**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Kierunek: Informatyka Stosowana

PROTOTYP GRY STRATEGICZNEJ CZASU RZECZYWISTEGO. BADANIA SZTUCZNEJ INTELIGENCJI PRZECIWNIKÓW

Praca magisterska

napisana pod kierunkiem

dr Jana Argasińskiego

Spis treści

[Wstęp 4](#_Toc461644193)

[Wprowadzenie 5](#_Toc461644194)

[1. Reguły i założenia prototypu 8](#_Toc461644195)

[1.1 Konwencja 8](#_Toc461644196)

[1.2 Specyfikacja jednostek oraz budynków 8](#_Toc461644197)

[1.2.1 Typy jednostek 9](#_Toc461644198)

[1.2.2 Typy budynków 10](#_Toc461644199)

[1.3 Podstawowa mechanika 11](#_Toc461644200)

[2. Sposób zaprogramowania prototypu 12](#_Toc461644201)

[2.1 Opis najważniejszych elementów biblioteki Unity3D 15](#_Toc461644202)

[2.2 Obiekty globalne 17](#_Toc461644203)

[2.2.1 Singleton *Globals* 17](#_Toc461644204)

[2.2.2 Obiekty gracza i armii 18](#_Toc461644205)

[2.2.3 Obiekty środowiska 19](#_Toc461644206)

[2.2.4 Obiekty konfiguracyjne 20](#_Toc461644207)

[2.2.5 Klasy ze stałymi 21](#_Toc461644208)

[2.3 Podsystem mapy 21](#_Toc461644209)

[2.3.1 Klasy mapy 21](#_Toc461644210)

[2.3.2 Klasy drzewa czwórkowego 24](#_Toc461644211)

[2.4 Podsystem elementów mapy 26](#_Toc461644212)

[2.4.1 Implementacja elementu mapy 27](#_Toc461644213)

[2.4.2 Statystyki 33](#_Toc461644214)

[2.4.3 Technologie 34](#_Toc461644215)

[2.4.4 Rozkazy 34](#_Toc461644216)

[2.4.5 Akcje rozkazów 38](#_Toc461644217)

[2.5 Podsystem mgły wojny 39](#_Toc461644218)

[2.6 Podsystem poszukiwania ścieżek 41](#_Toc461644219)

[2.7 Podsystem GUI 42](#_Toc461644220)

[3. Kreacja graficzna prototypu 43](#_Toc461644221)

[3.1 Grafika dwuwymiarowa 43](#_Toc461644222)

[3.2 Grafika trójwymiarowa 43](#_Toc461644223)

[4. Zaprogramowanie sztucznej inteligencji i jej typy 44](#_Toc461644224)

[5. Badania skuteczności sztucznej inteligencji 45](#_Toc461644225)

[Zakończenie 46](#_Toc461644226)

[Streszczenie 47](#_Toc461644227)

[Bibliografia 48](#_Toc461644228)

[Aneks/Spis ilustracji 49](#_Toc461644229)

# Wstęp

[tu znajdzie się cel oraz podział pracy]

# Wprowadzenie

RTS, czyli tłumacząc dosłownie z języka angielskiego „strategia czasu rzeczywistego” jest odmianą gry strategicznej, w której gracze skupiają się na rozbudowie gospodarki ekonomicznej oraz militarnej w celu obronienia swoich budynków oraz jednostek przed przeciwnikiem.[[1]](#footnote-1) Jej odmienność polega na tym, iż dzieje się w czasie rzeczywistym, co oznacza, że cała rozgrywka nie jest podzielona na tury lub kolejki. [[2]](#footnote-2)

Strategie czasu rzeczywistego różnią się 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 od siebie struktur, ideę różnych stron, które posiadają różne bronie oraz jednostki walczące. [[3]](#footnote-3) Cechą wspólną gier strategicznych są warunki zwycięstwa, czyli pokonanie przeciwnika poprzez zniszczenie jego głównej struktury dowodzącej lub pozbawienie go zasobów budowlanych, dzięki czemu nie ma on możliwości odbudowy swoich jednostek. [[4]](#footnote-4)

Główną mechaniką gry jest wybór ścieżki rozwoju, którą gracz zamierza podążać podczas trwania rozgrywki. Do wyboru zazwyczaj mamy dwie podstawowe gospodarki – ekonomiczną oraz militarną, które później może krzyżować. Pierwsza z nich sprawia, że na początku mamy bardzo słabe lub nie mamy wcale jednostek militarnych, gdyż skupiamy się na rozwoju związanym z pozyskiwaniem surowców budulcowych. Pozwala to na dużą ilość zgromadzonych zasobów w krótkim czasie, co później pozwala na lepszą odbudowę oraz masowe zrekrutowanie jednostek bitewnych w późniejszym okresie. Druga droga jest bardziej militarna i skupia się na inwestowaniu w jednostki oraz badania z nimi związane, co pozwala na zbudowanie jednostek potrzebnych we wczesnej fazie gry. Dzieki tej ścieżce gracz nie może pozwolić sobie na szybki rozwój technologiczny, ale zazwyczaj jest w stanie odeprzeć ataki wroga. Podczas podejmowania decyzji odnośnie wyboru ścieżki, którą w każdej chwili może zmienić, gracz musi również zrozumieć konstrukcję mapy oraz rozmieszczenie poszczególnych surowców, na tyle, żeby jak najbardziej optymalnie jego zdaniem rozlokować budynki oraz jednostki.[[5]](#footnote-5)

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

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

Głównym problemem jaki pojawia się podczas rozgrywek przeciwko sztucznej inteligencji jest takie działania jak podejmowanie decyzji opartych na zbyt małej ilości lub braku informacji, a co za tym idzie – optymalnie zaplanowanie taktyk oraz rozlokowanie zasobów. Kolejną rzeczą po zdobyciu informacji jest wyselekcjonowanie, która z nich ma najwyższy priorytet w danej sytuacji oraz zrozumienie jej wagi w odniesieniu do całości. W związku z tym brakuje wysublimowanej sztucznej inteligencji, która pozwoli na skalowanie trudności nie tylko poprzez zwiększenie liczebności jednostek przeciwnika, w taki sposób, żeby cała rozgrywka nie utraciła waloru taktyczno-strategicznego.

W związku z powyższym celem naszej pracy dyplomowej jest stworzenie prototypu gry RTS, który będzie oparty o proste reguły. Głównym założeniem jest zaprogramowanie sztucznej inteligencji, która nie będzie polegała jedynie na prostolinijnej produkcji jednostek w jak najkrótszym czasie. Będzie ona również posiadała zdolność do pozyskiwania informacji, przetwarzania ich oraz wykonanie działań, które są ich konsekwencją. Powinno to pozwolić na jednoosobową rozgrywkę przeciwko AI lub nawet symulację rozgrywki dwóch różnych przeciwników posiadających sztuczną inteligencję przeciwko sobie. Zamierzamy zaprogramować jednostki w taki sposób, aby same reagowały na obecność przeciwnika, dzięki czemu gracz nie musi kontrolować każdego ich ruchu.

# 1. Reguły i założenia prototypu

Celem naszej pracy dyplomowej jest stworzenie sztucznej inteligencji dla prototypu gry RTS. Postanowiliśmy oprzeć go o proste zasady, gdyż nie chcemy skupiać się na rozbudowie mechaniki rozgrywki. Mimo to nie mogą one też być zbyt proste, gdyż za mocno ograniczy to możliwości oraz decyzje, które może podejmować sztuczna inteligencja. Nie zamierzamy komplikować reguł prototypu, żeby zachować optymalny balans rozgrywki. Prototyp zostanie umieszczony w konkretnej konwecji posiadającej określony zasób budowlany, badania technologiczne, kilka typów jednostek oraz budynków.

## 1.1 Konwencja

Program, którego będziemy używać do stworzenia prototypu gry RTS to Unity 3D. Modele budynków, jednostek, jak i otoczenia będą trójwymiarowe, jednak widok z kamery będzie typowym rzutem izometrycznym. Teren, na którym będzie miała miejsce rozgrywka będzie płaską i symetryczną względem środka mapą tak, aby obie strony miały równe szanse przy rozpoczęciu partii.

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

## 1.2 Specyfikacja jednostek oraz budynków

Budynki oraz jednostki, które zostaną wykorzystane w prototypie będą trójwymiarowymi modelami stworzonymi w całości na potrzeby niniejszej pracy. W celu optymalizacji oraz faktu, iż widok z kamery będzie typu izometrycznego usuniemy niewidoczne ściany. Dym z kominów niektórych budynków zostanie zaprojektowany dzięki stystemowi cząsteczkowemu w Unity 3D.

### 1.2.1 Typy jednostek

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

* szybkością obrotu jednostki,
* szybkością poruszania się,
* mocą ataku,
* szybkością ataku,
* zasięgiem ataku,
* ilością życia,
* polem widzenia.

Dwoma podstawowymi niemilitarnymi jednostkami będą jednośladowy zwiadowca oraz tzw. *harvester*, czyli zbieracz. Zwiadowca będzie miał za zadanie sprawdzenie etapu rozwoju przeciwnika, rozpoznania terenu oraz rozlokowania surowców na mapie. Jego podstawowymi zaletami będzie szybkie przemieszczanie się oraz zdobywanie i przekazywanie informacji graczowi. Głównymi wadami jakie bedzie posiadał to brak możliwości walki oraz mała ilość życia. *Harvester* będzie jednostką, której głównym celem będzie zbieranie, wydobywanie oraz transportowanie w określone miejsce złomu. Obie z wyżej wymienionych jednostek nie będą mogły walczyć.

Podstawowymi oddziałami militarnymi będą dwa typy zmechanizowanych jednostek kroczących – bliskozasięgowe oraz dalekozasięgowe.Obie będą poruszały się dość szybko, jednak nie będą posiadały zbyt wielkiej siły ataku. Oprócz walki z oddziałami przeciwnika będą mogły również niszczyć budynki.

Oprócz powyżej opisanych jednostek powstaną jeszcze pojazdy militarne. Jednym z nich będą czołgi, które będą wolnymi maszynami z dużym zasięgiem ataku. Drugimi będą wyrzutnie rakiet, które oprócz zwiększonego zasięgu będą posiadały również bardzo długi czas przeładowania pocisku. Oba typy pojazdów będą posiadały na tyle duże obrażenia, by w bardzo szybkim czasie zniszczyć budynki oraz rozjeżdżać podstawowe oddziały militarne.

### 1.2.2 Typy budynków

Budynki będą służyły wytwarzaniu jednostek, opracowywaniu technologii oraz przetwarzaniu podstawowego surowca. W celu wybudowania ich muszą zostać spełnione pewnego typu warunki. Niektóre z nich będą potrzebowały opracowanej technologii, a inne wybudowania konkretnego budynku. Każdy z budynków będzie posiadał możliwość rozbudowy, co pozwoli na odblokowywanie kolejnych technologii oraz wzmocnienie statystyk budynków i produkowanych przez niego jednostek. Kominy przy niektórych budynkach będą generować dym utworzony za pomocą systemu cząsteczkowego w programie Unity 3D.

Typy planowanych przez nas budynków możemy podzielić na dwie podstawowe kategorie – militarne oraz przemysłowo-rozwojowe. Do militarnych budynków możemy zaliczyć wszystkie budowle odpowiedzialne za produkcję, budowe defensywy oraz wspomaganie dowodzenia w trakcie rozgrywki. Do tej kategorii budowli będą należeć:

* centrum dowodzenia, które będzie pozwalało na budowe nowych jednostek,
* fabryka jednostek,
* struktury obronne - takie jak zmechanizowana wieża obronna oraz mur.

Przemysłowo-rozwojowe budowle będą miały za zadanie wspomaganie rozwoju gracza oraz jego jednostek podczas całej rozgrywki. Do tego typu budowli będą zaliczać się:

* rafineria złomu, która będzie odpowiedzialna nie tylko za odbieranie i magazynowanie surowca, ale także za przetwarzanie go na metal,
* kopalnia, z której będzie można pozyskać złom,
* labolatorium technologii, które będzie umożliwiało opracowywanie nowych ścieżek rozwoju odpowiedzialnych za produkcję nowych budynków oraz jednostek .

## 1.3 Podstawowa mechanika

W podstawowej mechanice gry przewidujemy dwa tryby gry - gracz przeciwko sztucznej inteligencji oraz rozgrywka pomiędzy dwoma sztucznymi inteligencjami. Do żadnego z nich nie zamierzamy tworzyć ani fabuły ani kampanii ze względu na to, że głównym celem naszej pracy jest stworzenie sztucznej inteligencji w prototypie. Zamierzamy zaimplementować podstawowe zachowania jednostek kierowanych zarówno przez fizycznego gracza, jak i przez sztuczną inteligencję. Zaliczają się do nich:

* przemieszczanie się jednostek po mapie,
* wykonywanie rozkazów gracza,
* atakowanie oddziałów przeciwnika,
* zbieranie podstawowego zasobu w rozgrywce.

Oprócz działań zamierzamy również zastosować mechanizm tak zwanej „mgły wojny”. Polega on na tym, że tylko pewna część mapy znajduje się w polu widzenia jednostek, a zatem gracza. W związku z tym pozostałych miejscach lokalizacja jednostek przeciwnika pozostaje nieznana. Jest ona odpowiednikiem poziomu niepewności gracza względem znajomości sytuacji, w której znajduje się przeciwnik w danym momencie. Może ona również dostarczać informację odnośnie położenia surowców na mapie[[8]](#footnote-8). Dzięki jednostkom takim jak zwiadowca będziemy mogli odkryć fragment terytorium i zebrać informacje na temat stanu zaawansowania przeciwnika w rozwoju oraz położenia surowców.

Dodatkowym elementem będzie własna sztuczna inteligencja jednostek, która będzie polegać na zaprogramowaniu oddziałów w taki sposób, żeby mogły „myśleć” bez ingerencji gracza. W skład tego będą wchodziły nie tylko pojedyncze rozkazy odnośnie przemieszczania się, ale także automatyczne atakowanie, gdy przeciwnik pojawi się w zasięgu pola widzenia danej jednostki. Kolejnym przykładem tego elementu jest automatyczny odwrót niemilitarnych jednostek takich jak *harvester*. Będzie on uciekać jeżeli zostanie zaatakowany przez przeciwnika bez względu na wydawane rozkazy przez gracza.

# 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 (w nawiasie podana została nazwa jednej lub kilku najważniejszych klas z danego podsystemu — wyróżniających ten podsystem):

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

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

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

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

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

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

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

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

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

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

Zaprogramowana sztuczna inteligencja jest drugim co do obszerności modułem i szczegółowo została opisana w osobnym głównym rozdziale. W skrócie opiera się ona na systemie wielu agentów (klasa *Agent*), choć zastosowanym do obsługi jednej strony konfliktu. Każdy agent spełnia inne zadanie: jest agent od wiedzy, od rekonesansu, od zbierania zasobów tudzież konstrukcji budynków. Dla każdej jednostki w grze również istnieje osobny agent, można powiedzieć że niższego poziomu. Agenci symulują równoczesne działanie (obiekt *AIBrain* — będącym odpowiednikiem Spectatora po stronie sztucznej inteligencji — uruchamia aktualizację każdego istniejącego agenta raz na cykl pętli gry). Komunikują się między sobą za pomocą niezbyt rozbudowanego systemu wiadomości. Agenci jednostek mogą zostać „wzięci” przez innych agentów — agent, który taką jednostkę „wziął” chwilowo nią zarządza i żaden inny nie ma do niej dostępu, dopóki nie zostanie ona „zwolniona”. Agenci (głównie agenci 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* uwzględniając *VisibilityTable* tak, by AI korzystała z informacji tylko o tym, co sterowana przez nią armia widzi. *MapElementKind* opisuje zastosowania rodzajów jednostek i budynków (np. Harvester w 80% służy do zbierania zasobów i w 20% do zwiadów, podczas gdy Scout w 100% do zwiadów). 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 Unity3D do układania GUI), licznik zasobów (*ResourceCounter*), wizualizacje statusów zaznaczonych elementów mapy (*StatusDisplayDrawer*), przyciski akcji rozkazów (*OrderActionButtton*), obsługa ich skrótów klawiszowych, wyświetlanie dymków pomocy.

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

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

## 2.1 Opis najważniejszych elementów biblioteki Unity3D

Prototyp stworzony został przy pomocy silnika Unity3D, co wiązało się z niejednokrotnym korzystaniem z jego API w programie. Aby więc móc zrozumieć kod prototypu, trzeba wpierw rozumieć działanie Unity3D.

Unity3D w wygodny sposób zarządza tworzoną grą. Silnik oddaje nam do dyspozycji edytor, w którym programista może zarządzać tzw. sceną gry. Scena jest kontenerem na obecnie znajdujące się w grze obiekty. Tylko jedna scena naraz może być wczytana.

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

W najczystszej postaci obiekt gry jest punktem materialnym z niewielką funkcjonalnością. Może on zostać stworzony (do czego służy funkcja *GameObject.Instantiate()*), zniszczony (*GameObject.Destroy()*), a do tego posiada aktualną transformację (obiekt klasy *Transform*): translację, rotację i skalę przedstawione w panelu *Inspector* jako trójwymiarowe wektory. Dla rotacji pokazane są jej kąty Eulera (i to nimi można zarządzać z poziomu edytora), ale Unity3D przechowuje ją w postaci kwaternionu. Oprócz tego obiekty gry mogą zostać ułożone wobec siebie w relacji rodzic-dziecko (o czym informację również trzyma *Transform*) — co sprawia, że transformacja rodzica staje się bazą dla dziecka. W ten sposób można tworzyć całe hierarchie obiektów gry, których strukturę pokazuje panel *Hierarchy*. Dla tych obiektów gry Unity3D automatycznie zarządza pętlą gry, jednak aby miało to znaczenie, należy rozszerzyć im funkcjonalność.

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

Skrypty pozwalają w nieograniczony sposób modyfikować zachowanie obiektu gry. By to było możliwe muszą one spełniać pewną konwencję. Każdy skrypt jest klasą dziedziczącą po klasie *MonoBehaviour*. Może być klasą abstrakcyjną, lecz wtedy nie da się go przypisać do obiektu gry, ponieważ komponenty-skrypty są instancjami klas tych skryptów. Unity3D tworząc obiekt gry woła bezparametrowe konstruktory we wszystkich przypisanych do niego skryptach, by utworzyć ich komponenty.

Publiczne pola skryptu są rozpoznawane przez edytor — dla każdego z nich w panelu *Inspector* Unity3D 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; Unity3D 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 rzucać wyjątki — Unity3D łapie je i wyświetla w konsoli nie przerywając działania gry.

Ostatnią kwestią do poruszenia w związku z Unity3D są tzw. prefaby. Prefab jest zapisanym w projekcie Unity3D obiektem gry — skopiowanym wprost ze sceny (ang. prefabricated). Prefab zawiera wszystkie informacje (położenie, hierarchia, komponenty i wartości ich parametrów), jakie zawierał obiekt gry w momencie jego zapisywania. Funkcjonalność ta daje rozległe możliwości. Podstawowym przypadkiem użycia prefabu jest stworzenie szablonu do obiektu gry, który może być potem powielany na scenie. Można jednak wykorzystać je również do zapisania parametrów konfigurujących różne aspekty gry. Wiedząc, że Unity3D 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 klas w projekcie. Klasa dziedziczy po *MonoBehaviour*, by można było j$ej skrypt umieścić na obiekcie gry — dzięki temu Unity3D 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. 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 na obiekcie *Globals* znajduje się kilka publicznych pól z parametrami (możliwymi do ustawienia w Inspectorze), a na klasie — duża liczba statycznych właściwości do pobierania innych komponentów-skryptów z obiektu gry *Globals*. Są to m.in. takie skrypty jak: *MapSettings*, *Map*, *Textures*, *Prefabs*, *WallNeighbourhoodDictionary*. Można się też dostać do instancji *ShapeDatabase* i *LOSShapeDatabase* — które nie są skryptami Unity3D, ale istnieją na obiekcie *Globals*.

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

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

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

### 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 kilka — obecnie jednak prototyp jest zaprogramowany na pracę z maksymalnie dwoma. Każda z tych klas dziedziczy po MonoBehaviour — skrypty będące ich instancjami są przypisane do obiektów gry odpowiedzialnych za armie, graczy oraz „widza” (Spectator).

Obiekt Army reprezentuje armię (stronę konfliktu w rozgrywce) i zawiera wszystkie niezbędne informacje z nią związane: zbiór jej jednostek, zbiór jej budynków, obiekt TechnologyController (zarządzający rozwojem jej technologii), liczbę aktualnie posiadanych zasobów oraz obiekt VisibilityTable (gromadzący dane o widzialności pól mapy). Oprócz tego w trzech obiektach QuadTree (opisanych w rozdziale 3.3.2) przetrzymuje widoczne przez nią na mapie zasoby, własne elementy mapy i wrogie elementy mapy. Jedyne publiczne metody tej klasy: AddMapElement() oraz RemoveMapElement() pozwalają na zarządzanie zbiorami jednostek i budynków. W prototypie są umieszczone dwa obiekty gry posiadające skrypt Army — dwie przeciwne sobie strony konfliktu.

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

Na podobnej zasadzie (dzięki kompozycji) obiekt Player przypisany jest do jego sterowania. Spectator — „widz” — jest jednym z obiektów, które mogą sterować graczem. Klasa Spectator jest również prosta (choć nie tak prosta jak Player). Posiada publiczne pole typu Player, oraz publiczną właściwość InputController — obiekt ten wykonuje niemal wszystkie zadania potrzebne do sterowania graczem i armią przez gracza-człowieka. W metodzie Update() Spectator wywołuje metodę InputController.Update() (ponieważ InputController nie jest skryptem, więc Unity3D 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 skryptem *MonoBehaviour*, gdyż potrzebuje skorzystać z metody *Update()*. Wewnątrz znajdują się:

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

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

### 2.2.4 Obiekty konfiguracyjne

Fragment kodu : Ciało klasy *DayAndNight*

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);

}

}

}

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

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

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

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

### 2.2.5 Klasy ze stałymi

Do tej kategorii należą dwie klasy statyczne: *Tag* oraz *Layer*. Obydwie zawierają zestaw publicznych stałych typu string z nazwami tagów oraz warstw (dwóch mechanizmów kategoryzowania obiektów gry przez Unity3D).

## 2.3 Podsystem mapy

Podsystem mapy jest stosunkowo niewielki – zawiera 4 klasy umieszczone w namespace *Mapping*. Są to: *MapSettings*, *Map*, *QuadTree* oraz *QuadTreeMapElement*.

### 2.3.1 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), lista graczy, lista obiektów gry armii, *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()*, rzucany jest stosowny wyjątek.

Fragment kodu : Najważniejsze fragmenty klasy *Map*

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;

}

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. Posiada też słownik, który ma odwróconą zależność: dla klucza *MapElement* trzyma listę jego pozycji. Mapa śledzi też położenia duchów w 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()* do sprawdzania czy dana pozycja znajduje się w granicach mapy (często wykorzystywane, by nie został rzucony wyjątek *IndexOutOfBoundsException*), a także funkcje do sprawdzania czy zadane pole jest zajmowane przez zadany element mapy, tworzenia i zwalniania rezerwacji na pola przez elementy mapy (*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 ruchomym elementem mapy, musi więc na bieżąco aktualizować swoje położenie. W momencie gdy ma wykonać ruch z jednego pola na inne, sprawdza najpierw, czy jest ono wolne, lub zarezerwowane przez nią samą. Jeśli tak nie jest, ruch jest odwoływany. Jeśli nowe pole jest wolne, zostaje zarezerwowane, a ze starego pola rezerwacja jest zwalniana. Jeśli element mapy podejmie próbę zarezerwowania już zajętego pola, spowoduje to rzucenie wyjątku.

Metoda *MakeReservation()* przyjmuje *MapElement* (element mapy) oraz *IVector2* (wektor dwóch liczb całkowitych — pozycję do zarezerwowania). Jeśli nie nastąpi konflikt rezerwacji, wstawia *MapElement* do zadanego miejsca tablicy. Następnie pobiera ze słownika rezerwacji listę pozycji *MapElementu* 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 odpowiedniej listy współrzędnych w słowniku *reservationDictionary*. Następnie usuwany jest *MapElement* z zadanego miejsca w tablicy *reservationTable*.

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

### 2.3.2 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 mapy 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 pod-obszarów dopóty, dopóki każdy obszar nie będzie zawierał maksymalnie jednego elementu mapy.

Ponieważ dwuwymiarowa przestrzeń jest dyskretna i za każdym razem dzielona na 4 (w każdym wymiarze na 2), maksymalną głębokością drzewa będzie zawsze **log2(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 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 *MapElement* oraz pozycję w *QuadTree* (*IVector2*). Ponieważ istnieją budynki które zajmują więcej niż jedno pole, może być kilka *QuadTreeMapElementów* z tym samym *MapElementem* ale różnymi pozycjami.

Fragment kodu : Pola klasy *QuadTree*

QuadTreeMapElement QuadTreeMapElement;

SquareBounds bounds;

QuadTree x0y0;

QuadTree x0y1;

QuadTree x1y0;

QuadTree x1y1;

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

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

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

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

Fragment kodu : Metoda *QuadTree.QueryRange()*

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;

}

## 2.4 Podsystem elementów mapy

Elementy mapy to na tyle rozległy podsystem, że nie sposób go tu opisać dokładnie w całości tak, by nie zajęło to zbyt dużej liczby stron. Dlatego szczegółowo omówiono tylko wybrane fragmenty modułu.

Klasa *MapElement* będąca skryptem *MonoBehaviour* jest tu głównym typem. Po tej klasie dziedziczą 3 następne, które uzupełniają jej funkcjonalność: *Unit*, *Building* oraz *Resource* (które poprzez dziedziczenie również są skryptami). Każdy obiekt gry, który znajduje się na mapie jako jednostka, zasób, budynek bądź przeszkoda ma przypisany odpowiedni z tych skryptów. Ponieważ zarówno jednostka jak i budynek potrafią wykonywać rozkazy, obsługę rozkazów finalnie zdecydowano się umieścić w klasie *MapElement.* To, czy dany rodzaj *MapElementu* potrafi wykonać dane rozkazy czy też nie zależy zamiast tego od jego konfiguracji — czyli tego, jakie ma przypisane akcje rozkazów. Rozdział ten opisuje więc też obydwie klasy abstrakcyjne za to odpowiedzialne — *Order* i *OrderAction* — oraz ich przykładowe implementacje. Dodatkowo do tego omówione są pobieżnie statystyki, technologie i zarządzanie nimi oraz mechanizm ustalania sąsiedztwa murów.

### 2.4.1 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* jest stringiem — nazwą określającą rodzaj (jest wspólna dla elementów jednego rodzaju i różna dla każdego z rodzajów — np. każdy czołg ma tu ustawione „Tank”). Pole *id* to generowany automatycznie *int* unikalny dla każdego elementu mapy znajdującego się na scenie.

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

Fragment kodu : Wybrane publiczne pola klasy *MapElement*

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;

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

Lista *aims* obiektów *GameObject* to lista celów obieranych przez pociski wystrzeliwywane w kierunku *MapElementu* 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. Dla każdego prefabu elementu mapy jest ona skonfigurowana w panelu *Inspector*. Każda przypisana akcja rozkazu umożliwia wydanie jednego rodzaju rozkazu.

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

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);

}

}

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 *MapElementu* wokół osi Y (pionowej).

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

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

Fragment kodu : Właściwości klasy *MapElement* związane z kolejką rozkazów oraz definiujące jego możliwości

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); } }

*MapElement* zawiera też szereg wirtualnych właściwości boolowskich 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. Nie zostaną one w szczegółach opisane, ale należy chociaż pobieżnie wyjaśnić pojęcie ducha. Duch to klon elementu mapy, cechujacy 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 ma skopiowane wszystkie wartości statystyk i wygląd — pokazuje zatem ostatni stan *MapElementu*, kiedy jeszcze go było widać. W ten sposób ograniczana jest wiedza gracza o sytuacji poza polem widzenia. Na przykład gracz nie może wiedzieć, że zasób ukryty przez mgłę wojny jest właśnie zbierany przez przeciwnika. Duch jest niszczony dopiero, gdy *MapElement* stanie się na powrót widoczny. Każda armia widzi własne duchy, dlatego *MapElement* posiada słownik 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 zaznacyć jego ducha. Oprócz tego duch musi mieć możliwość być celem rozkazu (np. ataku), a jednocześnie taki rozkaz musi zmienić cel na oryginalny *MapElement*, gdy tylko ten zacznie być widoczny. Duchy muszą automatycznie zamieniać się z oryginalnymi elementami mapy w *QuadTree* trzymanych przez armie. Widać zatem, że konieczne jest, by duchy miały mieć referencję do oryginalnych elementów mapy, choć jednocześnie powinny mieć możliwość istnieć niezależnie od nich (bo element mapy może zostać zniszczony gdy znajduje się poza polem widzenia i gracz nie powinien o tym wiedzieć). Podsumowując, duch musi być dla gracza nieodróżnialny od oryginalnego elementu mapy i imitować wszelakie jego zachowania. Apsekty te sprawiły, że implementacja mechanizmu duchów była niezwykle skomplikowana.

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

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 zawołana funkcja *Start()*, dlatego wołane są tu również *UpateDying()* i *UpdateAlive()*. Następnie *InitializeInMap()* rezerwuje pola w obiekcie *Map*, na których znajduje się *MapElement*, a *InitializeMinimapMarker()* (opisana szerzej w części poświęconej GUI) tworzy marker widziany przez kamerę minimapy. Na końcu ustawiana jest widoczność elementu mapy (*VisibleToSpectator* i *VisibleToArmies*) oraz tworzony jest słownik duchów. Po tym *MapElement* jest gotowy do działania.

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

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 \*\*\*\*

}

}

Funkcja *ReadStats()* wczytuje statystyki z pliku zapisanego w standardzie XML. Wykorzystywana jest do tego .NETowa klasa *System.Xml.XmlDocument*. Przykład takiego pliku pokazany jest w rozdziale o statystykach.

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

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

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

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;

}

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 *Spectatora*. Aktualizowany jest atak, jeśli został przygotowany i uruchomiony. Aktualizowana jest kolejka rozkazów (która wykonuje aktualizację rozkazu, lub uruchamia następny rozkaz, gdy dotychczasowy się zakończył). Na końcu wołane są metody *UpdateDying()* i *UpdateAlive()*, by na bieżąco zarządzać czasem życia elementu mapy.

### 2.4.2 Statystyki

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

Przykład 1: Plik XML ze statystykami jednostki MechMachinegun

<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>

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

### 2.4.3 Technologie

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

*Technology* jest skryptem *MonoBehaviour*. Jako publiczne pola udostępnia m.in. nazwę (*string*) i listę bonusów (*StatBonus*), które można dzięki temu skonfigurować w edytorze Unity3D. Wymagania są konfigurowane w innym miejscu — na akcjach rozkazów.

Każda armia posiada jeden obiekt klasy TechnologyController, który służy do składowania technologii które już są odkryte oraz tych, które właśnie są w trakcie odkrywania. Zawiera metody które pozwalają na kontrolowanie tych aspektów, jak również pomocnicze funcje pozwalające pobrać bonusy dla zadanego elementu mapy.

### 2.4.4 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()*. Razem stanowią one szkielet programu każdego rozkazu, co spełnia założenia wzorca „template method”. Jednocześnie za konstrukcję obiektów *Order* odpowiadają akcje rozkazów (klasa *OrderAction* i jej potomne), a wyżej wspomniane publiczne metody uruchamiane są przez kolejkę rozkazów *OrderQueue* znajdującą się na elemencie mapy. Implementacja operacji do wykonania jako obiektu, który może być tworzony w jednym miejscu a wykorzystywany w innym i oddzielenie tej operacji od obiektu na którym pracuje (*MapElement*) spełnia więc z kolei założenia wzorca „command”.

Każdy rozkaz znajduje się w pewnym stanie (enum *OrderState*) —etapie jego życia. Świeżo utworzony obiekt *Order* ma stan *BrandNew*. Po zawołaniu 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* (jeśli rozkaz jest zatrzymywany z zewnątrz przy pomocy metody *Stop()*). Jeśli wystąpił błąd i rozkaz trzeba natychmiast przerwać, wołana jest metoda *Terminate()* i rozkaz przechodzi w stan *Terminated*. Stany *Finished*, *Stopped* i *Terminated* są stanami końcowymi pozwalającymi na usunięcie rozkazu z kolejki *OrderQueue*.

**[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 w której jednostka zatrzymałaby się pomiędzy kratkami.

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

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

Fragment kodu : Główne funkcje i właściwości klasy *Order*

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;

}

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).

Fragment kodu : Główne funkcje i właściwości klasy *ComplexOrder*

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();

}

}

Obiekt *ComplexOrder* samodzielnie trzyma referencję na aktualny i następny pod-rozkaz nie korzystając z kolejki *OrderQueue*. Nadpisuje i pieczętuje funkcję *LateOnUpdate()* (słowo kluczowe „sealed” — następni potomkowie nie mogą nadpisać tej funkcji), w której implementuje automatyczną obsługę pod-rozkazów. Udostępnia następne metody wirtualne w których rozkazy potomne mogą zareagować na zmianę stanu pod-rozkazu (dalsze użycie wzorca „template method”). Poza tym zmienia też zachowanie właściwości decydujących, czy rozkaz może być zatrzymany (*CanStop* itd.). Dzięki zwracanym przez nie nowym wartościom *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 zaimplementowane w taki sposób, by powodowały zatrzymanie pod-rozkazów.

### 2.4.5 Akcje rozkazów

Abstrakcyjna klasa *OrderAction* służy dwóm celom.

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

Po drugie klasa *OrderAction* dziedziczy po *MonoBehaviour*, może więc służyć do konfiguracji. W projekcie Unity3D zapisany jest szereg prefabów zawierających jako komponenty skrypty potomne do *OrderAction*. Prefaby te poustawiane są każdemu prefabowi elementu mapy na jego liście. W ten sposób skonfigurowane jest to, jakie rozkazy potrafi wykonywać jaki *MapElement*.

Fragment kodu : Główne metody klasy *OrderAction*

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);

## 2.5 Podsystem mgły wojny

Działanie mgły wojny zaimplementowane zostało za pomocą klasy *VisibilityTable*. Każda armia ma własną tablicę widzialności. Jest to obiekt zawierający dwie dwuwymiarowe tablicę odwzorowujące rozmiarem planszę gry. Dla każdego pola gry w tablicy *bool[,] fieldsUncovered* znajduje się informacja, czy dane pole w ogóle zostało odkryte przez armię. Z kolei tablica *int[,] fieldsSeenByUnits* trzyma dla każdego pola liczbę jednostek, która je widzi. Odkryte pola, dla których liczba ta jest równa zeru, znajdują się we mgle wojny — nie widać na nich jednostek, a zamiast budynków i zasobów widać ich duchy.

**[wpleść jakoś źródło z GameProgrammingGems]**

W *VisibilityTable* znajdują się metody *IncreaseVisibility()* i *DecreaseVisibility()*. Służą do inkrementacji i dekrementacji tablicy *fieldsSeenByUnits* przez *MapElementy*. Przykładowo gdy jednostka się porusza, dekrementuje pola tablicy w swoim zasięgu widzenia względem do starego położenia, a następnie inkrementuje te względem nowego. By uniknąć kosztownego wyliczania pól znajdujących się wewnątrz okręgu pola widzenia wyznaczanego przez statystykę „View range” (promień okręgu), tablica widzialności korzysta z obiektu *LOSShapeDatabase*, który w leniwy sposób generuje kształty pola widzenia 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.

bool[,] fieldsUncovered;

int[,] fieldsSeenByUnits;

void IncreaseVisibilityOfTile(int x, int y)

{

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

bool justUncovered = !fieldsUncovered[x, y];

fieldsUncovered[x, y] = true;

fieldsSeenByUnits[x, y]++;

if (VisibilityChanged != null)

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

Visibility.Unknown, Visibility.Visible);

else VisibilityChanged(new IVector2(x, y),

Visibility.Fogged, Visibility.Visible);

}

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

{

var meShape = mapElement.Shape;

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

if (radiusStat == null) return;

var radius = radiusStat.Value;

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

IncreaseVisibility(x, y, losShape);

}

void DecreaseVisibilityOfTile(int x, int y)

{

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

fieldsSeenByUnits[x, y]--;

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

VisibilityChanged(new IVector2(x, y),

Visibility.Visible, Visibility.Fogged);

}

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

{

var meShape = mapElement.Shape;

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

if (radiusStat == null) return;

var radius = radiusStat.Value;

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

DecreaseVisibility(x, y, losShape);

}

Fragment kodu 13: Wybrane składowe klasy *VisibilityTable*

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

Graficzny efekt mgły wojny jest renderowany na bazie *VisibilityTable*. Skrypt *VisualFog* (dziedziczący po *MonoBehaviour*) generuje w locie teksturę o wymiarach takich jak plansza, gdzie jeden piksel odpowiada jednemu polu. Piksele są czarne, ale różnią się składową alpha. Dla pól nieodkrytych są całkowicie czarne, pola we mgle są półprzezroczyste a widzialne — całkowicie przezroczyste. Tekstura ta nakładana jest następnie na płaszczyznę zasłaniającą planszę, stanowiąc coś w rodzaju filtru koloru dla tego, co widzi kamera. Dzięki użyciu warstw w Unity3D, mgła jednak zasłania jedynie podłoże, nie przykrywając żadnych elementów mapy. Klasa *MinimapFog* kopiuje teksturę z *VisualFog* i nakłada ją na własną płaszczyznę widoczną przez kamerę minimapy.

## 2.6 Podsystem poszukiwania ścieżek

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

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

Przeszacowanie odległości pozwala poprawić czas wykonywania algorytmu (mniej węzłów zostaje odwiedzonych), choć sprawia, że algorytm nie gwarantuje już najlkrótszej ścieżki. Jest to tak zwana ograniczona relaksacja heurystyki. Liczba stanowi ograniczenie relaksacji gwarantujące, że rozwiązanie nie będzie gorsze niż najlepsze rozwiązanie pomnożone przez. Heurystyka taka to heurystyka dopuszczalna. Istnieje kilka dopuszczalnych odmian algorytmu A\*. Wybraną przez nas jest odmianą jest A\* statycznie ważony. W tej odmianie stanowi wagę przez którą mnoży się wartość heurystyki. U nas wynosi zatem 5.

**[źródło=https://en.wikipedia.org/wiki/A\*\_search\_algorithm]**

## 2.7 Podsystem GUI

[część Natalii]

# 3. Kreacja graficzna prototypu

## 3.1 Grafika dwuwymiarowa

Na grafikę dwuwymiarową składa się interfejs gracza, czyli UI oraz ekran startowy, zwany również głównym menu. W związku z obowiązującymi aktualnie trendami obie rzeczy zostały zaprojektowane w stylu flat design. Polega on na .... Możemy się z nim spotkać w ...

Kodzenie głównego menu [stary typ kodzenia]

Kodzenie interfejsu [nowy typ kodzenia]

## 3.2 Grafika trójwymiarowa

Na grafikę trójwymiarową składają się modele mechów oraz budynków wewnątrz gry, a także otoczenia, które je otacza. Dodatkowo występują particle zaprojektowane wewnątrz Unity [dym z kominów]. Dodatkową rzeczą jest ruszające sie tło głównym menu.

# 4. Zaprogramowanie sztucznej inteligencji i jej typy

* zachowania ekonomiczne
* zachowania taktyczne
* jak to zostało zrobione/zaprogramowane
* mechanika gry?

# 5. Badania skuteczności sztucznej inteligencji

* AI vs AI
* AI vs gracz

# Zakończenie

* Co było do zrobienia
* Co zostało zrealizowane
* Z jakim skutkiem

# Streszczenie

# Bibliografia

# Aneks/Spis ilustracji

1. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-1)
2. web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real\_time, tłumaczenie własne, 11.18.2015r [↑](#footnote-ref-2)
3. web.archive.org/web/20110628235716/http://www.gamespot.com/gamespot/features/all/real\_time/p2\_02.html, tłumaczenie własne, 11.18.2015r [↑](#footnote-ref-3)
4. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-4)
5. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-5)
6. http://webdocs.cs.ualberta.ca/~cdavid/pdf/ecgg15\_chapter-rts\_ai.pdf, tłumaczenie własne, 11.30.2015r [↑](#footnote-ref-6)
7. http://pwp.detritus.net/in/1997/rf.html, tłumaczenie własne, 12.28.2015r [↑](#footnote-ref-7)
8. https://www.rand.org/content/dam/rand/pubs/papers/2008/P7511.pdf, tłumaczenie własne, 12.28.2015r [↑](#footnote-ref-8)