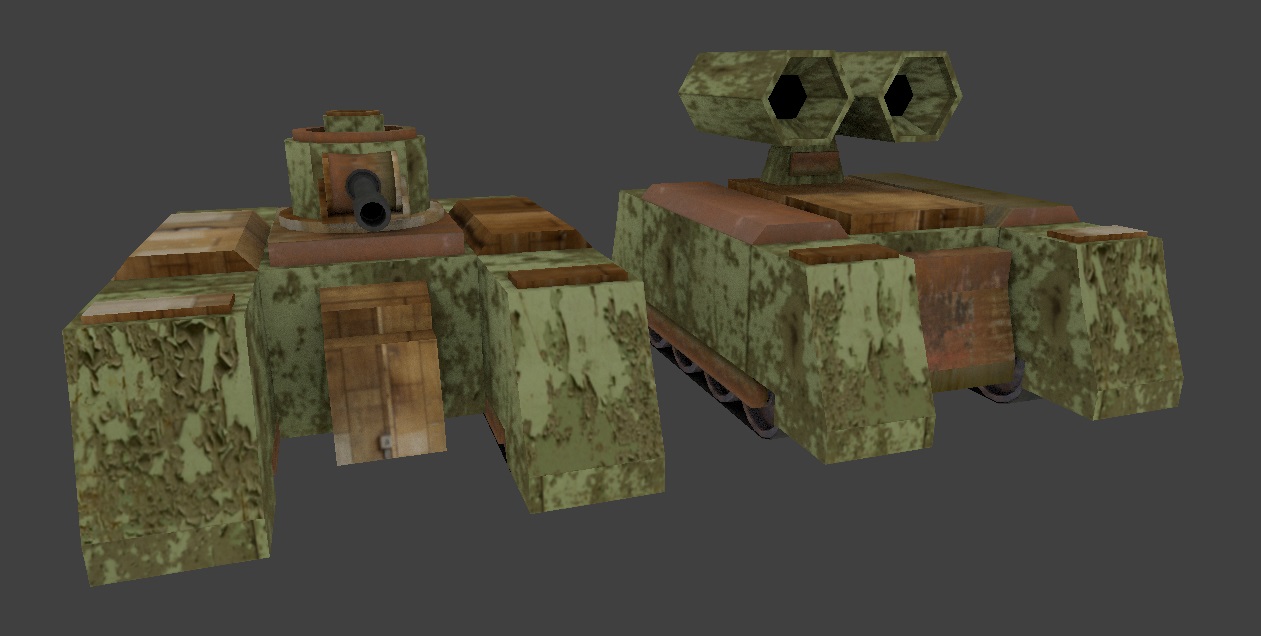


**Ilustracja 9.** Podgląd trójwymiarowego modelu dalekozasięgowego mecha



**Ilustracja 10.** Podgląd trójwymiarowego modelu bliskozasięgowego mecha

Kolejnymi jednostkami są dwa pojazdy militarne. Czołg posiada obrotową wieżyczkę dowódcy. Jest do niej zamontowany wielokalibrowy karabin maszynowy, który może zmienić kąt nachylenia, dzięki czemu może strzelać zarówno do mniejszych, jak i większych celów. Drugą jednostką jest pojazd, który zamiast wieżyczki oraz lufy posiada dwie połączone ze sobą wyrzutnie rakiet. Oba modele są zbliżone do siebie wyglądem i poruszają się na metalowych kołach.



**Ilustracja 11.** Podgląd trójwymiarowych modeli czołgu i mobilnej wyrzutni rakiet

*Harvester* jest jedną z dwóch niemilitarnych jednostek w prototypie **MechWars**. Jego zadaniem jest wydobywanie zasobów oraz transportowanie ich do rafinerii. W związku z tym został zaprojektowany tak, by jego kończyny uginały się podczas zbierania, a korpus był obrotowy i posiadał chwytacze. Drugą jednostką jest zwiadowca, który powinien się szybko poruszać, w związku z czym został skonstruowany z grubej opony opasanej metalowymi obręczami i felg z kolcami. Dodatkowo zamiast korpusu został wyposażony w kamerę na okrągłej głowicy. Dzięku takiemu zabiegowi może z łatwością zmieniać kąt patrzenia bez konieczności obracania się całym ‘ciałem’.

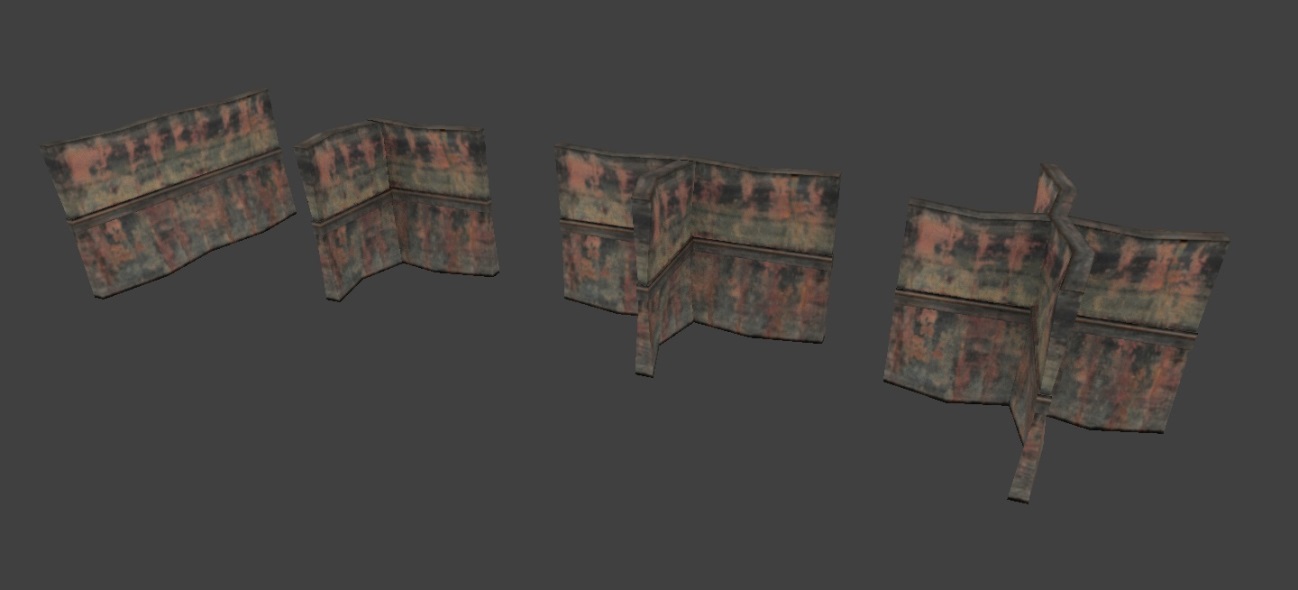


**Ilustracja 12.** Podgląd trójwymiarowych modeli *Harvestera* i zwiadowcy

Kolejnymi modelami są dwie fortyfikacje, które w założeniu mają bronić budynki oraz niemilitarne jednostki gracza przed przeciwnikiem. W związku z tym, że koncepcją prototypu był świat pełen złomu - mur jest podobny wyglądem do wygiętej blachy, która nosi ślady rdzy. Drugim modelem jest wieżyczka, która posiada obrotową głowicę razem z lufą.



**Ilustracja 13.** Podgląd trójwymiarowego modelu wieżyczki



**Ilustracja 14.** Podgląd trójwymiarowych modeli murów

Pozostałymi trójwymiarowymi modelami użytymi w prototypie są dwie, różniące się od siebie hałdy złomu stanowiące złoża surowców. Skrypt MapElement zawiera funkcję generującą zasoby po zniszczeniu jednostki lub budynku. Model każdego z nich jest losowo wybierany spośród wspomnianych dwóch. Dodatkowo do sceny zostały dodane przeszkody w postaci białych szcześcianów stworzonych w *Unity*.



**Ilustracja 15.** Podgląd trójwymiarowych modeli zasobów

### System cząsteczkowy dymu

Dla dodatkowych walorów estetycznych postanowiliśmy dodać wypuszczanie różnego rodzaju dymów z kominów budynków. Aby to osiągnąć skorzystaliśmy z systemu cząsteczkowego w *Unity*. Do dowolnego obiektu gry można przypisać komponent *ParticleSystem*, który służy do generowania cząsteczek. Można go skonfigurować pod kątem:

* koloru cząsteczek,
* częstotliwości ich emisji,
* ich szybkości, typu oraz kąta obrotu,
* ich wyglądu.

Z systemu cząsteczek skorzystaliśmy tworząc puste obiekty gry i nadając im jedynie komponent *ParticleSystem*. Te obiekty umieszczono następnie w pożądanych dla generacji cząsteczek współrzędnych — na przykład w wylocie komina fabryki. Zgrupowano je dla każdego budynku, tworząc nowy obiekt gry, którego skonfigurowano jako ich rodzica. Do owego obiektu nadrzędnego przypisano skrypt, zarządzający momentem rozpoczęcia ich emisji.

W klasie ParticleGroup znajdują się dwie główne funkcje Start() oraz Update(). Pierwsza z nich zatrzymuje na starcie emisję cząsteczek danej grupy. Natomiast druga odpowiada za sprawdzenie czy budynek jest w trakcie konstrukcji. Jeżeli tak jest to generowanie dymu zostanie włączone dopiero po zakończonej akcji budowania.

public class ParticleGroup : MonoBehaviour

{

bool active;

public Building building;

public List<ParticleSystem> particles;

void Start()

{

foreach (var ps in particles)

ps.Stop();

}

void Update()

{

if (building == null) return;

if (!active && !building.UnderConstruction)

{

active = true;

foreach (var ps in particles)

ps.Play();

}

}

}

**Fragment kodu 24.** Klasa ParticleGroup

# Zaprogramowanie sztucznej inteligencji

Z teoretycznego punktu widzenia gra *RTS* znacznie różnie się od tradycyjnej gry planszowej, takiej jak szachy. Głównymi różnicami są:

* *RTS* posiada równoczesność ruchów — wielu graczy może wykonywaćakcje w tym samym czasie.
* Akcje w grze *RTS* trwają pewien czas — nie wykonują się w jednym momencie.
* Każdy gracz ma bardzo niewielki czas na decyzję o następnych ruchach — gra wykonująca się w tempie 24 klatek na sekundę oznacza, że gracz może działać nawet co około 42 ms, zanim stan gry ulegnie zmianie.
* Stan rozgrywki poprzez mechanizm mgły wojny jest tylko częściowo widoczny.

Wreszcie, złożoność tych gier zarówno pod względem rozmiaru przestrzeni stanów jak i liczby akcji możliwych do podjęcia w każdym cyklu decyzyjnym jest olbrzymia. Przykładowo przestrzeń stanów szachów jest estymowana na około 1050, pokera *Texas Hold’em* w odmianie *no limit* — około 1080, a *Go* — około 10170. W porównaniu do tego, przestrzeń stanów w grze *StarCraft* dla typowej mapy jest szacowana na wiele rzędów wielkości większą.

Z tych powodów standardowe techniki sztucznej inteligencji w grach, takie jak przeszukiwanie drzew rozwiązań, nie mają zastosowania w grach *RTS* — przynajmniej nie bez zdefiniowania pewnego poziomu abstrakcji lub innego uproszczenia.[[24]](#footnote-24)

Aby sztuczna inteligencja była kompletna (czyli mogła pełnoprawnie symulować gracza ludzkiego) koniecznym jest, by sprostała szeregowi wyzwań. Po zanalizowaniu tematu sformułowaliśmy te wyzwania i możliwe ich rozwiązania.

Okazuje się, że *AI* musi spełniać szereg zachowań prowadzących do zwycięstwa. Są to zarówno zachowania ekonomiczne, takie jak zbieranie zasobów, wznoszenie budynków, produkcja jednostek oraz odkrywanie technologii, jak i zachowania taktyczne: zwiady, tworzenie obrony, przewidywanie ataków wroga i wybieranie mniej uczęszczanych przez niego ścieżek. Sztuczna inteligencja powinna synchronizować własne ataki, obsadzać jednostkami miejsca o walorze strategicznym, tworzyć zasadzki i priorytetyzować cele ataku.

Przytoczone zachowania są bardzo różnorodne. *AI* musi wykonywać wiele odmiennych zadań w jednym momencie. Oprócz tego zadania te mogą być wysokiego poziomu (np. decyzja o kolejności w jakiej będą opracowywane technologie) jak i niskiego poziomu (np. sterowanie pojedynczą jednostką). Potrzebny jest system który umożliwi takie wieloaspektowe i wielopoziomowe sterowanie armią.

Sztuczna inteligencja nie potrafi w prosty sposób przetwarzać informacji na temat planszy, gdy zapisane są po prostu w postaci dwuwymiarowej tablicy pól. Konieczne jest stworzenie struktur danych, które grupują kratki mapy w obszary o pewnym konkretnym znaczeniu dla *AI* — na przykład regiony o wysokiej koncentracji zasobów, albo terytorium wrogiej bazy.

Podobnie sztuczna inteligencja może mieć problem, by rozróżnić jednostki między sobą ze względu na funkcje i to, do czego się nadają, bazując wyłącznie na ich statystykach i przypisanych akcjach rozkazów. To samo tyczy się sposobów na wytworzenie budynków, jednostek lub odkrycie technologii, które znajdują się głębiej w drzewku technologicznym. Powinien zatem istnieć jakiś mechanizm informujący o tym, do czego służy dany typ jednostki, lub budynku oraz jakie uprzednie czynności są niezbędne, by móc stworzyć jego reprezentanta.

Ponieważ informacja o planszy jest ograniczona przez mgłę wojny, sztuczna inteligencja musi w czasie rzeczywistym gromadzić wiedzę i dane statystyczne, aby mogła podejmować rozsądne decyzje. Należy więc stworzyć jakieś repozytorium, w którym wiedza ta będzie składowana.

Wiele czynności wykonywanych jest przez jakiś dłuższy, często nieokreślony czas. Przykładowym zadaniem jest odnalezienie skupiska zasobów przez zwiadowcę. Skaut ma w tym wypadku zwiedzać nowy teren, dopóki ich nie znajdzie – może to trwać dłużej bądź krócej. Wymagane jest istnienie mechanizmu przedłużania wykonania takich zadań w czasie.

Z powodu ograniczeń czasowych zaimplementowano niewiele wspomnianych wcześniej zachowań. Głównie skupiono się na zbieraniu zasobów oraz zwiadach w ich poszukiwaniu. Jednak zaprezentowane w następnych rozdziałach rozwiązania omówionych powyżej wyzwań stanowią solidny fundament do zaprogramowania algorytmów radzących sobie z bardziej złożonymi problemami.



## Mechanizmy pomocnicze

Wiele z klas rozwiązujących problem wieloaspektowości i wielozadaniowości sterowania armią korzysta z obiektów związanych z organizacją wiedzy na temat świata i zasad gry. Rozdział ten opowiada właśnie o nich.

### Regiony

Pomysł na grupowanie pól planszy w większe obszary mające pewne znaczenie został zaczerpnięty z książki *Perełki programowania gier*.[[25]](#footnote-25) Opisuje ona konieczność stworzenia mechanizmu, dzięki któremu AI będzie w stanie rozumieć, że zbiór blisko położonych budynków jednej armii stanowi jej bazę, duże skupiska pól oddzielone od siebie wodą to kontynenty, a kratki położone wysoko są wzniesieniami. Dzięki temu sztuczna inteligencja jest w stanie np. lepiej oceniać taktyczne znaczenie odpowiednich rejonów planszy. Oczywiście w prototypie **MechWars** nie ma wody ani wysokości terenu, jednak wciąż istnieje przypadek z bazą oraz, co ważniejsze, z zasobami. Łatwiej *AI* jest zarządzać regionami o wysokiej koncentracji surowca, niż pojedynczymi polami z zasobem.

Klasa Region, realizująca funkcjonalność obszarów, została zaimplementowana na zasadzie czegoś, co książka określa jako „pojemnik pudełka od zapałek”. Mówiąc zwięźle, jest to dwustronna lista list przedziałów. Szkic poglądowy (**Rysunek 3**) wyjaśnia to zawiłe pojęcie.

Każdy przedział stanowi dwie liczby całkowite: współrzędne Y początku i końca przedziału. Jest on reprezentowany w projekcie przez klasę RegionStripPart.

Klasa RegionStrip — pasek regionu — opakowuje wewnętrzną listę przedziałów. Jest to sposób na przechowywanie całego pionowego wycinka regionu. Przedziały na tej liście są rozłączne i posortowane w kolejności rosnącej. Jeśli do obiektu RegionStrip dodane zostanie pole nie sąsiadujące z żadnym przedziałem, tworzony jest dla niego nowy przedział. Jeśli jednak współrzędna Y pola różni się o 1 od początku lub końca któregoś przedziału, ów przedział jest rozszerzany. Gdy w wyniku tego rozszerzenia dwa przedziały zaczynają sąsiadować, są one łączone w jeden większy. Usuwanie pól działa analogicznie — rozdziela przedziały na mniejsze, zawęża je i kasuje, jeśli zawierają tylko jedno pole.

Wreszcie klasa Region posiada dwie listy list przedziałów — czyli inaczej: dwie listy obiektów RegionStrip. Gdyż obszar, mający określony środek, powienien móc być rozszerzany w obie strony na osi X. Dlatego właśnie listy występują w parze. Wspomniany środek ten oznaczony jest liczbą całkowitą offset. Paski regionu są poukładane w kolejności współrzędnych osi X. Jeśli dla którejś współrzędnej poziomej brakuje w regionie pól, w odpowiednim miejscu listy znajduje się pusty RegionStrip. Kolekcja pionowych pasków jest automatycznie dostosowywana tak, by prawa lista miała przynamniej jeden element, oraz by na obydwu listach ostatni element nie był pustym paskiem. Wyjątkami są dwie sytuacje: lewa lista może być pusta (a więc nie mieć ostatniego elementu), oraz obie listy mogą być puste.



**Rysunek 3.** Szkic poglądowy struktury danych Region

Organizacja powyższej struktury danych w taki sposób służy przede wszystkim oszczędności pamięciowej. Tablica dwuwymiarowa zmiennych bool dla każdego regionu zajmuje znacznie więcej miejsca, niż tablica początków i końców przedziałów.

Klasa Region udostępnia funkcje do jej obsługi operujące na współrzędnych mapy: AddTile() (dodająca pole do regionu), RemoveTile() (usuwająca pole z regionu) oraz IsInside() (sprawdzająca czy pole należy do regionu). Ukrywane są w ten sposób zawiłe przeliczenia związane z liczbą offset określeniem który przedział w którym pasku należy poszerzyć, zawęzić bądź sprawdzić.

Region udostępnia też właściwości Width, Left i Right do określenia poziomych wymiarów i granic obszaru, jak również metody CalculateVerticalStart() i CalculateVerticalEnd() wyliczające granice pionowe. Natomiast właściwość AllTiles zwraca obiekt typu IEnumerable<IVector2> pozwalający przeiterować po wszystkich polach regionu.

Klasa Region jednak nie stanowi całości mechanizmu regionów. Obszary zaprogramowane w powyższy sposób wciąż są rozumiane z zewnątrz jako zbiór pól. Znacznie łatwiej by było natomiast operować na nich jak na figurach geometrycznych, dlatego dążono do stworzenia tzw. otoczki wypukłej — minimalnego wielokąta wypukłego opisanego na regionie.

Etapem pośrednim do stworzenia wielokąta wypukłego jest RegionHull. Obiekt tej klasy stanowi najmniejszy wielokąt niewypukły opisujący region, na bazie którego jest tworzony. W swoim konstruktorze iteruje on po wszystkich polach regionu i te, które leżą na brzegu obszaru (brakuje im przynajmniej jednego z sąsiadów w poziomie lub w pionie) dodaje ich współrzędne do listy punktów brzegowych. Po wykonaniu tego działania na liście znajdować się będą redundantne punkty: np. może wystąpić seria przynajmniej trzech sąsiednich punktów o tej samej współrzędnej Y. Nie ma to jednak znaczenia, gdyż RegionHull jest jedynie półproduktem służącym masowej redukcji liczby punktów, zanim zostanie wygenerowana otoczka wypukła.

Docelowym obiektem, jaki chcemy otrzymać jest RegionConvexHull. Stosuje on algorytm Grahama, który odrzuca zbędne punkty i ustawia pozostałe w kolejności, w jakiej są połączone ścianami wielokąta. Pierwszym krokiem algorytmu jest wybranie wierzchołka o minimalnej współrzędnej Y, a jeśli takich jest kilka, to z nich: o minimalnej współrzędnej X. Następnie punkty sortowane są rosnąco według kąta, jaki tworzą z wyróżnionym wierzchołkiem — w efekcie ułożone są w kierunku przeciwnym do ruchu wskazówek zegara. Później algorytm wrzuca kolejne punkty na stos, sprawdzając trzy z jego szczytu. Jeśli tworzą one skręt w prawo (liczony jest znak iloczynu wektorowego), lub nie tworzą skrętu w ogóle, środkowy punkt jest odrzucany i test jest powtarzany. Algorytm wrzuca następny punkt dopiero, kiedy trójka wierzchołków zakręca w lewo. Gdy skończą się punkty w liście RegionHull, algorytm dobiega końca: wszystkie wierzchołki otoczki wypukłej znajdują się na stosie we właściwej kolejności. **[źródło]**

Posiadanie regionu zapisanego w postaci wielokąta wypukłego pozwala na wprowadzenie kilku użytecznych funkcji. Właściwość Center posiada obliczony środek masy wielokąta. Metoda Contains() sprawdza, czy zadane współrzędne zawiera się w wielokącie. Funkcja GetDistanceTo() pozwala poznać odległość od określonego punktu do granicy wielokąta. Wreszcie GetPointClosestTo() zwraca współrzędne na brzegu wielokąta będące najbliżej zadanego punktu.

Obiekty Region, RegionHull oraz RegionConvexHull są zagregowane w jednym obiekcie typu RegionBatch. Klasa ta automatycznie aktualizuje obiekty RegionHull i RegionConvexHull, kiedy struktura regionu ulegnie zmianie. Ponadto kilka obiektów RegionBatch można połączyć w jeden funkcją ConcatBatches(). Wykorzystywana jest ona, np. gdy teren z jednym skupiskiem zasobów został odkryty z paru różnych stron, więc powstało parę różnych regionów. Niestety nie ma prostej metody na czynność odwrotną — podział regionu na mniejsze. Wynika to z tego, że trzebaby najpierw określić kryterium rozdzielenia pól między regiony.

Istnieją wyspecjalizowane klasy implementujące interfejs IRegionBatch. ResourceRegionBatch posiada zbiór obiektów Resource, które znajdują się w regionie i pozwala za pomocą właściwości TotalResourceValue wyliczyć całkowitą liczbę jednostek zasobów w regionie. BaseRegionBatch jest wykorzystywany do oznaczenia bazy, zawiera więc zbiór obiektów Building. Regiony użyto jeszcze w określaniu miejsc do zwiedzenia przez *Scouta*, do czego służy ReconRegionBatch. Sam interfejs wymusza na nich (i na oryginalnym RegionBatch) posiadanie m.in. właściwości Region, Hull i ConvexHull oraz bezparametrowej metody UpdateBatch().

### Rodzaje elementów mapy

Sztuczna inteligencja w prototypie **MechWars** postrzega rodzaje elementów mapy jako obiekty klasy MapElementKind. W tych obiektach trzyma informacje o nazwie elementu mapy, jego kształcie i prefabie z którego jest tworzony. W klasie MapElementKind znajdują się również słownik obiektów MapElementPurpose, które służą do określenia przeznaczenia elementu mapy, oraz lista obiektów CreationMethod mówiących, co jest potrzebne do stworzenia danego elementu mapy. Informacje te potrzebne są *AI*, aby mogła symulować zrozumienie sposobu, w jaki należy z danej jednostki lub budynku korzystać.

Klasa MapElementPurpose jest bardzo prosta — ma jedynie dwie właściwości: nazwę (Name) i zmiennoprzecinkową wartość (Value). Nazwa określa czynność i jest kluczem we wspomnianym słowniku, a wartość od 0 do 1 oznacza jak bardzo dany element mapy się nadaje do tej czynności. Zdefiniowane są dwa rodzaje czynności: „*Scouting*” oraz „*Harvesting*”. Wartości przeznaczeń elementów mapy są dostosowywane metodą MapElementKind.NormalizePurposes() tak, by ich suma dla jednego rodzaju elementu mapy zawsze wynosiła 1 (chyba że w słowniku nie ma żadnych przeznaczeń).

Klasa CreationMethod jest trochę bardziej złożona. Znajdują się w niej właściwości takie jak prefab tworzonego elementu mapy, prefab elementu mapy mającego akcję rozkazu produkcji rzeczonego elementu mapy, a także koszt w jednostkach zasobów oraz czas jaki tworzenie zajmie. Poza tym CreationMethod zawiera dwie listy określające wymagania. BuildingRequirements określa jakie budynki muszą znajdować się w bazie, a TechnologyRequirements — jakie technologie potrzeba opracować.

Dwie klasy *proxy*: CreationMethodDictionary oraz MapElementKindDictionary zawierają słowniki z informacjami na temat elementów mapy. W metodach InitializeDictionary() słowniki te na sztywno są wypełniane. Zewnętrzny mechanizm ich inicjalizacji nie został zaimplementowany z powodu niedostatecznego czasu. Jest to rozwiązanie tymczasowe, a nie docelowe.

MapElementKindDictionary definiuje rodzaje dla elementów mapy:

* *„Scout”* — przeznaczenie: „*Scouting*” = 1,
* *„Harvester”* — przeznaczenie: „*Scouting*” = 0.2, „*Harvesting*” = 0.8,
* *„ConstructionYard”*,
* *„Refinery”*,
* *„Factory”*.

Wartości przeznaczeń dla jednostek zostały dobrane eksperymentalnie. Kształty elementów mapy funkcja inicjalizująca pobiera z Globals.ShapeDatabase, więc ustawiane są automatycznie.

CreationMethodDictionary opisuje metody tworzenia dla rodzajów:

* *„Refinery”*,
* *„Factory”*,
* *„Scout”*,
* *„Harvester”*.

Wszystkie parametry, które muszą się znaleźć w obiekcie CreationMethod pobierane są z akcji rozkazu służącej konstrukcji danego budynku lub produkcji danej jednostki. Sztywne zaprogramowanie tej funkcji polega na tym, że programista wie gdzie szukać tych informacji.

## System wieloagentowy

Idea systemu wieloagentowego została zaczerpnięta z książki *Programming Game AI by Example*[[26]](#footnote-26). Stanowi ona rozwiązanie problemu, jakim jest wieloaspektowość i wielopoziomowość sterowania armią. Przedstawiony tam system został jednak mocno zmodyfikowany na potrzeby projektu.

Głównym założeniem takiego systemu jest istnienie agentów — czyli autonomicznych obiektów realizujących własne zadania i zdolnych do komunikowania się między sobą. W książce system wieloagentowy omawiany jest na najczęstszym przykładzie jego użycia — gdy każdy agent zarządza pojedynczym obiektem gry. W przypadku prototypu **MechWars** również istnieją takie agenty (UnitAgent zarządzający jednostką), jednak główne zastosowanie mają agenty stanowiące pewną abstrakcję i zarządzające osobnymi aspektami sterowania armią. Istnieją zatem:

* KnowledgeAgent, który gromadzi wiedzę (i rozwiązuje w ten sposób drugi z opisanych problemów),
* ReconAgent odpowiedzialny za zwiady,
* ConstructionAgent rozbudowujący bazę,
* ProductionAgent tworzący nowe jednostki,
* ResourceCollectorAgent, którego zadaniem jest zbieranie zasobów,
* MainAgent, który tworzy wszystkie powyższe agenty i trzyma ich referencje.

Poza tym książka opisuje dla tych agentów implementację automatu skończonego (*FSM*), której tutaj nie zrealizowano. Zamiast tego zaczerpnięto z niej prosty system wiadomości służący do komunikacji między agentami. Dodano też mechanizm zadań, opisany w innym rozdziale książki.[[27]](#footnote-27)

### Gracz AI

Klasą zarządzającą wszystkimi agentami jest AIBrain. Ideowo stanowi on „mózg” sztucznej inteligencji, gdyż agreguje wszystkie jej aspekty i umożliwia dowolnemu agentowi publiczny dostęp do innych agentów. Jednocześnie z zewnątrz można patrzeć na obiekt tej klasy, jak na sterowanie gracza — czyli odpowiednika klasy Spectator po stronie AI.

AIBrain jest skryptem MonoBehaviour, którego publiczne pola to Player — gracz, którym AI steruje — oraz kilka parametrów określających zachowanie pewnych agentów w określonych sytuacjach. Parametry te opisane zostały w rozdziałach o agentach, które z nich korzystają.

Prywatne pola: zbiory agentów agents oraz agentsToAdd trzymają wszystkie obecnie istniejące agenty. Publiczna funkcja AddAgent() umożliwia dodawanie nowych agentów do zbioru tymczasowego, z którego zostają przeniesione do głównego w funkcji Update(). Agenty usuwane są również w tej funkcji, gdy tylko skończyły swoje zadanie, czyli ich właściwość Finished zwraca **true**. Natomiast dla agentów pozostających w działaniu wołana jest ich funkcja Start() (jeśli są nowo dodane) oraz Update() (za każdym razem).

public Player player;

public float resourceRegionDistance = 2;

public TextAsset harvestingImportanceFunction;

public int reconRegionSize = 8;

public int coarseReconRegionPercentage = 90;

public ReconRegionOrderCalculator reconRegionOrderCalculator;

HashSet<Agent> agentsToAdd;

HashSet<Agent> agents;

public MainAgent MainAgent { get; private set; }

public FilteringMapProxy MapProxy { get; private set; }

void Start()

{

agentsToAdd = new HashSet<Agent>();

agents = new HashSet<Agent>();

MainAgent = new MainAgent(this);

MapProxy = new FilteringMapProxy(player.army);

InitializeResourceRegionDetectionShape();

}

void Update()

{

agents.UnionWith(agentsToAdd);

agentsToAdd.Clear();

foreach (var a in agents)

{

if (!a.Started) a.Start();

a.Update();

}

agents.RemoveWhere(a => a.Finished);

}

public void AddAgent(Agent agent)

{

agentsToAdd.Add(agent);

}

**Fragment kodu 25.** Wybrane elementy klasy AIBrain

Metoda AIBrain.Start() inicjalizuje właściwości i tworzy pierwszego agenta: MainAgent, który startuje wszystkie następne. Nie używa funkcji AddAgent() by dodać go do listy — każdy agent w swoim konstruktorze robi to sam.

Spośród właściwości na uwagę zasługuje MapProxy. Zawiera ona obiekt klasy FilteringMapProxy, który pośredniczy w pozyskiwaniu informacji z klasy Map przez agentów. Uwzględnia on VisibilityTable armii sterowanej przez AI maskując działania jednostek i zmiany w strukturach znajdujących się poza polem widzenia. Można zatem powiedzieć, że symuluje dla sztucznej inteligencji mechanizm, który dla gracza ludzkiego istnieje w postaci wizualizacji mgły wojny.

### Klasa agenta

Klasa abstrakcyjna Agent jest dosyć złożona. Stanowi ona bazową klasę dla wszystkich agentów. Zajmuje się wykonywaniem zadań, przesyłaniem, logowaniem i odbieraniem wiadomości a także uruchamianiem akcji czasowych. Posiada więc prywatną kolejkę wiadomości[[28]](#footnote-28), chronioną listę zadań (Goal) oraz słowniki z akcjami (ActionToPerform, ArgActionToPerform). Każdy agent ma szereg właściwości skracających dostęp do innych agentów (przechowywanych w obiekcie MainAgent), jak również gracza i MapProxy (trzymanych przez AIBrain). Ponadto istnieją tu właściwości mówiące czy agent już rozpoczął swoje działanie oraz czy je zakończył, jak również takie, które pozwalające pobrać informacje o obecnym zadaniu.

Konstruktor każdego agenta automatycznie dodaje go do zbioru agentów w AIBrain. Publiczna metoda GiveGoal() pozwala dać agentowi zadanie i określić jego ważność. Chronione funkcje SendMessage() (różniące się parametrami) tworzą nowy obiekt Message, kolejkują go i zapisują do logu. Wszystkie wysyłane wiadomości są zawsze logowane, dzięki czemu po wykonaniu przebiegu gry można obejrzeć plik tekstowy z informacjami na temat komunikacji między agentami. Nazwa tego pliku jest określona publicznym polem obiektu Globals: string aiMessageLogFileName. Metodą ReceiveMessage() klasy agentów potomnych mogą wyciągnąć wiadomość z kolejki, jeśli jakaś tam jest, by móc na nią zareagować.

Dwie metody o nazwie PerformEvery() stanowią mechanizm do wykonywania powtarzalnych czynności, które nie muszą wykonywać się co każdy cykl aktualizacji gry. Służą więc do wywoływania określonych funkcji co zadany w sekundach interwał.[[29]](#footnote-29) Dla przekazanej w parametrze akcji (System.Action, lub System.Action<object>) tworzony jest obiekt z tą akcją i informacjami o czasie jej wykonania (odpowiednio: ActionToPerform, lub ArgActionToPerform) i dodawany do odpowiedniego słownika. Jednakże gdy taka akcja znajduje się już w słowniku, metoda PerformEvery() jedynie aktualizuje jej czas wykonania i, jeśli interwał został przekroczony, zeruje czas i uruchamia ją. Informacja o zwiększeniu czasu jest zapisywana i resetowana dopiero podczas aktualizacji pętli gry, dzięki czemu nie da się jej wykonać dwukrotnie w czasie jednej aktualizacji (co byłoby błędem). Owo okresowe wywoływanie czynności można zatrzymać korzystając z odpowiedniej z dwóch bliźniaczych funkcji StopPerform().

Queue<Message> messages;

protected List<Goal> Goals { get; private set; }

Dictionary<System.Action, ActionToPerform> actionsToPerform;

Dictionary<System.Action<object>, ArgActionToPerform> argActionsToPerform;

public bool Started { get; private set; }

public bool Finished { get; private set; }

public void Start()

{

CheckForArmy();

OnStart();

Started = true;

}

public void Update()

{

CheckForArmy();

ResetActionsIncrements();

OnUpdate();

if (!Goals.Empty())

{

var goal = Goals.First();

if (goal.State == GoalState.BrandNew)

goal.Start();

if (goal.State == GoalState.Started)

goal.Update();

if (goal.InFinalState)

Goals.RemoveFirst();

}

}

**Fragment kodu 26.** Wybrane fragmenty klasy Agent

Agenty posiadają publiczne funkcje Start() i Update() wywoływane przez AIBrain w celu ich inicjalizacji i aktualizacji. Równolegle istnieją uruchamiane przez nie odpowiednio wirtualne, chronione metody OnStart() i OnUpdate(). Dzięki nim możliwa jest personalizacja agentów pod ich cele. Dodatkowo Start() sprawdza, czy AIBrain ma przypisanego gracza z armią i ustawia flagę Started, a Update() zajmuje się resetowaniem informacji o aktualizacji czasu akcji czasowych oraz obsługą zadań agenta. Metoda Finish() ustawia flagę Finished. Powoduje to usunięcie agenta ze zbioru w AIBrain po zakończeniu cyklu aktualizacji agentów.

### Wiadomości i żądania

Komunikacja pomiędzy agentami została zaimplementowana w postaci wiadomości. Klasa Message nie ma żadnej własnej funkcjonalności, ale posiada szereg właściwości. Są to: agent nadawca (Sender), agent odbiorca (Receiver), nazwa wiadomości (Name) oraz tablica argumentów (Arguments). Nazwa wiadomości jest ważna gdyż określa jej rodzaj, a zatem sposób postępowania agenta po jej otrzymaniu. Argumenty są dodatkowymi informacjami typu priorytet prośby albo typ i liczba jednostek do wyprodukowania. Dodatkowo obiekt Message posiada właściwość InnerMessege, w której może się znaleźć się inna wiadomość (np. taka, na którą obecna jest odpowiedzią).

Większość agentów po otrzymaniu wiadomości zapisuje sobie wiadomość jako tzw. żądanie. Realizująca je klasa Request również jest bez funkcjonalności. Jej właściwości to: agent żądający (RequestingAgent), gdzie zazwyczaj przypisywany jest Message.Sender, nazwa (Name) najczęściej kopiowana z wiadomości, priorytet (Priority) — liczbowy argument o minimalnej wartości 0, im mniejszy, tym żądanie ważniejsze. Oprócz tego Request może przechowywać InnerMessage — wiadomość na bazie której powstał — oraz Position — dwuwymiarowy wektor intów, tylko wtedy gdy ma to sens w kontekście wiadomości (np. „wybuduj mi tutaj budynek”).

### Klasa zadania

Zadania, jakie mogą wykonywać agenty zostały zrealizowane w postaci klasy abstrakcyjnej Goal. Podobnie jak klasa Order, jest ona zaprogramowana zgodnie ze wzorcem projektowym *template method* **[źródło]**, a jej obiekty posiadają zmienny stan określający przebieg ich życia. W ogólności zadania działają bardzo podobnie do rozkazów, różni je jednak od nich to, że funkcjonują na znacznie ogólniejszym poziomie. Podczas gdy rozkaz stanowi pewną powtarzalną, pospolitą czynność bez konkretnego znaczenia, poziom abstrakcji zadań pozwala zobaczyć cel realizowany przez sztuczną inteligencję.

Klasa Goal posiada cztery główne właściwości opisujące jej obiekty. Name jest nazwą zadania. Agent, to agent je wykonujący. State trzyma stan zadania, a Importance jest liczbą float, konwencjonalnie z przedziału zamkniętego (0, 1), określającą jego ważność.

State to właściwość typu enuma GoalState. Wartości, jakie może przyjmować, to:

* BrandNew — po użyciu konstruktora, przez wywołaniem funkcji Start(),
* Started — po wywołaniu funkcji Start(); obiekt Agent dla zadania w tym stanie woła cyklicznie jego metodę Update(),
* Finished — stan końcowy uzyskiwany przez wywołanie funkcji Finish() i oznaczający, że zadanie wykonane zostało pomyślnie,
* Canceled — stan końcowy otrzymywany przez użycie funkcji Cancel() i oznaczający, że zadanie anulowano.

Metody publiczne Start(), Update(), Finish() i Cancel() stanowią szablon zachowania obiektu Goal. Oprócz sterowania stanem (a więc czasem życia) zadania, wołają one puste, wirtualne funkcje: OnStart(), OnStarted(), OnUpdate(), OnFinishing(), OnFinished(), OnCanceling() oraz OnCanceled(). Każda z nich może zostać nadpisana w klasie potomnej, by zaimplementować odpowiednie zachowanie agenta wykonującego zadanie. Dodatkowa właściwość InFinalState ułatwia sprawdzenie, czy Goal jest w którymś z końcowych stanów.

Mimo zauważalnych podobieństw rozkazów do zadań, te drugie są znacznie prostszym mechanizmem. Operując na wyższym poziomie abstrakcji nie wymagają takiej sztywnej konstrukcji, co sprawia, że łatwiej implementować poszczególne przypadki zadań. Mimo, że klasę Goal zaprogramowano z myślą o wykorzystaniu jej przez dowolnego agenta, w praktyce okazało się, że ma sens tylko dla agentów jednostek. Z drugiej strony możliwe, że gdyby kontynuować rozbudowywanie *AI*, znalazłyby się przypadki użycia zadań przez agentów ogólniejszych. W tym momencie istnieją tylko dwie klasy dziedziczące po Goal. Zadanie CoarseRegionGoal[[30]](#footnote-30) wykonywane przez jednostkę sprawia, że odkrywa ona niezbadany teren i zwiedza planszę, natomiast zadanie HarvestGoal[[31]](#footnote-31) może realizować *Harvester*, by zbierać zasoby z planszy.

### Agent jednostki

Obiekt klasy UnitAgent jest odpowiedzialny za zachowanie pojedynczej jednostki. Cechują go takie publiczne właściwości jak generowane automatycznie Id, jednostka, której zachowaniem steruje (Unit), rodzaj elementu mapy, jakim ona jest (Kind) oraz inny agent, który obecnie jest właścicielem agenta jednostki (Owner). Dodatkowo właściwość Busy służy do szybkiego sprawdzenia, czy UnitAgent ma właściciela.

Istnienie właściwości Owner wynika z potrzeby zaprogramowania mechanizmu podobnego, do współdzielenia zasobów w systemach wielowątkowych. Każdy obiekt UnitAgent może być posiadany przez innego agenta. Udostępnia publiczne metody Take(), Release() oraz HandOn(), żeby można go było odpowiednio wziąć, wypuścić i przekazać innemu agentowi. Właściciel obiektu UnitAgent jest tym agentem, którego cele realizuje jednostka. To on daje agentowi jednostki zadania i ogólnie — zarządza nim. Inny agent może za pomocą wiadomości „*Hand me on Units*” poprosić o jego przekazanie, ale to, czy UnitAgent zostanie oddany zależy tylko i wyłącznie od właściciela. W następnych rozdziałach pracy pojawiają się w kontekście posiadania agentów jednostek sformułowania: „ma”, „trzyma”, „wziął”, „bierze”, „zwalnia”, „wypuszcza”, „oddaje”, „przekazuje”. Oznaczają one użycie jednej z trzech metod służących do zmiany właściciela.

public bool Take(Agent takingAgent)

{

if (Busy) return false;

Owner = takingAgent;

return true;

}

public void Release(Agent releasingAgent)

{

if (releasingAgent != Owner)

throw new System.Exception(string.Format(

"Agent {0} cannot release UnitAgent - it is not the Owner.",

releasingAgent));

if (!Busy)

throw new System.Exception(

"Cannot release UnitAgent - it's not Busy.");

Owner = null;

}

public void HandOn(Agent releasingAgent, Agent takingAgent)

{

Release(releasingAgent);

Take(takingAgent);

}

**Fragment kodu 27.** Metody służące do zmiany właściciela obiektu UnitAgent.

Nadpisana metoda wirtualna OnUpdate() sprawdza jedynie, czy jednostka została zniszczona. Jeśli tak, to usuwa jej agenta ze słownika KnowledgeAgent.UnitAgents[[32]](#footnote-32) i woła funkcję Finish(), by usunięto go również ze zbioru w AIBrain.

### Agent gromadzący wiedzę

Problem repozytorium wiedzy został rozwiązany poprzez stworzenia agenta, którego celem jest zbieranie i udostępnianie informacji na temat rozgrywki. Klasa KnowledgeAgent (dziedzicząca po Agent) zawiera stałą wiedzę — omówione już słowniki MapElementKindDictionary i CreationMethodDictionary. Poza tym KnowledgeAgent gromadzi też zmienne dane: agenty jednostek w słowniku (klasa UnitAgentDictionary), a także wiedzę na temat rozlokowania na planszy zasobów (ResourcesKnowledge) i własnej bazy (AllyBaseKnowledge). Dzięki składowaniu wszystkich informacji w jednym miejscu dowolny agent ma do nich łatwy dostęp — zarówno w celu ekstrakcji i analizy danych obecnych, jak i poszerzeniu ich o nowo odkryte.

public MapElementKindDictionary MapElementKinds { get; private set; }

public TechnologyKindDictionary TechnologyKinds { get; private set; }

public CreationMethodDictionary CreationMethods { get; private set; }

public UnitAgentDictionary UnitAgents { get; private set; }

public ResourcesKnowledge Resources { get; private set; }

public AllyBaseKnowledge AllyBase { get; private set; }

**Fragment kodu 28.** właściwości agenta KnowledgeAgent

Obiekt UnitAgentDictionary tak naprawdę w środku przechowuje dwa słowniki: kindDict dla klucza MapElementKind zwraca wartość — zbiór obiektów UnitAgent, natomiast unitDict jako klucz przyjmuje Unit, a wartością jest UnitAgent. W ten sposób możliwe jest pobranie wszystkich agentów jednostek danego rodzaju, jak również otrzymanie agenta sterującego wybraną jednostką. UnitAgentDictionary słowniki przechowuje w polach prywatnych — udostępnia za to metody Add(), Remove() do ich modyfikacji oraz dwie właściwości-indeksatory by móc pobrać wartość z każdego z tych słowników.

Klasa ResourcesKnowledge zawiera dwuwymiarową tablicę obiektów ResourceInfo o wymiarach mapy oraz zbiór pakietów regionów (RegionBatch). Obiekt ResourceInfo jest niedużą paczką informacji o zasobie na mapie. Jego właściwości to Location — położenie zasobu, RegionBatch — region, który zawiera ten zasób oraz Resource — zwracająca sam zasób. Początkowa wartość RegionBatch to **null**, ale właściwość ta ma publiczny *setter*. Jest ustawiana dopiero w momencie wstawienia nowego obiektu ResourceInfo lub wartości **null** do tablicy ResourcesKnowledge.resourceInfos. We właściwości-indeksatorze, która się tym zajmuje, wołana jest metoda UpdateResourceRegions(). Korzysta ona z funkcji RemoveFromRegion() lub AddToRegion(), zależnie od tego, czy obiekt został właśnie do tablicy dodany, czy też z niej usunięty. To właśnie te dwie metody zajmują się zarządzeniem zbiorem pakietów regionów.

Metoda AddToRegion() używa funkcji FindSurroundingResourceInfos(), by wyszukać w tablicy resourceInfos wszystkie zasoby w pewnym obszarze dookoła zadanej pozycji. Obszar ten określony jest przez MapElementSurroundingShape, obiekt będący okrągłym kształtem dookoła elementu mapy. Generowany jest on w funkcji AIBrain.Start(), jego promień określa jeden z parametrów liczbowych: publiczne pole AIBrain.resourceRegionDistance. Gdy sąsiedzi nowego zasobu zostaną określeni, determinowany jest zbiór obiektów RegionBatch, na których się znajdują. Jeśli zbiór jest pusty, nowy zasób nie ma żadnych innych w okolicy, więc dla niego tworzony jest nowy region. W przeciwnym wypadku wybierany jest dowolny z regionów. Zasób i jego pole jest dodawane do skonstruowanego lub znalezionego regionu, który z kolei jest ustawiany jako RegionBatch w obiekcie ResourceInfo. Na koniec, jeśli pobliskich regionów jest kilka, są one łączone w jeden funkcją ConcatBatches().

ResourceInfo[,] resourceInfos;

HashSet<ResourceRegionBatch> resourceRegions;

public ResourceInfo this[int x, int y]

{

get { return resourceInfos[x, y]; }

set

{

if (resourceInfos[x, y] == value) return;

var oldValue = resourceInfos[x, y];

resourceInfos[x, y] = value;

UpdateResourceRegions(x, y, oldValue, value);

}

}

void UpdateResourceRegions(int x, int y,

ResourceInfo oldValue, ResourceInfo newValue)

{

if (oldValue != null) RemoveFromRegion(x, y, oldValue);

if (newValue != null) AddToRegion(x, y, newValue);

}

void AddToRegion(int x, int y, ResourceInfo resource)

{

var others = FindSurroundingResourceInfos(x, y, resource);

var regionBatches = others.SelectDistinct(ri => ri.RegionBatch);

if (resource.RegionBatch != null) throw new System.Exception(

"newValue is already in some Region.");

ResourceRegionBatch regionBatch;

if (regionBatches.Empty())

{

regionBatch = new ResourceRegionBatch(knowledge.Brain);

resourceRegions.Add(regionBatch);

}

else regionBatch = regionBatches.First();

resource.RegionBatch = regionBatch;

regionBatch.Resources.Add(resource);

regionBatch.Region.AddTile(x, y);

if (regionBatches.HasAtLeast(2))

{

var newRegionBatch = regionBatches.ConcatBatches(knowledge.Brain);

resourceRegions.ExceptWith(regionBatches);

resourceRegions.Add(newRegionBatch);

}

}

void RemoveFromRegion(int x, int y, ResourceInfo resource)

{

var rb = resource.RegionBatch;

resource.RegionBatch = null;

rb.Resources.Remove(resource);

rb.Region.RemoveTile(resource.Location.X, resource.Location.Y);

if (rb.RegionEmpty)

resourceRegions.Remove(rb);

}

**Fragment kodu 29.** Wybrane składowe klasy ResourceKnowledge

Metoda RemoveFromRegion() jest prostsza — jedynie usuwa ResourceInfo i jego wspóółrzędne z regionu, a pod właściwość RegionBatch wstawia **null**. Jeśli w wyniku tego region stał się pusty, jest on wyrzucany ze zbioru. Nie został stworzony mechanizm dzielenia regionów, gdyż problem sprawdzenia w jaki sposób należy je podzielić był zbyt złożony.

Klasa AllyBaseKnowledge działa w bardzo podobny sposób do ResourceKnowledge. Również wykorzystuje pomocniczy obiekt — BuildingInfo. Obiekty tej klasy są analogiczne do ResourceInfo: też posiadają element mapy, którego dotyczą — tym razem budynek, jego współrzędne i pakiet regionów, do którego należy. Jednak ponieważ budynki mogą znajdować się na kilku polach, BuildingInfo zawiera dodatkową właściwość AllCoords zwracającą listę wszystkich zajmowanych współrzędnych.

BuildingInfo[,] buildingInfos;

public BaseRegionBatch BaseRegion { get; private set; }

public BuildingInfo this[int x, int y]

{

get { return buildingInfos[x, y]; }

}

public void AddBuilding(BuildingInfo building)

{

if (building.AllCoords.Any(c => this[c] != null))

throw new System.Exception(

"Cannot AddBuliding - at least one coord is not empty.");

foreach (var c in building.AllCoords)

buildingInfos[c.X, c.Y] = building;

building.RegionBatch = BaseRegion;

BaseRegion.Buildings.Add(building);

foreach (var c in building.AllCoords)

BaseRegion.Region.AddTile(c);

}

public void RemoveBuilding(BuildingInfo building)

{

if (building.AllCoords.All(c => this[c] == null))

throw new System.Exception(

"Cannot RemoveBuliding - none of its coords contain it.");

foreach (var c in building.AllCoords)

buildingInfos[c.X, c.Y] = null;

building.RegionBatch = null;

BaseRegion.Buildings.Remove(building);

foreach (var c in building.AllCoords)

BaseRegion.Region.RemoveTile(c);

}

**Fragment kodu 30.** Wybrane składowe klasy AllyBaseKnowledge

Zarządzanie wiedzą o bazie jest znacznie prostsze od zarządzania wiedzą o zasobach, gdyż znajduje się tu tylko jeden obiekt RegionBatch. Klasa AllyBaseKnowledge wciąż jednak posiada dwuwymiarową tablicę obiektów BuildingInfo, analogiczną do tablicy ResourceKnowledge.resourceInfos. Istnieje tu też właściwość‑indeksator, lecz nie posiada ona *settera*. Do modyfikowania tablicy i regionu służą metody AddBuilding() i RemoveBuilding().

Sama klasa KnowledgeAgent jest pasywna. Nie implementuje metody OnUpdate(), więc nie aktualizuje się co cykl. Zamiast tego agent wiedzy nasłuchuje na zdarzeniach obiektu Army gracza *AI*, które powiadamiają go o zmianach w widoczności pól mapy, oraz o tym, czy powstaje lub jest niszczony element mapy w zasięgu widzenia. W metodzie OnStart() pod te zdarzenia podpinane są metody, aktualizują obiekty ResourceKnowledge i AllyBaseKnowledge. Natomiast dla każdej z już istniejących w czasie inicjalizacji jednostek tworzony jest agent i dodawany do słownika UnitAgentDictionary.

void VisibilityTable\_VisibilityChanged(IVector2 tile,

Visibility from, Visibility to)

{

if (to == Visibility.Visible)

{

var mapElement = MapProxy[tile];

if (from == Visibility.Unknown)

{

if (mapElement is Resource)

Resources[tile] = new ResourceInfo(MapProxy, tile);

}

else if (from == Visibility.Fogged)

{

var resInfo = Resources[tile];

if (mapElement == null)

{

if (resInfo != null)

Resources[tile] = null;

}

else if (mapElement is Resource)

{

if (resInfo == null)

Resources[tile] = new ResourceInfo(MapProxy, tile);

}

}

}

}

void Army\_OnVisibleMapElementCreated(MapElement mapElement)

{

var tile = mapElement.Coords.Round();

if (mapElement is Resource)

Resources[tile] = new ResourceInfo(MapProxy, tile);

else if (mapElement is Building)

{

var b = (Building)mapElement;

if (b.Army == Army)

MyBase.AddBuilding(new BuildingInfo(MapProxy, b));

}

}

void Army\_OnVisibleMapElementDied(MapElement mapElement)

{

var tile = mapElement.Coords.Round();

if (mapElement is Resource)

Resources[tile] = null;

else if (mapElement is Building)

{

var b = (Building)mapElement;

if (b.Army == Army)

MyBase.RemoveBuilding(MyBase[tile]);

}

}

**Fragment kodu 31.** Funkcje obsługi zdarzeń dotyczących zmiany widzalności  
oraz tworzenia i niszczenia elementu mapy w obrębie pola widzenia armii

### Agent odpowiedzialny za zwiady

Klasą agenta wywiadowczego jest ReconAgent. Zajmuje się ona wysyłaniem jednostek poza bazę, by odkryć nieznany teren. W teorii agent ten powinien też odpowiadać za stałe patrolowanie często uczęszczanych szlaków i szpiegowanie na przeciwniku. Ponieważ jednak aspekt gry *AI* przeciwko wrogowi nie został zrealizowany, skupiono się na przeczesywaniu niezbadanych obszarów w poszukiwaniu nowych miejsc z zasobami.

ReconAgent korzysta z regionów by określić niepoznane obszary mapy. Dzieli całą planszę na równe, kwadratowe wycinki. Długość boku kwadratu określa parametr AIBrain.reconRegionSize. Dla każdego z tych wycinków tworzony jest obiekt ReconRegionBatch, który następnie agent umieszcza w dwuwymiarowej tablicy ReconRegions. Procedura ta przeprowadzana jest przez GenerateReconRegions(), funkcję wołaną w metodzie OnStart().

Odmiana pakietu regionów dla zwiadu (poza standardowymi składowymi wymuszanymi przez interfejs IRegionBatch) udostępnia kilka właściwości n.t. stopnia zwiedzenia regionu. A zatem: UnknownTilesCount kalkuluje i zwraca liczbę niezbadanych pól, KnownTilesCount z kolei podaje liczbę pól poznanych. ExplorationPercentage pokazuje względny stopień zwiedzenia obszaru w procentach, a EntirelyExplored jest flagą ustawianą w momencie, gdy żadna z kratek nie jest już ukryta.

Obiekt ReconAgent reaguje na wysłane do niego wiadomości. W funkcji OnUpdate() woła metodę ProcessMessages(), w której z kolei w pętli odbiera wszystkie wiadomości o nazwie „*Find me Resources*” (prośby o odnalezienie zasobów, wysyłane przez agenta ResourceCollectorAgent). Na ich bazie tworzone są obiekty Request, które trafiają na listę żądań. Oprócz tego, dla każdego z tych żądań do słownika ReconUnits dodawany jest nowy obiekt RequestUnitAgentSet — zbiór agentów UnitAgent oddelegowanych do wykonywania żądania.

Następnie w OnUpdate() uruchamiana jest metoda ProcessRequest(), która iteruje po wszystkich żądaniach. Dla każdego mającego nazwę „*Find me Resources*” uruchamia za pomocą funkcji PerformEvery() metodę ProcessFindMeResources() z 1 sekundą interwału. Żądanie z listy usunąć może tylko owa metoda — kiedy zakończy zadanie. Sprawia to, że funkcja PerformEvery() będzie wołana co cykl aktualizacji, a w jej efekcie też i metoda ProcessFindMeResources() — co 1 sekundę. Jest to konieczne, gdyż przetwarzanie żądania poszukiwania zasobów trwa dłuższy czas. W ten sposób zrealizowany został na tym agencie mechanizm przedłużania zadań w czasie.

List<Request> requests;

public Dictionary<Request, RequestUnitAgentSet>

ReconUnits { get; private set; }

protected override void OnUpdate()

{

ProcessMessages();

ProcessRequests();

}

void ProcessMessages()

{

Message message;

while ((message = ReceiveMessage()) != null)

{

if (message.Name == AIName.FindMeResources)

{

var req = requests.FirstOrDefault(r =>

r.Name == AIName.FindMeResources);

if (req == null)

{

req = new Request(message.Sender, message.Name,

int.Parse(message.Arguments[0]), message);

requests.Add(req);

ReconUnits.Add(req, new RequestUnitAgentSet(this, req));

}

else req.Priority = int.Parse(message.Arguments[0]);

}

}

requests.Sort((r1, r2) => r1.Priority.CompareTo(r2.Priority));

}

void ProcessRequests()

{

var processed = new List<Request>();

foreach (var r in requests)

{

if (r.Name == AIName.FindMeResources)

PerformEvery(1, ProcessFindMeResources,

new ProcessFindMeResourcesArgs(r, processed));

}

foreach (var r in processed)

requests.Remove(r);

}

**Fragment kodu 32.** Realizacja mechanizmu przedłużania zadań w czasie wewnątrz agenta ReconAgent

Ponieważ na podobnej zasadzie zaprogramowana jest obsługa żądań w agentach ConstructionAgent i ProductionAgent, należałoby ją wyekstrahować jako osobną funkcjonalność i, być może, skorzystać z niej w klasie bazowej (Agent), a nie w potomnych. Jednak w momencie pisania tej pracy mechanizm ów był jeszcze w fazie eksperymentalnej, stanowi więc jeden z niedokończonych elementów prototypu.

Aby móc wykonać zwiad, ReconAgent musi skorzystać z jednostek. Armia może nie mieć w tej chwili jednostek *Scout*, a *Harvester* też potrafi przeprowadzać rekonesans. W momencie, gdy ReconAgent potrzebuje zwiadowców bardziej, niż ResourceCollectorAgent zbieraczy złomu, ten pierwszy może wysłać do drugiego wiadomość — prośbę o przekazanie agenta. Dzieje się to szczególnie, gdy nie ma dostępnych żadnych lepszych jednostek nadających się do zwiadu.[[33]](#footnote-33) Mija jednak pewien czas, zanim ResourceCollectorAgent przetworzy zapytanie i zastosuje się do niego. Dlatego w klasie RequestUnitAgentSet istnieją tak naprawdę trzy zbiory obiektów UnitAgent. Zbiór All zawiera wszystkie agenty przypisane do żądania. Zbiór Ready gromadzi te, które ReconAgent już ma. Wreszcie do zbioru Awaiting dodawane są jednostki, o które poproszono innego agenta, ale jeszcze nie zostały przekazane. Funkcja RequestUnitAgentSet.ReadyAgents, wykonywana co cykl aktualizacji, przemieszcza z Awaiting do Ready agentów już posiadanych przez ReconAgent i daje im zadanie określone w parametrze.

Metoda ProcessFindMeResources() korzysta z dwóch trzyelementowych tablic ustawionych sztywno parametrów. Funkcja pobiera z każdej tablicy taki element, którego indeks jest priorytetem obsługiwanego żądania. Ów priorytet otrzymywany jest razem z wiadomością „*Find me Resources*”. Zatem pole scoutsNeededByPriority zawiera parametry określające liczbę jednostek zwiadowczych, które ReconAgent ma przypisać do tego zadania, natomiast scoutsImportanceByPriority — mieszczące się między 0 a 1 ważności zadania rekonesansu.

bool waitingForAnyScout;

bool waitingForNonBusyScout;

int[] scoutsNeededByPriority = { 3, 1, 1 };

float[] scoutsImportanceByPriority = { 0.8f, 0.6f, 0.35f };

void ProcessFindMeResources(object args)

{

var concreteArgs = (ProcessFindMeResourcesArgs)args;

var r = concreteArgs.request;

var processed = concreteArgs.processed;

// Get all available MapElementKinds of UnitAgents suitable for Scouting

var kinds =

from k in Knowledge.UnitAgents.Kinds

let p = k.GetPurposeValue(AIName.Scouting)

where p > 0

orderby p descending

select k;

// Send request for production of Scouts if no kinds

if (kinds.Empty())

{

if (!waitingForAnyScout)

{

SendMessage(Production, AIName.ProduceMeUnits, "1", AIName.Scout);

waitingForAnyScout = true;

}

return;

}

else waitingForAnyScout = false;

// Determine how many scouts are needed and how important is their task

int scoutsNeeded = scoutsNeededByPriority[r.Priority];

float scoutsImportance = scoutsImportanceByPriority[r.Priority];

// Get UnitAgents currently assigned to this Request and update them

var uaSet = ReconUnits[r];

uaSet.ReadyAgents(ua => new CoarseReconGoal(ua), scoutsImportance);

foreach (var ua in uaSet.Ready)

ua.CurrentGoal.Importance = scoutsImportance;

// Determine how many more scouts are needed

int scoutsNeededLeft = scoutsNeeded - uaSet.All.Count;

// Release surplus scouts

for (; scoutsNeededLeft < 0; scoutsNeededLeft++)

{

var toRemove = uaSet.All.SelectMin(

ua => ua.Kind.GetPurposeValue(AIName.Scouting));

toRemove.CurrentGoal.Finish();

toRemove.Release(this);

uaSet.RemoveAgent(toRemove);

}

// Get all not assigned UnitAgents and sort them by their Suitability

var unitAgentsSuitabilities =

from ua in Knowledge.UnitAgents.All

where !uaSet.All.Contains(ua)

let p = ua.Kind.GetPurposeValue(AIName.Scouting)

where p > 0

let i = ua.CurrentGoalImportance

where i < scoutsImportance

let s = CalcSuitability(i, p)

orderby s descending

select new { Agent = ua, Suitability = s };

**Fragment kodu 33.** Metoda ProcessFindMeResources() – część 1

// As long as there is not enough scouts assigned to this Request

for (; scoutsNeededLeft > 0; scoutsNeededLeft--)

{

// Send request for production, if there are no more scouts to take

if (unitAgentsSuitabilities.Empty())

{

if (!waitingForNonBusyScout)

{

SendMessage(Production, AIName.ProduceMeUnits, "1", AIName.Scout);

waitingForNonBusyScout = true;

}

break;

}

else waitingForNonBusyScout = false;

// Look for the most suitable UnitAgent, that can be taken

var uas = unitAgentsSuitabilities.First();

TakeAgentNowOrLater(uas.Agent, uaSet, scoutsImportance);

}

// Get all UnitAgents assigned to this Request and sort them by their Purposes

var requestUnitAgentsPurposes =

from ua in uaSet.All

let p = ua.Kind.GetPurposeValue(AIName.Scouting)

orderby p descending

select new { Agent = ua, Purpose = p };

// Replace current UnitAgents with more suitable if available

bool replaced;

do

{

replaced = false;

if (requestUnitAgentsPurposes.Empty()) break;

if (unitAgentsSuitabilities.Empty()) break;

var firstUAS = unitAgentsSuitabilities.First();

var lastRUAP = requestUnitAgentsPurposes.Last();

if (firstUAS.Suitability > lastRUAP.Purpose)

{

replaced = true;

lastRUAP.Agent.CurrentGoal.Cancel();

lastRUAP.Agent.Release(this);

uaSet.RemoveAgent(lastRUAP.Agent);

TakeAgentNowOrLater(firstUAS.Agent, uaSet, scoutsImportance);

}

}

while (replaced);

// Determine total map exploration percentage

float sum = 0;

float total = 0;

foreach (var rb in AllReconRegions)

{

sum += rb.ExplorationPercentage;

total++;

}

float totalExplorationPercentage = sum / total;

**Fragment kodu 34.** Metoda ProcessFindMeResources() – część 2

// If coarse recon is done, finish request and release all agents

if (totalExplorationPercentage >= Brain.coarseReconRegionPercentage)

{

processed.Add(r);

foreach (var ua in uaSet.Ready)

{

ua.CurrentGoal.Cancel();

ua.Release(this);

}

foreach (var ua in uaSet.Awaiting)

awaitingNoLongerNeededUnitAgents.Add(ua);

uaSet.Clear();

}

}

**Fragment kodu 35.** Metoda ProcessFindMeResources() – część 3

Metoda obsługi żądania poszukiwania zasobów stanowi długi algorytm postępowania. Na początku przeszukuje ona zbiór agentów jednostek w KnowledgeAgent i wyłuskuje ich rodzaje. Automatycznie odrzuca przy tym te rodzaje, które nie są przeznaczone do zwiedzania. Jeśli okaże się, że nie ma agentów nadających się do rekonesansu, wysyłana jest wiadomość „*Produce me Units*” do agenta ProductionAgent — prośba o produkcję jednostek *Scout* — i na tym funkcja w się kończy. Przy każdym następnym uruchomieniu wiadomość już nie zostanie wysłana — metoda będzie czekać, aż odpowiednie jednostki zostaną stworzone.

Zakładając, że już istnieją agenty odpowiedniego przeznaczenia, by ich użyć, ReconAgent musi je najpierw wziąć, a następnie przypisać im zadanie CoarseReconGoal[[34]](#footnote-34). W tym celu metoda określa ile należy ich pobrać i jaką ważność przypisać ich zadaniom. Liczby te wybierane są ze wspomnianych tablic parametrów na podstawie priorytetu żądania. Jeśli okazuje się, że obecnie ReconAgent ma za dużo agentów, nadmiarowe są zwalniane.[[35]](#footnote-35) Bez względu na to algorytm przechodzi do następnego kroku.

Określiwszy parametry metoda generuje listę par UnitAgent i Suitability — to drugie jest liczbą z przedziału zamkniętego (0, 1) określającą, na ile właściwe jest wzięcie agenta do rekonesansu. Suitability oblicza funkcja CalcSuitability(). Jej argumenty (również od 0 do 1) to przeznaczenie agenta do zwiadu oraz ważność jego obecnego zadania (lub 0, jeśli zadania brak). W efekcie CalcSuitability() stanowi dwuwymiarową funkcję matematyczną. Obecnie jest ona obliczana wzorem:

,

gdzie: — ważność obecnego zadania agenta, — przeznaczenie agenta do zwiadu.

Lista par jest posortowana malejąco po ich Suitability. Dzięki temu w następnym kroku można wybrać najbardziej pasujących agentów. Jeśli nie ma więcej dostępnych, nadających się jednostek, znów wysyłana jest prośba o ich produkcję do ProductionAgent. Dopóki jednak lista zawiera agenty, ReconAgent bierze je i daje im zadanie CoarseRegionGoal (przypisując mu ważność parametru z tablicy scoutsImportanceByPriority), lub (jeśli agent już jest zajęty) wysyła wiadomość do jego właściciela z prośbą przekazanie. Funkcja TakeAgentNowOrLater(), która do tego służy, dodatkowo umieszcza obiekt UnitAgent w zbiorze ReconUnitAgentSet związanego z przetwarzanym żądaniem (odpowiednio na listę Ready, bądź Awaiting). Gdy zajęty agent nie może zostać oddany natychmiast, zadanie otrzyma dopiero w momencie przekazania i przepisania z listy Awaiting na Ready.

Kolejnym etapem algorytmu jest wymiana obecnych agentów na lepszych, gdyż podczas wykonywania żądania mogły się pojawić nowe. Metoda przetwarza zbiór obecnie posiadanych przez ReconAgent zwiadowców na listę par agent-przeznaczenie, posortowaną malejąco po przeznaczeniach. Następnie brany jest agent z poprzedniej listy o najwyższej wartości Suitability i porównywany z agentem z nowej listy o najniższej wartości Purpose. Gdy okaże się, że nowy UnitAgent nadaje się do zwiadu bardziej, obecnie posiadany agent jest zwalniany i zastępowany lepszym z nich. Jeśli doszło do zamiany, procedura jest ponawiana. W przeciwnym wypadku na pewno nie ma sensu brać innych agentów, gdyż najlepszy z nich okazał się nie bardziej nadawać do zadania zwiadu, niż najgorszy z obecnie je wykonujących.

Wreszcie ostatnim krokiem jest zdeterminowanie warunku końca. Metoda sprawdza procent, w jakim odkryty jest cały teren. Jeśli jest on większy, niż globalny parametr sztucznej inteligencji AIBrain.coarseReconRegionPercentage (ustalony na 90%), przetwarzane żądanie dodawane jest na listę processed, dzięki czemu obiekt Request zostanie usunięty z listy ReconAgent.requests. Przy tym zwalniane są wszystkie posiadane agenty, a te oczekujące na przekazanie dodawane są do zbioru awaitingNoLongerNeededUnitAgents. Agent, którego ReconAgent poprosił o oddanie jednostki pyta się go najpierw, czy prośba wciąż jest aktualna (korzystając z funkcji MakeSureIfHandOn()). ReconAgent usuwa wtedy agenta ze wspomnianego zbioru, jeśli tam jest, i zwraca odpowiednią wartość bool.

W ten sposób realizacja żądania „*Find me resources*” doprowadza w rzeczywistości do odkrycia prawie całej planszy. Żądanie mogłoby mieć warunek zakończenia wcześniej, ale tak naprawdę zawsze warto zlokalizować następne zasoby, a mogą znajdować się one w dowolnym rejonie mapy. Nie jest też problemem, że żądanie zajmuje obiekt UnitAgent, który mógłby być użyty do innego celu, ponieważ wystarczy dopisać klasie ReconAgent obsługę wiadomości „*Hand me on Unit*”. Może ona oddawać jednostkę po uprzednim porównaniu ważności zadania zwiadu z ważnością drugiego zadania — tego, które agent proszący chce jednostce dać. W ogólności samą ważność zwiadu możnaby też uzależnić od tego, ile regionów zasobów jest widocznych i jakie są ich rozmiary. Podsumowując, istnieją przy przetwarzaniu żądania rekonesansu aspekty, które bez wątpienia warto ulepszyć, lecz jednocześnie istnieją i narzędzia, za pomocą których owe ulepszenia da się zrealizować.

### Zadanie zgrubnego rekonesansu

Klasa CoarseRegionGoal realizuje pobieżny zwiad całej planszy gry. Owa pobieżność zależy od wartości parametru *AI*: pola AIBrain.coarseReconPercentage. Wyraża ona procent odkrytych pól mapy, jaki jest wystarczająco dobry dla algorytmu. Domyślnie parametr ustawiony jest na 90%.

Zadanie zwiadu zgrubnego zarządza agentem jednostki wybranym na zwiadowcę, wydając mu co pewien czas rozkaz MoveOrder. By zorientować się w postępie zwiedzania mapy, korzysta z tablicy ReconRegions, z regionami generowanymi przez obiekt ReconAgent. Obszary te monitorują stopień, w jakim są odkryte i udostępniają właściwości do pobrania tej informacji. [[36]](#footnote-36) Jeśli wydany jest rozkaz ruchu, na początku metody OnUpdate() może on zostać zatrzymany — dzieje się to, gdy obecnie wybrany region jest już widoczny w części ustalonej przez coarseReconPercentage. Taki region dodawany jest do zbioru visited, by więcej do niego nie wracać. Zadanie wybiera więc następny, korzystając z funkcji TakeNextReconRegion(). Na koniec zadanie wydaje zwiadowcy rozkaz ruchu do pozycji będącej środkiem nowo wybranego regionu.

Metoda TakeNextReconRegion() jest dosyć ważna, gdyż decyduje o ścieżce, jaką porusza się skaut. Funkcja musi przeszukać wszystkie regiony i wybrać z nich ten, który w danym momencie najlepiej odwiedzić. Pomija przy tym wszystkie elementy zbioru visited oraz regiony zwiedzone w procencie większym, niż coarseRegionPercentage. Regiony są sortowane przy pomocy funkcji abstrakcyjnej Calculate() obiektu ReconRegionOrderCalculator (skryptu MonoBehaviour), parametryzowanego w panelu *Inspector* dla komponentu AIBrain. Następnie wybierany jest pierwszy region, który nie jest już docelowym regionem innego agenta-zwiadowcy. Jeśli regionu nie dało się wybrać, zadanie zostaje ukończone sukcesem (gdyż oznacza to, że plansza jest zwiedzona w stopniu zadowalającym).

protected override void OnUpdate()

{

if (CurrentReconRegion == null) return;

if (currentMoveOrder != null &&

CurrentReconRegion.ExplorationPercentage >=

Agent.Brain.coarseReconPercentage)

currentMoveOrder.Stop();

if (currentMoveOrder != null && currentMoveOrder.InFinalState)

currentMoveOrder = null;

if (currentMoveOrder == null)

{

visited.Add(CurrentReconRegion);

TakeNextReconRegion();

if (CurrentReconRegion == null) return;

}

var regCenter = CurrentReconRegion.ConvexHull.Center.Round();

var u = UnitAgent.Unit;

if (u.OrderQueue.CurrentOrder == null ||

!(u.OrderQueue.CurrentOrder is MoveOrder))

{

currentMoveOrder = new MoveOrder(u, regCenter);

u.OrderQueue.Give(currentMoveOrder);

}

}

void TakeNextReconRegion()

{

var orderCalculator = Agent.Brain.reconRegionOrderCalculator;

var recRegs = Agent.Recon.AllReconRegions;

var sortedRecRegs =

from reg in recRegs

where !visited.Contains(reg)

where reg.ExplorationPercentage < Agent.Brain.coarseReconPercentage

orderby orderCalculator == null ? 0 :

orderCalculator.Calculate(this, reg)

select reg;

var otherUnitAgents = Agent.Recon.ReconUnits

.SelectMany(kv => kv.Value.Ready)

.Where(ua => ua != UnitAgent)

.Where(ua => ua.CurrentGoal != null)

.Where(ua => ((CoarseReconGoal)ua.CurrentGoal)

.CurrentReconRegion != null);

CurrentReconRegion = sortedRecRegs.FirstOrDefault(

reg => !otherUnitAgents.Any(

ua => ((CoarseReconGoal)ua.CurrentGoal)

.CurrentReconRegion == reg));

if (CurrentReconRegion == null) Finish();

}

**Fragment kodu 36.** Treści metod OnUpdate() i TakeNextReconRegion() klasy CoarseReconRegion

Wspomniana klasa ReconRegionOrderCalculator jest abstrakcyjna. Jej potomkowie określają treść metody Calculate(), będącą czynnikiem decyzyjnym co do kolejności zwiedzania regionów, a zatem: skuteczności zwiadu. Z tego powodu kolejne, stopniowo coraz lepsze implementacje tej metody zostały opisane w rozdziale na temat badań *AI*.[[37]](#footnote-37)

### Agent rozbudowujący bazę

Obiekt ConstructionAgent w podobny sposób do agenta ReconAgent reaguje na wiadomości i przetwarza żądania przy pomoc metod ProcessMessages() oraz ProcessRequests().

Funkcja ProcessMessages() odbiera każdą wiadomość o nazwie „*Construct me Building*”. Stanowi ona prośbę innego agenta o skonstruowanie budynku. W reakcji na nią, ConstructionAgent wysyła zwrotną wiadomość „*Ok*” (ustawiając jej tę pierwszą jako InnerMessage), aby nadawca miał informację, że jego prośba zostanie zrealizowana. Następnie, na bazie otrzymanego obiektu Message, tworzony jest Request i dodawany do listy żądań. Należy nadmienić, że wiadomość ta, oprócz standardowego argumentu: priorytetu, przekazuje też drugi argument: nazwę budynku, jaki ma zostać skonstruowany. Wybór miejsca natomiast należy do agenta konstruującego — choć wystarczyłoby dodać do wiadomości trzeci argument i uwzględnić go w obsłudze żądania, by dać możliwość określenia pozycji budynku przez nadawcę wiadomości.

W funkcji ProcessRequests() przetwarzane jest żądanie konstrukcji budynku. Analogicznie do ReconAgent, ConstructionAgent posiada tu listę processed, do której można dodać obiekt Request, by został usunięty z listy żądań pod koniec funkcji. Ponieważ zadanie tworzenia budynku realizowane jest natychmiastowo — wystarczy wydać budynkowi ConstructionYard odpowiedni rozkaz — agent konstruujący nie korzysta tutaj z funkcji PerformEvery(), by rozciągnąć je w czasie. Obsługa tego żądania jest znacznie prostszym i krótszym algorytmem od obsługi żądania poszukiwania zasobów w ReconAgent.

var buildingName = r.InnerMessage.Arguments[1];

var buildingKind = Knowledge.MapElementKinds[buildingName];

var creationMethod = Knowledge.CreationMethods[buildingKind];

var creatorKind = creationMethod.Creator;

var startCost = creationMethod.StartCost;

var requiredBuildings = creationMethod.BuildingRequirements;

bool dontFinish = false;

Building creator = null;

BuildingConstructionOrderAction orderAction = null;

var completeBuildings = Army.Buildings.Where(b => !b.UnderConstruction);

var creators = completeBuildings.Where(b =>

b.mapElementName == creatorKind.Name);

if (creators.Empty()) {

if (!Construction.HasCurrentRequestOfKind(creatorKind) &&

!HasGivenOrdersOfKind(creatorKind) &&

!Army.Buildings.Any(\_b => \_b.mapElementName == creatorKind.Name))

SendMessage(this, AIName.ConstructMeBuilding,

r.Priority.ToString(), creatorKind.Name);

dontFinish = true;

} else {

creator = creators.SelectMin(c => c.OrderQueue.OrderCount);

orderAction = creator.orderActions

.OfType<BuildingConstructionOrderAction>().FirstOrDefault(oa =>

oa.Building.mapElementName == buildingKind.Name);

if (Army.resources < startCost) {

var isRefinery = buildingName == AIName.Refinery;

var hasRefineries = completeBuildings.Any(b =>

b.mapElementName == AIName.Refinery);

if (isRefinery && !hasRefineries) {

processed.Add(r);

SendMessage(MainAgent, AIName.NoRefineriesAndNoResources);

}

else SendMessage(ResourceCollector, AIName.HarvestMore);

dontFinish = true;

}

}

var requiredBuildingsLeft = requiredBuildings.Where(

b => !completeBuildings.Any(\_b => \_b.mapElementName == b.Name));

if (!requiredBuildingsLeft.Empty()) {

foreach (var b in requiredBuildingsLeft)

if (!Construction.HasCurrentRequestOfKind(b) &&

!HasGivenOrdersOfKind(b) &&

!Army.Buildings.Any(\_b => \_b.mapElementName == b.Name))

SendMessage(this, AIName.ConstructMeBuilding,

r.Priority.ToString(), b.Name);

dontFinish = true;

}

if (dontFinish) continue;

var place = PickBuildingPlace(buildingKind);

if (place.CannotBuild) continue; // handle the situation instead

var givenOrder = (BuildingConstructionOrder)orderAction.GiveOrder(creator, newAIOrderActionArgs(Brain.player, place));

if (givenOrder != null) {

givenOrder.BuildingFinished += GivenOrder\_BuildingFinished;

givenOrders.Add(givenOrder);

}

processed.Add(r);

**Fragment kodu 37.** Algorytm obsługi żądania „*Construct me Building*” w klasie ConstructionAgent

Na początku pobierane są wszelkie informacje potrzebne do podejmowania decyzji związanych z konstrukcją budynku. Są to: nazwa budynku, jego rodzaj oraz sposób tworzenia, który dostarcza informacji takich, jak rodzaj budynku konstruującego, koszt rozpoczęcia produkcji czy też lista wymaganych innych budynków. Potrzebne technologie nie są tu uwzględniane, gdyż agent nie został w na tyle rozwinięty, by tworzyć budynki ich wymagające.

Następnie tworzona jest lokalnie flaga dontFinish, która może zostać ustawiona w następnych krokach algorytmu. Jeśli tak się stanie, kilka kroków wprzód wciąż będzie wykonanych, lecz znajdujący się na końcu rozkaz konstrukcji nie zostanie wydany. Jednocześnie żądanie nie znajdzie się na liście processed, więc w następnym cyklu aktualizacji algorytm spróbuje wykonać się ponownie. Mechanizm flagi dontFinish istnieje, gdyż pewne wymagania mogą nie być jeszcze w tym momencie spełnione i należy na nie poczekać.

Pierwszym warunkiem sprawdzanym w obsłudze żądania, jest to, czy armia posiada inny budynek — taki, który potrafi skonstruować ten, o który poproszono. Jeśli nie, ConstructionAgent wysyła wiadomość do samego siebie, by go utworzyć (chyba, że proces jego produkcji jest już w trakcie realizacji). Od razu ustawiana jest flaga dontFinish.

Jeśli choć jeden budynek konstruujący istnieje, wybierany jest ten o najmniejszej liczbie rozkazów w kolejce i wyciągana jest akcja rozkazu służącego do konstrukcji żądanego budynku. Następnie sprawdzane jest, czy armię w ogóle stać na rozpoczęcie konstrukcji.[[38]](#footnote-38) Jeśli nie, realizowane są dwa przypadki. W standardowym trybie wysyłana jest wiadomość „Harvest more” do obiektu ResourceCollectorAgent, by pospieszył się ze zbieraniem zasobów. ResourceCollectorAgent w obecnym stanie prototypu nie reaguje na tę wiadomość, ale mógłby na przykład zwiększać ważność zadania zbierania zasobów. Przypadek nadzwyczajny następuje, gdy budynkiem do skonstruowania jest rafineria, a żadna inna nie istnieje. Fakt ten oznacza, że: **1)** *Harvestery* w tym momencie nie mają gdzie odnosić zasobów, więc armia ich nie pozyska, **2)** Rafineria nie powstanie, bo brakuje na to zasobów. Gracz nie ma żadnych możliwości rozwiązać tego problemu, dlatego ConstructionAgent dodaje żądanie do listy processed (by go dłużej nie obsługiwać) oraz wysyła do głównego agenta wiadomość „*No Refineries and no resources*”. Reakcja na tę wiadomość nie została zaimplementowana, ale mogłaby ona być poddaniem się (walkowerem). W obu powyższych przypadkach flaga dontFinish jest ustawiana.

Następny krok jest wykonywany bez względu na przebieg poprzednich. Sprawdzane są wymagania do konstrukcji — czy armia ma zbudowane potrzebne struktury. Dla każdego budynku, którego nie posiada, ConstructionAgent wysyła do siebie wiadomość proszącą o jego stworzenie (o ile już żądanie takiego budynku nie jest już wykonywane) i ustawia flagę dontFinish.

Algorytm dochodzi do momentu, w którym przerywa swe działanie, gdy flaga dontFinish jest ustawiona. Jeśli jednak przejdzie dalej, to wywoływana jest funkcja PickBuildingPlace() by określić w jakim miejscu budynek ma zostać skonstruowany.

Funkcja ta wpierw woła metodę GetAvailablePlacements()[[39]](#footnote-39), by pozyskać listę współrzędnych dookoła bazy, w których da się zbudować żądany budynek. Następnie wybiera jedną z pozycji z tej listy, analizując ją pod kątem różnych kryteriów — zależnych od tego, jaki budynek ma zostać stworzony. Jeśli konstruowana jest rafineria, to należy znaleźć miejsce o minimalnej odległości do najbliższego regionu zasobów. Dzięki temu *Harvester*y będą miały krótszą trasę do przebycia, co zwiększy przyrost jednostek zasobów w czasie. Natomiast przy budowie fabryki wybierana jest pozycja najdalej od rafinerii (jeśli istnieje), by w możliwie małym stopniu blokować *Harvesterom* ścieżkę, po której kursują.

Jeśli PickBuildingPlace() nie znajdzie żadnych współrzędnych, w których da się skonstruować budynek, następuje nieobsłużona sytuacja. W obecnym stanie algorytm po prostu jest przerywany, jednak żądanie nie zostaje usunięte.

Jeśli miejsce zostało wybrane, budynkowi konstruującemu wydawany jest rozkaz z pobranej uprzednio akcji. Następnie agent zapisuje sobie go w zbiorze wydanych rozkazów i przypina do niego metodę obsługi zdarzenia zakończenia konstrukcji budynku. Dzięki temu *AI* pamięta, że tworzenie danej struktury jest już w trakcie realizacji, co zapobiega powielaniu żądań. Ostatecznie Request zostaje dodany do listy processed, gdyż jego cel został spełniony.

W powyższym algorytmie prototyp potrzebuje rozwiązania problemu braku miejsca na budynek, który może wynikać z kilku powodów. Przykładowymi przyczynami mogą być jednostki tarasujące pozycję dobrze nadającą się na budowę, albo brak odkrytego terenu (nie można wznosić budynków na nieodkrytym terenie). Pierwszą kwestię można łatwo rozwiązać poprzez wzięcie agentów tych jednostek i wydanie im rozkazu opuszczenia wybranych pól. Druga wymaga trochę więcej wkładu — należałoby wysłać wiadomość do ReconAgent i poprosić go o zwiedzenie większej połaci terenu dookoła bazy. Mimo tego, podwaliny pod kompletne zachowanie rozbudowy bazy istnieją. Można łatwo sobie wyobrazić rozszerzenie tego algorytmu o realizowanie konstrukcji budunków w kolejności wynikającej zarówno z ich pełnego kosztu i obecnej liczby jednostek zasobów, jak i z zapotrzebowania na nie. Wyzwanie natomiast mógłby stanowić mechanizm konstrukcji murów i wieżyczek obronnych, gdyż musiałby wybierać istotne strategicznie miejsca (np. mury, by mieć sens, muszą zachowywać ciągłość, a zasięgi wieżyczek powinny w pewnym stopniu nachodzić na siebie; do tego niektóre rejony bazy mogą być bardziej wrażliwe na atak od innych, więc istnieją miejsca wyróżnione pod względem zapotrzebowania na struktury defensywne).

### Agent produkcyjny

Klasa ProductionAgent wykazuje wiele podobieństw do agenta konstrukcji budynków[[40]](#footnote-40). Jego struktura jest tak bardzo zbliżona, że gdyby rozbudowywać projekt, prawdopodobnie zostałaby wyciągnięta do osobnej klasy. Obie klasy posiadają listę żądań, obie odczytują jeden (choć nie ten sam) rodzaj wiadomości powodujący dodanie obiektu Request na tę listę. Algorytm przetwarzania żądania przez agenta produkcji jest analogiczny do sposobu, w jaki robi to ConstructionAgent — choć prostszy (kilka kroków jest pominiętych).

Wiadomość, na którą reaguje agent w swojej metodzie ProcessMessages() to „*Produce me Unit*”. Stanowi ona prośbę o produkcję jednostki i przetwarzana jest na żądanie, nadając mu priorytet z jej argumentu. Request obsługiwany jest w funkcji ProcessRequests(), która korzysta z identycznego mechanizmu usuwania żądań, co taka sama funkcja w klasie ConstructionAgent.

Algorytm na początku pobiera z wiadomości argument — nazwę jednostki do wyprodukowania. Na tej podstawie pobiera sobie od agenta wiedzy rodzaj tej jednostki i jej sposób produkcji. Ten drugi jest tu potrzebny by określić jedynie budynek, jaki służy do produkcji żądanej jednostki oraz koszt jej wytworzenia. Wymagania nie są pobierane, ponieważ obecnie ProductionAgent służy jedynie do produkowania *Scoutów* i *Harvesterów* — a te jednostki wymagań nie mają.

var unitName = r.InnerMessage.Arguments[1];

var unitKind = Knowledge.MapElementKinds[unitName];

var creationMethod = Knowledge.CreationMethods[unitKind];

var creatorKind = creationMethod.Creator;

var cost = creationMethod.Cost;

bool dontFinish = false;

Building creator = null;

UnitProductionOrderAction orderAction = null;

var completeBuildings = Army.Buildings.Where(b => !b.UnderConstruction);

var creators = completeBuildings.Where(b =>

b.mapElementName == creatorKind.Name);

if (creators.Empty())

{

if (!Construction.HasCurrentRequestOfKind(creatorKind) &&

!Construction.HasGivenOrdersOfKind(creatorKind) &&

!Army.Buildings.Any(\_b => \_b.mapElementName == creatorKind.Name))

SendMessage(Construction, AIName.ConstructMeBuilding,

r.Priority.ToString(), creatorKind.Name);

dontFinish = true;

}

else

{

creator = creators.SelectMin(c => c.OrderQueue.OrderCount);

orderAction = creator.orderActions.OfType<UnitProductionOrderAction>()

.FirstOrDefault(oa => oa.unit.mapElementName == unitKind.Name);

if (Army.resources < cost)

{

var isHarvester = unitName == AIName.Harvester;

var hasHarvesters = Army.Units.Any(b =>

b.mapElementName == AIName.Harvester);

if (isHarvester && !hasHarvesters)

{

processed.Add(r);

SendMessage(MainAgent, AIName.NoHarvestersAndNoResources);

dontFinish = true;

}

else SendMessage(ResourceCollector, AIName.HarvestMore);

}

}

if (dontFinish) continue;

var givenOrder = (UnitProductionOrder)orderAction

.GiveOrder(creator, new AIOrderActionArgs(Brain.player));

if (givenOrder != null)

{

givenOrder.UnitSpawned += GivenOrder\_UnitSpawned;

givenOrders.Add(givenOrder);

}

processed.Add(r);

**Fragment kodu 38.** Algorytm obsługi żądania „*Produce me Unit*” w klasie ProductionAgent

Flaga dontFinish działa identycznie jak w klasie ConstructionAgent. Raz ustawiona sprawia, że kilka kroków algorytmu wprzód się wykona, ale rozkaz produkcji nie zostanie wydany.

Pierwszym sprawdzanym warunkiem jest to, czy armia posiada choć jeden budynek, którym można wyprodukować żądaną jednostkę. Jeśli go nie ma, do agenta konstrukcji wysyłana jest wiadomość „*Construct me Building*” — o ile ten już się jego stworzeniem nie zajmuje. Poza tym ustawiana jest flaga dontFinish.

Gdy budynek produkcyjny istnieje, algorytm bierze ten o najkrótszej kolejce rozkazów i pobiera jego akcję rozkazu produkcji jednostki. Następuje sprawdzenie, czy armię stać na tę operację. Jeśli nie ma dość zasobów, agent poprzez wiadomość „*Harvest more*” wysyła do obiektu ResourceCollectorAgent prośbę o szybsze ich zebranie, chyba że armia nia ma ani jednego Harvestera, a jest on żądaną jednostką. Widać, że gracz nie jest wtedy w stanie pozyskać więcej zasobów, dlatego wysyła do obiektu MainAgent wiadomość „*No Harvesters and no resources*”, na którą reakcją (której nie zaprogramowano) powinno być poddanie partii.

Cały powyższy krok algorytmu jest niemal identyczny do jednego z kroków algorytmu w klasie ConstructionAgent. Message „*No Harvesters (...)*” jest analogią do „*No Refineries (...)”.* Jeśli brakuje zasobów, w obu wypadkach wysyłana jest prośba o więcej zasobów (którą jej adresat ignoruje). Jednak ProductionAgent ustawia flagę dontFinish tylko, gdy nastąpi sytuacja wymagająca walkowera. Wynika to z tego, że sprawdzany w agencie produkcji koszt całkowity jednostki nie stanowi wydatku natychmiastowego, lecz jest rozciągnięty w czasie. Ma więc sens, by agent nie dysponując dostatecznymi środkami wciąż zarządził produkcję, ale dodatkowo upomniał się o więcej surowca.

Ten algorytm nie posiada kroku sprawdzania wymagań. Od razu przerywane jest jego działanie, jeśli flaga dontFinish jest ustawiona. W przeciwnym wypadku program nie wybiera też żadnego miejsca, gdyż jednostka nie posiada stałej pozycji. Rozkaz UnitProductionOrder jest więc po prostu wydawany budynkowi produkcyjnemu. Tutaj również przypinana jest obsługa zdarzenia zakończenia produkcji jednostki, a on sam zapamiętywany jest w zbiorze. Tak jak w ConstructionAgent, służy to zapobieganiu duplikatom żądań.

### Agent zbierający zasoby

Obiekt klasy ResourceCollectorAgent zachowuje się dość odmiennie od dotychczas opisanych. Wcale nie korzysta on z mechanizmu żądań, odbiera wiadomości, jednak przetwarza je natychmiast. Znacznie większym stopniu wykorzystuje metodę wirtualną OnUpdate() — w niej zawiera się większość jego zachowania. Używa za to funkcji PerformEvery(), żeby móc pewne operacje wykonywać rzadziej.

LerpFunc2 harvestingImportanceFunction;

public HashSet<UnitAgent> Harvesters { get; private set; }

public HashSet<Building> Refineries { get; private set; }

public float HarvestingImportance { get; private set; }

protected override void OnUpdate()

{

ProcessMessages();

PerformEvery(1, UpdateRefineries);

if (Refineries.Count == 0)

{

if (!refineryOnTheWay)

PerformEvery(1, RequestForRefineryConstruction);

}

else refineryOnTheWay = false;

PerformEvery(1, RequestForHarvestersProduction);

PerformEvery(1, TryRequestForResourceSearch);

HarvestingImportance = CalcHarvestingImportance();

var resRegions = Knowledge.Resources.Regions;

if (!resRegions.Empty())

{

var freeHarvesters = Knowledge.UnitAgents[AIName.Harvester]

.Where(h => !h.Busy);

foreach (var h in freeHarvesters)

{

h.Take(this);

Harvesters.Add(h);

h.GiveGoal(new HarvestGoal(h, this), HarvestingImportance);

}

}

foreach (var h in Harvesters)

if (h.CurrentGoal is HarvestGoal)

h.CurrentGoal.Importance = HarvestingImportance;

}

float CalcHarvestingImportance()

{

if (harvestingImportanceFunction == null) return 0;

float res = Army.resources;

float harv = Harvesters.Count;

return harvestingImportanceFunction[harv, res];

}

**Fragment kodu 39.** Wybrane składniki klasy ResourceCollectorAgent

W ogólności agent ten zajmuje się zarządzaniem *Harvesterami* i procesem pozyskiwania surowców. Pierwszą czynnością w metodzie OnUpdate() jest odebranie i obsłużenie wiadomości. Ponieważ jednak nie jest ono związane z późniejszymi instrukcjami, zostanie opisane na końcu tego rozdziału.

Aby było gdzie odkładać zasoby, muszą istnieć do tego odpowiednie budynki. Agent więc posiada zbiór do przechowywania rafinerii. Co sekundę aktualizuje go za pomocą funkcji UpdateRefineries() — wyszukuje w armii nowe budynki, które dodaje do zbioru, a zniszczone usuwa. Jeśli zbiór jest pusty, wysyłana jest do agenta ConstructionAgent wiadomość „*Construct me Building*” z argumentem „*Refinery*”. Flaga refineryOnTheWay broni przed wysłaniem tej wiadomości wielokrotnie.

Oprócz tego ważne jest istnienie jednostek zbierających. Bez *Harvesterów* nie ma mowy o przyroście zasobów. Im zaś ich więcej, tym szybciej powinno się surowce pozyskiwać. Funkcja RequestForHarvestersProduction() ma za zadanie cyklicznie wysyłać do agenta ProductionAgent wiadomość „*Produce me Unit*” z argumentem „*Harvester*”. Korzysta ona z metody GetCurrentHarvestersCountRequired(), by obliczyć ile jednostek zbierających armia powinna w sumie posiadać w tym momencie gry. Różnica między rezultatem tej metody, a obecną liczbą *Harvesterów* określa ile razy prośba do agenta produkcji zostanie przekazana. Domyślnie działa to tak, aby w -tej minucie armia miała do dyspozycji *Harvesterów*.

Trzecim potrzebnym elementem jest sama wiedza o położeniu zasobów na planszy. Jeśli armia nie widzi żadnych zasobów, nie może ich też, oczywiście, pozyskać. Dlatego agent wysyła wiadomość „*Find me Resources*” do agenta ReconAgent. Priorytet żądania określany jest przez liczbę widocznych regionów z zasobami. Jeśli armia nie ma wiedzy o żadnym regionie, sytuacja jest krytyczna (priorytet 0). Gdy znany jest nie wiecej niż jeden region, wciąż warto poszukać zasobów, ale nie ma kryzysu (priorytet 1). Kiedy widać najwyżej trzy regiony, zwiad im poświęcony nie jest bardzo istotny (priorytet 2). Jeśli zaś surowców jest jeszcze więcej, prośba o ich znalezienie w ogóle nie jest wysyłana.

Wreszcie, po zapewnieniu trzech niezbędnych przy zbieraniu zasobów aspektów: posiadania rafinerii, *Harvesterów* oraz wiedzy o położeniach zasobów, ResourceCollectorAgent może zająć się sednem sprawy. Agent musi w tym momencie wyliczyć ważność zbierania zasobów, a następnie wziąć wszystkie niezajęte obiekty UnitAgent rodzaju „*Harvester*”, dodać je do zbioru Harvesters i dać im zadanie HarvestGoal o tej ważności. Ponadto owa ważność powinna też zostać zaktualizowana w zadaniach każdego z *Harvesterów* już obecnych w zbiorze.

Sposób obliczenia ważności jest osobnym problemem, któremu poświęcono trochę przemyśleń. Intuicyjnie można stwierdzić, że istotność zadania zbierania zasobu dla każdego z *Harvesterów* powinna zależeć od dwóch rzeczy: wartości zasobów posiadanych przez armię, oraz liczby samych *Harvesterów* pod jej kontrolą. Pierwszy parametr jest oczywisty: im więcej mamy surowców, tym mniej ich potrzebujemy. Drugi wynika z rozproszenia odpowiedzialności: im więcej armia ma do dyspozycji jednostek zbierających, tym mniejsza jest waga zadania przypadająca na każdą z nich.

Dzięki parametrowi ważności agenty rozdzielają między sobą jednostki. Przykładowo wewnątrz obsługi żądania „*Find me Resources*” przez ReconAgent[[41]](#footnote-41) znajduje się instrukcja pobierająca wszystkie jednostki zdolne do rekonesansu. Bierze ona pod uwagę ważność ich zadań, by określić, które z nich może wziąć z najmniejszą szkodą dla agenta obecnie je posiadającego. Jeśli więc liczby posiadanych zasobów i *Harvesterów* wzrosną, spadnie ważność zbierania zasobów, co da większą szansę na wzięcie jednego z nich przez ReconAgent.

Ważność obliczana jest przez metodę CalcHarvestingImportance(), która bierze pod uwagę właśnie liczbę zasobów i *Harvesterów*. Korzysta w tym celu z dwuargumentowej funkcji ciągłej. Dziedzina tej funkcji to , a przeciwdziedziną jest przedział zamknięty . Zdefiniowana jest ona poprzez obiekt klasy LerpFunction, będący dwuwymiarową tablicą wartości, z określonymi argumentami. Jeśli podane do indeksera tego obiektu argumenty trafią pomiędzy te zdefiniowane, stosowana jest interpolacja biliniowa, by wyliczyć wartość funkcji w punkcie. Macierz liczb determinująca funkcję składowana jest w pliku, będącym jednym z parametrów AIBrain: publicznym polem TextAsset harvestingImportanceFunction. Wpływ kształtu funkcji na wymianę jednostek między agentami miał zostać poddany badaniom w rozdziale 5, jednak nie zrobiono tego z powodu ograniczeń czasowych.

Funkcja ProcessMessages() reaguje na dwa rodzaje wiadomości. Jedną z nich jest „*Ok*”, będąca odpowiedzią od agenta konstrukcji na wysłanie mu „*Construct me Building*”. Gdy ResourceCollectorAgent otrzyma taki komunikat, wie, że jego żądanie jest realizowane, więc czyści flagę refineryOnTheWay i przestaje wołać metodę RequestForRefineryConstruction(). Druga wiadomość ma nazwę „Hand me on Unit”. W argumencie przychodzi liczba id — wskazanie na konkretnego agenta *Harvestera*. ResourceCollectorAgent woła funkcję nadawcy MakeSureIfHandOn(), by upewnić się, czy od wysłania wiadomości prośba o oddanie obiektu UnitAgent nie przestała być aktualna. Jeśli inny agent wciąż potrzebuje określonej przez niego jednostki, usuwana jest ona ze zbioru Harvesters, jej zadanie jest anulowane i wołana jest metoda HandOn().

Omówiwszy agenta zbierania zasobów można wysunąć spostrzeżenie pozwalające zrozumieć jego odmienność od innych agentów w szerszym zakresie. Obiekty ReconAgent, ConstructionAgent i ProductionAgent są czysto responsywne: wykonują żądania będące reakcjami na wiadomości. ResourceCollectorAgent jest natomiast swego rodzaju motorem wszystkich akcji, napędzającym pozostałe. To on wysyła prośby, to dla niego tworzone są jednostki i budynki, oraz przeprowadzane zwiady. Oczywiście planując kompletne *AI* szybko zauważyć można, że nie jest to uniwersalna zależność. ResourceCollectorAgent realizuje jedynie niewielką część zachowań ekonomicznych i to dość topornie. Brakuje mu choćby reakcji na wiadomość „*Harvest more*” (jaką mogłoby być podbicie czynnika Importance). Przy sztucznej inteligencji większego zakresu powstałyby agenty odpowiedzialne np. za rozwój technologii albo projektowanie układu bazy i one mogłyby również więcej funkcjonować samodzielnie i odpytywać inne agenty. Da się z tego wywnioskować, że agenty dzialą się na działające aktywnie oraz responsywnie. Można nawet zaryzykować stwierdzenie, że rozgraniczenie to jeszcze bardziej by się uwydatniło, jeśli wzrosłyby zakres odpowiedzialności *AI* i liczba rodzajów agentów.

### Zadanie zbierania zasobów

Klasa HarvestGoal jest prostym zadaniem które zarządza wyborem regionu obecnie zbieranych zasobów oraz wydawaniem *Harvesterowi* rozkazów. Posiada on publiczną właściwość HarvestedRegion i prywatne pole currentResource (zasób obecnie wybrany przez zadanie).

W metodzie OnUpdate() wpierw wybierany jest region zasobów przy pomocy funkcji PickHarvestedRegion(). Przeszukuje ona wszystkie pary region-rafineria by znaleźć tę o najmniejszym dystansie do granicy obszaru. W ten sposób preferowane są regiony znajdujące się jak najbliżej rafinerii. Następnie metoda OnUpdate() wydaje *Harvesterowi* nowy rozkaz HarvestOrder, jeśli takiego już nie ma. W przeciwnym wypadku sprawdza, czy obecny jego rozkaz jest prawidłowy, t.j. czy zasób wybrany przez ten rozkaz znajduje się w regionie wybranym przez zadanie. Należy zaznaczyć, że gdy rozkaz HarvestOrder się skończy (np. z powodu braku zasobów w zasięgu widzenia *Harvestera*), zadanie automatycznie wyda nowy. Służy do tego funkcja GiveNewOrder(), która na start określa dla rozkazu dowolny zasób z wybranego regionu.[[42]](#footnote-42)

protected override void OnUpdate()

{

if (HarvestedRegion != null && HarvestedRegion.RegionEmpty)

HarvestedRegion = null;

if (HarvestedRegion == null)

{

HarvestedRegion = PickHarvestedRegion();

if (HarvestedRegion == null) return;

}

var currentOrder = UnitAgent.Unit.OrderQueue.CurrentOrder;

if (!(currentOrder is HarvestOrder)) GiveNewOrder();

else

{

var harvestOrder = (HarvestOrder)currentOrder;

if (currentResource != harvestOrder.Resource)

{

currentResource = harvestOrder.Resource;

if (!HarvestedRegion.HasResource(currentResource))

GiveNewOrder();

}

}

}

**Fragment kodu 40.** Treść metody HarvestGoal.OnUpdate()

# Badania skuteczności sztucznej inteligencji

Zaimplementowana sztuczna inteligencja nie jest bardzo skomplikowana. Jej domyślne zachowanie polega na poszukiwaniu zasobów, ich zbieraniu oraz produkcji odpowiednich do tego celu jednostek i budynków. Mimo niewielkiej liczby zachowań jest już tu sporo aspektów, które można przebadać. W rozdziale przeprowadziliśmy kilka eksperymentów. Na początku zdefiniowaliśmy domyślny *setup* gry. Następnie uruchomiliśmy dla niego rozgrywkę i opisaliśmy nasze spostrzeżenia dotyczące zachowania gracza *AI* przy takim ustawieniu. Później przeszliśmy do badań zależności skuteczności rekonesansu od przyjętej funkcji sortowania regionów do zwiadu. Ekperymenty w tej części przeprowadziliśmy przy specyficznych warunkach początkowych. Testowaliśmy tam kolejne, coraz lepsze wersje algorytmu sortowania, otrzymując na koniec wynik, który przeszedł nasze oczekiwania. Mieliśmy jeszcze zamiar przebadać szybkość zbierania zasobów przez armię w zależności od częstości produkcji *Harvesterów*, jednak nie wykonaliśmy tego z powodu ograniczeń czasowych.



## Badanie przebiegu domyślnej rozgrywki

Ponieważ sztuczna inteligencja ma zaprogramowane pewne domyślne zachowania, nie można nie sprawdzić, jak wygląda przykładowy przebieg gry dla tych zachowań. Oczywiście nie jest to pełna rozgrywka — nie może być mowy o żadnych bitwach, opracowywaniu technologii czy nawet produkcji jednostek bojowych, a co dopiero o zniszczeniu przeciwnika. Jednak *AI* wykonuje takie czynności, jak przeprowadzenie zwiadu, pozyskiwanie surowców i podstawowa rozbudowa bazy. Dlatego uruchomiono przy standardowym ustawieniu planszy rozgrywkę i zaobserwowano pierwszych kilka minut jej przebiegu, by stwierdzić, czy zachowania te są skuteczne.

### Założenia eksperymentu

Przed przystąpieniem do testu należy sprecyzować wspomniany domyślny *setup* rozgrywki. Na początku każda strona konfliktu dysponuje jedynie budynkiem *ConstructionYard* oraz trzema *Harvesterami*. W swoim polu widzenia ma niewielki region surowców, a do wydania ma 500 jednostek zasobów (w skrócie: *RU*, ang. *resource units*). Armia startuje z górnego narożnika symetrycznej planszy[[43]](#footnote-43). Sama sztuczna inteligencja (obiekt AIBrain) zdefiniowana jest parametrami o następujących wartościach:

* resourceRegionDistance = 2,
* harvestingImportanceFunction: plik *StandardHarvestingImportance*,[[44]](#footnote-44)
* reconRegionSize = 4,
* coarseReconPercentage = 90,
* reconRegionOrderCalculator: prefab *BaseSelfProductAndTilesExploredOrderCalculator*.



**Ilustracja 16.** Startowa pozycja armii w standardowym ustawieniu rozgrywki.   
Czerwone linie — brzegi planszy, niebieska elipsa — region zasobów,   
zielona elipsa — położenie budynku *ConstructionYard* i *Harvesterów*.

### Przebieg i analiza symulacji

Poniżej zaprezentowano wyniki obserwacji przebiegu rozgrywki w kolejności chronologicznej. Kolejne zdarzenia opatrzone są czasami ich zajścia. Poniżej listy znajdują się również ilustracje pokazujące sytuację gry w danym momencie czasowym.

* **0:00** — Rozpoczęcie rozgrywki. ResourceCollectorAgent wysyła prośbę do ConstructionAgent o stworzenie rafinerii. Jej konstrukcja rozpoczyna się. Armia ma 460 *RU*.
* **0:07** — Rafineria w trakcie konstrukcji. Dwa *Harvestery* wysłane przez ResourceCollectorAgent do zbierania zasobów, trzeci zabrany przez ReconAgent wykonuje zwiad. Armia ma 318 *RU*.
* **0:18** — Rafineria skonstruowana. *Harvestery* zbierające wracają odłożyć zasoby. ReconAgent potrzebuje więcej *Scoutów*, a ResourceCollectorAgent *Harvesterów*, więc ConstructionAgent rozpoczyna konstrukcję fabryki. Armii pozostało 63 *RU* i liczba ta spada. *Harvester* zwiedzający eksploruje teren dookoła bazy.
* **0:23**— *Harvestery* odkładają zasoby do rafinerii, następuje szybki wzrost *RU*, przewyższający ich spadek wynikający z trwającej konstrukcji fabryki. Po odłożeniu zasobów armia ma 209 *RU*.
* **0:37** — Konstrukcja fabryki jest na wykończeniu. Zwiedzony został już cały teren najbliższy bazie, armia ma 133 *RU*.
* **0:45** — Fabryka zostaje ukończona, ale *Harvester* wciąż zwiedza. ProductionAgent zarządza konstrucję *Scouta*. Armia ma 144 *RU*.
* **0:51** — *Scout* został właśnie wyprodukowany. ReconAgent bierze go i zwalnia *Harvester* — teraz to *Scout* wyrusza na rekonesans. *Harvester* do tej pory zwiedzający jedzie zbierać zasoby. Armia ma 215 *RU*.
* **1:00**— Fabryka rozpoczyna produkcję czwartego *Harvestera*. Szybki zwiadowca dotarł już do granic widoczności i zaczyna eksplorować. Armia ma 310 *RU*.
* **1:20** — Powstaje czwarty *Harvester* i natychmiast jedzie zbierać zasoby. *Scout* zwiedza sąsiedni narożnik planszy. Armia ma 419 *RU*.
* **1:40** — Skierowanie dwóch nowych *Harvesterów* do zbierania zasobów znacznie przyspiesza ich przyrost w czasie — armia przekracza próg 1000 *RU*. Zasoby w pierwszym regionie właśnie się wyczerpały, *Harvestery* za chwilę będą jeździć w dalsze trasy. *Scout* eskploruje środek mapy.
* **2:00** — Fabryka rozpoczyna produkcję piątego *Harvestera*, *Scout* właśnie zbliżył się do drugiego z sąsiednich narożników mapy, wyczerpany został jednoelementowy region zasobów po drugiej stronie fabryki. Armia ma 1271 *RU*.
* **2:20** — Zakończona została produkcja piątego *Harvestera*. *Harvestery* zbierają zasoby z regionu ukrytego za przeszkodami. *Scout* zwiedza obszar przy dolnej krawędzi mapy i zbliża się do narożnika z wrogą bazą. Armia ma 1494 *RU*.
* **2:38** — *Scout* zwiedził 90% pól mapy i zatrzymał się. *Harvestery* kursują na trasie między rafinerią a zasobami. Armia ma 1849 *RU*.
* **2:50** — Armia przekroczyła próg 2000 *RU*.
* **3:00** — Rozpoczęła się produkcja szóstego *Harvestera*. Armia ma 2215 *RU*.
* **3:18** — Szósty *Harvester* wyprodukowany. Armia ma 2419 *RU*.
* **3:29** — Trzeci region z zasobami wyczerpany. Armia ma 2607 *RU*.
* **3:40** — *Harvestery* zaczynają kursować do dwóch regionów jednocześnie (3 do jednego, 3 do drugiego). Armia ma 2925 *RU* (niemal 3000). W tym momencie rozgrywkę przerwano.

 **Ilustracja 17.** Przebieg domyślnej rozgrywki, czas: 0:07. Strzałka żółta to zwiad,   
niebieska — zbieranie zasobów.



**Ilustracja 18.** Przebieg domyślnej rozgrywki, czas: 0:18. Żółta strzałka na minimapie  
to trasa zwiadu



**Ilustracja 19.** Przebieg domyślnej rozgrywki, czas: 0:37. Żółta strzałka na minimapie  
to trasa zwiadu.



**Ilustracja 20.** Przebieg domyślnej rozgrywki, czas: 0:45. Żółta strzałka na minimapie to trasa zwiadu. Pokazano kolejkę rozkazów fabryki.



**Ilustracja 21.** Przebieg domyślnej rozgrywki, czas: 0:51. Żółty okrąg to wyprodukowany *Scout*.



**Ilustracja 22.** Przebieg domyślnej rozgrywki, czas: 1:00. Żółta strzałka to zwiad,  
niebieska — powrót Harvestera.



**Ilustracja 23.** Przebieg domyślnej rozgrywki, czas: 1:20. Zielony okrąg to wyprodukowany *Harvester*, żółta strzałka — trasa zwiadu.



**Ilustracja 24.** Przebieg domyślnej rozgrywki, czas: 1:40. Żółta strzałka to trasa zwiadu.



**Ilustracja 25.** Przebieg domyślnej rozgrywki, czas: 2:00. Żółta strzałka to trasa zwiadu,   
niebieski okrąg — wyczerpany region zasobów.



**Ilustracja 26.** Przebieg domyślnej rozgrywki, czas: 2:20. Zielony okrąg to wyprodukowany *Harvester*, żółta strzałka — trasa zwiadu.



**Ilustracja 27.** Przebieg domyślnej rozgrywki, czas: 2:38 Żółta strzałka to trasa zwiadu i miejsce zatrzymania się *Scouta*, niebieska strzałka — trasa rafineria-zasoby.

Po obserwacji można śmiało powiedzieć, że AI zachowuje się w sposób niemal zgodny z zamierzeniem. Początkowo teren zwiedza jeden z *Harvesterów*, jednak armia szybko dąży do wyprodukawania *Scouta*. Gdy to osiągnie, jednocześnie można zauważyć przyspieszenie zarówno w pozyskiwaniu *RU*, jak i w eksploracji terenu. *Scout*, będąc znacznie szybszą jednostką o dużo większym polu widzenia, realizuje zadanie rekonesansu w dużo lepszym stopniu od *Harvestera*. Problemem jest, że po zakończeniu zwiadu nie wraca on do bazy — zachowanie to nie zostało w ogóle zaimplementowane. Widać też, że błędem jest stosowanie parametru zgrubnego zwiedzania (90%) do całej planszy — powinien być wykorzystywany tylko do pojedynczych regionów. *Scout* wyraźnie zignorował spore połacie terenu w lewej części mapy, jak również narożnik z bazą wrogiej armii. Taki błąd zachowania mógłby przeważać o wygranej lub przegranej w prawdziwej rozgrywce.

### Analiza przyrostu zasobów i odkrytego terenu

Rozgrywkę uruchomiono ponownie, uzyskując niemal identyczny przebieg wydarzeń w czasie 3 minut 40 sekund. (AI nie ma mechanizmów randomizacji). Symulację kontynuowano aż do końca 10 minuty gry. Wykonano przy tym pomiary liczby zwiedzonych kratek oraz liczby zebranych zasobów w czasie, w odstępach 1 sekundy. W rozdziale tym prezentujemy wykresy z wynikami.

**Wykres 1.** Liczba jednostek zasobów w czasie pierwszych dwóch minut gry, próbkowana co sekundę.

**Wykres 1** pokazuje cztery etapy pierwszych dwóch minut rozgrywki. Spadkowa tendencja niebieskiej części wynika z trwającej konstrukcji rafinerii. Jej koszt (400 *RU*) i czas budowy (20 s) są tu jedynym czynnikiem wpływającym na zmianę zasobów. Wydawanie ich w tempie 20 *RU*/s jest zgodne z oczekiwaniami. Trend liniowy jest doskonale odwzorowany.

Po wybudowaniu rafinerii widzimy już wzrost *RU* — to dlatego, że *Harvestery* mają gdzie teraz odkładać zasoby. W przerwach między przyrostami następuje spadek wynikający z konstrukcji fabryki (koszt 350 *RU*, czas 30 s). Choć regresja liniowa nie jest tu dopasowana zbyt dobrze, jest to jedyna, jaka ma sens. Armia w tym okresie dysponuje stałą liczbę *Harvesterów* i zbiera zasoby cały czas z tego samego regionu, co oznacza niezmienny dystans do rafinerii. Dlatego średni przyrost zasobów w tym czasie musi być stały. Natomiast gdy obciążenie z konstruowania fabryki przestanie mieć wpływ, (0:45) następuje produkcja *Scouta*, która jeszcze przez 5 s zmniejsza przyrost *RU*. Tylko przez 10 kolejnych sekund nie widzimy spadku zasobów. Od 1:00 zaczyna się produkcja *Harvestera*, trwająca aż do 1:20 (do 80 sekundy gry). Mamy zatem w tym 60-sekundowym odcinku czasu niemal stałą składową wzrostu, oraz mniejszą, niemal stałą składową spadku. Niska korelacja wynika tu z dyskretności procesu odkładania zasobów do rafinerii, co wpływa na „poszarpany” kształt wykresu.

Od 1:20 następuje moment, w którym nie dość, że nie ma wydatków (aż do 2:00, wtedy zaczyna się produkcja następnego *Harvestera*), to jeszcze armia dysponuje nowym *Harvesterem*, a ten który w pierwszej minucie zwiedzał okolice bazy, dociera do regionu z zasobami. Dotychczas zbieraniem surowców zajmowały się tylko dwa *Harvestery*, zatem ich liczba podwaja się. Dlatego trend w następnych 25 sekundach znacznie się wybija. Jest to niemal **pięciokrotny**[[45]](#footnote-45) wzrost średniego przychodu *RU* w czasie.

Stały etap do końca 2 minuty wynika z wyczerpania się regionu zasobów najbliższego rafinerii. Nagle *Harvestery* musiały zacząć kursować do odleglejszych źródeł, stąd większy czas między odcinkami przyrostu. Pod sam koniec wykresu widać, że pierwsze *Harvestery* powróciły z surowcem.

Można się pokusić o dodatkowe sprawdzenie, jak wielki wpływ miało faktycznie podwojenie liczby dostępnych *Harvesterów* w 1:20. Aby to zrobić, należy policzyć ile zasobów zostało dostarczonych do rafinerii między 20-tą, a 80-tą sekundą. W tym celu trzeba uwzględnić koszty fabryki, *Harvestera* i *Scouta* i dodać je do różnicy *RU*.

Jeśli w ciągu 60 sekund rafineria przetworzyła 984 jednostki zasobów, daje nam to **16.4** *RU*/s. Możemy tę liczbę porównać z przyrostem między 1:20 a 1:45. Odczytana z wykresu wartość **26.404** *RU*/s jest rzeczywiście większa, lecz tylko **1.61** raza, a nie **2** razy, jak się spodziewano.

Wynik ten trochę zaskakuje. Ciężko jest nam określić przyczynę, być może jest to konsekwencją zbyt małej przestrzeni między regionem zasobów, a rafinerią. W wyniku dużego zagęszczenia, jakie tworzą cztery *Harvestery*, może im zajmować więcej czasu przejście kursowanie po surowce. Hipoteza ta jednak nie została sprawdzona.

**Wykres 2.** Liczba jednostek zasobów w czasie pierwszych dziesięciu minut gry, próbkowana co sekundę

Przyrost zasobów w całym 10-minutowym odcinku został pokazany na **Wykresie 2**. Wydawać by się mogło, że w dłuższym odstępie czasu wzrost zasobów spełnia zależność kwadratową. Wynika to na pewno ze stałego przyrostu *Harvesterów* (produkowane są co minutę) — każdy z nich zwiększa pochodną *RU* po czasie o zbliżoną wartość. Kwadratowy przyrost nie jest jednak tutaj do końca prawdziwy.

Należy pamiętać, że regiony zasobów wyczerpują się i *Harvestery* muszą wykonywać coraz dalsze kursy. Owe regiony rozmieszczone są równomiernie na planszy. Te, które leżą w tej samej odległości od bazy, zajmują ćwiartkę okręgu. Średnio więc będzie ich tam , gdzie — dystans ćwiartki okręgu do rafinerii. Niech będzie maksymalnym , dla którego nie ma zasobów bliżej bazy. Im większe , tym więcej zasobów znajduje się dopiero na ćwiartce okręgu i tym dłużej zajmuje zebranie ich z tej ćwiartki. Poza tym, *Harvestery* muszą pokonać dystans między regionem a rafinerią. Ponieważ są to dwa niezależne czynniki **liniowe**, okres na zebranie regionów z ćwiartki okręgu rośnie **kwadratowo** od . Stały przyrost *Harvesterów* w czasie bilansuje wzrost tak, że jest ono **liniowe**. Całkowity zaś czas potrzebny do wyczerpania regionów ze środka ćwiartki jest całką okresów dla każdej mniejszej ćwiartki okręgu. Skoro rośnie liniowo od , to jego całka jest kwadratowa, więc funkcja odwrotna, jakiej szukamy jest **pierwiastkiem**. Jeśli dystans rośnie tak, jak pierwiastek całkowitego czasu rozgrywki, to tak samo rośnie czas trwania pojedynczego kursu między regionem zasobów, a rafinerią. Prędkość kursowania, czyli liczba wykonanych przez *Harvester* kursów w jednostce czasu od , to funkcja typu . Mnożąc ją przez wynikające ze stałego tempa produkcji *Harvesterów* otrzymujemy prawdziwą zależność, której nie dało się zobaczyć na wykresie: . Prawdopodobnie 10 minut było po prostu za małym czasem gry, by to pokazać.

Warto zaznaczyć, że powyższe rozważanie przestaje być prawdziwe, gdy osiągnie wartość równą rozmiarowi mapy — ponieważ wtedy fragmenty ćwiartki okręgu znajdą się poza planszą (ćwiartka zacznie szybko maleć). Jednak pod koniec eksperymentu *Harvestery* wciąż zbierały zasoby znajdujące się bliżej bazy — problem ten więc nie wystąpił.

Teraz przejdziemy do krótkiego omówienia wyników pomiaru eksploracji mapy.

**Wykres 3.** Przyrost odkrytych pól mapy w czasie pierwszych trzech minut gry.

Oś pionowa **Wykresu 3** jest opisana potęgami dwójki, ponieważ długość boku planszy to 64, zatem liczba wszystkich pól do zwiedzenia wynosi . Wykres jest podzielony na kilka wyraźnych fragmentów. Przez pierwszych 50 sekund gry rolę skauta wykonuje jeden z *Harvesterów*. Później wyprodukowany zostaje *Scout*, który przejmuje to zadanie. Przez ok. 6 sekund żadne nowe pola nie zostają odkryte, gdyż zwiadowca musi dotrzeć do granicy widoczności. Następnie *Scout* zwiedza teren w stałym tempie aż do ok. 140-tej sekundy gry. Od 2:19 do 2:27 znów nie ma postępów w eksploracji. Jest to moment, gdy zwiadowca wraca po własnych śladach. Rekonesans trwa jeszcze parę sekund i kończy się, gdy stosunek zwiedzonych pól do wszystkich przekroczy 0.9. Widać to na **Wykresie 4**, który (oczywiście) ma identyczny kształt jak **Wykres 3**.

**Wykres 4.** Procentowy stosunek odkrytych pól planszy do wszystkich pól.

Z wykresu widać, że zaprogramowany algorytm zwiedzania okazał się być liniowy. Z policzonej regresji widzimy, że *Harvester* przeprowadza rekonesans w tempie 17.75 pól/s. Szybszy od niego scout *Scout* odkrywa 30.337 pól/s, jest zatem ok. 1.71 raza szybszy. Liczba ta smuci — gdyż jest mniejsza, niż oczekiwano. Ponieważ *Scout* jest 1.6 raza szybszy od *Harvestera* (szybkości ruchu odpowiednio: 4 i 2.5 kratek na sekundę), a jego pole widzenia ma 1.4 raza większy promień (zasięgi widzenia odpowienio: 7 i 5 kratek), spodziewano się znacznie większej wartości. Optymalnie można szacować, że *Scout* powinien być lepszy razy od *Harvestera*. Najwyraźniej sposób wyboru regionów do odkrycia przez jednostkę zwiadowczą jest nie dość dobry i w efekcie *Scout* nie wykorzystuje w pełni swoich możliwości. Wciąż cieszy jednak fakt, że zdarzył się tylko jeden krótki okres bez odkrytych pól — obawiano się, że ścieżka zwiadu częściej będzie przechodzić przez już zwiedzone tereny.

## Badanie skuteczności rekonesansu

W rozdziale 4.2.8 wspomniano o klasie abstrakcyjnej, której potomkowie w funkcji polimorficznej Calculate() definiują sposób sortowania regionów. Jest on ważny, gdyż pozwala zadaniu CoarseReconGoal wybrać najlepszy w danym momencie region do odwiedzenia. Można zauważyć, że funkcja ta powinna brać pod uwagę takie czynniki, jak odległość regionu od bazy (by preferować regiony bliskie własnym strukturom) lub odległość regionu od zwiadowcy wykonującego zadanie (by zwiększyć optymalność zwiadów). Należałoby te parametry jakoś połączyć, by otrzymać wartość mówiącą o kolejności wyboru regionów. Im mniejsza to będzie liczba, tym bardziej powinien się nadawać region.

W rozdziale tym będziemy po kolei proponować implementacje funkcji Calculate(), testować je, wyciągać wnioski o tym, co można poprawić. Wprowadzone poprawki znów będą poddane testom, ich wyniki analizie itd. Proces ten powtórzymy kilkukrotnie aż otrzymamy zadowalający nas algorytm (z którego *notabene* skorzystano przy domyślnej rozgrywce[[46]](#footnote-46)).

### Założenia eksperymentów

Dla każdego z przeprowanych testów *setup* gry wygląda następująco:

* Gracz dysponuje jednym budynkiem *ConstructionYard* oraz jednym *Scoutem*,
* *Scout* zaczyna dokładnie z górnego narożnika, t.j. pola (63, 63),
* *ConstructionYard* ustawiony jest na polach (61, 63) i (62, 63),
* Armia nie posiada żadnych zasobów,
* Paremetry AIBrain zdefiniowano jako:
  + resourceRegionDistance = 2,
  + harvestingImportanceFunction: plik *StandardHarvestingImportance*,[[47]](#footnote-47)
  + reconRegionSize = 8,
  + coarseReconPercentage = 90
  + reconRegionOrderCalculator: zależny od eksperymentu.

Dzięki takim ustawieniom *AI* nie będzie rozpraszać się niepotrzebnymi zadaniami typu konstruowanie budynków, czy zbieranie zasobów. Jednocześnie parametry identyczne z tymi z domyślnej rozgrywki pozwolą jak najbardziej zbliżyć przypadek do rzeczywistego.

### Sortowanie według odległości do bazy

Pierwszą propozycją wyboru kolejności regionów, jest ich odległość od bazy. Motywowane jest to dwoma aspektami, wynikającymi z faktu, że w ten sposób najpierw zostaną odkryte tereny najbliższe bazie. Po pierwsze: *Harvestery* szybciej będą miały w polu widzenia regiony zasobów bliższe rafinerii. Po drugie: warto znać otoczenie bazy na wypadek niespodziewanego ataku wroga.

public class BaseOnlyOrderCalculator : ReconRegionOrderCalculator

{

public override float Calculate(

CoarseReconGoal goal, ReconRegionBatch region)

{

var baseReg = goal.Agent.Knowledge.AllyBase.BaseRegion;

var delta = region.ConvexHull.Center - baseReg.ConvexHull.Center;

var distToBase = delta.magnitude;

return distToBase;

}

}

**Fragment kodu 41.** Klasa BaseOnlyOrderCalculator

Treść funkcji w klasie BaseOnlyOrderCalculator pokazuje, że tak naprawdę kryterium jest dystans między **środkiem** regionu bazy, a **środkiem** regionu do zwiedzenia. Można zauważyć więc, że funkcja zacznie zwracać odrobinę inną wartość, gdy dostawione zostaną nowe budynki. Sytuacja ta jednak nie zajdzie w tym eksperymencie.

Po uruchomieniu symulacji szybko okazało się, że sama odległość od bazy jest bardzo niekorzystnym kryterium. *Scout* wielokrotnie odwiedzał już poznane miejsca i wracał po swoich śladach. W szczytowych przypadkach znalazł się w tym samym punkcie aż 5 razy. Ilustracja 7 doskonale pokazuje błądzenie zwiadowcy. W efekcie ukończył rekonesans dopiero po ok. 5 i pół minuty.



**Ilustracja 28.** algorytm BaseOnlyOrderCalculator.   
Żółta linia: trasa zwiadowcy, niebieska siatka: regiony zwiedzania.   
Skaut dziesiątki razy pokonuje te same trasy.

Na **Wykresie 5** widzimy, że trend nie przypomina kształtem liniowego — bardziej pasuje do niego pierwiastek. Po chwili zastanowienia przestaje to dziwić. *Scout* bardzo często wracał w odkryte obszary, zwłaszcza, gdy znajdował się mniej więcej na środku planszy. Na początku rekonesansu regionów zwiedzonych było znacznie mniej, co widać na wykresie w postaci krótkich fragmentów poziomych. W miarę zagłębiania się w planszę, czas spędzony na jeżdżeniu po diagonali tam i z powrotem znacznie się wydłużał, gdyż wydłużała się sama diagonala. Wykres obrazuje to jako dłuższe odcinki poziome.

Można jeszcze by się pokusić o oszacowanie, ile czasu *Scout* spędził na bezproduktywnym błądzeniu. W tym celu należy policzyć, wszystkie powtórzone wartości (bez pierwszego wystąpienia). Ponieważ czas próbkowania wynosi 1s, liczba próbek będzie równa przybliżonemu czasowi w sekundach. Okazuje się, że jest 142 takich wartości na 332 wszystkich. Oznacza to, że *Scout* przez niemal 43% czasu nie odkrywał żadnych pól i w ten sposób zmarnował 2 minuty i 22 sekundy. Średnie tempo rekonesansu wyniosło ok. 9.98 pól/s. Jest to niemal **dwa razy** mniej, niż tempo zwiedzania przez *Harvester* w domyślnej symulacji.[[48]](#footnote-48) Wydawać by się mogło, że optymalnym czasem na zwiedzenie planszy jest tu 190 sekund (332 – 142), ale późniejsze eksperymenty pokazały, że da się uzyskać znacznie lepszy wynik.

**Wykres 5.** Przyrost odkrytych pól mapy w czasie pierwszych 5 minut 30 sekund gry  
przy zastosowaniu algorytmu BaseOnlyOrderCalculator

### Sortowanie według sumy odległości do bazy i zwiadowcy

Problem poprzedniego sposobu polega na tym, że algorytm bezkrytycznie pobiera regiony euklidesowo najbliższe bazie. Wiadomo, że nad regiony po przekątnej od bazy będą preferowane te, których pozycja różni się od położenia bazy tylko jedną współrzędną. Powoduje to kursowanie *Scouta* od jednego boku mapy do drugiego. Żeby temu zaradzić postanowiliśmy przy określaniu kolejności regionów uwzględnić również ich odległość od zwiadowcy. Naturalnie, trzeba w jakiś sposób połączyć ją z dystansem do bazy. Na początek zdecydowano się na zwykłą sumę tych dwóch liczb. Powinno to penalizować regiony, które choć są najbliższe bazie, znajdują się bardzo daleko od obecnego położenia skauta.

public class BaseSelfSumOrderCalculator : ReconRegionOrderCalculator

{

public override float Calculate(

CoarseReconGoal goal, ReconRegionBatch region)

{

var baseReg = goal.Agent.Knowledge.AllyBase.BaseRegion;

var dToBase = region.ConvexHull.Center - baseReg.ConvexHull.Center;

var distToBase = dToBase.magnitude;

var dToSelf = region.ConvexHull.Center - goal.UnitAgent.Unit.Coords;

var distToSelf = dToSelf.magnitude;

return distToBase + distToSelf;

}

}

**Fragment kodu 42.** Klasa BaseSelfSumOrderCalculator



**Ilustracja 30.** algorytm BaseSelfSumOrderCalculator.   
Żółta linia: trasa zwiadowcy, niebieska siatka: regiony zwiedzania.   
Zwiadowca wyraźnie preferuje chodzenie po koncentrycznych okręgach dookoła bazy.

Wyniki eksperymentu, pokazane na **Wykresie 6**, i **Ilustracji 8** pokazują znaczną poprawę. Zastosowanie tak prostej modyfikacji znacznie zwiększyło tempo wykonywania zwiadu. Nie zaobserwowano również, żeby zwiadowca wracał po swoich śladach. Jego trasa bardziej przypominała koncentryczne ćwierć-okręgi: gdy dojechał do krawędzi mapy, oddalał się od bazy i wracał odkrywając dalej.

Na wykresie wyraźnie widać, że:

* niemal nie występują okresy bez zwiedzania terenu (widać jeden, bardzo krótki moment między 120 a 130 sekundą),
* uzyskane tempo zwiedzania 26.01 pól/s jest ok. 2.6 razy większe, niż tempo pierwszego algorytmu,
* *Scout* zakończył zwiad w 143 sekundzie (nieco ponad 2 minuty), co jest znacznie lepszym wynikiem, niż szacowane 190.

Z ostatniego spostrzeżenia wynika, że błądzenie nie było jedynym problemem poprzedniego algorytmu. Ponieważ możliwy jest wynik większy od 190, to znaczy, że BaseOnlyOrderCalculator musiał w jakiś sposób powodować mniejsze tempo zwiadu, nawet gdy Scout nie wracał po swoich śladach.

**Wykres 6.** Porównanie przyrostu odkrytych pól w czasie dla algorytmów:   
BaseOnlyOrderCalculator (linia niebieska) oraz   
BaseSelfSumOrderCalculator (linia czerwona).

Jedynym zauważonym problemem tego podejścia jest tak naprawdę to, że zwiadowca za płytko zwiedza nieznane regiony i zbliża się niepotrzebnie za bardzo do krawędzi. Przez to połowa jego pola widzenia praktycznie nie jest wykorzystywana. Pokazuje to **Ilustracja 9**. *Scout* ma na tyle duże pole widzenia, że obejmuje nim dwa regiony w każdą stronę. Dlatego powinien odchodzić na większą odległość od ścieżki już przez siebie pokonanej. W późniejszych eksperymentach spróbowaliśmy na to coś zaradzić.



**Ilustracja 32.** Płytki zwiad nieznanych regionów

### Sortowanie według iloczynu odległości do bazy i zwiadowcy

Chociaż uwzględnianie dystansu zwiadowcy do regionu znacznie poprawiło jakość rekonesansu, nie zamierzaliśmy spocząć na laurach. Postanowiliśmy sprawdzić inne metody łączenia odległości od skauta z odległością od bazy. Zamiast więc je dodawać, BaseSelfSumProductOrderCalculator mnoży je przez siebie.

public class BaseSelfProductOrderCalculator : ReconRegionOrderCalculator

{

public override float Calculate(

CoarseReconGoal goal, ReconRegionBatch region)

{

var baseReg = goal.Agent.Knowledge.AllyBase.BaseRegion;

var dToBase = region.ConvexHull.Center - baseReg.ConvexHull.Center;

var distToBase = dToBase.magnitude;

var dToSelf = region.ConvexHull.Center - goal.UnitAgent.Unit.Coords;

var distToSelf = dToSelf.magnitude;

return distToBase \* distToSelf;

}

}

**Fragment kodu 43.** Klasa BaseSelfProductOrderCalculator

W efekcie otrzymaliśmy trasę, która wchodzi nieco głębiej w nieznane regiony, niż poprzednik, lecz jednocześnie za bardzo oddala się od bazy, gdy blisko niej wciąż są niezbadane obszary do których mógłby w efektywny (pod względem rekonesansu) sposób dotrzeć.



**Ilustracja 33.** algorytm BaseSelfProductOrderCalculator.  
Żółta linia: trasa zwiadowcy, niebieska siatka: regiony zwiedzania.   
Skaut za bardzo oddala się od bazy na zbyt wczesnym etapie.

**Wykres 7.** Porównanie przyrostu odkrytych pól dla algorytmów:   
BaseSelfProductOrderCalculator (linia pomarańczowa)  
oraz BaseSelfSumOrderCalculator (linia czerwona).

**Wykres 7** pokazuje, że algorytm z iloczynem wypadł nieznacznie lepiej od algorytmu z sumą. Zwiad zostaje ukończony w 124 sekundzie, a więc o 19 sekund szybciej. Tempo wyniosło 29.22 pól/s, czyli ok. 1.12 raza szybciej niż w poprzednim przypadku, oraz ok. 2.93 razy szybciej od metody uwzględniającej tylko dystans do bazy. Jest to niewątpliwie pewna poprawa, lecz niezadowalająca.

Zbytnie oddalanie się od bazy wynika prawdopodobnie z tego, że gdy różnica między dwoma liczbami (dodatnimi) jest bardzo duża, ich iloczyn przyjmuje niskie wartości — w porównaniu z wartościami gdy różnica jest mała. **Wykres 8** pokazuje funkcję , na dziedzinach (1, 20).

**Wykres 8.** Funkcja iloczynu dwóch zmiennych przyjmuje znacznie  
większe wartości, gdy zmienne niewiele się od siebie różnią.

### Sortowanie według iloczynu oraz odsetka zbadanych pól

Ponieważ algorytm z iloczynem okazał się być lepszy, zostawiliśmy go, natomiast postanowiliśmy sprostać problemowi zbyt małego zagłębiania się w niezbadany teren. Idea rozwiązania jest prosta: przy określeniu kolejności sortowania należy uwzględnić odsetek pól zbadanych w obszarze pola widzenia umieszczonym w centrum regionu. Liczba ta będzie implikować ile nowych pól nie odkryjemy, a zatem, w jakim stopniu zostanie zmarnowany duży zasięg widzenia jednostki zwiadowczej.

Algorytm ten został zaimplementowany wcześniej, niż go zbadano. Przy późniejszych testach okazało się, że, niestety, zawierał on błędy — przez co działał w sposób niezgodny z zamierzeniem. Ów błędna wersja metody została wykorzystana w symulacji przebiegu domyślnej rozgrywki,[[49]](#footnote-49) co musiało rzutować na skuteczność zwiadu. Stąd otrzymany tam niski stosunek szybkości odkrywania *Scouta* do *Harvestera* przestaje dziwić.[[50]](#footnote-50) W tym rozdziale przytaczamy jedynie prawidłowy wariant algorytmu.

public class BaseSelfProductAndTilesExploredOrderCalculator

: ReconRegionOrderCalculator

{

public override float Calculate(

CoarseReconGoal goal, ReconRegionBatch region)

{

var baseReg = goal.Agent.Knowledge.AllyBase.BaseRegion;

var dToBase = region.ConvexHull.Center - baseReg.ConvexHull.Center;

var distToBase = dToBase.magnitude;

var dToSelf = region.ConvexHull.Center - goal.UnitAgent.Unit.Coords;

var distToSelf = dToSelf.magnitude;

var baseSelfProduct = distToBase \* distToSelf;

var uShape = goal.UnitAgent.Unit.Shape;

var radiusStat = goal.UnitAgent.Unit.Stats[StatNames.ViewRange];

if (radiusStat == null) return baseSelfProduct;

var losShape = Globals.LOSShapeDatabase[radiusStat.Value, uShape];

var center = region.ConvexHull.Center.Round();

int x = center.X;

int y = center.Y;

int tilesInShape = 0;

int tilesExploredInShape = 0;

for (int rx = losShape.GetXMin(x), i = 0;

rx <= losShape.GetXMax(x); rx++, i++)

for (int ry = losShape.GetYMin(y), j = 0;

ry <= losShape.GetYMax(y); ry++, j++)

{

if (!losShape[i, j]) continue;

tilesInShape++;

if (!Globals.Map.IsInBounds(rx, ry)) continue;

var vis = goal.Agent.Army.VisibilityTable[rx, ry];

if (vis != Visibility.Unknown)

tilesExploredInShape++;

}

var explorationRatio = (float)tilesExploredInShape / tilesInShape;

float minVal = 0.1f;

float power = 0.78f;

var factor = minVal + (explorationRatio \* (1 - minVal));

return baseSelfProduct \* Mathf.Pow(factor, power);

}

}

**Fragment kodu 44.** Klasa BaseSelfProductAndTilesExploredOrderCalculator

Klasa BaseSelfProductAndTilesExploredOrderCalculator jest niewątpliwie bardziej złożona od poprzednio przytaczanych. Pierwszych sześć linijek metody Calculate() wykonuje dokładnie to samo, co cała metoda Calculate() w klasie BaseSelfProductOrderCalculator. Na tym jednak podobieństwa się kończą. Aby móc policzyć ile pól zostanie odkryte, jeśli zwiadowca znajdzie się w centrum regionu, algorytm najpierw pobiera kształt jednostki (zawsze domyślny: 1 kratka 1x1) oraz wartość jej statystyki zasięgu widzenia (promień okręgu). Te dwie rzeczy są potrzebne, by uzyskać kształt pola widzenia z Globals.LOSShapeDatabase.

Tworzone są zmienne-akumulatory tilesInShape i tilesExploredInShape i rozpoczyna się podwójna pętla, która przykłada szablon pola widzenia do planszy i liczy pola widoczne. Licznik tilesInShape zwiększany jest zawsze, jeśli pole znajduje się w szablonie. tilesExploredInShape natomiast inkrementuje się, gdy kratka jest wewnątrz planszy i nie jest nieznana. Po zakończeniu pętli liczby dzielone są przez siebie by uzyskać odsetek pól zwiedzonych wewnątrz pola widzenia.

Następnie tworzone są dwa parametry. Zmienna minVal skaluje wartość stosunku pól zwiedzonych do przedziału zamkniętego (minVal, 1). Powód tego zabiegu jest bardzo ważny. Dla większości regionów we wczesnym etapie gry odsetek pól zwiedzonych osiągał wartość 0. Po pomnożeniu tego czynnika przez iloczyn wszystkie regiony całkowicie nieodkryte przestawały uzwględniać odległość od bazy i zwiadowcy. W testach powodowało to, że *Scout* poruszał się bardzo chaotycznie. Żeby za bardzo nie zniekształcać przedziału, przyjęto minVal równe 0.1. Poza tym przeskalowanie jest przekształceniem liniowym i nie powinno mieć wpływu na wzajemne zależności między wyliczonymi dla różnych regionów kolejnościami sortowania.

Druga zmienna jest ważniejsza i została wprowadzona po pierwszych testach. Początkowo iloczyn mnożony był wprost factor. Wpływ tego czynnika okazał się jednak zbyt wysoki. Zwiadowca owszem wchodził znacznie głębiej w niezbadany teren, ale pomijał mnóstwo obszarów, często blisko bazy. Tak wysoka zgrubność rekonesansu nie tylko skutkowała nieodkrytymi połaciami terenu w bezpośredniej bliskości własnych struktur. Później rzutowało to na czas całego zwiadu, gdyż *Scout* musiał dużo wędrować by dokończyć zadanie. **Ilustracja 11** pokazuje zrzut ekranu minimapy po pierwszych testach, na których widać ten problem.



**Ilustracja 34.** Zwiad przy czystym mnożeniu przez factor  
jest zdecydowanie zbyt niedokładny

Widać zatem, że należy zmodyfikować siłę czynnika. Pomnożenie przez liczbę jest przekształceniem liniowym i nic nie zmieni. Zmienna factor musi mieć różny wpływ dla różnych stopni zwiedzenia regionu. Gdyby miała stałą wartość 1, to oczywiście, algorytm ten byłby identyczny z samym iloczynem. Jeśli byłaby równa zawsze 0, wtedy otrzymalibyśmy zachowanie losowe. Można zauważyć, że na przedziale (0, 1) funkcja potęgowa ma następujące własności (które można zaobserwować na **Wykresie 9**):

* Wartości zachowują przedział (0, 1),
* Dla jest funkcją liniową,
* Dla jest funkcją wklęsłą, im większe , tym większa wklęsłość,
* Dla dąży do funkcji stałej , bez ,
* Dla jest funkcją wypukłą, im większe , tym większa wypukłość,
* Dla dąży do funkcji stałej , bez ,

Jeśli więc potraktujemy factor jako argument funkcji potęgowej, to kontrolując parametr (wyrażony przez zmienną power), będziemy mogli dostosować zachowanie dla różnych odsetków odkrycia terenu. Przykładowo zastosowanie pierwiastka kwadratowego, czyli powinno spowodować, że regiony odkryte w bardzo niewielkim stopniu będą zyskiwały przewagę nad pozostałymi.

**Wykres 9.** Funkcja potęgowa na przedziale (0, 1) dla różnych wartości parametru .

Ostatnia linijka metody Calculate() uwzględnia funkcję potęgową. W pokazanym kodzie zmienna power ustawiona jest na 0.78. Jest to eksperymentalnie znaleziona wartość, dla której algorytm osiągał najlepsze efekty. Na samym początku jednak przetestowano wartość 0.5. Zaobserwowano poprawę, jeśli chodzi o głębszą eksplorację terenów niezwiedzonych, jednak była ona niezadowalająca, co pokazuje **Ilustracja 12**.



**Ilustracja 35.** Czerwony obszar pokazuje jakiej części pola widzenia *Scouta* nie wykorzystano przy .

Po wielu testach zdecydowano się na wspomnianą liczbę 0.78. Ilustracja 13 pokazuje przebieg trasy zwiadowcy dla tej wartości, zaś na **Ilustracji 14** widać problem, jaki trasa ta generuje. Sortowanie przy takim parametrze pozostawia za sobą spore połacie niezwiedzonego terenu — na ilustracji widzimy zaczerniony obszar na środku planszy.



**Ilustracja 36.** zastosowany algorytm sortowania regionów stworzony w klasie BaseSelfProductAndTilesExploredOrderCalculator znacznie skraca trasę...



**Ilustracja 37.** ...lecz zostawia po drodze dużo nieodkrytego terenu.

Wykonaliśmy pomiary przyrostu odkrytych pól w czasie, których wyniki pokazuje **Wykres 10**. Skuteczność rozwiązania, jeśli chodzi o szybkość zwiadu, przechodzi wszelkie oczekiwania. Zwiadowca ani razu nie wracał po swoich śladach. Każdy jego ruch był istotny. W efekcie skończył rekonesans już po 103 sekundach. Jego średnia prędkość odkrywania wyniosła 37.531 pól/s, co jest ok. 1.27 raza lepszym wynikiem, niż prędkość przy samym iloczynie. Jednocześnie porównując ten wynik z metodą opartą na samej odległości do bazy uzyskujemy poprawę **niemal czterokrotną** (3.76). Jest to doskonały rezultat, którego nie spodziewaliśmy się uzyskać.

**Wykres 10**. Porównanie przyrostu odkrytych pól dla algorytmów: BaseSelfProductOrderCalculator (linia pomarańczowa), oraz BaseSelfProductAndTilesExploredOrderCalculator (linia granatowa).

Na koniec wypadało zadośćuczynić błędowi, który wkradł się w porównanie szybkości *Harvestera* i *Scouta* z rodziału 5.1. Ustawiliśmy więc na szybko test z identycznymi warunkami początkowymi, jak tutaj, jedynie jednostkę *Scout* zastąpiliśmy właśnie *Harvesterem*. Wyniki tej krótkiej symulacji pokazaliśmy na **Wykresie 11**.

**Wykres 11**. Porównanie przyrostu odkrytych pól dla algorytmu BaseSelfProductOrderCalculator wykonywanego przez *Scouta* (linia granatowa) i *Harvestera* (linia czerwona).

Przypomnijmy, że w trakcie domyślnej rozgrywki *Scout* zwiedzał w tempie 30.337 pól/s, a *Harvester* w tempie 17.75 pól/s. Wzajemny ich stosunek wynosił ok. 1.71, gdy przewidywano 2.24. Odczytując czynniki liniowe regresji z wykresu 11 widzimy, że po poprawieniu błedu wynik *Scouta* poprawił się ok. 1.24 raza, wynik 1.13 raza, natomiast nowy wzajemny stosunek ma wartość 1.88. Jest bliższy oczekiwanemu, choć nieznacznie.

Wnioskujemy z tego, że istnieje sposób by algorytm ulepszyć. Nie jest to już jednak priorytetem tej pracy, gdyż otrzymany wynik okazał się znacznie więcej niż zadowalający. Badanie skuteczności rekonesansu można więc uznać za zakończone.

# Zakończenie

Celem pracy było zaprogramowanie prototypu gry z gatunku *RTS*, oraz zaimplementowanie sztucznej inteligencji sterującej jedną ze stron konfliktu w rozgrywce. Następnie należało przebadać jej zachowanie oraz skuteczność i finalnie wyciągnąć z tego wnioski.

Pierwsza część tematu została zrealizowana w znacznie większym stopniu, niż planowaliśmy. Chociaż prototyp ***MechWars*** miał mieć proste reguły (i, w porównaniu z prawdziwymi *RTS*-ami, miał), to jego stworzenie zajęło 90% czasu poświęconego na całe przedsięwzięcie. Wynika to z tego, że gra *RTS* jest niezwykle złożonym projektem informatycznym, składającym się z mnóstwa różnych komponentów. Nasz projekt nie bez przyczyny jest jedynie prototypem, gdyż do bycia kompletną grą *RTS* na miarę dzisiejszych czasów brakuje mu takich elementów, jak: kampania jednoosobowa, rozgrywki w trybie multiplayer, złożone plansze, kompletne menu z opcjami zapisywania, wczytywania i zmiany wielu ustawień, grupowanie jednostek, fabuła, muzyka, dźwięki. Mimo to, realizuje on większość pozostałych aspektów rozgrywki, takich, jak gracze, armie oraz zasoby przez nie posiadane oraz plansza gry okryta mgłą wojny. Znajdują się w nim też liczne elementy mniejszego kalibru: jednostki, budynki, ich rozkazy i działania (np. poruszanie się, atakowanie, zbieranie zasobów, produkcja innych budynków i jednostek), statystyki liczbowe i bonusy, odkrywanie technologii oraz zależności technologicznych. Od strony algorytmicznej pojawiają się problemy poszukiwania ścieżek (*A\**), oraz podziału i organizacji przestrzeni (*quad-tree*, tablice widzialności). Prototyp wymagał odpowiedniego sterowania: mysz w nim służy do zaznaczania obiektów i wydawania rozkazów, a klawisze shift, alt, control modyfikują domyślne zachowanie; za pomocą klawiatury da się wywoływać komendy skrótami (tzw. *hotkey*). Operowanie kamerą pozwala na podgląd dowolnego miejsca planszy zarówno w sytuacji taktycznej (w małej skali) jak i strategicznej (z daleka). Prototyp ***MechWars*** posiada czytelny interfejs graficzny wyświetlający licznik zasobów, stan całej planszy w postaci minimapy, a także przyciski do wydawania rozkazów. Menu główne na osobnej scenie pozwala na wybór trybu gry i zmianę ustawień. Do prototypu utworzono też samą treść: temat przewodni, rodzaje jednostek, budynków, technologii i zależności oraz prostą planszę. Dochodzi też cała strona graficzna: komplet modeli jednostek, budynków, zasobów i model terenu, a także płaskie grafiki takie jak tekstury oraz wygląd interfejsu graficznego. Wreszcie w prototypie zaimplementowano prosty mechanizm *AI* realizujący zachowania zwiedzania planszy i zbierania zasobów. Powyższe wyliczenie w żadnym wypadku nie jest wyczerpujące, w wielu wymienionych aspektach tkwią nieopisane szczegóły.

Przy implementacji sztucznej inteligencji posiłkowano się licznymi źródłami. Głównym zainspirowanym mechanizmem jaki pomógł zorganizować działania *AI* był system wieloagentowy. Stworzono agenty sterujące pojedynczymi jednostkami, a także kilka agentów abstrakcyjnych, które nie reprezentowały żadnych bytów, a kontrolowały poczynania jednego z aspektów całego „procesu myślowego” sztucznej inteligencji. Były to osobne agenty od gromadzenia wiedzy, przeprowadzania zwiadów, konstrukcji budynków, produkcji jednostek oraz zbierania zasobów. Miały one ustalone zachowanie, podczas gdy agenty jednostek wykonywały wymienne cele. *AI* musiała w jakiś sposób uporządkować sobie informacje, dlatego stworzono pomocnicze mechanizmy regionów, przeznaczeń i metod tworzenia. Wiedzę dla sztucznej inteligencja filtrowano przy pomocy mgły wojny, by uzyskać równe szanse między oboma armiami biorącymi udział w bitwie.

Stopień zaprogramowania sztucznej inteligencji pozwolił na realizację drugiej części tematu. Wykonano badania rozgrywki imitującej zwykły scenariusz dla dwóch stron konfliktu, z ustawionymi parametrami i warunkami początkowymi. Obserwacja przebiegu wykazała, że *AI* zachowywała się niemal tak, jak zamierzono. Błędem w zaprojektowaniu zachowań natomiast było stosowanie czynnika zgrubności 90% w odniesieniu do całej planszy. Przeanalizowano wygenerowane dane liczbowe. Porównano szybkości zbierania zasobów przez różne liczby *Harvesterów* i wysunięto spróbowano wytłumaczyć wynik. Przeprowadzono teoretyczne rozważanie o przyroście zasobów w czasie. Na koniec porównano szybkości zwiedzania terenu przez jednostki *Harvester* i *Scout*. Błąd w programie, z którego się wytłumaczono, zaburzył ten ostatni pomiar.

Zrekompensowano go w serii eksperymentów dotyczącej rekonesansu. Zmieniono tam warunki początkowe na tak, by skupić się na samym zwiedzaniu i zniwelować wpływ innych aspektów gry na wynik pomiarów. Obranym celem było znalezienie jak najskuteczniejszego algorytmu wyznaczania regionów do zwiedzania, zarówno pod kątem szybkości, jak i dokładności rekonesansu. Okazało się, że pierwszeństwo regionów wyrażone jedynie przez odległość od bazy jest fatalnym kryterium — *Scout* marnował czas wracając po własnych śladach. Po kilku próbach otrzymano ostateczny algorytm uwzględniający iloczyn dystansu regionu do bazy i do zwiadowcy mnożony przez współczynnik wynikający ze stopnia wykorzystania zasięgu widzenia. Dzięki temu rozwiązano problem płytkiego wchodzenia *Scouta* w niezbadany teren. Wyniki najlepszej z testowanych metod były powyżej oczekiwań. Na koniec przyszedł czas na zadośćuczenienie przekłamaniu z pierwszych badań. Porównano więc na nowo wyniki *Scouta* i *Harvestera* przy warunkach początkowych z drugiego eksperymentu. Ich wynik był bliższy oczekiwaniom od poprzedniego.

# Streszczenie

Celem pracy magisterskiej było stworzenie prototypu gry *RTS*, zaimplementowanie w nim sztucznej inteligencji i zbadanie jej skuteczności. W pierwszym rozdziale opisano założenia prototypu, mechanikę rozgrywki oraz rodzaje elementów gry i interfejs użytkownika. Drugi rozdział stanowi ogólny przegląd implementacji większości podsystemów prototypu, takich jak plansza, elementy mapy, mgła wojny, sterowanie i interfejs gracza. Trzeci rozdział poświęcono grafice: utworzonym modelom oraz zaprogramowaniu kursora myszy i efektów cząsteczkowych. Czwarty rozdział traktuje o teoretycznych rozwiązaniach problemu sztucznej inteligencji w grze *RTS*, sposobie w jaki zaimplementowano ją w prototypie i metodach z jakich skorzystano. W piątym rozdziale wykonano dwa rodzaje badań stworzonej przez nas sztucznej inteligencji: przetestowano domyślną rozgrywkę oraz znaleziono optymalny algorytm wyznaczania trasy rekonesansu. Efektem końcowym zarówno pracy jak i projektu magisterskiego jest istniejący prototyp gry *RTS* z ograniczoną grywalnością oraz wykonane na jego podstawie badania sztucznej inteligencji.

# Bibliografia

# Aneks/Spis ilustracji

1. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-1)
2. web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real\_time, tłumaczenie własne, 11.18.2015r [↑](#footnote-ref-2)
3. web.archive.org/web/20110628235716/http://www.gamespot.com/gamespot/features/all/real\_time/  
   p2\_02.html, tłumaczenie własne, 11.18.2015r [↑](#footnote-ref-3)
4. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-4)
5. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-5)
6. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-6)
7. http://pwp.detritus.net/in/1997/rf.html, tłumaczenie własne, 12.28.2015r [↑](#footnote-ref-7)
8. https://www.rand.org/content/dam/rand/pubs/papers/2008/P7511.pdf, tłumaczenie własne, 12.28.2015r [↑](#footnote-ref-8)
9. Szczegółowo sztuczna inteligencja została opisana w rozdziale 4: Zaprogramowanie sztucznej inteligencji, str. X [↑](#footnote-ref-9)
10. Klasę QuadTree opisano w rozdziale 3.3.2: Klasy drzewa czwórkowego, str. X [↑](#footnote-ref-10)
11. *http://www.gamedev.net/topic/489276-generating-line-of-sight-in-tile-based-rts/* [↑](#footnote-ref-11)
12. Funkcja ta została opisana szerzej w rozdziale 3.7: Podsystem *GUI*, str. X [↑](#footnote-ref-12)
13. Przykład pliku *XML* ze statystykami pokazany jest w rozdziale 3.4.2: Statystyki, str. X [↑](#footnote-ref-13)
14. Patrz: Fragment kodu 9 [↑](#footnote-ref-14)
15. Nie: „konfigurowane”. To, czy są konfigurowalne, czy nie, zależy od tego, czy klasa potomna do OrderAction zawiera jakieś publiczne pola. [↑](#footnote-ref-15)
16. Nie: „publiczną metodę”. Są dwie metody o tej samej nazwie i różnych parametrach. [↑](#footnote-ref-16)
17. Często są używane, by uchronić się przed wyjątkiem IndexOutOfBoundsException [↑](#footnote-ref-17)
18. *Perełki programowania gier. Vademecum profesjonalisty, Tom 2*, 2002, Matt Pritchard, *Rozdział 3.5: Wysokowydajny system widoczności i wyszukiwania oparty na siatkach*, str. 317 [↑](#footnote-ref-18)
19. W książce zwana jest ona „mapą widoczności”. [↑](#footnote-ref-19)
20. Książka określa je jako „miraże” [↑](#footnote-ref-20)
21. Więcej na temat klasy MinimapFog napisano w rozdziale 2.8.1: Wygląd minimapy, str X [↑](#footnote-ref-21)
22. *http://en.wikipedia.org/wiki/A\*\_search\_algorithm* [↑](#footnote-ref-22)
23. E. Adams, *Fundamentals of Game Design*, New Riders, 2014, s. 284, tłumaczenie własne [↑](#footnote-ref-23)
24. *http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf* [↑](#footnote-ref-24)
25. *Perełki programowania gier. Vademecum profesjonalisty, Tom 3*, 2003, Daniel Higgins, *Rozdział 3.4: Analiza terenu w grach RTS – ukryta siła*, str. 321 [↑](#footnote-ref-25)
26. Mat Buckland, *Programming Game AI by Example*, *Chapter 2: State-Driven Agent Design*, 2005, str. 43 [↑](#footnote-ref-26)
27. Mat Buckland, *Programming Game AI by Example, Chapter 9: Goal-Driven Agent Behavior*, 2005, str.379 [↑](#footnote-ref-27)
28. Klasa Message, opisana w rodziale 4.2.3 [↑](#footnote-ref-28)
29. Motywem dla tej funkcjonalności jest oczywiście wydajność. [↑](#footnote-ref-29)
30. Rozdział 4.2.8. Zadanie zgrubnego rekonesansu, str. X [↑](#footnote-ref-30)
31. Rozdział 4.2.11. Agent zbierający zasoby, str. X [↑](#footnote-ref-31)
32. Patrz: rozdział 4.2.6. Agent gromadzący wiedzę, str. X [↑](#footnote-ref-32)
33. Podobna sytuacja mogłaby wystąpić pomiędzy dowolnymi dwoma agentami, więc mechanizm jak najbardziej nadaje się do rozszerzenia na więcej przypadków. [↑](#footnote-ref-33)
34. Opisane w rozdziale 4.2.8. Zadanie zgrubnego rekonesansu, str X [↑](#footnote-ref-34)
35. Może tak być, gdy właśnie zmniejszono priorytet zadania, więc spadła liczba potrzebnych jednostek zwiadowczych. [↑](#footnote-ref-35)
36. Rozdział 4.2.7. Agent odpowiedzialny za zwiady, str. X [↑](#footnote-ref-36)
37. Rozdział 5.2 Badania skuteczności rekonesansu [↑](#footnote-ref-37)
38. Koszt rozpoczęcia konstrukcji skonfigurowano w prototypie jako 10% pełnego kosztu budynku. [↑](#footnote-ref-38)
39. Sposób funkcjonowania tej metody nie został omówiony. [↑](#footnote-ref-39)
40. Rozdział 4.2.9. Agent rozbudowujący bazę, str. X [↑](#footnote-ref-40)
41. Rozdział 4.2.7 Agent odpowiedzialny za zwiady, str. X [↑](#footnote-ref-41)
42. Oczywiście stwierdzenie kilka zdań wcześniej: „zasób wybrany przez ten rozkaz” jest jak najbardziej prawidłowe; kiedy zasób zostanie zebrany, rozkaz już samodzielnie zajmuje się wyszukaniem następnego — najbliższego w zasięgu widzenia *Harvestera*. [↑](#footnote-ref-42)
43. Patrz: **Ilustracja 1**. [↑](#footnote-ref-43)
44. Treść pliku w aneksie, str. X [↑](#footnote-ref-44)
45. (stosunek czynników liniowych regresji wynosi ~4.81) [↑](#footnote-ref-45)
46. Rozdział 5.1 Badanie przebiegu domyślnej rozgrywki [↑](#footnote-ref-46)
47. Treść pliku w aneksie, str. X [↑](#footnote-ref-47)
48. Rozdział 5.1.3. Analiza przyrostu zasobów i odkrytego terenu [↑](#footnote-ref-48)
49. Rozdział 5.1 [↑](#footnote-ref-49)
50. Rozdział 5.1.3 [↑](#footnote-ref-50)