Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Sławomir Tomaszewski

Numer albumu: 1118787

Natalia Zmysłowska

Numer albumu: 1117040

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

KRAKÓW 2016

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

Wstęp

Tematem niniejszej pracy magisterskiej jest przeprowadzenie badań nad zachowaniem sztucznej inteligencji (ang. *artificial intelligence*, w skrócie *AI*). Zostaną one przedstawione na przykładzie stworzonego przez nas specjalnie do tego celu prototypu gry strategicznej czasu rzeczywistego (ang. *real-time strategy*, w skrócie *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 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 uruchomiona i zanalizowana rozgrywka dla domyślnych ustawień paramterów i planszy gry. Następnie przeprowadzimy szereg owocnych eksperymentów związanych z zadaniem rekonesansu.

Należy podkreślić, że praca była pisana przez dwie osoby. Aby można było rozróżnić, kto przyczynił się do stworzenia określonych części projektu, w **Aneksie** umieszczono informację o podziale prac.[1]

Ze względu na obszerność pracy, szczególnej uwadze polecamy rozdziały:

- Od 1.2 do 1.5, dotyczące zawartości tematycznej gry,
- 2.3, w tym 2.3.1, 2.3.4 oraz 2.3.5 dotyczące elementów mapy i rozkazów,
- 2.4 i 2.5 skupiające się na mapie i mgle wojny,
- 3 (cały), opisujący wygląd prototypu i sposób jego zaprogramowania,
- 4.2, w tym: 4.2.2 i od 4.2.7 do 4.2.12, dotyczące systemu wieloagentowego,
- 5 (cały), dotyczący badań nad zaimplementowaną sztuczną inteligencją.

[1] Aneks, A. Podział prac nad projektem, str. 136

Wprowadzenie

„*RTS*, czyli *real-time strategy* (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)”.[2] Jej główna cecha polega na tym, iż rozgrywka nie jest podzielona na tury lub kolejki, lecz dzieje się w czasie rzeczywistym.[3]

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 zależność struktur od siebie, a także ideę osobnych stron konfliktu, różniących się pod względem dostępnych jednostek i broni. [4] Cechą wspólną gier strategicznych są warunki zwycięstwa, czyli pokonanie przeciwnika poprzez zniszczenie jego kluczowych struktur lub pozbawienie go zasobów, dzięki czemu nie będzie miał on możliwości odbudowy swoich jednostek.[5]

Główną mechaniką gry jest wybór ścieżki rozwoju, którą gracz zamierza podążać. Do wyboru zazwyczaj ma dwie podstawowe drogi: ekonomiczną oraz militarną, które później może krzyżować. Pierwsza z nich sprawia, że na początku gracz ma bardzo słabe jednostki zbrojne (lub nie ma ich wcale), gdyż skupiamy się na rozwoju związanym z pozyskiwaniem surowców budulcowych. W ten sposób jednak 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 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, gracz musi również zrozumieć konstrukcję mapy oraz rozmieszczenie poszczególnych surowców, w stopniu wystarczającym, by jak najbardziej optymalnie jego zdaniem rozlokować budynki oraz jednostki.[6]

---

[2] http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 30.11.2015r [1]
[3] http://web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real_time/index.html, 18.11.2015r [2]
[4] http://web.archive.org/web/20110628235716/http://www.gamespot.com/gamespot/features/all/real_time/p2_02.html, 18.11.2015r [3]
[5] http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 30.11.2015r [1]
[6] http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 30.11.2015r [1]

Sztuczna inteligencja w grach typu *RTS* ma między innymi za zadanie postępować jak prawdziwy gracz. W skład tego wchodzą takie zachowania jak:[7]

- 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 działań do wybranego poziomu trudności rozgrywki,
- wysyłanie jednostek zwiadowczych w celu zebrania informacji o poziomie zaawansowania gracza,
- przewidywanie strategii przeciwnika.

Głównym problemem dla sztucznej inteligencji, jaki pojawia się podczas rozgrywek, jest podejmowanie decyzji opartych o zbyt małą liczbę informacji, a co za tym idzie optymalne 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 zatem, 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* posiadającej 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.

[7] http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 30.11.2015r [1]

1. Reguły i założenia prototypu

Celem niniejszej pracy magisterskiej było zaprogramowanie prototypu gry, stworzenie sztucznej inteligencji w ograniczonym stopniu sterującej graczem i przebadanie jej skuteczności. Prototyp otrzymał roboczą nazwę **MechWars**. Postanowiliśmy oprzeć go o proste zasady, gdyż nie chcemy skupiać się zbyt mocno na rozbudowie mechaniki rozgrywki. Mimo to nie mogły one też być zbyt proste, gdyż za mocno ograniczyłoby to możliwości oraz decyzje sztucznej inteligencji. Prototyp został umieszczony w konkretnej konwencji określającej rodzaje jednostek, budynków i badań technologicznych oraz surowiec, z którego zbudowane są elementy świata.

1.1. Konwencja

Do stworzenia prototypu MechWars skorzystaliśmy z silnika gier *Unity*. Postanowiliśmy, że modele budynków, jednostek i otoczenia będą trójwymiarowe, a perspektywiczna kamera zostanie ustawiona pod kątem 45° do planszy. Teren rozgrywki został zaplanowany jako płaska i symetryczna względem środka mapa po to, by obie strony miały równe szanse przy rozpoczęciu partii.

Przy określaniu konwencji prototypu zainspirowaliśmy się wizją świata po apokalipsie (tzw. *postapo*), w którym jedynymi ocalałymi są maszyny. Dodatkowo wygląd otoczenia odwzorowywać miał styl retrofuturystyczny, czyli wyobrażenie przyszłości zazwyczaj stylizowane na erę wiktoriańską. W związku z tym ustaliliśmy temat przewodni naszego prototypu jako walki maszyn na wielkim złomowisku. Armie w takim świecie mogą wznosić budynki produkcyjne, a także konstruować jednostki zmechanizowane (tzw. *mechy*) i pojazdy gąsienicowe. Głównym zasobem jest złom rozsypany po powierzchni. Wraki jednostek i zniszczone budynki mogą zostać zebrane jako dodatkowe źródło surowców.

Budynki oraz jednostki, które zaplanowaliśmy do prototypu miały być trójwymiarowymi modelami stworzonymi w całości na potrzeby niniejszej pracy. Uzgodniliśmy, że dym z kominów niektórych budynków zaprojektujemy dzięki systemowi cząsteczkowemu w *Unity*.

1.2. Podstawowa mechanika

W podstawowej mechanice przewidzieliśmy jeden funkcjonalny tryb gry — gracz ludzki przeciwko sztucznej inteligencji. Dodatkowo, jeśli udałoby nam się zaprogramować dość zachowań *AI*, zostałby wprowadzony drugi tryb gry: rozgrywka pomiędzy dwoma sztucznymi inteligencjami. Warunek ten jednak nie został spełniony, więc drugiej opcji nie wprowadziliśmy. Do żadnego z tych trybów gry nie zamierzaliśmy tworzyć ani fabuły, ani kampanii, ponieważ chcieliśmy skupić się na programowaniu sztucznej inteligencji.

Planszę (zwaną również mapą), na której toczyć się miała rozgrywka, podzieliliśmy na kwadratowe pola. Ustaliliśmy, że elementami mapy będą jednostki, budynki, zasoby i przeszkody. Jednostki i zasoby zajmować miały dokładnie jedno pole mapy, budynki natomiast mogły znajdować się na kilku z nich (według nadanego im kształtu). Kratki każdego elementu mapy miały być zawsze określone — z wyjątkiem jednostki, ta mogła w trakcie ruchu z jednego pola na sąsiednie tymczasowo przebywać pomiędzy nimi.

Postanowiliśmy zaimplementować podstawowe zachowania jednostek kierowanych zarówno przez fizycznego gracza, jak i przez *AI*. Zaliczać się do nich miały:

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

Jednostki i budynki posiadać miały własną, ograniczoną sztuczną inteligencję zrealizowaną w postaci rozkazów wydawanych im przez gracza. Rozkazy te wymyśliliśmy, jako pewnego rodzaju procesy wykonujące szereg czynności. W ich skład miały wchodzić nie tylko pojedyncze komendy przemieszczania się, lecz także bardziej złożone instrukcje typu automatyczne atakowanie wrogów w zasięgu, albo eskorta grupy własnych jednostek. Ostatecznie zaplanowaliśmy następujące rodzaje rozkazów:

- *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),[8]
- *Move* — jednostka przemieszcza się do określonej pozycji docelowej,
- *FollowAttack* — jednostka atakuje wroga i goni go, dopóki nie zniszczy,
- *StandAttack* — jednostka atakuje wroga w miejscu i przerywa atak, gdy wróg wyjdzie z zasięgu,

[8] Wszystkie zaplanowane jednostki miały potrafić się obracać, nie wszystkie atakować, natomiast jeden rodzaj budynku — wieżyczka obronnna — miał potrafić zmieniać rotację głowicy i przeprowadzać atak.

- *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 i przechodzi do rozkazu *Idle*,
- *UnitProduction* — budynek produkuje określony rodzaj jednostki,
- *BuildingConstruction* — budynek konstruuje inny, określony budynek we wzkazanym miejscu,
- *TechnologyDevelopment* — budynek opracowuje określoną technologię.

Idle zaprojektowaliśmy jako rokaz domyślny — jeśli jednostka lub budynek nie miałaby żadnych wydanych rozkazów, wykonywałby się właśnie rozkaz *Idle*. Sprecyzowaliśmy różnice pomiędzy jednostkomi i budynkami co do możliwych rozkazów. Przykładowo ustaliliśmy, że zwiadowca nie będzie mógł atakować ani odkrywać technologii, a wieżyczka obronna poruszać się.

Zamierzaliśmy zastosować mechanizm tzw. mgły wojny (ang. *fog of war*). 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 sił wroga pozostaje nieznana. „Mgła wojny jest odpowiednikiem poziomu niepewności gracza co do sytuacji, w której znajduje się przeciwnik”.[9] W naszym prototypie dzięki jednostkom takim jak zwiadowca możliwe ma być jednak przeprowadzanie rekonesansu: odkrycie fragmentu terytorium i zebranie informacji na temat stanu zaawansowania przeciwnika oraz położenia surowców.

1.3. Typy jednostek

W związku z tym, iż chcieliśmy zachować optymalny balans rozgrywki, stworzyliśmy jedynie jednostki naziemne. Różnią się one między sobą nie tylko wyglądem, ale również statystykami, takimi jak:

- prędkość obrotu i poruszania się,
- siła i szybkość ataku,
- zasięg ataku i widzenia,
- wytrzymałość.

[9] https://www.rand.org/content/dam/rand/pubs/papers/2008/P7511.pdf, 12.28.2015r [4]

Dwoma podstawowymi nieuzbrojonymi jednostkami są jednośladowy zwiadowca (*Scout*) oraz zbieracz zasobów (*Harvester*). Zwiadowca charakteryzuje się dużym zasięgiem widzenia, i szybkością, ale jednocześnie nie ma możliwości ataku i bardzo łatwo go zniszczyć. Jego zadaniem jest zatem rekonensans: sprawdzenie terenu, odnalezienie zasobów, a także szpiegowanie przeciwnika i monitorowanie jego postępów. *Harvester* natomiast to jednostka, która będzie potrafi wydobywać zasoby (złom) i transportować je do rafinerii. On również nie potrafi walczyć.

Najprostszymi typami oddziałów militarnych są dwa rodzaje zmechanizowanych jednostek kroczących — tzw. *mechów*. *Mech* wyposażony w karabiny maszynowymi (*MechMachinegun*) ostrzeliwuje z daleka wrogów, ale nie zadaje wysokich obrażeń. Wytrzymalszy i silniejszy od niego jest *mech* z miotaczem ognia (*MechFlamethrower*), jednka musi on podejść bezpośrednio do wroga. Obie jednostki poruszają się dość szybko, jednak stosunkowo łatwo je zniszczyć i nie zadają zbyt wysokich obrażeń innym jednostkom i budynkom.

Oprócz powyżej opisanych formacji zaprojektowaliśmy jeszcze dwa rodzaje uzbrojonych pojazdów opancerzonych. Jednym z nich jest masywny czołg (*Tank*): powolna, ale wytrzymała maszyna z dość dużym zasięgiem ataku i siłą ognia. Drugi natomiast to mobilna wyrzutnia rakiet (*RocketLauncher*), cechująca się lżejszym pancerzem, zrekompensowanym przez olbrzymim zasięgiem i siłę rażenia. Jej główną wada polega na tym, że z bliska jej rakiety są niezwykle niecelne — do tego stopnia, że mogą przypadkiem trafić we własne jednostki i struktury. Oba typy pojazdów posiadają na tyle duże obrażenia, by w bardzo szybkim czasie niszczyć struktury i całe oddziały wroga.

1.4. Typy budynków

Budynki służą konstrukcji nowych struktur, produkcji jednostek, opracowywaniu technologii, składowaniu zasobów oraz obronie bazy. Należy nadmienić, że użyte tu słowa dotyczące czynności — „konstrukcja”, „produkcja” — zawsze tyczą się odpowiednio: budynków i jednostek, nigdy na odwrót. Jest to jedna z przyjętych w projekcie konwencji.

Rodzaje planowanych przez nas budynków to:

- *ConstructionYard* — warsztat konstrukcyjny, budynek służący do konstrukcji nowych struktur; nie można go wybudować, ale każda armia posiada jeden jego egzemplarz na początku rozgrywki,
- *Refinery* — rafineria złomu, będąca punktem, do którego *Harvestery* znoszą zasoby (koncepcyjnie służy ona do przetapiania złomu na użyteczny metal),
- *Factory* — fabryka produkująca jednostki,

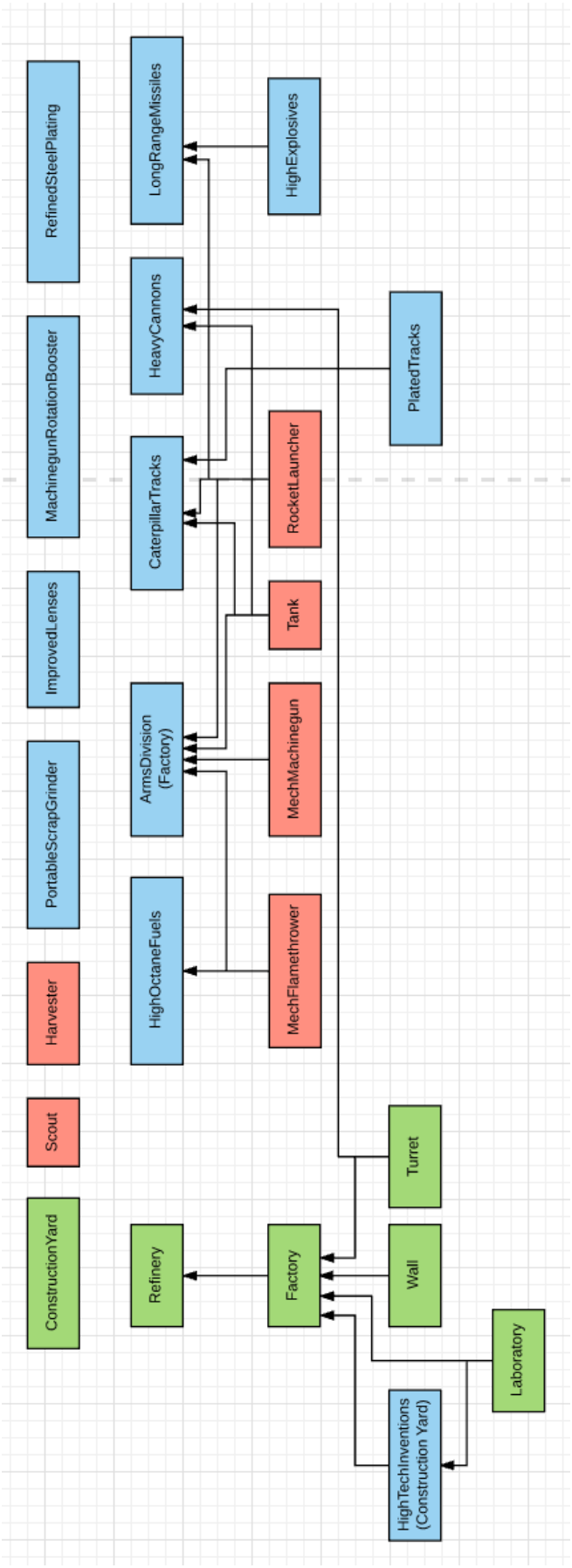
- *Laboratory* — laboratorium naukowe służące do odkrywania nowych technologii, dających premie do statystyk oraz odblokowujących nowe jednostki i budynki,
- *Turret* — zmechanizowana wieżyczka obronna, która potrafi strzela do wrogów,
- *Wall* — mur blokujący jednostkom przejazd.

1.5. 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 takim wymaganiem. Gdy wziąć wszystkie zależności wymagań pomiędzy technologiami, budynkami i jednostkami, tworzą one tak zwane drzewko technologiczne, pokazujące drogi rozwoju ekonomicznego i militarnego. Przedstawiono je na **Rysunek 1** (na następnej stronie). W następnym akapicie znajduje się natomiast jego objaśnienie.

Czerwone prostokąty oznaczają jednostki, zielone budynki, a niebieskie technologie. Wszystkie jednostki są produkowane w *Factory*, a budynki konstruowane w *ConstructionYard*. Technologie opracowuje się głównie w *Laboratory*, za wyjątkiem dwóch z nich — te w nawiasie mają podany budynek, który służy do ich odkrycia. Strzałka wychodzi od obiektu wymagającego i wskazuje na obiekt wymagany. Oczywiście, by stworzyć jednostkę potrzeba *Factory*, a do opracowania dowolnej technologii musi najpierw istnieć *Laboratory*. Nie ma jednak konieczności egzekwować takiego wymagania, gdyż jest to zrealizowane po prostu 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 żadna z jednostek nie jest do niczego wymagana. Oczywiście jest to celowe, gdyż taki zabieg nie miałby sensu.

Przykładem wymagań może być 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 w fabryce musi zostać opracowany dział zbrojeniowy (*ArmsDivision*).



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

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

- *PortableScrapGrinder*: szybkość zbierania zasobów przez jednostkę *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.

1.6. Interfejs użytkownika

Na graficzny interfejs użytkownika (ang. *Graphic User Interface*, w skrócie *GUI*) można podzielić na dwa warianty: główne menu pozwalające wybrać tryb gry i skonfigurować ustawienia oraz *GUI* dostępne podczas samej rozgrywki. Głównym elementem tego drugiego jest panel dolny zawierający przyciski rozkazów oraz tzw. minimapę (czyli miniaturę całej planszy, na której widać jej stan — położenie jednostek, budynków, zasobów oraz kształt mgły wojny). Poza tym w górnej części ekranu wyświetlana jest obecna liczba jednostek zasobów, a nad przyciskami rozkazów, po po najechaniu na dowolny z nich kursorem myszy, pokazuje się „dymek" (ang. *tooltip*) z jego opisem.

Sama myszka zachowuje się w sposób zbliżony do znanego ze współczesnych *RTS*-ów. Lewy przycisk służy do zaznaczania elementów mapy. Przy przytrzymaniu go i przeciągnięciu myszy powstaje ramka zaznaczenia, która wpływa na wybieranie wszystkich elementów mapy znajdujących się wewnątrz niej. Z kolei prawy przycisk myszy wydaje rozkazów w sposób kontekstowy — na przykład przy kliknięciu na teren jednostka otrzymuje rozkaz *Move*, na zasób: *Harvest*, a na wroga: *FollowAttack*. Natomiast po wybraniu rozkazu przyciskiem z panelu dolnego, to lewy przycisk myszy służy do jego wydania, a prawy — wraca do domyślnego trybu działania.

Wokół zaznaczonych jednostek i budynków pojawiają się ramki z tzw. paskami życia. Długość tych pasków wskazuje na to, ile punktów życia pozostało elementowi mapy, a ich kolor określa z której armii on pochodzi.

2. 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. Poniżej przedstawiamy ich listę:[10]

- 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 jej stanie, 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

[10] W nawiasie podana została nazwa jednej lub kilku najważniejszych klas z danego podsystemu. Znajduje się w nich zazwyczaj większa część jego funkcjonalności.

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 trzy podmoduły. Podmoduł ataków zawiera klasy obsługujące wyprowadzanie ataków przez jednostki. Podmoduł statystyk dotyczy utrzymywania liczbowych 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 tej produkcji.

Obiekty mapy utrzymują dane na temat 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 widzianym przez armię terenem i jednostkami. Aktualizowana przez `MapElement` tablica `VisibilityTable` zawiera informacje o tym, które pola są widoczne, które poza zasięgiem widzenia, a które jeszcze 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 zastosowanie także i innej metody, 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, np. w zależności od jej stanu kliknięcie może wydać taki lub inny rozkaz), podświetlania i zaznaczania elementów mapy (`HoverBox` i `SelectionMonitor`), wyboru miejsca konstrukcji budynku, decydowania o kolorach ramek zaznaczonych jednostek.

Zaprogramowana sztuczna inteligencja jest drugim co do obszerności modułem.[11] 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 odrębny agent — niższego poziomu. Agenty symulują równoczesne działanie, podobne do procesów wielowątkowych. 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 „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 elementów interfejsu),[12] licznik zasobów (`ResourceCounter`), wizualizacje statusów zaznaczonych elementów mapy (`StatusDisplayDrawer`), przyciski akcji rozkazów (`OrderActionButttton`), obsługa ich skrótów klawiszowych, wyświetlanie „dymków" pomocy.

Ostatnim modułem jest moduł narzędziowy. Zawiera on bardzo różne funkcjonalności pomocnicze: często używane typy które mogłyby istnieć w oderwaniu od projektu

[11] Szczegółowo *AI* została opisana w Rozdziale 4. Zaprogramowanie sztucznej inteligencji, str. 70
[12] Rozdział 3.1.2. Budowa interfejsów gracza, str. 59, fragment dotyczący *Canvas GUI*

(np. `IVector2`, `SquareBounds`, `BinaryHeap`) albo statyczne klasy z metodami rozszerzającymi do innych typów[13] (np. `UnityExtensions`, `EnumerableExtensions`, `DotNetExtensions`).

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

2.1. 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 najpierw zrozumieć 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.[14] Oprócz tego obiekty gry mogą zostać ułożone wobec siebie w relacji rodzic-dziecko[15] — 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, oraz, najważniejsze dla

[13] tzw. *extension methods*.
[14] Dla rotacji pokazane są jej kąty Eulera (i to nimi można zarządzać z poziomu edytora), ale *Unity* przechowuje ją w postaci kwaternionu.
[15] Informację o tej relacji również trzyma `Transform`.

nas: wykonywania własnej obsługi obiektu gry w postaci skryptu. Dla każdego rodzaju funkcji istnieje osobny komponent. Wszystkie przypisane do obiektu gry komponenty wyświetlają się w panelu *Inspector*, poniżej części *Transform*.[16] Tam to możliwa jest konfiguracja wszystkich komponentów. Najistostniejszym i najbardziej wykorzystywanym przez prototyp **MechWars** komponentem jest skrypt *C#*.

Skrypty pozwalają w rozległy sposób modyfikować zachowanie obiektu gry. Aby było to możliwe, muszą one utrzymywać pewną konwencję. Każdy skrypt jest klasą dziedziczącą po klasie `MonoBehaviour`. Może być on 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 klas wszystkich przypisanych do niego skryptów, 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 przerywając działanie gry, lecz nie edytora.

Ostatnią kwestią do poruszenia w związku z *Unity* są tzw. prefaby. Prefab (ang. *prefabricated*) jest zapisanym w projekcie *Unity* obiektem gry — skopiowanym do katalogu projektu wprost ze sceny. Prefab zawiera wszystkie informacje (położenie, hierarchia, komponenty i wartości ich parametrów), jakie zawierał obiekt gry w momencie jego

[16] *Transform* również jest komponentem, jednak wpisanym w każdy obiekt gry na stałe.

zapisywania. Funkcjonalność ta daje rozległe możliwości. Podstawowym przypadkiem użycia prefabu jest stworzenie szablonu do obiektu gry, który może być później powielany na scenie. Prefaby można jednak wykorzystać 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.

2.2. 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.

2.2.1. Singleton `Globals`

`Globals` jest jedną z ważniejszych typów 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. [17] 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

[17] http://www.dofactory.com/net/singleton-design-pattern, 23.09.2016r [6]

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 tu też dostać do instancji `ShapeDatabase` i `LOSShapeDatabase` — które nie są skryptami `Unity`, ale istnieją w polach obiektu `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 trzymającego główną kamerę i skrypt `GLRenderer` (do którego delegowane są zadania renderowania linii, wykorzystane w wizualizacji zaznaczonych jednostek i ramki zaznaczenia).

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.

2.2.2. 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 po kilka. Obecnie prototyp jest zaprogramowany na pracę z maksymalnie dwoma każdego rodzaju. Wszystkie trzy klasy dziedziczą 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`[18] armia przetrzymuje widoczne przez nią na mapie własne elementy mapy, wrogie elementy mapy i zasoby. Jedyne publiczne metody tej klasy: `AddMapElement()` oraz `RemoveMapElement()` pozwalają na zarządzanie

[18] Klasę `QuadTree` opisano w Rozdziale 2.4.2 Klasy drzewa czwórkowego, str. 44

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. Drugim jest `AIBrain`, omówiony w rozdziale o zaprogramowaniu sztucznej inteligencji.[19] Klasa `Spectator` jest również prosta (choć nie tak prosta jak `Player`). Posiada publiczne pole typu `Player`, oraz publiczną właściwość z obiektem `InputController`, który wykonuje niemal wszystkie zadania potrzebne do sterowania graczem i armią przez gracza-człowieka. W metodzie `Update()` obiekt `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.

2.2.3. Obiekty środowiska

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

- lista typu `GameObject` zawierającą dwa obiekty gry — światła imitujące księżyc oraz słońce obracające się dookoła sceny,
- pole `cycleTime`, z informacją mówiącą ile czasu trwa jeden cykl obrotu świateł,
- metoda `Update()` aktualizująca rotację słońca i księżyca.

Wewnątrz metody znajduje się warunek obsługujący sytuację, w której wartość `cycleTime` jest równa 0 (jest ono traktowane jak 1). Gdyby nie to, program wygenerowałby wyjątek dzielenia przez 0, gdyż prędkość obrotu jest obliczana poprzez odwrotność okresu.

[19] Rozdział 4. Zaprogramowanie sztucznej inteligencji, str, 70

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);
        }
    }
}
```

Listing 1. Klasa `DayAndNight`.

2.2.4. 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`, w których ustawione są prefaby do instancjonowania na scenie (zasoby, tzw. 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 markerów).

Obie klasy zapewniają jedynie bardzo ogólną konfigurację. 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*.

2.2.5. 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*).

2.3. 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.[20] Ponieważ zarówno jednostka, jak i budynek potrafią wykonywać rozkazy, ich obsługę finalnie zdecydowaliśmy 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. Dlatego rozdział ten opisuje też obie klasy abstrakcyjne za to odpowiedzialne — `Order` i `OrderAction`. Dodatkowo omówione są pobieżnie statystyki, technologie i zarządzanie nimi oraz mechanizm ustalania sąsiedztwa murów.

2.3.1. 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 ona 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 nowej.

[20] W przypadku przeszkody to po prostu `MapElement`, gdyż nie ma ona żadnej dodatkowej funkcjonalności.

```
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;
```

Listing 2. Wybrane publiczne pola klasy `MapElement`.

Publiczne pola `shapeFile` i `statsFile` typu `TextAsset` służą do przypisania plików tekstowych z informacjami na temat kształtu elementu mapy (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).

Publiczna lista obiektów `OrderAction` definiuje, jakie rozkazy można wydać elementowi mapy. Jest ona skonfigurowana w panelu *Inspector* we wszystkich prefabach elementów mapy. 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).

```
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);
    }
}
```

Listing 3. Właściwości klasy `MapElement` związane ze statystykami, kształtem i transformacją.

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 przerwać się 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.

`MapElement` kolejkuje wydane mu rozkazy na obiekcie klasy `OrderQueue`. Obiekt ten zawiaduje w całości kolejnością ich wykonywania. 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); } }
```

Listing 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*.[21] 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 było go widać. W ten sposób ograniczana jest wiedza gracza o sytuacji poza polem

[21] http://www.gamedev.net/topic/489276-generating-line-of-sight-in-tile-based-rts/, 23.09.2016r [5]

widzenia. Na przykład gracz nie powinien 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 osobne duchy, dlatego `MapElement` posiada słownik własnych 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 konieczne jest, by duch miał możliwość być celem rozkazu (np. ataku), a jednocześnie taki rozkaz musi zmienić cel na oryginalny `MapElement`, gdy tylko ten zacznie być na powrót 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 — ponieważ element mapy może zostać zniszczony, gdy znajduje się poza polem widzenia i o tym gracz też nie może 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.

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()`[22] 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.

[22] Funkcja opisana szerzej w Rozdziale 2.8.3. Wygląd minimapy, str. 55

```
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 ****
    }
}
```

Listing 5. Metoda `MapElement.OnStart()` dla elementu mapy nie będącego duchem.

Funkcja `ReadStats()` wczytuje statystyki z pliku zapisanego w standardzie *XML*. Wykorzystywana jest do tego .NETowa klasa `System.Xml.XmlDocument`.[23]

`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`. Przykładowo `GetClosestAimTo()` pobiera jeden z celów elementu mapy z listy `aims` — ten, który jest najbliższy zadanej pozycji. `HasMapElementInRange()` sprawdza z kolei, czy zadany element mapy znajduje się w zasięgu, np. ataku. Natomiast `PickClosestResourceInRange()` zwraca najbliższy zasób w polu widzenia (używane przez *Harvestery* w rozkazie `HarvestOrder`).

[23] Przykład pliku *XML* ze statystykami pokazano w Rozdziale 2.3.2. Statystyki, str. 33

Metody związane z atakiem to `ReadyAttack()` i `MakeAttack()`. Atak musi zostać najpierw przygotowany — ponieważ przed jego wykonaniem należy wyliczyć różne parametry (takie, jak np. kąt o jaki atakujący musi obrócić siebie bądź głowicę). Dopiero potem można atak wykonać. Naliczane jest wtedy opóźnienie ataku (tzw. *cooldown*), dzięki któremu jednostka może zaatakować tylko raz na określony czas.

```
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;
}
```

Listing 6. Metoda `MapElement.OnUpdate()` dla elementu mapy nie będącego duchem.

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.

2.3.2. Statystyki

Za statystyki odpowiadają przede wszystkim klasy `Stat` i `Stats`. Klasa `Stats` jest pośrednikiem dla słownika statystyk, w których kluczem jest ich nazwa. Natomiast obiekt klasy `Stat` stanowi pojedynczą statystykę. Zawiera więc 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>
```

Listing 7. 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 premia 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.

2.3.3. Technologie

Technologie znalazły się w tym module, gdyż mają bezpośredni związek z elementami mapy. Są odkrywane za pomocą budynków, a ich opracowywanie spełnia wymagania do produkcji i konstrukcji nowych jednostek i budynków, a także odblokowuje premie 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 natomiast są konfigurowane w innym miejscu, a dokładniej: w akcjach rozkazów.[24]

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.

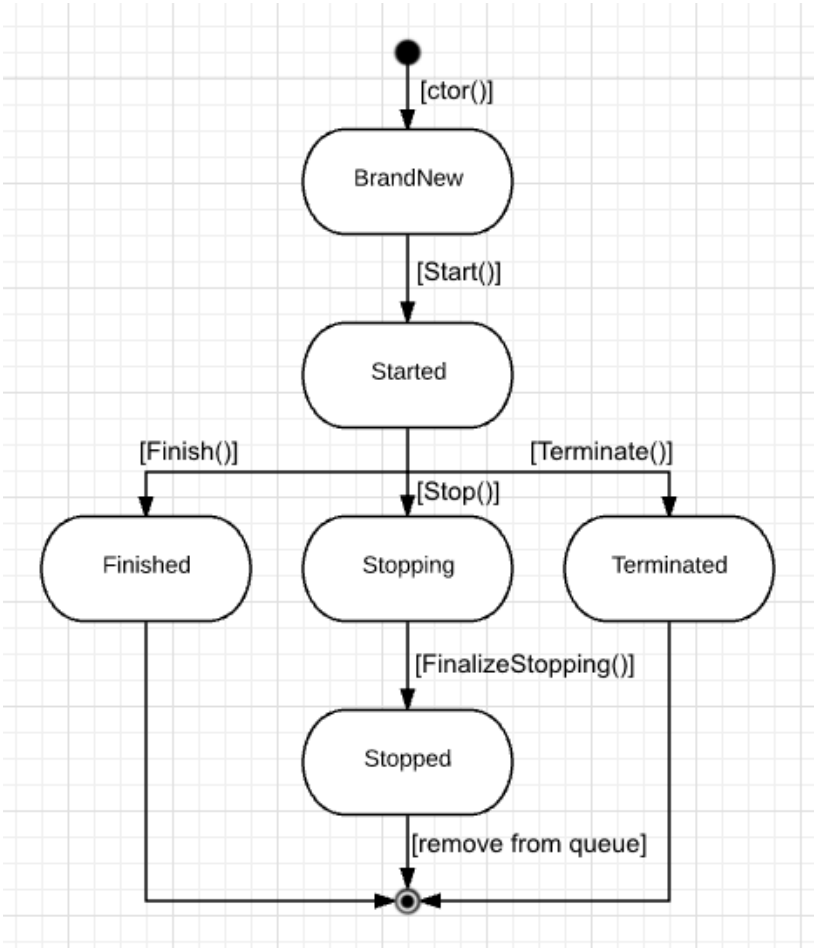
2.3.4. Rozkazy

Klasa `Order` i jej potomne są stworzone jako połączenie wzorców projektowych *template method* oraz *command*. 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*.[25] 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*.[26]

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

[24] Rozdział 2.3.5. Akcje rozkazów, str. 39
[25] http://www.dofactory.com/net/template-method-design-pattern, 23.09.2016r [7]
[26] http://www.dofactory.com/net/command-design-pattern, 23.09.2016r [8]

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. Jedyna więc możliwość jego zatrzymania, to użycie metody `Terminate()`, która jest zarezerwowana dla sytuacji i tak błędnych.

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`**).

```
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;
}
```

Listing 8. Główne funkcje i właściwości klasy `Order`.

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 zależnie od jego wartości zmieniają stan w odpowiedni sposób. 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).

Obiekt `ComplexOrder` samodzielnie trzyma referencję na aktualny i następny pod rozkaz, nie korzystając przy tym z kolejki `OrderQueue`. Nadpisuje oraz pieczętuje[27] funkcję `LateOnUpdate()`, 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,[28] `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.

[27] Słowo kluczowe `sealed` — następni potomkowie nie mogą nadpisać tej funkcji.
[28] Patrz: **Listing 9**

```
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();
    }
}
```

Listing 9. Główne funkcje i właściwości klasy `ComplexOrder`.

2.3.5. Akcje rozkazów

Abstrakcyjna klasa `OrderAction` służy dwóm celom. Po pierwsze, konstruując obiekty dziedziczące po `Order`, akcja rozkazu realizuje wzorzec projektowy *abstract factory*.[29] Abstrakcyjna metoda `CreateOrder()` przyjmuje wykonawcę rozkazu (`MapElement`) oraz parametry (takie jak cel ataku albo ruchu), a zwraca nowo stworzony rozkaz. Dla każdej klasy rozkazu, który gracz może wydać elementowi mapy, istnieje klasa tworzącej go akcji rozkazu. 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 sprawdzeniu, 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);
```

Listing 10. Główne metody klasy `OrderAction`.

Po drugie, klasa `OrderAction` dziedziczy po `MonoBehaviour`, więc może służyć do konfiguracji. W projekcie `Unity` zapisany jest szereg prefabów zawierających (jako komponenty) skrypty potomne do `OrderAction`. Natomiast każdy prefab elementu mapy ma powyższe prefaby ustawione na jego liście akcji rozkazów. W ten właśnie sposób skonfigurowane jest to, jakie rozkazy potrafi wykonywać każdy `MapElement`.

Same akcje rozkazów również mogą być konfigurowalne (sic![30]). Dla trzech akcji rozkazów: `UnitProductionOrderAction`, `BuildingConstructionOrderAction` oraz `TechnologyDevelopmentOrderAction` istnieje szereg prefabów ustawionych w różny

[29] http://www.dofactory.com/net/abstract-factory-design-pattern, 23.09.2016r [9]
[30] Nie: „konfigurowane". To, czy są konfigurowalne, czy nie, zależy od tego, czy klasa potomna do `OrderAction` zawiera jakieś publiczne pola.

sposób. Przykładowo komponentowi `UnitProductionOrderAction` przypisać można prefab jednostki, która jest tworzona, koszt i czas trwania produkcji oraz wymagania: listę budynków, jakie muszą być wybudowane oraz listę technologii koniecznych do uprzedniego opracowania.

2.3.6. Sąsiedztwo murów

Wygląd muru zależy od tego, z czym on sąsiaduje. Na przykład jeżeli stawiamy go na otwartej przestrzeni, gdzie z niczym się nie styka, jego model jest podstawowy (w kształcie litery I). Natomiast gdy przynajmniej dwóch stron będzie otoczony innymi budynkami, zamiast bazowego wyglądu jego model zastąpiony zostaje innym, o kształcie pasującym do otoczenia. Do zarządzania wyborem modelu muru, który ma zostać wybrany w konkretnej sytuacji, używamy czterech skryptów:

- `WallModelReplacer`,
- `WallNeighbourhood`,
- `WallNeighbourhoodDefinition`,
- `WallNeighbourhoodDictionary`.

Skrypt `WallNeighbourhood` zawiera strukturę składającą się z czterech zmiennych `bool` (dla każdego kierunku) określających, obok których pól sąsiednich do muru znajdują się inne budynki. `WallNeighbourhoodDefinition` jest skryptem `MonoBehaviour`, posiadającym cztery publiczne pola `bool`, również odpowiadające kierunkom świata. Stworzono go po to, by opisane wyżej sąsiedztwo muru dało się skonfigurować w panelu *Inspector*. Dzięki przypisaniu go do prefabów odpowiednich modeli można wybrać, do jakiego sąsiedztwa pasuje który model. Dodatkowo `WallNeighbourhoodDefinition` posiada posiada publiczną właściwość `Neighbourhood`, która na podstawie wartości jego publicznych pól tworzy i zwraca instancję struktury `WallNeighbourhood`.[31]

`WallNeighbourhoodDictionary` jest kolejnym skryptem `MonoBehaviour`. Posiada publiczną listę obiektów `WallNeighbourhoodDefinition`, którą również można konfigurować w panelu *Inspector*. W swojej metodzie `Start()` kopiuje on zawartość tej listy do wewnętrznego słownika, gdzie kluczem jest `WallNeighbourhood`.

Skrypt `WallModelReplacer` przypisany jest do prefabu elementu mapy *Wall* (i tylko do tego elementu mapy). Sprawdza on otoczenie, z którym sąsiaduje mur. Na tej podstawie

[31] Istotny jest fakt, że `WallNeighbourhood` jest strukturą, a nie klasą. Struktury w języku C# zachowują się jak typy wartościowe: przy podstawianiu i przekazywaniu do zmiennych są kopiowane, ale porównanie dwóch różnych instancji struktury o identycznych wartościach jej pól zwraca **`true`**. Zastosowano to rozwiązanie, by obiekt `WallNeighbourhood` mógł być kluczem w słowniku `WallNeighbourhoodDictionary`. Ma to sens, gdyż na `WallNeighbourhood` można patrzeć jak na 4-wymiarowy wektor wartości `bool`.

tworzy on nową instancję struktury `WallNeighbourhood`, którą podaje jako klucz do słownika `WallNeighbourhoodDictionary`. W ten sposób pobiera odpowiedni prefab modelu, za pomocą którego tworzy obiekt gry z modelem i umieszcza go na scenie.

```
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;
    }
}
```

Listing 11. Ciało klasy `WallModelReplacer`.

2.4. Podsystem mapy

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

2.4.1. 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()`, wygeneruje stosowny wyjątek.

```
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;
}
```

Listing 12. Najważniejsze fragmenty klasy `Map`.

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 `IsInBounds()` (sic![32]) do sprawdzania, czy dana pozycja znajduje się w granicach mapy,[33] a także funkcje do: sprawdzania, czy dane pole jest zajmowane przez element mapy, tworzenia i zwalniania rezerwacji (`MakeReservation()`, `ReleaseReservation()`) oraz dodawania i usuwania duchów (`AddGhost()`, `RemoveGhost()`).

Nazwa „rezerwacja" wynika z tego, że żaden element mapy nie może zajmować pola już zajętego przez inny element mapy. Jednostka jest a, 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 jednak nowe pole jest wolne, zostaje zarezerwowane, a ze starego pola rezerwacja jest zwalniana. Gdy 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 działają one analogicznie do funkcji `MakeReservation()` i `ReleaseReservation()`. Warto jedynie nadmienić, że dla duchów 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 jest zwykły element mapy. Dlatego właśnie duchy

[32] Nie: „publiczną metodę". Są dwie metody o tej samej nazwie i różnych parametrach.
[33] Często są one używane, by uchronić się przed wyjątkiem `IndexOutOfBoundsException`.

umieszczone są w odrębnych strukturach danych, a `ghostsTable` jest tablicą **list** obiektów `MapElement`.[34]

2.4.2. 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 nowy obszar nie będzie zawierał maksymalnie jednego elementu mapy.

Ponieważ dwuwymiarowa przestrzeń jest dyskretna i za każdym razem dzielona na 4, a w każdym wymiarze na 2, to maksymalną głębokością drzewa będzie zawsze $\boldsymbol{log_2(S)}$, gdzie **S** 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 obszarów 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.

[34] Podczas gdy `reservationTable` jest tablicą **pojedynczych** obiektów `MapElement`.

```
QuadTreeMapElement QuadTreeMapElement;
SquareBounds bounds;

QuadTree x0y0;
QuadTree x0y1;
QuadTree x1y0;
QuadTree x1y1;
```

Listing 13. 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 również będące typu `QuadTree` pola, które reprezentują węzły potomne. Działa to, gdyż każdy węzeł jednocześnie stanowi osobne pod-drzewo oparte na czterokrotnie mniejszym obszarze. 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) zawierającego współrzędną podanego w argumencie elementu mapy, który następnie jest tam wstawiany. 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 funkcji `Insert()` zależy natomiast od lokalnej głębokości drzewa, więc jej złożoność wynosi $\boldsymbol{O}(\boldsymbol{log}(\boldsymbol{n}))$.

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 tak, 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.

Ostatnią, i najważniejszą 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 prostokąt podany w argumencie. Niepuste liście zwracają jednoelementowe listy, a węzły na coraz wyższych poziomach łączą je w listy coraz większe. Na koniec metoda zwraca pełną listę obiektów `QuadTreeMapElement` z całego wycinka drzewa.[35]

```
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;
}
```

Listing 14. Metoda `QuadTree.QueryRange()`.

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 maksymalną. Niech $\boldsymbol{n}$ będzie liczbą elementów mapy w drzewie, a $\boldsymbol{h}$ jego głębokością. Najszersze zapytanie (o obszarze całej mapy) zwróci oczywiście wszystkie elementy, więc jego złożoność to $\boldsymbol{O}(\boldsymbol{n})$. Najwęższe z kolei będzie musiało przejść od korzenia do dokładnie jednego liścia, stąd $\boldsymbol{O}(\boldsymbol{h})$. Razem daje to złożoność $\boldsymbol{O}(\boldsymbol{n}+\boldsymbol{h})$.

2.5. Podsystem mgły wojny

Pomysł na działanie mgły wojny został zaczerpnięty z książki *Perełki programowania gier*. [36] Przedstawia ona tam kompletny sposób na stworzenie efektywnego systemu widoczności dla gry *RTS*. Opisany tu podsystem jest bardzo inspirowany mechanizmami tam zawartymi.

[35] https://en.wikipedia.org/wiki/Quadtree#Query_range, 23.09.2016r [11]
[36] *Perełki programowania gier. Vademecum profesjonalisty, Tom 2*, 2002, Matt Pritchard, *Rozdział 3.5: Wysokowydajny system widoczności i wyszukiwania oparty na siatkach*, str. 317 [15]

Zarządzaniem danymi o kształcie mgły wojny zajmuje się klasa `VisibilityTable`. Każda armia ma własną tablicę widzialności. [37] Jest to obiekt zawierający dwie dwuwymiarowe tablice odwzorowujące rozmiarem planszę gry. Dla każdego pola mapy w tablicy `bool[,] fieldsUncovered` znajduje się informacja, czy dane pole w ogóle zostało odkryte przez armię. Z kolei tablica `int[,] fieldsSeenByUnits` trzyma dla każdej kratki liczbę widzących ją jednostek. Odkryte pola, dla których liczba ta równa jest zeru, znajdują się we mgle wojny — nie widać na nich jednostek, a zamiast budynków i zasobów pokazane są ich duchy.[38]

W klasie `VisibilityTable` znajdują się metody `IncreaseVisibility()` oraz `DecreaseVisibility()`. Służą one do inkrementacji i dekrementacji elementów tablicy `fieldsSeenByUnits` przez `MapElementy`. Przykładowo gdy jednostka deklaruje ruch, dekrementuje pola tablicy w swoim zasięgu widzenia względem 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[39] 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 [40] zwracającą wygodną w użyciu wartość *enuma* `Visibility`.[41] Elementy mapy sprawdzają ją, gdy określają 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 pola widzialne — całkowicie przezroczyste. Tekstura ta nakładana jest następnie na płaszczyznę zasłaniającą planszę, stanowiąc coś w rodzaju filtru koloru dla

[37] W książce zwana jest ona „mapą widoczności".
[38] Książka określa je jako „miraże".
[39] W książce zwane „szablonami linii widoczności".
[40] Mechanizm języka C#. Istnienie właściwości-indeksatora sprawia, że można korzystać z obiektów klasy ją zawierającej zupełnie tak, jakby były tablicami. Indeksem może być tu dowolny typ — taki, jaki zdefiniujemy pisząc indeksator. Wewnątrz *gettera* i *settera* mamy dostęp do argumentu podanego jako ów indeks.
[41] *Enum* Visibility przyjmuje wartości: `Unknown`, `Fogged` oraz `Visible`

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.[42]

```
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);
}
```

Listing 15. Wybrane składowe klasy `VisibilityTable`.

[42] Rozdział 2.8.3. Wygląd minimapy, str. 55

2.6. Podsystem poszukiwania ścieżek

Podsystem ten został stworzony w ogólny sposób tak, by można było zaimplementować dowolny algorytm na znajdowanie ścieżek. Interfejs `IPathdfinder` wystawia fukcję `FindPath()`, którą każda klasa poszukująca ścieżek musi posiadać. Przyjmuje ona punkt startowy, punkt docelowy oraz `MapElement`, który potrzebuje tej ścieżki. Istnieje jednak w projekcie tylko jedna klasa implementująca ten interfejs — jest nią `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 1, a po przekątnej — $\sqrt{2}$. 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. Nazywane jest to ograniczoną relaksacja heurystyki algorytmu. Liczba ε stanowi „ograniczenie relaksacji gwarantujące, że rozwiązanie nie będzie gorsze niż najlepsze rozwiązanie pomnożone przez $(1 + \varepsilon)$. Algorytm z takż heurystyką, to algorytm ε-dopuszczalny. Istnieje kilka ε-dopuszczalnych odmian algorytmu *A**".[43] Wybraną przez nas odmianą jest *A** statycznie ważony. Liczba ε stanowi w niej wagę, przez którą mnoży się wartość heurystyki. U nas ε wynosi zatem 5. Wartość tę dobrano eksperymentalnie jako taką, która wystarczająco poprawiała wydajność, jednocześnie nie zmniejszając skuteczności algorytmu w widocznym stopniu.

2.7. Podsystem sterowania

Z modułu tego zdecydowano się przedstawić jedynie jeden jego mechanizm. Jest nim sposób decydowania o wyborze miejsca budynku, poprzez wyświetlanie tzw. cienia.[44]

Klasa `InputController`, której jedyna instancja łączy wszelkie aspekty sterowania grą, posiada właściwość `BuildingShadow`, oraz publiczną metodę `CreateShadow()`, służącą do stworzenia cienia i podstawienia go pod wspomnianą właściwość. Z funkcji tej

[43] https://en.wikipedia.org/wiki/A*_search_algorithm#Bounded_relaxation, 23.09.2016r [12]
[44] Pojęcia **cienia budynku** nie wolno mylić z pojęciem **ducha elementu mapy**!

korzystają przyciski *GUI* wprowadzające myszkę w tryb wyboru miejsca do postawienia budynku. `BuildingShadow` odpowiada za zintancjonowanie elementu mapy będącego cieniem i następnie wyświetlanie go w określonych warunkach. Cień nie jest w jakikolwiek sposób funkcjonalny — ma jedynie w trybie wyboru miejsca na postawienie budynku pokazywać to miejsce. Wewnątrz klasy znajduje się pare funkcji, jednak najważniejsze z nich to `UpdateLocation()` oraz `UpdateLook()`.

```
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 horizontalOffset = W % 2 == 0 ? 0.5f : 0;
    float verticalOffset = H % 2 == 0 ? 0.5f : 0;

    float xSnap1 = Mathf.Floor(p.x - horizontalOffset) + horizontalOffset;
    float xSnap2 = Mathf.Ceil(p.x - horizontalOffset) + horizontalOffset;
    float ySnap1 = Mathf.Floor(p.y - verticalOffset) + verticalOffset;
    float ySnap2 = Mathf.Ceil(p.y - verticalOffset) + verticalOffset;

    float closerXSnap =
        Mathf.Abs(p.x - xSnap1) < Mathf.Abs(p.x - xSnap2) ? xSnap1 : xSnap2;
    float closerYSnap =
        Mathf.Abs(p.y - ySnap1) < Mathf.Abs(p.y - ySnap2) ? ySnap1 : ySnap2;

    shadow.Coords = Position;
    var allCoords = shadow.AllCoords.ToList();
    bool cannotBuild = allCoords. Any(c => Globals.Map[c] != null ||
        !inputController.ConstructionRange.FieldInRange(c));

    CannotBuild = cannotBuild;
    shadow.transform.position = new Vector3(Position.x, 0, Position.y);
}
```

Listing 16. Funkcja `BuildingShadow.UpdateLocation()`.

Metoda `UpdateLocation()` odpowiada za położenie cienia budynku oraz przyciąganie go do pełnych kratek. Najpierw odczytywany jest punkt (rzeczywisty) planszy, nad którym znajduje się kursor. Na jego podstawie wyliczane są nabliższe współrzędne, do których cień zostanie przyciągnięty. Nie zawsze są to współrzędne całkowite — jeśli kształt budynku ma parzysty jeden z wymiarów, to odpowiednia współrzędna będzie połówkowa. Dzięki temu zabiegowi stawiany budynek zawsze pasuje do kratek planszy. Na końcu funkcja sprawdza warunki konieczne. Jeżeli chociaż jedno z wybranych pól znajduje się poza dozwolonym zasięgiem budowania, lub jest zajęte przez inny element mapy — cień zostanie

pokolorowany na czerwono, co pokaże, że miejsce nie jest dozwolone. Za barwę cienia odpowiada funkcja `UpdateLook()`.

2.8. Podsystem interfejsu gracza

Jedną z podstawowych rzeczy w każdej grze jest interfejs gracza. Pozwala on na wykorzystywanie elementów mechaniki przez użytkownika. Większa część *GUI* w grze została stworzona przy pomocy wbudowanego w *Unity* mechanizmu *Canvas*. Osadzono w nim przyciski rozkazów, opisy do tych przycisków oraz tzw. minimapę.

2.8.1. Przyciski rozkazów

Choć w większości gier *RTS* rozkazy można wydawać bezpośrednio za pomocą myszy, często pojawiającym się elementem *GUI* jest panel z przyciskami, wśród których dostęp mamy do wszystkich możliwych rozkazów. Jego zawartość zależy od kontekstu — jeśli pewna jednostka nie potrafi wykonać danego rozkazu, odpowiedni przycisk nie pojawi się, gdy ją zaznaczymy. Dodatkowo większość przycisków posiada skróty klawiszowe (ang. *hotkey*) dzięki którym gracz może przyspieszyć swoje działania. W prototypie **MechWars** za całą powyższą funkcjonalność odpowiadają skrypty: `OrderActionButton`, `OrderUtilityButton`, oraz `OrderActionButtonController`.

Skrypt `OrderActionButton` przypisuje się do przycisku. Posiada on dwa publiczne pola: `orderAction` oraz `hotkey`. Dzięki nim w panelu *Inspector* da się ustawić, akcję, jaką wywołuje przycisk i skrót klawiszowy, którym można to również wykonać.

W klasie `OrderActionButton` znajdują się trzy funkcje obsługi zdarzenia kliknięcia przycisku. Dla jednego przycisku przypisuje się tylko jedną z nich. `OnClick_ChangeCarriedOrderAction()` odpowiada za wydanie rozkazu, który wymaga kliknięcia w obszar gry — określając w ten sposób np. pozycję docelową ruchu, albo cel ataku. Przypisuje ona akcję przycisku do pola `CarriedOrderAction` w obiekcie `InputController`. Czynność ta zmienia domyślne działanie myszki, która od tego momentu „niesie” akcję rozkazu i wyda ją, gdy tylko gracz kliknie lewym przyciskiem myszy. Podobnie działa funkcja `OnClick_ChangeCarriedBuildingAction()`, jednak ją przypisuje się jedynie do przycisków akcji `BuildingConstructionOrderAction`, gdyż dodatkowo tworzy ona cień budynku, oraz obiekt wizualizujący zasięg konstrukcji. Funkcja `OnClick_InvokeOrderAction()` natomiast wywołuje rozkazy, które nie wymagają żadnych argumentów — robi to więc natychmiast po kliknięciu przycisku.

Ostatnie dwie funkcje odpowiadają za podświetlenie przycisków. `OnPointerEnter()` wywoływana jest, gdy gracz najedzie kursorem myszy na przycisk, więc ustawia go na

podświetlony. `OnPointerExit()` natomiast uruchamia się, gdy mysz opuści obszar przycisku, dlatego czyści podświetlenie.

```
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);
}
```

Listing 17. Metody klasy `OrderActionButton`.

Skrypt `OrderUtilityButton` jest bardzo podobny do klasy `OrderActionButton`. Nie posiada funkcji odpowiadających za blokowanie akcji po naciśnięciu przycisku. Zamiast tego zaprogramowano działanie przycisków do anulowania obecnego rozkazu z kolejki `OrderQueue`, (funkcja `OnClick_CancelProduction()`; nazwa pochodzi stąd, że przycisk ten widoczny jest tylko dla budynków produkcyjnych, więc anuluje jedynie rozkaz produkcji, konstrukcji albo odkrywania), a także do odznaczania wszystkich aktualnie wybranych elementów mapy (funkcja `OnClick_CancelSelection()`).

```
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();
}
```

Listing 18. Funkcje obsługi zdarzeń przycisków *CancelProduction* i *CancelSelection* w klasie `OrderUtilityButton`.

Skrypt `OrderActionButtonController` obsługuje mechanizm decydujący, które przyciski powinny być widoczne. Posiada on publiczne pola na obydwa przyciski typu `OrderUtilityButton` oraz listę wszystkich przycisków `OrderActionButton`. Wszystkie te pola muszą zostać skonfigurowane w panelu *Inspector*, inaczej przyciski będą zawsze pokazane.

Całą funkcjonalność wykonuje metoda `Update()`. Na początku obsługiwany jest przypadek, gdy wybrano pojedynczy budynek. Jedynie wtedy pokazywany jest przycisk anulujący produkcję. Jeśli zaznaczenie jest puste, lub zawiera elementy mapy nienależące do armii *Spectatora*, to wszystkie przyciski dotyczące rozkazów są ukrywane. Jeżeli jednak wybrane zostały sojusznicze elementy mapy to pokazywane są te przyciski, których zbiór akcji rozkazów stanowi część wspólną zbiorów akcji każdego z zaznaczonych elementów mapy. Przykładowo, gdy gracz wybierze jednocześnie jednostki *Harvester* i *Tank*, to ukryte będą rozkazy ataku oraz zbierania zasobów, gdyż *Harvester* nie potrafi atakować, a *Tank* zbierać zasobów. Mimo wszystko, aby przycisk mógł zostać pokazany, trzeba jeszcze spełnić wymagania technologiczne, co jest sprawdzane w następnej pętli. Na końcu wywoływana jest funkcja `HandleHotkeys()`, która obsługuje skróty klawiszowe. Iteruje ona w pętli po wszystkich przyciskach i, jeśli został naciśnięty klawisz oznaczony jako ich skrót, wywołuje zdarzenie `onClick`.

```
void HandleHotkeys()
{
    var possibleButtons = AllButtons.Where(b => b.gameObject.activeSelf);
    foreach (var b in possibleButtons)
    {
        var button = b.gameObject.GetComponent<Button>();
        if (Input.GetKeyDown(b.Hotkey))
            button.onClick.Invoke();
    }
}
```

Listing 19. Kod odpowiedzialny za obsługę skrótów klawiszowych.

```
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);
    }

    HandleHotkeys();
}
```

Listing 20. Funkcja `OrderActionButtonController.Update()`.

2.8.2. Opisy pomocnicze przycisków

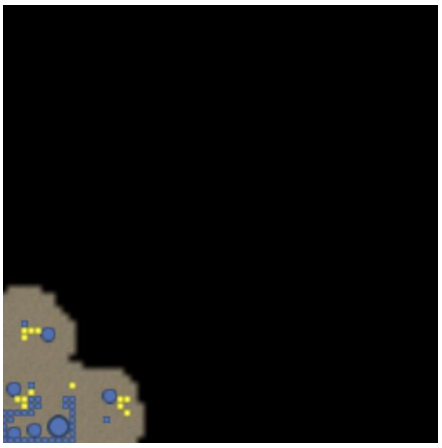
W celu lepszego zrozumienia przez gracza funkcji poszczególnych przycisków zaprogramowano skrypt `ButtonsInfoDisplayer`, który przypisano do pola tekstowego wyświetlanego bezpośrednio nad panelem przycisków. Odpowiada on za wyświetlanie podpowiedzi, w których umieszcza opis słowny zawarty jako publiczne pole w skryptach `OrderActionButton` i `OrderUtilityButton`. W zdefiniowanych opisach znajdują się: nazwa rozkazu oraz streszczenie jego działania i skrót klawiszowy.

```
Button hoveredButton;

public void SetHoveredButton(Button button)
{
    if (button == hoveredButton) return;
    hoveredButton = button;

    var text = GetComponent<Text>();
    if (button == null) text.text = "";
    else text.text = button.GetComponent<IDescriptionProvider>().Description;
}
```

Listing 21. Wybrane skadowe skryptu `ButtonsInfoDisplayer`.

2.8.3. Wygląd minimapy

„Minimapa to miniaturowy obraz, który pokazuje świat gry lub jego część w rzucie od góry. (...) Dzięki niej gracz może w łatwy sposób zorientować się w położeniu ważnych elementów w przestrzeni świata gry. (...) Ponieważ minimapa musi być mała (...) pokazuje ona tylko najważniejsze cechy geograficzne i minimum niekluczowych informacji. Główne jednostki i budynki zazwyczaj oznaczone są kolorowymi kropkami. Obszary świata ukryte przez mechanizm mgły wojny, ukrywa również i minimapa.”[45]



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

[45] E. Adams, *Fundamentals of Game Design*, New Riders, 2010, str. 227 [14]

W naszym prototypie minimapa jest teksturą renderowaną w czasie rzeczywistym ze znajdującej się nad sceną ortograficznej kamery. Generowany przez nią co klatkę obraz nie trafia jednak na ekran — zamiast tego jest on przekierowywany na teksturę *MinimapRenderTexture*. Teksturę tę nałożono na element *GUI*, aby ją móc wyświetlić wyrenderowaną minimapę.

Przy tworzeniu minimapy bardzo istotne było wyświetlanie w wyraźny sposób tylko konkretnych elementów mapy. W *Unity* można to osiągnąć poprzez przypisanie pożądanym obiektom gry odpowiednich warstw sceny, a następnie określenie, które z tych warstw kamera ma renderować, a których nie. W naszym przypadku chcemy, by widoczne były:

- teren (warstwa *Enviro*),
- markery, czyli symbole elementów mapy (warstwa *Minimap*),
- płaszczyzna mgły wojny (warstwa *MinimapFogOfWar*).[46]

Komponent *Camera* przypisany do obiektu gry *MinimapCamera* skonfigurowano zatem tak, aby kamera renderowała jedynie powyższe warstwy.

```
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);
}
```

Listing 22. Metoda `MapElement.InitializeMinimapMarker()`.

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, to barwa markera

[46] Dla minimapy stworzono osobną wersję płaszczyzny mgły wojny, gdyż w przeciwieństwie do głównej, ta nie musi się przesuwać i skalować tak, by znajdować się zawsze w tym samym miejscu na osi kamera-teren. Powodem tego jest oczywiście to, że kamera gry jest perspektywiczna, a minimapy — ortograficzna.

jest biała. `Resource` natomiast zwraca zawsze żółty kwadrat. Dzięki temu markery jednoznacznie określają rodzaj i przynależność elementu mapy.

Aby zdeterminować odległość markera od kamery, a więc wzajemne zasłanianie się markerów, stosowana jest wirtualna funkcja `GetMarkerHeight()`. Zwracana wysokość markera nad powierzchnią planszy zależy od implementacji w klasach potomnych. W efekcie jednostki są w stanie zasłaniać zasoby, które z kolei przykrywać mogą budynki. Z reguły nie powinno do tego dochodzić, gdyż nie może być dwóch elementów mapy na tym samym polu. Jednak przy odmiennym ustawieniu istniałaby szansa, że markery budynków zasłonią pozostałe markery. Wynika to z faktu, że budynek może zajmować więcej niż jedno pole i mieć niekwadratowe wymiary, ale jego marker zawsze będzie okręgiem (większym bądź mniejszym), więc może przykrywać pola, na których budynku nie ma.

Renderowanie mgły wojny na minimapie jest realizowane dzięki skryptowi `MinimapFog`. Klasa ta w funkcji `Update()` kopiuje teksturę z `VisualFog` i nakłada ją na płaszczyznę na warstwie *MinimapFogOfWar* (która jest wyświetlaną jedynie przez kamerę minimapy). Dzięki temu 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;
    }
}
```

Listing 23. Klasa `MinimapFog`.

3. 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.

3.1. Grafika dwuwymiarowa

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

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

3.1.1. 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 niewielka klasa `CursorController`, która zajmuje się przemieszczaniem kursora po ekranie. Sam kursor reprezentuje umieszczony na *Canvasie*[47] obiekt gry *Cursor*, o ustawionej w panelu *Inspector* teksturze. Skrypt `CursorController` jest do niego przypisany.

Klasa `CursorController` zawiera funkcje `Start()` oraz `Update()`. Pierwsza z nich wyłącza podstawowy wskaźnik *Windows*. Druga ustala położenie naszego kursora. W tym celu pobiera aktualne współrzędne myszki z obiektu `InputController` i ustawia je jako pozycję w transformacji obiektu gry *Cursor*.

[47] Rozdział 3.1.2. Budowa interfejsów gracza, str. 59, fragment dotyczący *Canvas GUI*

```
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;
}
```

Listing 24. Ciało klasy `CursorController`.

3.1.2. Budowa interfejsów gracza

W związku z obecnie popularnym trendem *flat designu* postanowiliśmy wykorzystać go w naszej pracy. Mimo iż nie jest on spotykany w grach *RTS*, wygląda czysto i schludnie, a jego głównymi cechami jest minimalizm i prostota.

Unity udostępnia kilka sposobów zaprogramowania interfejsu gracza:

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

Interfejs *IMGUI* wykorzystano do stworzenia głównego menu. W tym celu stworzony został skrypt `MainMenuScript`, gdzie oprócz działania opisana została część wytycznych dotyczących wyglądu. Definiuje się go poprzez przypisanie tzw. „skórki" (ang. *skin*). Obiekty *skin* stanowią w *Unity* zestaw styli graficznych. Można zdefiniować go w edytorze i przypisać w panelu *Inspector* (oczywiście do tego skrypt musi mieć publiczne pole typu `GUISkin`). Przy wywoływaniu metod służących do pokazania odpowiednich kontrolek należy podać `string` będący nazwą stylu użytego w skórce.

Elementy interfejsu *IMGUI* są definiowane za pomocą kodu źródłowego. Przykładowo instrukcja `GUI.Label()` tworzy we wskazanych w parametrach współrzędnych ekranu etykietę, podczas gdy funkcja `GUI.Button()` powoduje wyświetlenie przycisku i dodatkowo zwraca rezultat typu `bool`. Jeśli jest on wartością wartość **`true`**, oznacza to, że w danym cyklu aktualizacji przycisk został naciśnięty — co pozwala zaimplementować obsługę tego zdarzenia.

Na przykład w funkcji `FirstMenu()` tworzony jest przycisk, którego rezultat **`true`** powoduje włączanie konkretnego podmenu. Przycisk ten ma określone wymiary, bazujące częściowo na pionowej rozdzielczości ekranu. Jednak już marginesy pomiędzy elementami a danym przyciskiem wpisane są na sztywno. Jest to bardzo problematyczne, ponieważ nie można od razu zobaczyć efektu pracy — człowiek nie potrafi wpisując np. liczbę pikseli od

lewej krawędzi wyobrazić sobie dokładnie, gdzie element *GUI* się znajdzie. Ponadto kontrolki nie zawsze dobrze się skalują przy zmianie rozdzielczości ekranu.

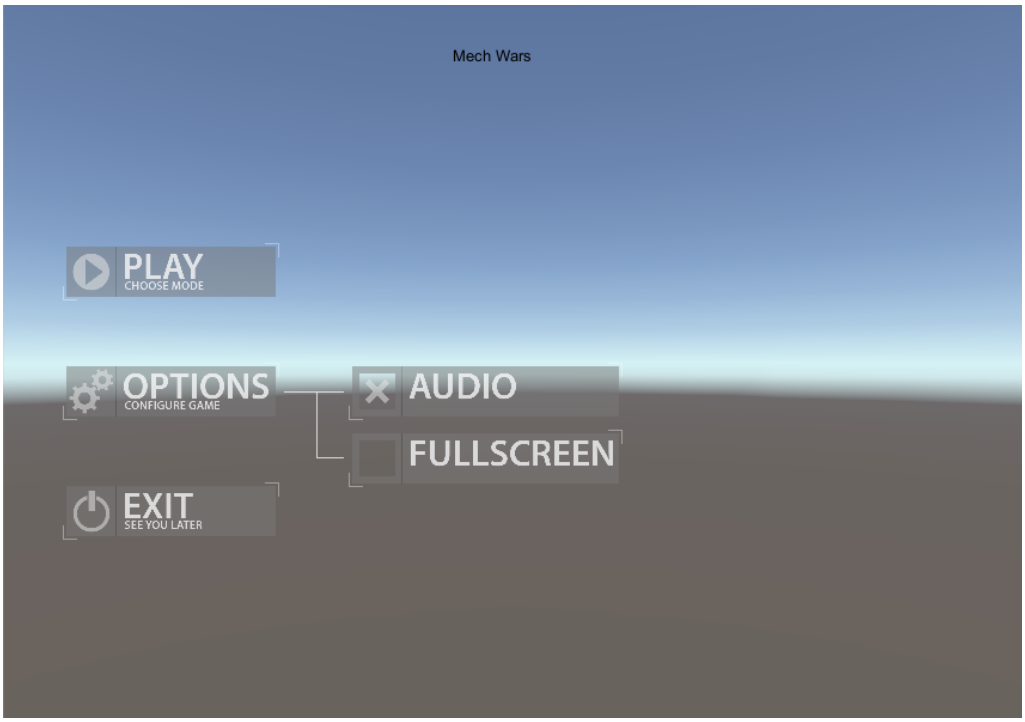
```
void OnGUI()
{
    GUI.skin = GameSkin;

    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" **//
}
```

Listing 25 Wybrane składowe klasy `MainMenuScript`.

Niektóre z przycisków w głównym menu po kliknięciu otwierają podmenu, co pozwala na wybór dodatkowych opcji. Poniżej wylistowana jest zbudowana w ten sposób struktura menu głównego:

- „**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 bitwy 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 (co nie działa — patrz niżej),
    - „**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 wizualizacji włączenia i wyłączenia, nie powoduje on faktycznego pojawienia się w prototypie dźwięków, gdyż po prostu prototyp takich dźwięków nie posiada.



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

Interfejs graficzny wyświetlany w czasie rozgrywki został stworzony przy pomocy *Canvas GUI*. W metodzie tej umieszcza się elementy interfejsu na specjalnym obiekcie gry o nazwie *Canvas*. Każdy z nich to po najczęściej po prostu dwuwymiarowy obrazek (tzw. *Sprite*). Ponieważ obrazki te są obiektami gry, to można je rozszerzać za pomocą poszczególnych komponentów, np. o funkcjonalność przycisku, lub pola tekstowego. Żeby projektant interfejsu nie musiał tych rzeczy konfigurować, *Unity* posiada szereg predefiniowanych kontrolek z ustalonymi zestawami komponentów. Warto jednak pamiętać o istnieniu tych komponentów, gdyż w panelu Inspector można przy ich pomocy w bardzo szeroki sposób skonfigurować elementy *GUI* — a także przypisać własne skrypty *C#*. Dużym ułatwieniem w tworzeniu tego typu interfejsu jest możliwość ustawienia hierarchii jego elementów (tak samo zresztą, jak w przypadku dowolnych obiektów gry). Dzięki temu podczas skalowania i przemieszczania obiektu nadrzędnego, jego dzieci będą zmieniać swój rozmiar i pozycję razem z nim (tak jakby stanowiły jedną całość). Można śmiało stwierdzić, że ten sposób tworzenia *GUI* jest znacznie bardziej wygodny i przystępny, jak również łatwiejszy w utrzymaniu od *IMGUI*.

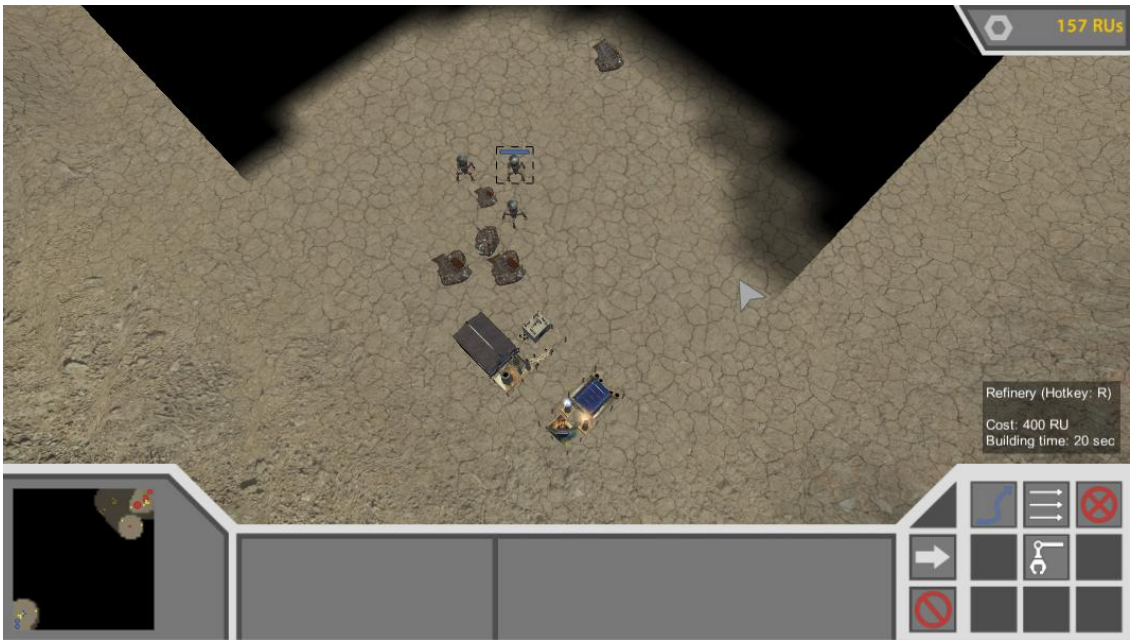
W interfejsie wyświetlanym podczas rozgrywki w prawym górnym rogu ekranu położony jest licznik zasobów, które aktualnie posiada gracz. Na dole natomiast umieszczono panel z najważniejszymi funkcjami. Po jego lewej stronie jest minimapa, natomiast po prawej znajdują się panel przycisków rozkazów,[48] którego zawartość zmienia

[48] Rozdział 2.8.1. Przyciski rozkazów, str. 51

się, w zależności od zaznaczonych elementów mapy. Stałymi przyciskami są anulowanie zaznaczenia (przekreślone, czerwone kółko), oraz przejście do następnej strony panelu (biała strzałka). Ten drugi został zrobiony na wypadek, gdyby przycisków istniało więcej, niż miejsc na panelu (czyli więcej, niż dziewięć). Później jednak okazało się, że takiej sytuacji nie ma. Dlatego przycisk ten nie wykonuje żadnej akcji.

Jeżeli na przykład gracz zaznaczy jednostkę *Harvester*, pokażą się cztery przyciski:[49]

- *Move*, po wskazaniu celu zaznaczona jednostka udaje się do wybranego przez gracza miejsca,
- *Escort*, jednostka eskortuje inną sojuszniczą jednostkę,
- *Stop*, zatrzymuje jednostkę (przerywa dowolny wykonywany przez nią obecnie rozkaz),
- *Harvest*, po wskazaniu konkretnego zasobu jednostka udaje się w tamto miejsce i zaczyna go zbierać.



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

3.2. Grafika trójwymiarowa

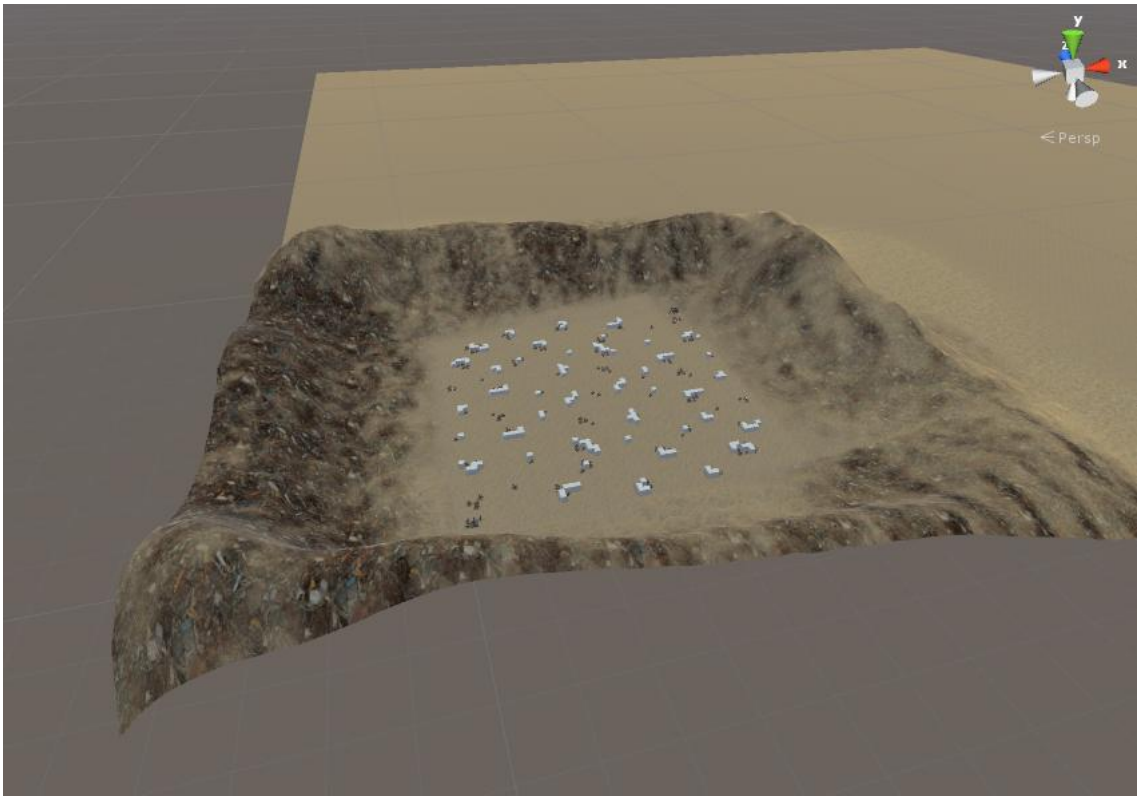
Większość dzisiejszych gier typu *RTS* wykorzystuje grafikę przestrzenną do przedstawienia świata, w którym toczy się rozgrywka. W naszym prototypie z grafiki trójwymiarowowej skorzystaliśmy w formie modeli terenu, zasobów, budynków i jednostek oraz systemu cząteczkowego dymu. Do stworzenia wszystkich tych elementów wykorzystaliśmy programy *Autodesk 3ds Max® 2016* (modele), *Adobe Photoshop* (tekstury) oraz *Unity* (teren).

[49] Dodatkowo po najechaniu na nie kursorem wyświetlą się ich opisy.

3.2.1. Modele

Obszar, na którym toczy się rozgrywka, wykonano wewnątrz *Unity* korzystając z tzw. obiektu *Terrain*. Jest to wbudowane w program narzędzie do modelowania terenu pozwalające dowolnie kształtować wzniesienia i doliny. Jednakże jednym z przyjętych uproszczeń prototypu jest płaska plansza — pozbawiona ukształtowania terenu w obszarze gry. Oczywiście teren w postaci płaskiego kwadrata psuje walory estetyczne, ponieważ gdy kamerę umieścić blisko jego krawędzi, widoczne jest czarne tło za nim. By temu zaradzić, stworzono większy teren i wypiętrzono obszar dookoła rzeczywistej planszy tworząc w ten sposób 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 *Unity* 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*

W skład dostępnych w prototypie **MechWars** trójwymiarowych budynków wchodzą:

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

Pomimo tego, że wszystkie budynki otoczono drucianym płotem-siatką, są one urozmaicone, dzięki czemu gracz może je z łatwością od siebie odróżnić. Każdy z nich, oprócz laboratorium, posiada kominy przemysłowe.

Ilustracja 5. Podgląd trójwymiarowego modelu warsztatu budowlanego.

Ilustracja 6. Podgląd trójwymiarowego modelu rafinerii złomu.

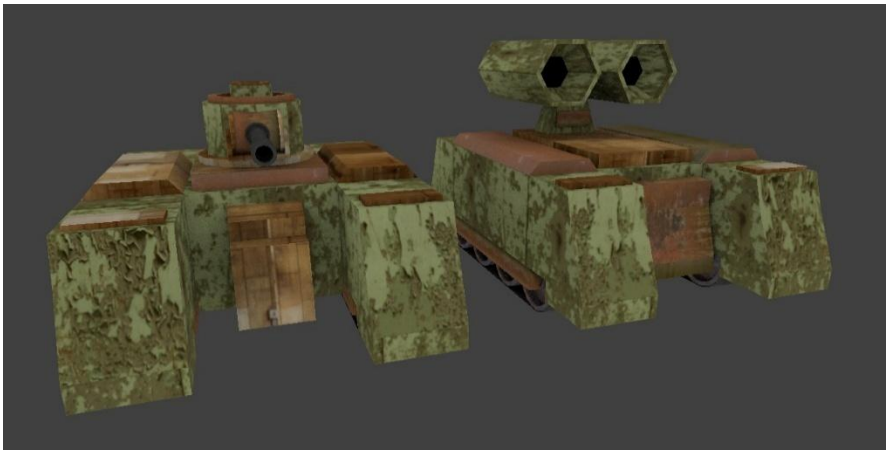
Ilustracja 7. Podgląd trójwymiarowego modelu fabryki jednostek.

Ilustracja 8. Podgląd trójwymiarowego modelu laboratorium.

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

Ilustracja 9. Podgląd trójwymiarowego modelu dalekozasięgowego mecha.

Ilustracja 10. Podgląd trójwymiarowego modelu bliskozasięgowego mecha.

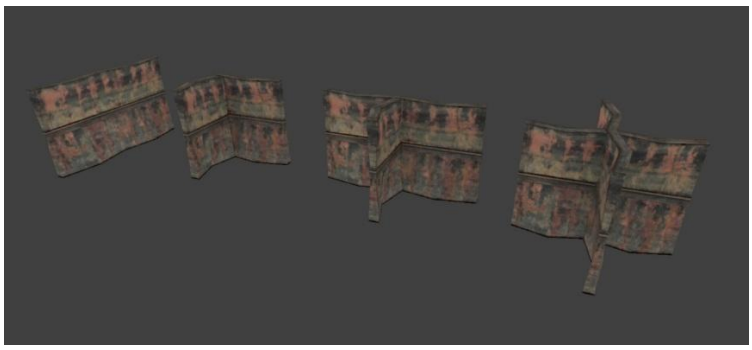


Ilustracja 11. Podgląd trójwymiarowych modeli czołgu i mobilnej wyrzutni rakiet.

Kolejnymi jednostkami są dwa ciężkie pojazdy opancerzone. Pierwszym z nich jest czołg, który posiada obrotową wieżyczkę. Zamontowano do niej wielkokalibrowe działo przeciwpancerne, z którego pociski mają dużą siłę rażenia. Drugą jednostką jest mobilna wyrzutnia rakiet z dwoma prowadnicami. Miota ona pociskami samonaprowadzającymi bardzo dalekiego zasięgu powodujące spustoszenie w miejscu trafienia. Oba modele są zbliżone do siebie wyglądem i poruszają się na chronionych blachami gąsienicach.

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

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 cywilną jest zwiadowca, który powinien się szybko poruszać, dlatego został skonstruowany z grubej opony opasanej dla wytrzymałości metalowymi obręczami oraz felg z kolcami utrzymującymi zwiększoną przyczepność. Dodatkowo zamiast korpusu posiada on jedynie kamerę na okrągłej głowicy. Dzięku takiemu zabiegowi może z łatwością zmieniać kąt patrzenia bez konieczności obracania się całym ‘ciałem’. Mimo wszystko w prototypie nie zostały zaimplementowane żadne animacje, zatem powyższe zachowania *Harvestera* i *Scouta* istnieją jedynie jako koncepcja.

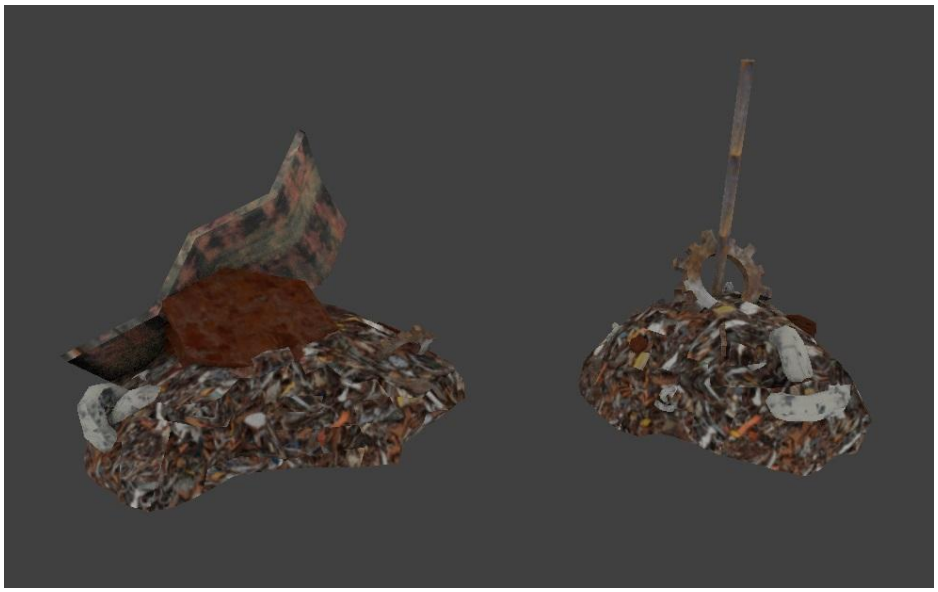


Ilustracja 13. Podgląd trójwymiarowych modeli murów.

Ilustracja 14. Podgląd trójwymiarowego modelu wieżyczki.

Ostatnimi trójwymiarowymi modelami są dwie, różniące się od siebie hałdy złomu stanowiące złoża surowców. Skrypt `MapElement` zawiera funkcję `TurnIntoResource()` wywoływaną przy niszczeniu jednostki, lub budynku. Generuje ona zasoby (wrak/zgliszcza), dla których model losowo wybiera spośród wspomnianych dwóch.

Dodatkowo do sceny zostały dodane przeszkody w postaci białych szcześcianów, będących jednym ze standardowych obiektów *Unity*.



Ilustracja 15. Podgląd trójwymiarowych modeli zasobów.

3.2.2. 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*. W silniku gry istnieje wbudowany komponent *ParticleSystem*, który służy do generowania cząsteczek. Można go przypisać do obiektu gry i skonfigurować pod kątem takich cech cząsteczek jak kolor, częstotliwości emisji, szybkości, typu, kąta obrotu oraz sam wygląd, będący własną teksturą.

Z systemu cząsteczek skorzystaliśmy poprzez stworzenie pustych obiektów gry, a następnie nadanie im jedynie komponentu *ParticleSystem*. Później obiekty umieściliśmy we współrzędnych, które miały być źródłem cząsteczek — na przykład w wylocie komina fabryki. Zgrupowaliśmy je dla każdego budynku, tworząc nowy obiekt gry, którego skonfigurowaliśmy jako ich rodzica. Do owego nadrzędnego obiektu gry przypisaliśmy skrypt zarządzający momentem rozpoczęcia ich emisji.

W skrypcie `ParticleGroup` przypisanym do obiektu-rodzica skorzystaliśmy z funkcji `Start()` i `Update()`. W pierwszej z nich znajduje się pętla zatrzymująca emisję wszystkich cząsteczek danej grupy. Druga natomiast odpowiada za rozpoczęcie generacji dymu dopiero, gdy budynek zostanie w całości wzniesiony (chyba że znajduje się na planszy od początku rozgrywki).

```
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();
        }
    }
}
```

Listing 26. Klasa `ParticleGroup`.

4. Zaprogramowanie sztucznej inteligencji

„Z teoretycznego punktu widzenia gra *RTS* jest bardzo odmienna od tradycyjnej gry planszowej, takiej jak szachy. Głównymi różnicami są:

- *RTS* posiada równoczesność ruchów — wielu graczy może wykonywać akcje 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 10^{50}, pokera *Texas Hold'em* w odmianie *no limit* — około 10^{80}, a *Go* — około 10^{170}. 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."[50]

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.

[50] http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 24.09.2016r [1]

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 tego typu 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 rekonesansie przeprowadzonym w celu ich poszukiwania. Jednak zaprezentowane w następnych rozdziałach rozwiązania omówionych powyżej wyzwań stanowią solidny fundament do zaprogramowania algorytmów radzących sobie z bardziej złożonymi problemami.

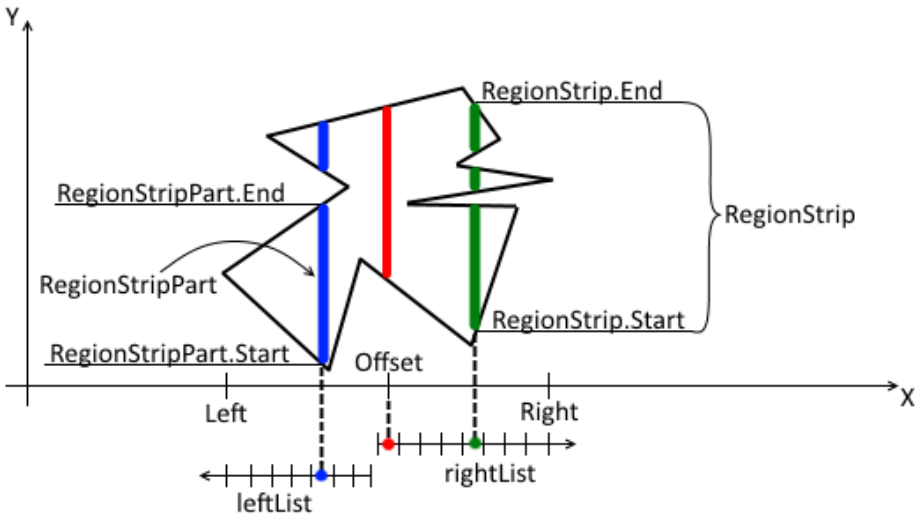
4.1. Mechanizmy pomocnicze

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.

4.1.1. Regiony

Pomysł na grupowanie pól planszy w większe obszary mające pewne znaczenie został zaczerpnięty z książki *Perełki programowania gier*.[51] 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.

Klasę `Region`, realizującą funkcjonalność obszarów, stworzono na zasadzie czegoś, co książka określa jako „pojemnik pudełka od zapałek". Mówiąc krótko, jest to dwustronna lista list przedziałów. Szkic poglądowy (**Rysunek 3**) wyjaśnia to zawiłe pojęcie.



Rysunek 3. Szkic poglądowy struktury danych `Region`

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

[51] *Perełki programowania gier. Vademecum profesjonalisty, Tom 3*, 2003, Daniel Higgins, *Rozdział 3.4: Analiza terenu w grach RTS – ukryta siła*, str. 321 [16]

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. 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`. Obszar, mający określony środek, powienien móc być rozszerzany w obie strony na osi X. Dlatego właśnie listy występują w parze. Wspomniany środek 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 (czyli cały region jest pusty).

Organizacja powyższej struktury danych w taki sposób służy przede wszystkim oszczędności pamięciowej. Tablica dwuwymiarowa zmiennych `bool` dla każdego pola 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()` oraz `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 byłoby 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 dla tych, które leżą na brzegu obszaru (czyli: 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.[52]

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 zawierają się w wielokącie. Funkcja `GetDistanceTo()` pozwala poznać odległość od określonego punktu, do granicy wielokąta. Wreszcie `GetPointClosestTo()` zwraca pozycję na brzegu wielokąta będącą 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

[52] https://en.wikipedia.org/wiki/Graham_scan#Algorithm, 24.09.2016r [13]

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, a jest to dosyć nietrywialne zagadnienie.

Istnieją wyspecjalizowane klasy implementujące interfejs `IRegionBatch`. Zostało to zaprogramowane na zasadzie wzorca projektowego *proxy*.[53] `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()`.

4.1.2. 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.

[53] http://www.dofactory.com/net/proxy-design-pattern, 24.09.2016r [10]

`BuildingRequirements` określa jakie budynki muszą znajdować się w bazie, a `TechnologyRequirements` — jakie technologie potrzeba opracować.

Dwie klasy: `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 ograniczeń czasowych. 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.

4.2. System wieloagentowy

Idea systemu wieloagentowego została zaczerpnięta z książki *Programming Game AI by Example.*[54] Stanowi ona rozwiązanie największego 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.

[54] Mat Buckland, *Programming Game AI by Example*, *Chapter 2: State-Driven Agent Design*, 2005, str. 43 [17]

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 zajmujące się 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*), którego 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.[55]

4.2.1. 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).

[55] Mat Buckland, *Programming Game AI by Example, Chapter 9: Goal-Driven Agent Behavior*, 2005, str. 379 [18]

```
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);
}
```

Listing 27. 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.

4.2.2. 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,[56] 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, a także, które pozwalają 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`. Jeśli w kolejce znajduje się jakaś nowa wiadomość, metodą `ReceiveMessage()` obiekty agentów potomnych mogą ją wyciągnąć, 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.[57] Dla przekazanej w parametrze akcji (`System.Action`, lub `System.Action<object>`) tworzony jest obiekt z tą akcją oraz 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 oraz, 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 zresztą byłoby błędem). Owo okresowe wywoływanie czynności można zatrzymać korzystając z odpowiedniej z dwóch bliźniaczych funkcji `StopPerform()`.

[56] Klasa `Message`, opisana w Rodziale 4.2.3. Wiadomości i żądania, str. 80
[57] Motywem dla tej funkcjonalności jest oczywiście wydajność.

```
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();
    }
}
```

Listing 28. 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 obsługą zadań agenta oraz resetowaniem informacji o tym, że akcje czasowe już wykonały się w tym cyklu. Metoda `Finish()` ustawia flagę `Finished`. Powoduje to usunięcie agenta ze zbioru w `AIBrain` po zakończeniu cyklu aktualizacji agentów.

4.2.3. 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 takimi, jak priorytet prośby, albo typ i liczba jednostek do wyprodukowania.

Dodatkowo obiekt `Message` posiada właściwość `InnerMessege`, w której może się znaleźć się pierwotna wiadomość (gdy obecna stanowi odpowiedź).

Większość agentów po otrzymaniu wiadomości zapisuje sobie ją 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 liczb całkowitych, ale, tylko wtedy gdy ma to sens w kontekście wiadomości (np. „wybuduj mi tutaj budynek").

4.2.4. 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*,[58] 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

[58] http://www.dofactory.com/net/template-method-design-pattern, 23.09.2016 [7]

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 to 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`[59] wykonywane przez jednostkę sprawia, że odkrywa ona niezbadany teren i zwiedza planszę, natomiast zadanie `HarvestGoal`[60] może być realizowane przez *Harvester*, by zbierać zasoby z planszy.

4.2.5. 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.

[59] Rozdział 4.2.8. Zadanie zgrubnego rekonesansu, str. 95
[60] Rozdział 4.2.11. Agent zbierający zasoby, str. 103

```
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);
}
```

Listing 29. 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` i woła funkcję `Finish()`, by usunięto go również ze zbioru w `AIBrain`.

4.2.6. Agent gromadzący wiedzę

Problem repozytorium wiedzy rozwiązano poprzez stworzenia agenta, który zbiera i udostępnia informacje na temat rozgrywki. Klasa `KnowledgeAgent` (dziedzicząca po `Agent`) zawiera stałą wiedzę — omówione już słowniki `MapElementKindDictionary` i `CreationMethodDictionary`.[61] 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; }
```

Listing 30. właściwości agenta `KnowledgeAgent`.

[61] Rozdział 4.1.2. Rodzaje elementów mapy, str. 75

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 zarówno pobranie wszystkich agentów jednostek danego rodzaju, jak i również otrzymanie agenta sterującego wybraną jednostką. `UnitAgentDictionary` słowniki przechowuje w polach prywatnych — udostępnia za to metody `Add()` i `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, gdy do tablicy `ResourcesKnowledge.resourceInfos` wstawiany jest nowy obiekt `ResourceInfo`, albo wartość **`null`**. 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 zarządzaniem 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()`, a jego promień określny jest jednym 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 więcej niż jeden, są one łączone 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);
}
```

Listing 31. Wybrane składowe klasy `ResourceKnowledge`.

Metoda `RemoveFromRegion()` jest prostsza — jedynie usuwa `ResourceInfo` i jego współrzędne 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 okazał się być 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);
}
```

Listing 32. 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()`.

```
void VisibilityTable_VisibilityChanged(IVector2 tile,
    Visibility from, Visibility to)
{
    if (to == Visibility.Visible)
    {
        var mapElement = MapProxy[tile];
        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);
        }
        else if (from == Visibility.Unknown)
            if (mapElement is Resource) 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]);
    }
}
```

Listing 33. Funkcje obsługi zdarzeń dotyczących zmiany widzialności oraz tworzenia i niszczenia elementu mapy w obrębie pola widzenia armii

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 niszczony jest element mapy w zasięgu widzenia. W metodzie `OnStart()` pod te zdarzenia podpinane są metody, które aktualizują obiekty `ResourceKnowledge` i `AllyBaseKnowledge`. Oprócz tego, również w `OnStart()`, dla każdej z już istniejących w czasie inicjalizacji jednostek tworzony jest agent i dodawany do słownika `UnitAgentDictionary`.

4.2.7. 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, gdy nie ma on już kratek ukrytych.

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 z nich, mającego nazwę „*Find me Resources*", wywołuje ona w pośredni sposób (za pomocą funkcji `PerformEvery()`) metodę `ProcessFindMeResources()` z 1 sekundą interwału. Żądanie z listy usunąć może tylko owa metoda — kiedy zakończy zadanie. Funkcja `PerformEvery()` będzie tu wołana co cykl aktualizacji, a jej efektem będzie wykonywanie `ProcessFindMeResources()` tylko co 1 sekundę. To konieczne, gdyż obsługa żądania szukania zasobów trwa dłuższy czas. W ten sposób zrealizowany został w 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);
}
```

Listing 34. 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 `ResourceCollectorAgent` nie potrzebuje zbieraczy złomu aż tak bardzo, jak `ReconAgent` zwiadowców, ten drugi może wysłać do pierwszego wiadomość: prośbę o przekazanie agenta. Dzieje się to szczególnie, gdy nie ma dostępnych żadnych lepszych jednostek nadających się do zwiadu.[62] Mija jednak pewien czas, zanim `ResourceCollectorAgent` przetworzy zapytanie i zastosuje się do niego. Dlatego dla każdego przetwarzanego żądania `ReconAgent` posiada w słowniku obiekt typu `RequestUnitAgentSet` — specjalną, potrójną strukturę danych do przechowywania agentów jednostek oddelegowanych do żądania. Wewnątrz niej istnieją 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 parametrów ustawionych na sztywno w kodzie. 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` określa liczby jednostek zwiadowczych, którym `ReconAgent` ma przydzielić zadanie rekonesansu, a `scoutsImportanceByPriority`: mieszczące się między 0 a 1 ważności tego zadania.

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

[62] Podobna sytuacja mogłaby wystąpić pomiędzy dowolnymi dwoma agentami, więc mechanizm jak najbardziej nadaje się do rozszerzenia na więcej przypadków.

```
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 };
```

Listing 35. Metoda `ProcessFindMeResources()` — część 1 z 3.

```
// 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;
```

Listing 36. Metoda `ProcessFindMeResources()` — część 2 z 3.

```
    // 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();
    }
}
```

Listing 37. Metoda `ProcessFindMeResources()` — część 3 z 3.

jest wiadomość „*Produce me Units*" do agenta `ProductionAgent` — prośba o produkcję jednostek *Scout* — i na tym funkcja 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`.[63] 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.[64] 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), która określa, na ile właściwym jest wzięcie czyjegoś agenta. Jej wartość 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:

$$S(i, p) = \min(1 - i,\ p),$$

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

Lista par jest posortowana malejąco po ich `Suitability`. Dzięki temu w następnym kroku można wybrać najlepiej pasujących agentów. Jeśli nie ma więcej dostępnych,

[63] Rozdział 4.2.8. Zadanie zgrubnego rekonesansu, str. 95
[64] Może tak się stać w momencie, w którym właśnie zmniejszono priorytet zadania, a więc spadła liczba potrzebnych jednostek zwiadowczych.

nadających się jednostek, do ProductionAgent znów wysyłana jest prośba o ich produkcję. Dopóki jednak lista zawiera agenty, ReconAgent bierze je i daje im zadanie CoarseRegionGoal (z tablicy scoutsImportanceByPriority przypisuje im ważność parametru), lub (jeśli agent już jest zajęty) wysyła wiadomość do jego właściciela z prośbą o przekazanie. Służy do tego funkcja TakeAgentNowOrLater(), która dodatkowo umieszcza agenta jednostki w zbiorze ReconUnitAgentSet związanym z przetwarzanym żądaniem. Gdy zajęty agent nie może zostać oddany natychmiast trafi do zbioru Awaiting, a zadanie otrzyma dopiero w momencie przekazania go i przepisania do 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 zwiadu z ważnością drugiego zadania (tego, które agent proszący

chce jednostce dać). W ogólności samą ważność zwiadu można też uzależnić od ilości widocznych regionów zasobów, a także od ich rozmiarów. Podsumowując, występują przy przetwarzaniu żądania rekonesansu aspekty, które bez wątpienia warto ulepszyć, lecz jednocześnie istnieją i narzędzia, za pomocą których owe ulepszenia da się zrealizować.

4.2.8. Zadanie zgrubnego rekonesansu

Klasa `CoarseRegionGoal` realizuje pobieżny zwiad całej planszy gry. Owa zgrubność zależy od wartości parametru *AI*: pola `AIBrain.coarseReconPercentage`, o domyślnej wartości 90%. Jest to procent odkrytych pól mapy wystarczająco dobry dla algorytmu.

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. [65] 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 ciężkości 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). Sama klasa `ReconRegionOrderCalculator` jest abstrakcyjna. Jej potomkowie określają treść wspomnianej 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*.[66]

[65] Rozdział 4.2.7. Agent odpowiedzialny za zwiady, str. 88
[66] Rozdział 5.2. Badanie skuteczności rekonesansu, str. 118

```
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();
}
```

Listing 38. Treści metod `OnUpdate()` i `TakeNextReconRegion()` klasy `CoarseReconRegion`.

4.2.9. Agent rozbudowujący bazę

Obiekt `ConstructionAgent` w podobny sposób, jak 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.

Funkcji `ProcessRequests()` przetwarza 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`.

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.

```
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);
```

Listing 39. Algorytm obsługi żądania *„Construct me Building”* w klasie `ConstructionAgent`.

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.[67] 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 wykonuje się 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 samego 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()`,[68] 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

[67] Koszt rozpoczęcia konstrukcji skonfigurowano w prototypie jako 10% pełnego kosztu budynku.
[68] Sposób funkcjonowania tej metody nie został omówiony.

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. Wszystkie pozycje jednak wybierane są również pod kątem tego, by budynki oddzielone były od siebie przejściem o szerokości jednego pola.

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` dodaje się 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).

4.2.10. Agent produkcyjny

Klasa `ProductionAgent` wykazuje wiele podobieństw do agenta konstrukcji budynków.[69] Ich struktura jest tak bardzo do siebie zbliżona, że gdyby rozbudowywać projekt, prawdopodobnie zostałaby wyciągnięta do osobnej klasy. Oba agenty posiadają listę żądań, oba 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.

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 tworzeniem nie zajmuje. Poza tym ustawiana jest flaga `dontFinish` (która działa tu identycznie, jak w klasie `ConstructionAgent`).

Gdy jednak 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.

[69] Rozdział 4.2.9. Agent rozbudowujący bazę, str. 97

```
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);
```

Listing 40. Algorytm obsługi żądania *„Produce me Unit”* w klasie `ProductionAgent`.

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 surowców, 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.

4.2.11. 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.

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.

```
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];
}
```

Listing 41. Wybrane składniki klasy `ResourceCollectorAgent`.

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 zaprogramowano to tak, żeby, aby w *n*-tej minucie armia miała do dyspozycji *n 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: 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`[70] znajduje się instrukcja pobierająca wszystkie jednostki zdolne do rekonesansu. Dzięki ważności ich zadań określa, 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

[70] Rozdział 4.2.7 Agent odpowiedzialny za zwiady, str. 88

funkcji ciągłej. Dziedzina tej funkcji to nieujemny zbiór $\mathbb{R}^2$, a przeciwdziedziną jest przedział zamknięty $(0, 1)$. Funkcja zdefiniowana jest 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 jako jeden z parametrów obiektu `AIBrain`, czyli publiczne pole `TextAsset harvestingImportanceFunction` reprezentujące plik tekstowy. Wpływ kształtu funkcji na wymianę jednostek między agentami miał zostać poddany eksperymentom w rozdziale n.t. badań,[71] jednak nie zrobiono tego z powodu ograniczeń czasowych. Zamiast tego w aneksie zawarto treść domyślnie używanej macierzy definiującej przyjętą funkcję.[72]

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.

[71] Rozdział 5. Badania skuteczności *AI*, str. 108
[72] Aneks, B. Treść domyślnej funkcji ważności zadania zbierania zasobów, str. 140

Da się z tego wywnioskować, że agenty dzielą 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.

4.2.12. Zadanie zbierania zasobów

Klasa `HarvestGoal` jest prostym zadaniem, które wybiera region do zbierania zasobów oraz wydawaje *Harvesterowi* rozkazy. Posiada ono publiczną właściwość `HarvestedRegion` i prywatne pole `currentResource` (obecnie wybrany zasób).

W metodzie `OnUpdate()` wpierw przy pomocy funkcji `PickHarvestedRegion()` wybierany jest region zasobów. 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.[73]

```
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
    {
        if (currentResource != (HarvestOrder)currentOrder.Resource)
        {
            currentResource = (HarvestOrder)currentOrder.Resource;
            if (!HarvestedRegion.HasResource(currentResource))
                GiveNewOrder();
        }
    }
}
```

Listing 42. Treść metody `HarvestGoal.OnUpdate()`.

[73] Oczywiście stwierdzenie sprzed kilku zdań: „zasób wybrany przez ten rozkaz" jest prawidłowe; gdy zasób zostanie zebrany, rozkaz samodzielnie zajmuje się wyszukaniem następnego, najbliższego w zasięgu widzenia.

5. Badania skuteczności *AI*

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. Jednak 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*, a także zależność przekazywania agentów jednostek między agentami głównymi od kształtu funkcji ważności zadania zbierania zasobów, niestety nie wykonaliśmy tego z powodu ograniczeń czasowych.

5.1. Badanie przebiegu domyślnej rozgrywki

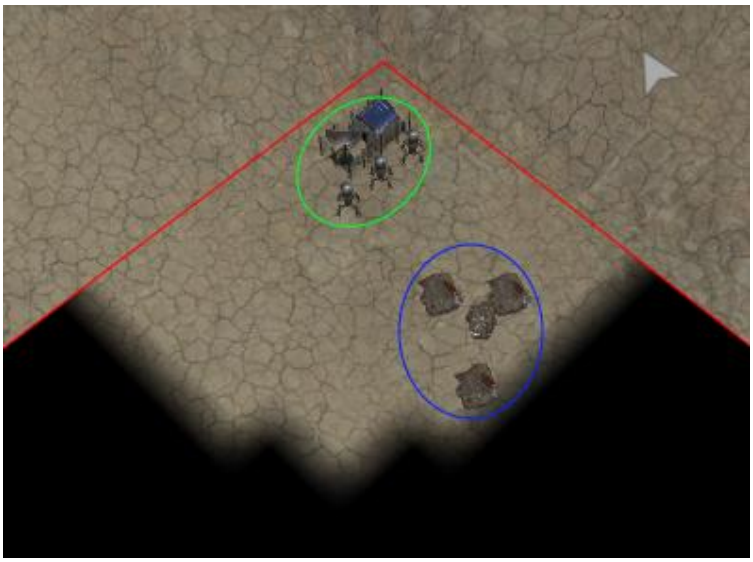
Ponieważ sztuczna inteligencja ma zaprogramowane pewne domyślne zachowania, nie można nie sprawdzić, jak dla nich wygląda przykładowy przebieg gry. Oczywiście nie jest to pełna rozgrywka — nie może być mowy o żadnych bitwach, opracowywaniu technologii czy nawet produkcji jednostek bojowych, a co dopiero 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.

5.1.1. 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 jej polu widzenia znajduje się niewielki region surowców, a do wydania ma 500 jednostek zasobów (w skrócie: *RU*, ang. *resource units*). Armia startuje z górnego narożnika symetrycznej planszy (**Ilustracja 16**). Sama

sztuczna inteligencja (obiekt `AIBrain`) zdefiniowana jest parametrami o następujących wartościach:

- `resourceRegionDistance` = 2,
- `harvestingImportanceFunction`: plik *StandardHarvestingImportance*,[74]
- `reconRegionSize` = 4,
- `coarseReconPercentage` = 90,
- `reconRegionOrderCalculator`: prefab *BaseSelfProductAndTilesExploredOrderCalculator*.[75]



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

5.1.2. Przebieg i analiza symulacji

Poniżej zaprezentowano wyniki obserwacji przebiegu rozgrywki w porządku chronologicznym. Kolejne zdarzenia opatrzone są czasami zajścia. Poniżej listy znajdują się również ilustracje pokazujące sytuację gry w wybranych momentach czasowych.

- **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*,

[74] Aneks, B. Treść domyślnej funkcji ważności zadania zbierania zasobów, str. 140
[75] Rozdział 5.2.5. Sortowanie według iloczynu oraz odsetka zbadanych pól, str. 125

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* eksploruje 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*.

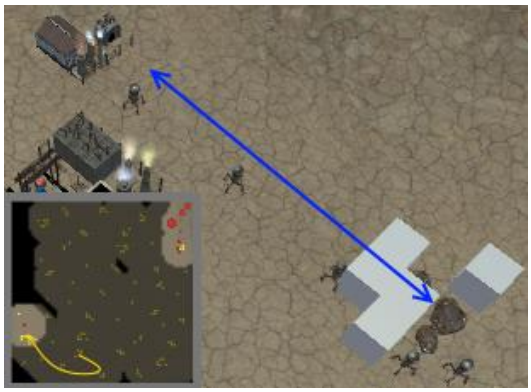
Ilustracja 22. Przebieg domyślnej rozgrywki, czas: 1:00. Żółta strzałka to zwiad, niebieska — powrót *Harvestera*.

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.



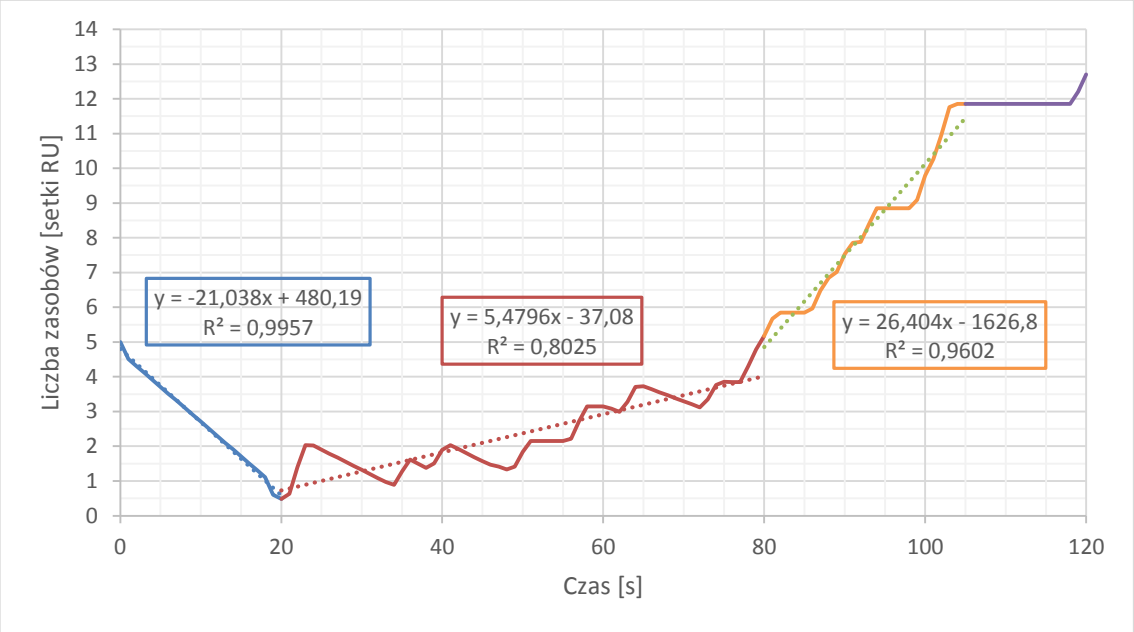
Ilustracja 27. Przebieg domyślnej rozgrywki, czas: 2:38 Żółta strzałka to trasa zwiadu i miejsce zatrzymania się *Scouta*, niebieska strzałka — trasa rafineria-zasoby.

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 jedynie w odniesieniu 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.

5.1.3. 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 teraz gdzie 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 zatem jedynie z nieciągłości 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**[76] wzrost średniego przychodu *RU* w czasie.

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

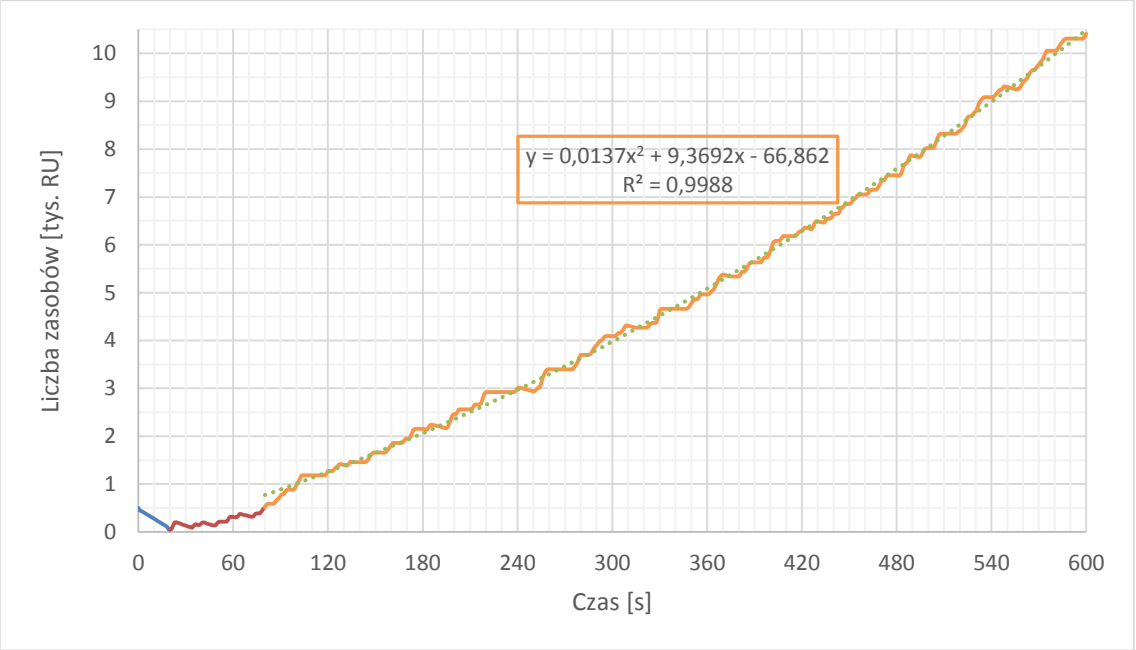
$$R_{20} = 48,$$
$$R_{80} = 517,$$
$$\Delta R_b = R_{80} - R_{20} = 469,$$
$$\Delta R = \Delta R_b + C_F + C_H + C_S = 469 + 350 + 130 + 35 = 984.$$

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

[76] Stosunek czynników liniowych regresji wynosi ~4.81.

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*, kursowanie po surowce może im zajmować więcej czasu. Hipoteza ta jednak nie została sprawdzona.



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

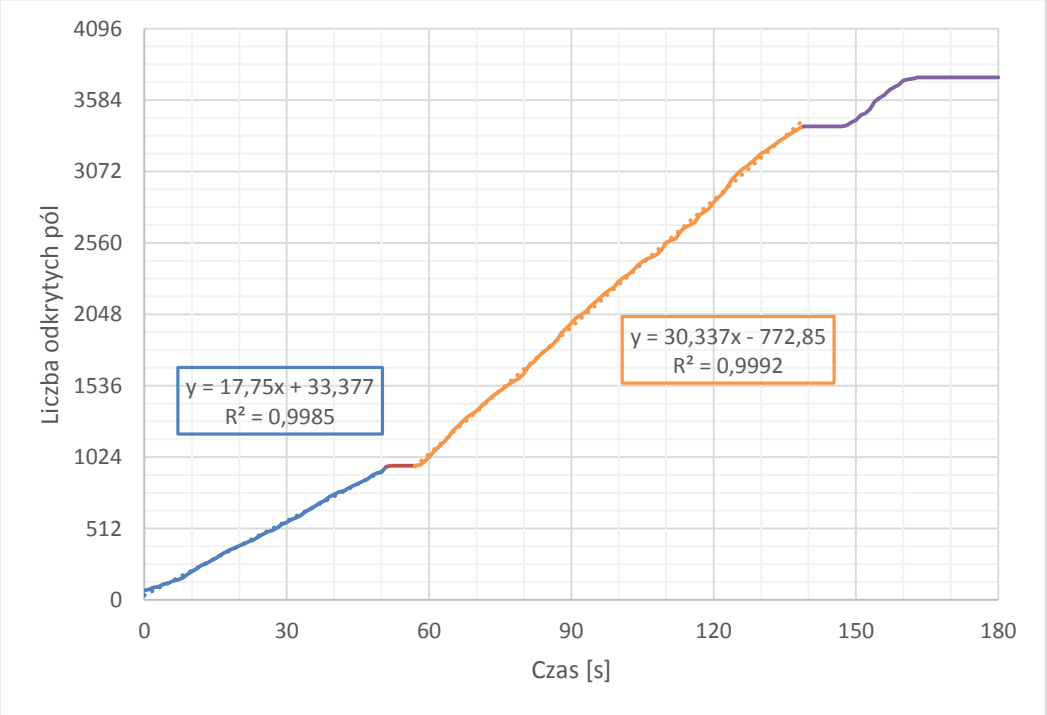
Przyrost zasobów w całym 10-minutowym odcinku został pokazany na **Wykres 2**. Wydawać by się mogło, że w dłuższym odstępie czasu wzrost zasobów spełnia zależność kwadratową. Wynika to na pewno ze stałego przyrostu *Harvesterów* (produkowane są co minutę) — każdy z nich zwiększa pochodną *RU* po czasie o zbliżoną wartość. Regresja również tutaj pasuje, z bardzo wysokim czynnikiem korelacji. Ale jednak, jeśli przeprowadzić głębsze rozważania teoretyczne, można dojść do wniosku, że kwadratowy przyrost nie jest 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 $\frac{\pi r}{2}$, gdzie r — dystans ćwiartki okręgu do rafinerii. Niech r_0 będzie maksymalnym r, dla którego nie ma zasobów bliżej bazy. Im większe r_0, tym więcej zasobów znajduje się dopiero na ćwiartce okręgu i tym dłużej zajmuje zebranie ich z tej ćwiartki. Poza tym, *Harvestery* muszą pokonać dystans r_0 między regionem a rafinerią. Ponieważ są to dwa

niezależne czynniki **liniowe**, okres T na zebranie regionów z ćwiartki okręgu rośnie **kwadratowo** od r_0. Stały przyrost *Harvesterów* w czasie bilansuje wzrost T tak, że jest ono **liniowe**. Całkowity zaś czas potrzebny do wyczerpania regionów ze środka ćwiartki jest całką okresów dla każdej mniejszej ćwiartki okręgu. Skoro T rośnie liniowo od r_0, to jego całka jest kwadratowa, więc funkcja odwrotna, jakiej szukamy jest **pierwiastkiem**. Jeśli dystans rośnie tak, jak pierwiastek całkowitego czasu t_C rozgrywki, to tak samo rośnie czas t_k trwania pojedynczego kursu między regionem zasobów, a rafinerią. Prędkość kursowania, czyli liczba wykonanych przez *Harvester* kursów w jednostce czasu od t_C, to funkcja typu $\frac{1}{\sqrt{t_C}}$. Mnożąc ją przez t^2 wynikające ze stałego tempa produkcji *Harvesterów* otrzymujemy prawdziwą zależność, której nie dało się zobaczyć na wykresie: $t^{\frac{3}{2}}$. Prawdopodobnie 10 minut było po prostu za małym czasem gry, by to pokazać.

Warto zaznaczyć, że powyższe rozważanie przestaje być prawdziwe, gdy r_0 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.

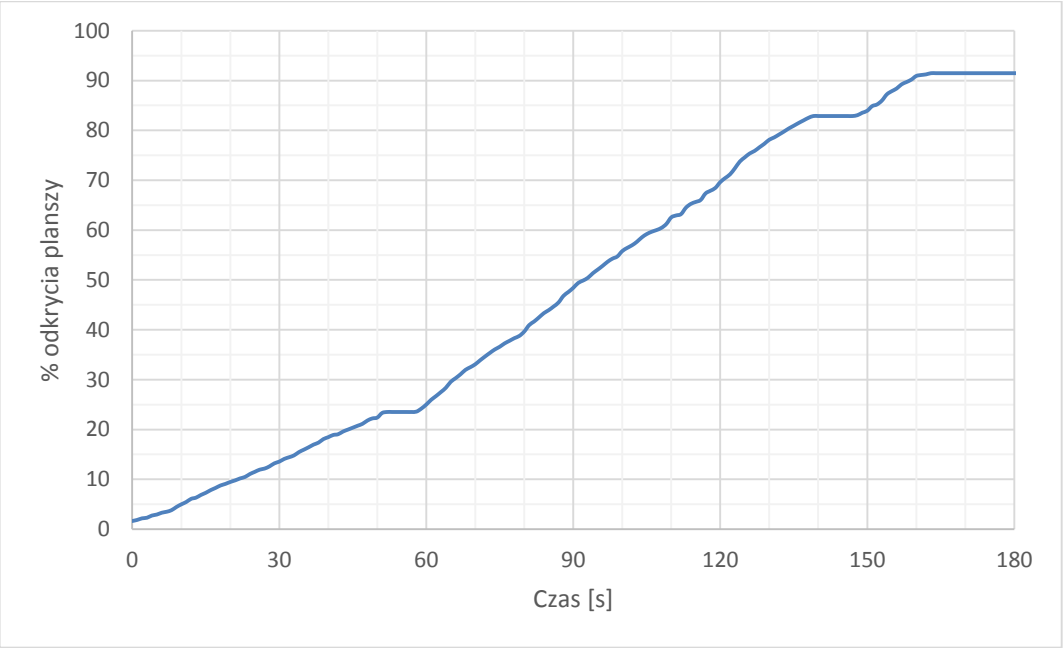
Teraz przejdziemy do krótkiego omówienia wyników pomiaru eksploracji mapy.



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

Oś pionowa **Wykres 3** jest opisana potęgami dwójki, ponieważ długość boku planszy to 64, zatem liczba wszystkich pól do zwiedzenia wynosi $64^2 = 4096$. 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 **Wykres 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 $1.6 \cdot 1.4 = 2.24$ 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.

5.2. Badanie skuteczności rekonesansu

Rozdział 4.2.8 [77] wspomnina 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 czynniki takie, 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 postulować wersje funkcji `Calculate()`, testować je i wyciągać wnioski o tym, co można poprawić. Wprowadzone korekty znów będą poddane testom, ich wyniki analizie itd. Proces powtórzymy kilkukrotnie, aż otrzymamy zadowalający nas algorytm (*notabene* skorzystano z niego przy domyślnej rozgrywce[78]).

5.2.1. 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*,[79]
  - `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.

[77] Rozdział 4.2.8. Zadanie zgrubnego rekonesansu, str. 95
[78] Rozdział 5.1 Badanie przebiegu domyślnej rozgrywki, str. 108
[79] Aneks, B. Treść domyślnej funkcji ważności zadania zbierania zasobów, str. 140

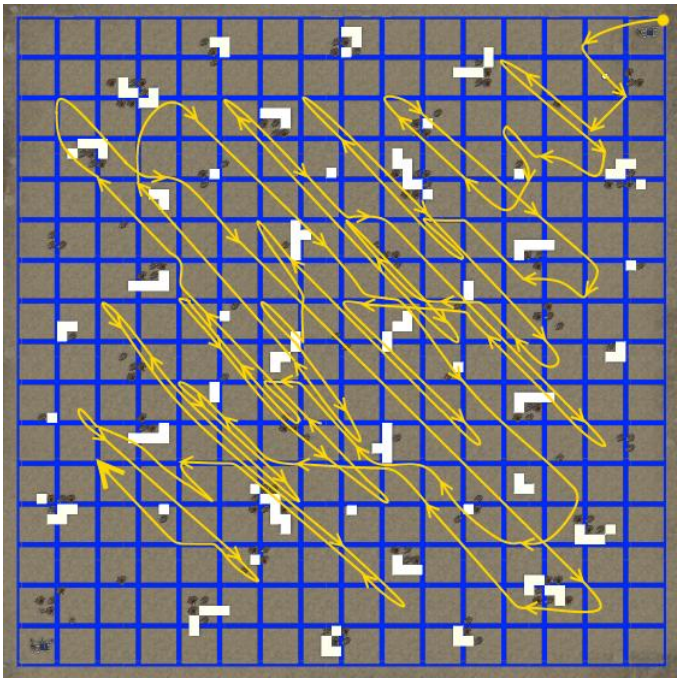
5.2.2. Sortowanie według odległości do bazy

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;
    }
}
```

Listing 43. Klasa `BaseOnlyOrderCalculator`.

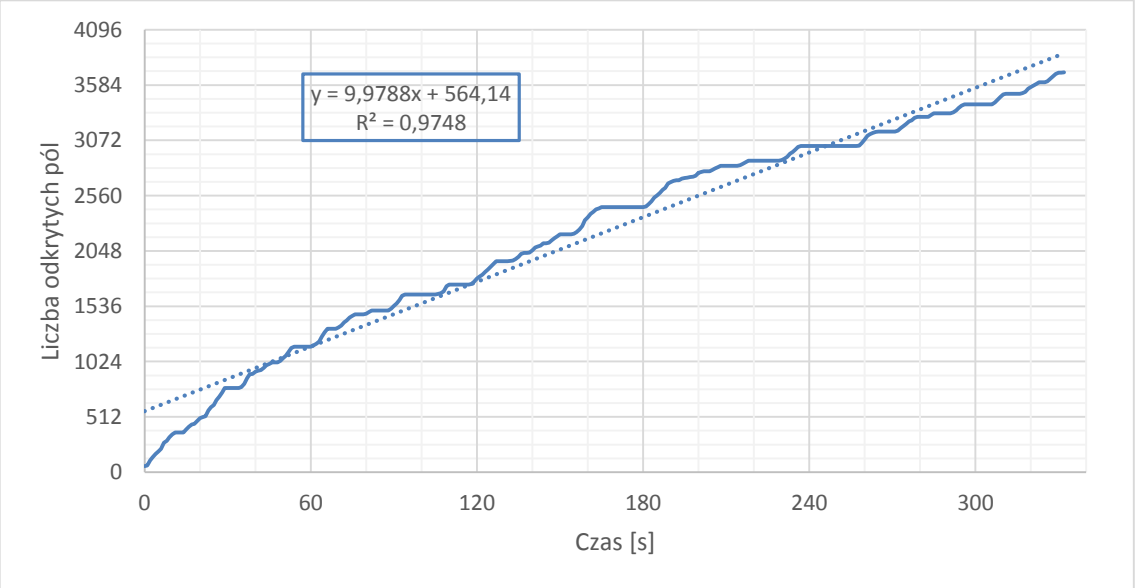
Treść funkcji w klasie `BaseOnlyOrderCalculator` pokazuje, że tak naprawdę kryterium jest dystans między **środkiem** regionu bazy, a **środkiem** regionu do zwiedzenia. Można zauważyć więc, że funkcja zacznie zwracać odrobinę inną wartość, gdy dostawione zostaną nowe budynki. Sytuacja ta jednak nie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



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

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 28** doskonale pokazuje błądzenie zwiadowcy. W efekcie ukończył rekonesans dopiero po ok. 5 i pół minuty.

Na **Wykres 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.



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

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.[80] Wydawać by się mogło, że optymalnym czasem na zwiedzenie planszy jest tu

[80] Rozdział 5.1.3 Analiza przyrostu zasobów i odkrytego terenu, str. 113

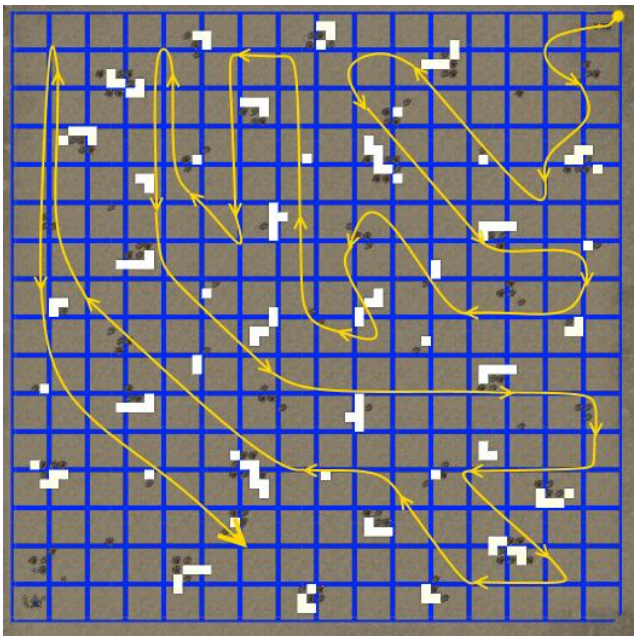
190 sekund (332 – 142), ale późniejsze eksperymenty pokazały, że da się uzyskać znacznie lepszy wynik.

5.2.3. Sortowanie według sumy odległości do bazy i zwiadowcy

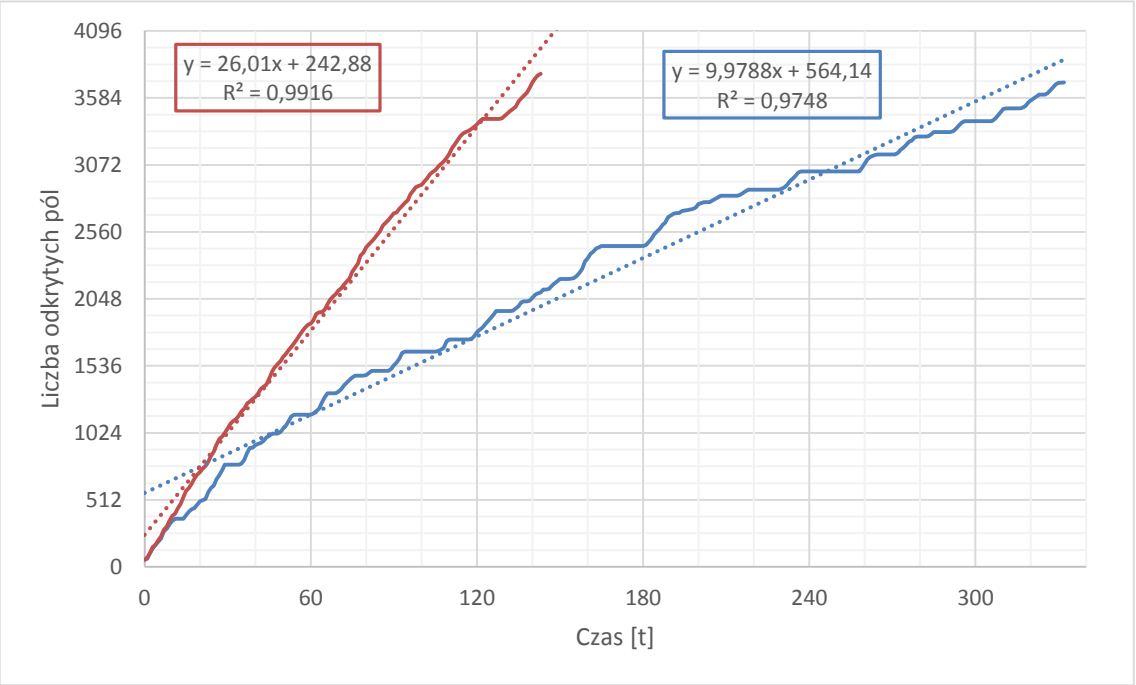
Problem poprzedniego sposobu polega na tym, że algorytm bezkrytycznie pobiera regiony euklidesowo najbliższe bazie. Wiadomo, że nad regiony po przekątnej od bazy będą preferowane te, których pozycja różni się od położenia bazy tylko jedną współrzędną. Powoduje to kursowanie *Scouta* od jednego boku mapy 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;
    }
}
```

Listing 44. Klasa `BaseSelfSumOrderCalculator`.



Ilustracja 29. algorytm `BaseSelfSumOrderCalculator`.
Żółta linia: trasa zwiadowcy, niebieska siatka: regiony zwiedzania.
Zwiadowca wyraźnie preferuje chodzenie po koncentrycznych okręgach dookoła bazy.



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

Wyniki eksperymentu (**Wykres 6** i **Ilustracja 29**) pokazują znaczną poprawę. Zastosowanie tak prostej modyfikacji znacznie zwiększyło tempo wykonywania zwiadu. Nie zaobserwowano również, żeby zwiadowca często wracał po swoich śladach. Jego trasa bardziej przypominała koncentryczne ćwierć-okręgi: gdy dojechał do krawędzi mapy, oddalał się od bazy i wracał odkrywając dalej.

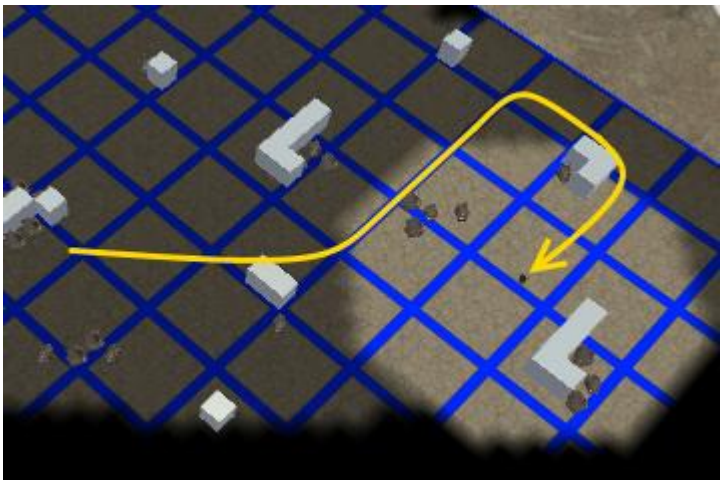
Na wykresie wyraźnie widać, że:

- 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.

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 30**. *Scout* ma na tyle duże pole widzenia, że obejmuje nim obszar o średnicy trzech i pół regionów. Dlatego powinien odchodzić na większą odległość od ścieżki już przez siebie pokonanej. W późniejszych eksperymentach spróbowaliśmy na to coś zaradzić.



Ilustracja 30. Płytki zwiad nieznanych regionów

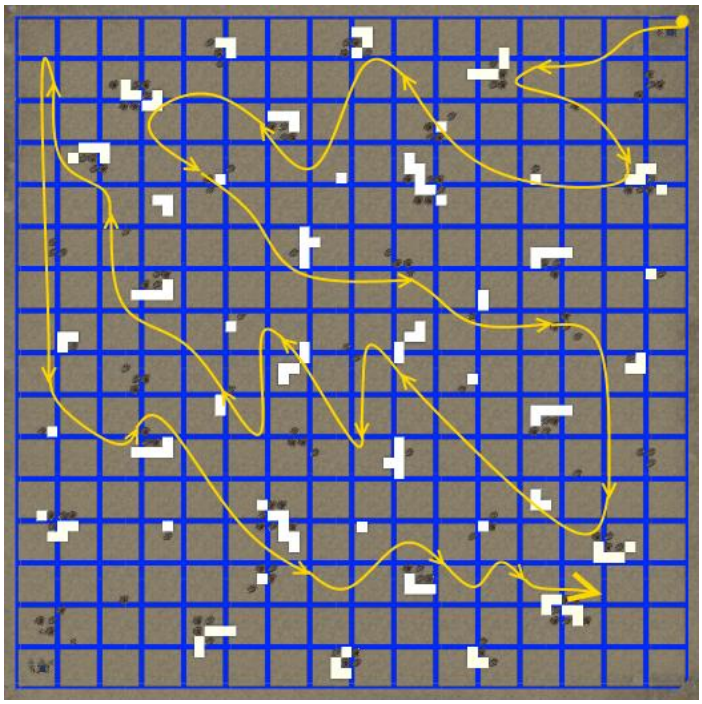
5.2.4. Sortowanie według iloczynu odległości do bazy i zwiadowcy

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.

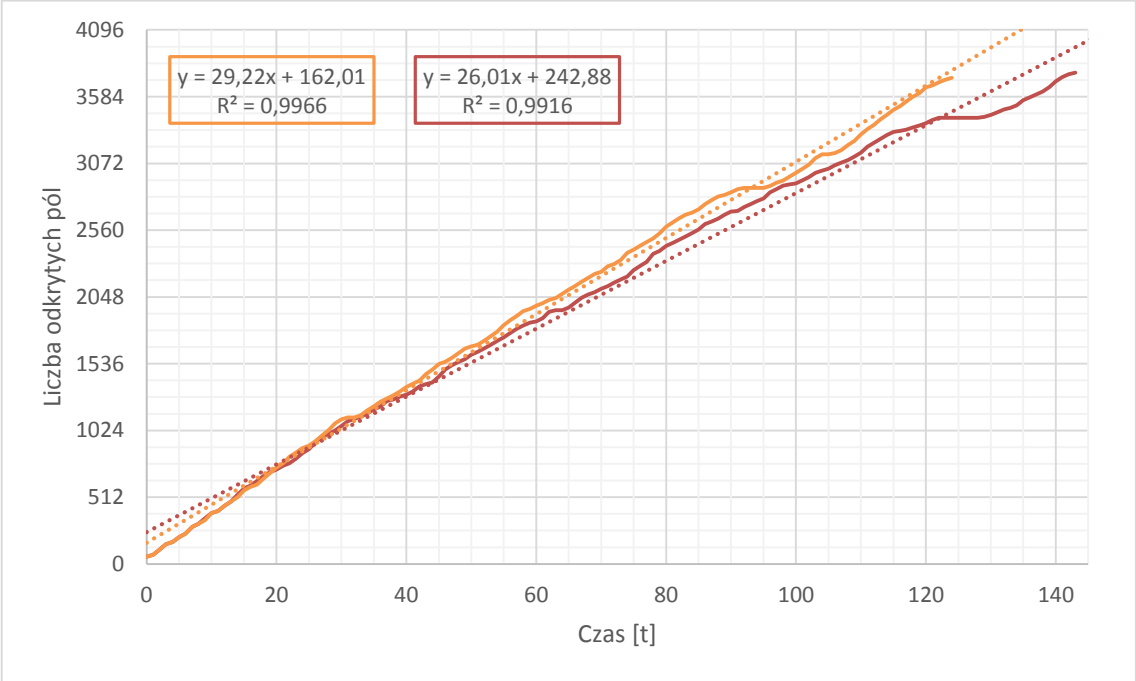
```
public class BaseSelfProductOrderCalculator : ReconRegionOrderCalculator
{
    public override float Calculate(
        CoarseReconGoal goal, ReconRegionBatch region)
    {
        var baseReg = goal.Agent.Knowledge.AllyBase.BaseRegion;
        var dToBase = region.ConvexHull.Center - baseReg.ConvexHull.Center;
        var distToBase = dToBase.magnitude;
        var dToSelf = region.ConvexHull.Center - goal.UnitAgent.Unit.Coords;
        var distToSelf = dToSelf.magnitude;
        return distToBase * distToSelf;
    }
}
```

Listing 45. 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 31. algorytm `BaseSelfProductOrderCalculator`.
Żółta linia: trasa zwiadowcy, niebieska siatka: regiony zwiedzania.
Skaut za bardzo oddala się od bazy na zbyt wczesnym etapie.

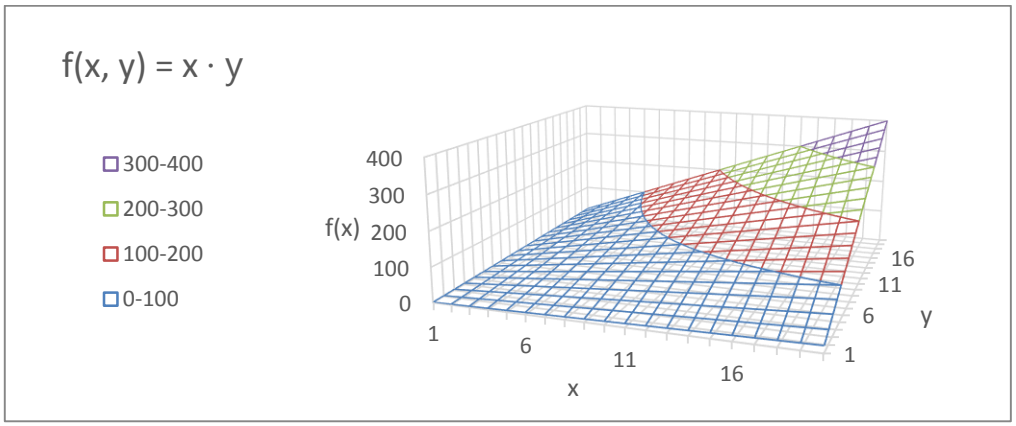


Wykres 7. Porównanie przyrostu odkrytych pól dla algorytmów:
`BaseSelfProductOrderCalculator` (linia pomarańczowa)
oraz `BaseSelfSumOrderCalculator` (linia czerwona).

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 $f(x, y) = xy$, na dziedzinach (1, 20).



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

5.2.5. 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,[81] co musiało rzutować na skuteczność zwiadu. Stąd otrzymany tam niski stosunek szybkości odkrywania *Scouta* do *Harvestera* przestaje dziwić.[82] W tym rozdziale przytaczamy jedynie prawidłowy wariant algorytmu.

Klasa `BaseSelfProductAndTilesExploredOrderCalculator` jest niewątpliwie bardziej złożona od poprzednio przytaczanych. Pierwszych sześć linijek metody `Calculate()` wykonuje dokładnie to samo, co cała metoda `Calculate()` w klasie

[81] Rozdział 5.1 Badanie przebiegu domyślnej rozgrywki, str. 108
[82] Rozdział 5.1.3 Analiza przyrostu zasobów i odkrytego terenu, str. 113

`BaseSelfProductOrderCalculator`. Na tym jednak podobieństwa się kończą. Aby móc policzyć ile pól zostanie odkryte, gdy 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`.

```
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);
    }
}
```

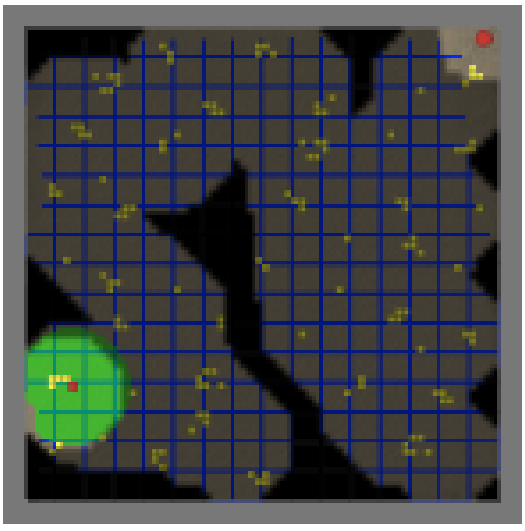
Listing 46. Klasa `BaseSelfProductAndTilesExploredOrderCalculator`.

Tworzone są zmienne-akumulatory `tilesInShape` oraz `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. Zmienna `tilesExploredInShape` natomiast inkrementuje się, gdy kratka jest wewnątrz planszy i nie jest nieznana. Po zakończeniu pętli liczby dzielone są one 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* ich nie rozróżniał, więc 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 32** przedstawia zrzut ekranu minimapy po pierwszych testach, na których pokazany jest ten problem.



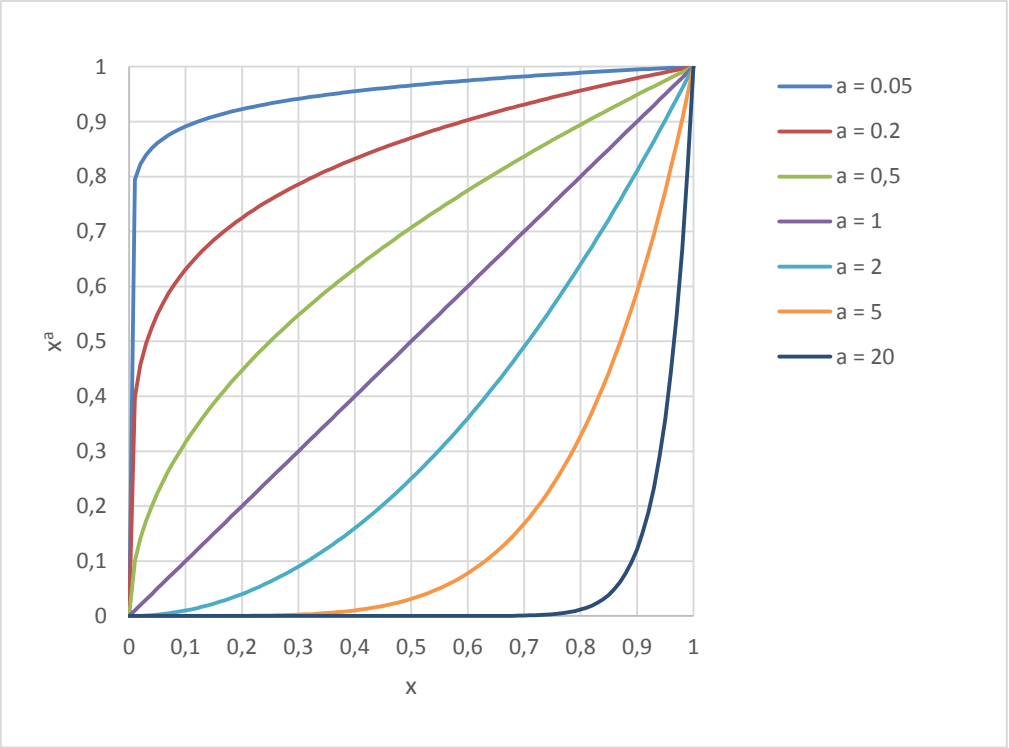
Ilustracja 32. Zwiad przy czystym mnożeniu przez `factor` jest zdecydowanie zbyt niedokładny

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 x^a ma następujące własności (które można zaobserwować na **Wykres 9**):

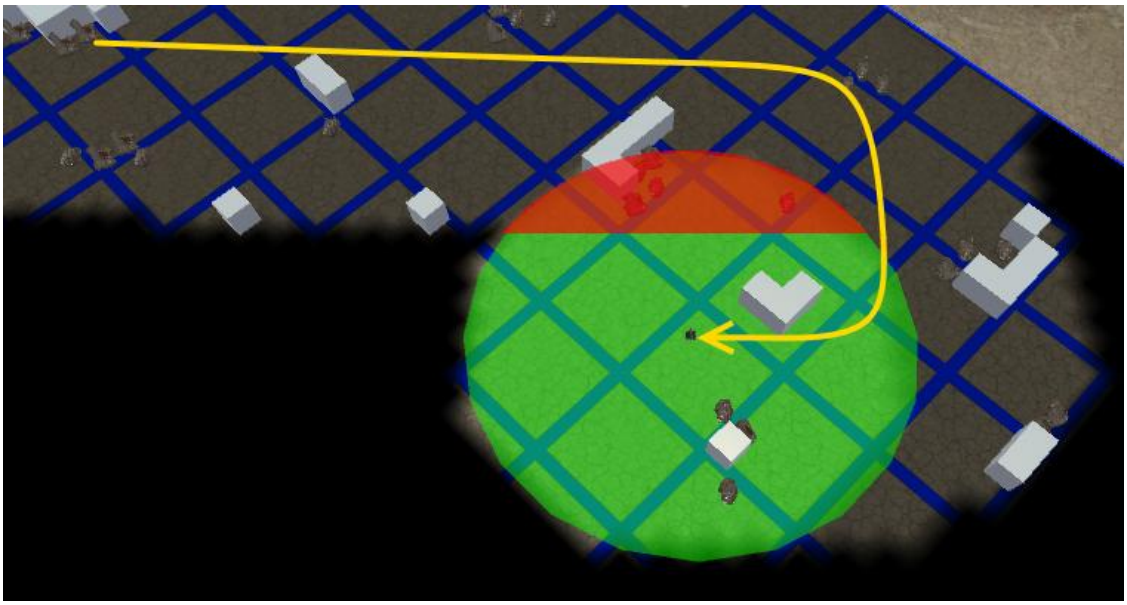
- Wartości zachowują przedział (0, 1),
- Dla $a = 1$ jest funkcją liniową,
- Dla $a > 1$ jest funkcją wklęsłą, im większe a, tym większa wklęsłość,
- Dla $a \to \infty$ dąży do funkcji stałej $f(x) = 0$, bez $x = 1$,
- Dla $0 < a < 1$ jest funkcją wypukłą, im większe a, tym większa wypukłość,
- Dla $a \to 0^+$ dąży do funkcji stałej $f(x) = 1$, bez $x = 0$,

Jeśli więc potraktujemy `factor` jako argument funkcji potęgowej, to kontrolując parametr a (wyrażony przez zmienną `power`), będziemy mogli dostosować zachowanie dla różnych odsetków odkrycia terenu. Przykładowo zastosowanie pierwiastka kwadratowego, czyli $a = 0.5$ powinno spowodować, że regiony odkryte w bardzo niewielkim stopniu będą zyskiwały przewagę nad pozostałymi.



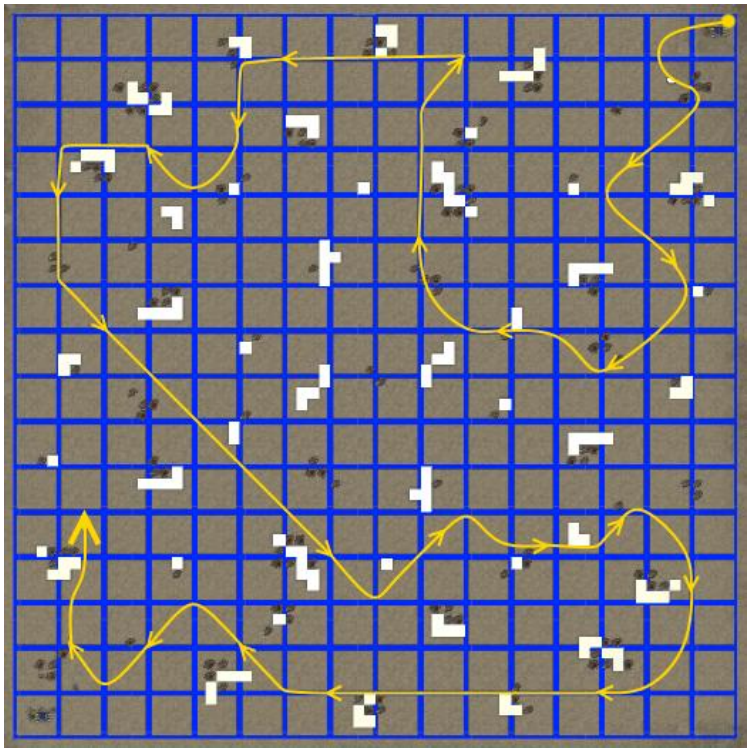
Wykres 9. Funkcja potęgowa x^a na przedziale (0, 1) dla różnych wartości parametru a.

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 33**.



Ilustracja 33. Czerwony obszar pokazuje jakiej części pola widzenia *Scouta* nie wykorzystano przy $a = 0.5$.

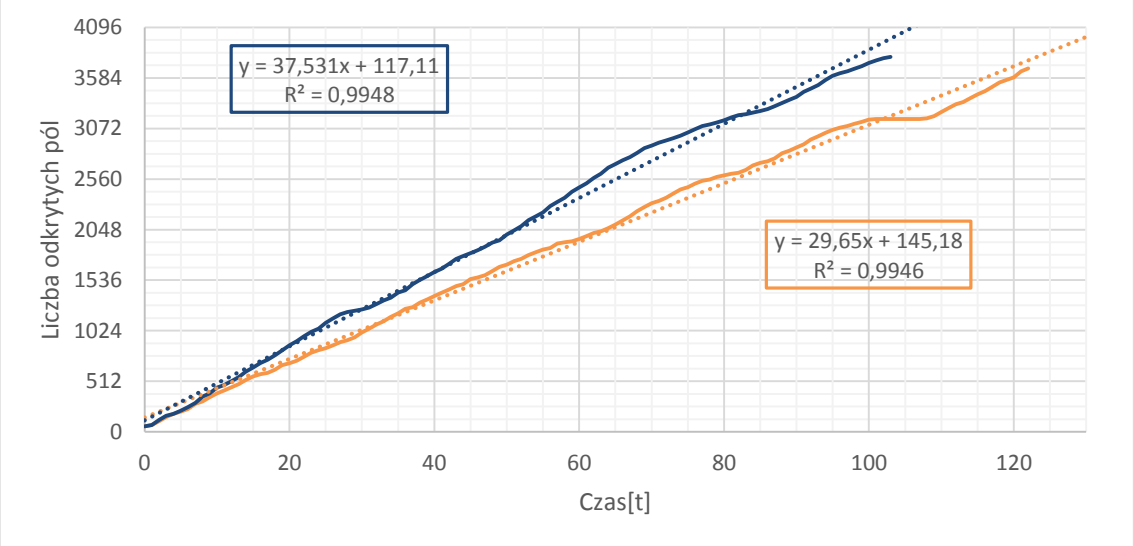
Po wielu testach zdecydowano się na wspomnianą liczbę 0.78. **Ilustracja 34** pokazuje przebieg trasy zwiadowcy dla tej wartości, zaś na **Ilustracja 35** widać problem, jaki trasa ta generuje. Sortowanie przy takim parametrze pozostawia za sobą spore połacie niezwiedzonego terenu — na ilustracji widzimy zaczerniony obszar na środku planszy.



Ilustracja 34. zastosowany algorytm sortowania regionów stworzony w klasie `BaseSelfProductAndTilesExploredOrderCalculator` znacznie skraca trasę...

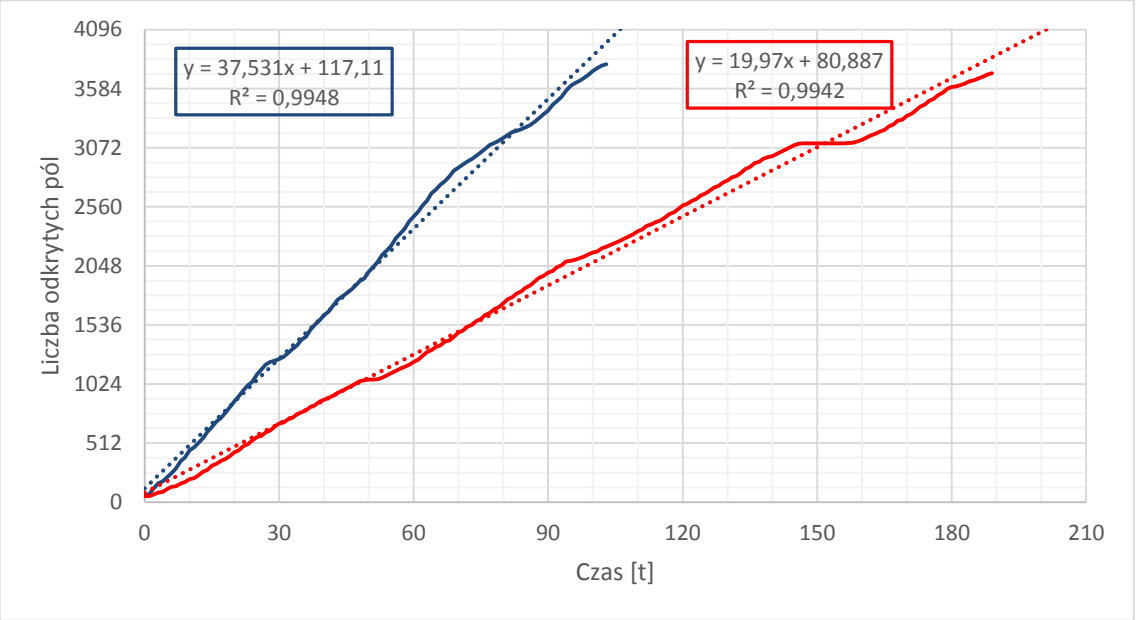
Ilustracja 35. ...lecz zostawia po drodze dużo nieodkrytego terenu.

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 Rozdziału 5.1.[83] Ustawiliśmy więc na szybko test z identycznymi warunkami początkowymi, jak tutaj, jedynie jednostkę *Scout* zastąpiliśmy właśnie *Harvesterem*. Wyniki tej krótkiej symulacji pokazaliśmy na **Wykres 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, podczas gdy przewidywano 2.24. Odczytując czynniki liniowe regresji z **Wykres 11** widzimy, że po poprawieniu błędu wynik *Scouta* poprawił się ok. 1.24 raza, wynik *Harvestera* 1.13 raza, natomiast nowy wzajemny stosunek ma wartość 1.88. Jest więc on bliższy oczekiwanemu, choć nieznacznie.

Wnioskujemy z tego, że wciąż 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.

[83] Rozdział 5.1. Badanie przebiegu domyślnej rozgrywki, str. 108

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 planowano. 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 grą *RTS* na miarę dzisiejszych czasów brakuje mu takich rzeczy, jak: kampania jednoosobowa, rozgrywki *multiplayer*, złożone plansze, kompletne menu z opcjami zapisu i odczytu gry oraz zmiany wielu ustawień, fabuła, muzyka, dźwięki. Mimo to, realizuje on większość pozostałych aspektów rozgrywki, czyli graczy, armie, zasoby przez nie posiadane oraz planszę 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; komendy do się wywoływać skrótami klawiszowymi (tzw. *hotkey*). Operowanie kamerą pozwala na wgląd w każde miejsce planszy w sytuacji taktycznej (w małej skali) 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 wykonano pomocnicze mechanizmy regionów, przeznaczeń i metod tworzenia. Wiedzę dla sztucznej inteligencji 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 spróbowano wytłumaczyć wynik tego porównania. 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 było 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

1. **Artykuł „RTS AI: Problems and Techniques”**
   http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15_chapter-rts_ai.pdf, 27.09.2016r
2. **Wprowadzenie do artykułu „A History of Real-Time Strategy Games”**
   http://web.archive.org/web/20110525140756/http://www.gamespot.com/gamespot/features/all/real_time/index.html, 18.11.2015r
3. **Rozdział artykułu „A History of Real-Time Strategy Games” n.t. gry Dune II**
   http://web.archive.org/web/20110628235716/http://www.gamespot.com/gamespot/features/all/real_time/p2_02.html, 18.11.2015r
4. **Artykuł “Simulating the Fog of War”**
   https://www.rand.org/content/dam/rand/pubs/papers/2008/P7511.pdf, 12.28.2015r
5. **Wypowiedź jednego z twórców *Age of Empires* n.t. sposobu stworzenia mgły wojny**
   http://www.gamedev.net/topic/489276-generating-line-of-sight-in-tile-based-rts/, 23.09.2016r.
6. **Opis wzorca projektowego *singleton***
   http://www.dofactory.com/net/singleton-design-pattern, 23.09.2016r
7. **Opis wzorca projektowego *template method***
   http://www.dofactory.com/net/template-method-design-pattern, 23.09.2016r
8. **Opis wzorca projektowego *template command***
   http://www.dofactory.com/net/command-design-pattern, 23.09.2016r
9. **Opis wzorca projektowego *abstract factory***
   http://www.dofactory.com/net/abstract-factory-design-pattern, 23.09.2016r
10. **Opis wzorca projektowego *proxy***
    http://www.dofactory.com/net/proxy-design-pattern, 24.09.2016r
11. **Artykuł na temat drzew czwórkowych — algorytm funkcji *queryRange()***
    https://en.wikipedia.org/wiki/Quadtree#Query_range, 23.09.2016r
12. **Artykuł na temat drzew algorytmu A* — wariant ograniczonej relaksacji**
    https://en.wikipedia.org/wiki/A*_search_algorithm#Bounded_relaxation, 23.09.2016
13. **Artykuł na temat Algorytmu Grahama do tworzenia otoczki wypukłej**
    https://en.wikipedia.org/wiki/Graham_scan#Algorithm, 24.09.2016r
14. E. Adams, Fundamentals of Game Design, New Riders, 2010, str. 227
15. *Perełki programowania gier. Vademecum profesjonalisty, Tom 2*, 2002, Matt Pritchard, *Rozdział 3.5: Wysokowydajny system widoczności i wyszukiwania oparty na siatkach*, str. 317
16. *Perełki programowania gier. Vademecum profesjonalisty*, Tom 3, 2003, Daniel Higgins, *Rozdział 3.4: Analiza terenu w grach RTS – ukryta siła*, str. 321
17. Mat Buckland, *Programming Game AI by Example, Chap. 2: State-Driven Agent Design*, 2005, str. 43
18. Mat Buckland, *Programming Game AI by Example, Chap. 9: Goal-Driven Agent Behavior*, 2005, str. 379

Aneks

A. Podział prac nad projektem i pracą magisterską

Praca została zrealizowana przez dwie osoby, więc każda z nich pracowała nad innymi jej częściami, z wyjątkiem nielicznych, wykonanych razem. Poniżej wyspecyfikowano, kto był odpowiedzialny za które fragmenty projektu.

Razem:

- Opracowanie założeń dla:
  - Typów jednostek,
  - Typów budynków,
  - Technologii,
  - Drzewka technologicznego.

Natalia Zmysłowska:

- Opracowanie założeń dla:
  - Konwencji tematycznej prototypu,
  - Interfejsu użytkownika.
- Kod źródłowy:
  - Obiekty środowiska — system dnia i nocy,
  - Sąsiedztwo murów — konfiguracja i wymiana modeli w zależności od sąsiadów,
  - Sterowanie — tylko wyświetlanie cienia budynku,
  - Interfejs gracza:
    - Menu główne:
      - Włączanie trybów gry,
      - Ustawienie trybu pełnoekranowego.
    - Przyciski rozkazów:
      - Ustawianie „niesionego” przez myszkę rozkazu,
      - „Cień” budynku,
      - Skróty klawiszowe,
      - Obsługa *tooltipów*.
    - Minimapa:
      - Podmienianie elementów mapy na odpowiednie markery,
      - Rysowanie mgły wojny na minimapie.
    - Licznik zasobów,
    - Obsługa kursora myszy,

        - Kontrola stanu systemu cząsteczkowego.
- Grafika:
    - Dwuwymiarowa:
        - Tekstury dla:
            - Jednostek,
            - Budynków,
            - Zasobów,
            - Terenu,
            - Cząsteczek.
        - Zaprojektowanie interfejsu graficznego:
            - Wyglądu i rozłożenia głównego menu,
            - Rozłożenie elementów w trakcie rozgrywki (zasobów, minimapy, przycisków rozkazów, całego panelu dolnego).
        - Narysowanie *GUI* w stylu flat-design:
            - Przycisków w menu głównym,
            - Piktogramów przycisków rozkazów,
            - Ramek od panelu dolnego, panelu minimapy, panelu przycisków rozkazów i panelu zasobów.
    - Trójwymiarowa:
        - Modele (wymodelowanie i oteksturowanie):
            - Jednostek,
            - Budynków,
            - Zasobów,
            - Terenu.
        - Konfiguracja systemów cząsteczkowych.
- Badania:
    - Eksperymenty dotyczące skuteczności rekonesansu
        - Z wyjątkiem zastosowania funkcji potęgowej, by zmienić wpływ czynnika factor

<u>Sławomir Tomaszewski:</u>

- Opracowanie założeń dla:
    - Podstawowej mechaniki.
- Kod źródłowy:
    - Obiekty globalne:
        - Globals,
        - Gracz, armia, `Spectator`,
        - Obiekty konfiguracyjne i klasy statyczne.
    - Podsystem elementów mapy:
        - Element mapy:

        - Jednostki, budynki, zasoby,
        - Obsługa głowic obrotowych,
        - Duchy (automatyczna podmiana w `QuadTree`, wszystkich rodzajach rozkazów, zaznaczeniu i podświetleniu),
        - I wszystkie inne, liczne, nieopisane tu aspekty związane z ich obsługą, za wyjątkiem tworzenia markerów do minimapy, oraz sąsiedztwa murów.
    - Statystyki,
    - Technologie i bonusy,
    - Rozkazy i akcje rozkazów,
        - W tym: wszystkie klasy potomne (rodzaje) rozkazów i akcji rozkazów,
        - Produkty.
    - Ataki i pociski (zwykłe i kierowane),
- Podsystem mapy:
    - Klasa mapy,
    - Struktura danych `QuadTree`.
- Podsystem mgły wojny:
    - Mechanizm tablicy widzialności,
    - Generowanie tekstury minimapy:
        - Rysowanie na płaszczyznie zgodnie z tablicą widzialności,
        - Przesuwanie i skalowanie płaszczyzny tak, by była zawsze na jednej linii kamera-teren (perspektywa),
- Podsystem poszukiwania ścieżek:
    - Interfejs do dowolnych algorytmów,
    - Algorytm *A** statycznie ważony,
- Sterowanie:
    - Cała klasa `InputController`, w tym:
        - Obsługa kamery (zoom, przesuwanie),
        - Obsługa myszy (*LPM* — zaznaczenie, *PPM* — domyślny rozkaz),
        - Kontroler podświetlenia (filtruje według rodzaju podświetlane elementy mapy),
        - Monitor selekcji,
        - `HoverBox`, czyli ramka zanzaczania,
        - Modyfikatory do stanów myszy za pomocą *shift*, *alt*, *ctrl*.
- Sztuczna inteligencja:
    - Klasa `AIBrain` (gracz AI),
    - System wieloagentowy:
        - Klasa `Agent`,
        - Wiadomości i żądania,
        - System powtarzania metody co jakiś czas (`PerformEvery()`),

        - Wszystkie implementacje agenta: dla zbierania zasobów, rekonesansu, konstrukcji, produkcji, wiedzy, jednostek,
        - Zadania (`Goal`) (zgrubny rekonesans oraz zbieranie zasobów).
    - Organizacja informacji:
        - Regiony,
        - Rodzaje elementów mapy i ich przeznaczenia,
        - Sposoby kreacji elementów mapy.
- Badania
    - Obserwacja przebiegu rozgrywki
    - Z eksperymentów dotyczących skuteczności rekonesansu:
        - Jedynie zastosowanie funkcji potęgowej, by zmienić wpływ czynnika `factor`

B. Treść domyślnej funkcji ważności zadania zbierania zasobów

Treść funkcji zawarta jest w pliku o względnej ścieżce (względem głównego katalogu projektu): `./Assets/AIConfig/StandardHarvestingImportance.txt`. Zawartością tego pliku jest tekst:

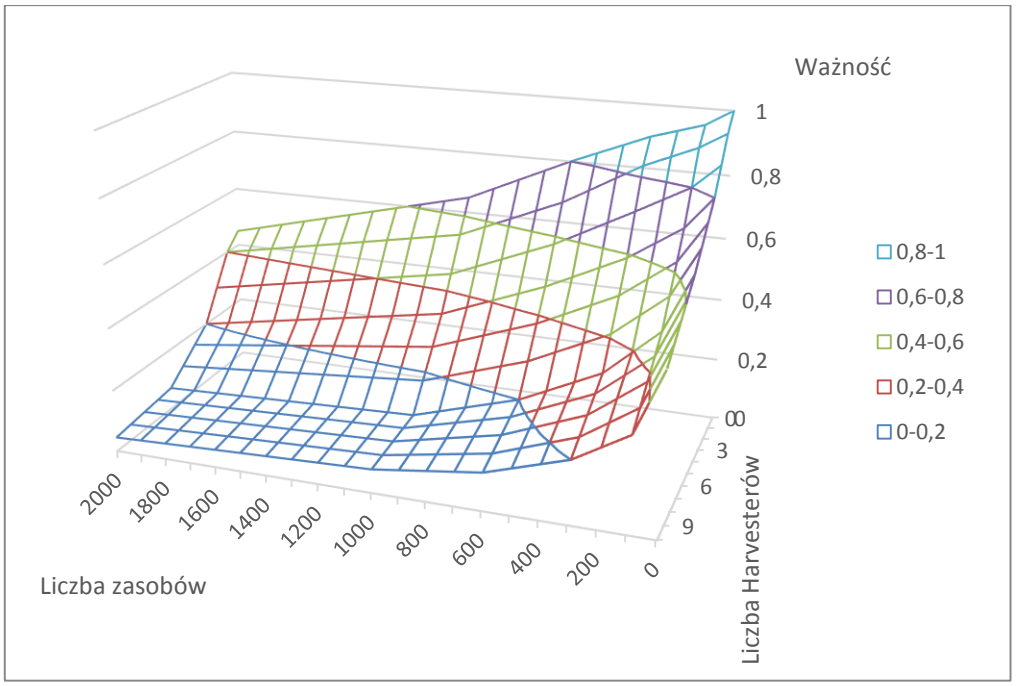
```
Harvesters 0 1 3 6 10
Resources 0 100 300 600 1000 2000
1 0.95 0.9 0.8 0.65 0.45
0.95 0.9 0.83 0.69 0.55 0.4
0.8 0.7 0.59 0.47 0.34 0.2
0.65 0.47 0.32 0.2 0.1 0.07
0.5 0.3 0.2 0.12 0.075 0.045
```

Pierwsze dwie linijki oznaczają osi wykresu dwuwymiarowego (z nazwami), natomiast poniżej nich znajduje się macierz wartości na skrzyżowaniach współrzędnych tych osi. Pierwsza z osi oddwzorowuje wiersze macierzy, a druga — kolumny. Tabelarycznie zatem wygląda to w taki sposób:

Harvesters \ Resources	0	100	300	600	1000	2000
0	1	0,95	0,9	0,8	0,65	0,45
1	0,95	0,9	0,83	0,69	0,55	0,4
3	0,8	0,7	0,59	0,47	0,34	0,2
6	0,65	0,47	0,32	0,2	0,1	0,07
10	0,5	0,3	0,2	0,12	0,075	0,045

Natomiast poglądowy wykres płaszczyzny przedstawiono na następnej stronie. Jest to zgodne z tym, jak się zachowuje cała funkcja, gdyż przy pobraniu wartości spomiędzy określonych argumentów stosowana jest interpolacja biliniowa by wartość wyznaczyć. Natomiast jeśli pobierana jest wartość dla argumentu spoza zakresu, argument jest „przyciągany" do najbliższego w zakresie (np. dla współrzędnych $H = 15, R = 600$ funkcja „przyciągnie" H na 10 i zwróci 0,12).

Wartości te dobrano eksperymentalnie. Zostawiono takie, ponieważ zachowanie przekazywania *Harvesterów* od `ResourceCollectorAgent` do `ReconAgent` było zgodne z oczekiwaniami. Jednakże sposób, w jaki funkcja jest wczytywana umożliwia dowolne jej skonfigurowanie. Wystarczy zapisać inne osi i macierz w pliku. Ważne jest tylko, by zachowany był format: dwie pierwsze liniki muszą być osiami, pierwsza oznaczająca wiersze macierzy, a druga kolumny, pierwszym elementem osi jest jej etykieta, argumenty na osiach muszą być w kolejności rosnącej, a wymiary macierzy muszą się zgadzać z wymiarami osi.


Ważność
1
0,8
0,6
0,4
0,2
0
0,8-1
0,6-0,8
0,4-0,6
0,2-0,4
0-0,2
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Liczba zasobów
0
3
6
9
Liczba Harvesterów