Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Fizyki, Matematyki i Informatyki

Mikołaj Raźny

Numer albumu: 104987

Wykorzystanie metod sztucznej inteligencji do konstrukcji gry opartej o środowisko Unity

Artificial Intelligence Methods in the game based on Unity environment

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Uzgodniona ocena:.....

.....
podpisy promotora i recenzenta

Kraków 2018

Spis Treści

1. Wstęp

1.1 Cel pracy

Celem tej pracy jest przybliżenie sposobów implementacji i wykorzystania SI w różnego typu grach komputerowych. Równolegle z powstawaniem niniejszego dokumentu stworzony ma być także projekt gry zawierającej elementy sztucznej inteligencji. W związku z czym wynikiem końcowym jest także bliższe poznanie procesu tworzenia gier oraz wykorzystywanych do tego, zarówno przez deweloperów jak i menadżerów projektu, profesjonalnych narzędzi i metod.

1.2 Zakres pracy

Pracę rozpoczyna omówienie pojęcia sztucznej inteligencji (SI) w rozdziale 2: jej historii, rozwoju oraz krótki opis, do czego wykorzystuje się ją w grach komputerowych. Sposoby implementacji i projektowania SI przedstawione są w rozdziale 3. Następnie przechodzimy do omówienia typów gier komputerowych (rozdział 4) - pokrótce opisujemy najpopularniejsze z nich, jednocześnie wskazując przykłady zastosowania w nich SI. Rozdział 5 skupia się na procesie tworzenia gier od strony teoretycznej - opisane są w nim silniki do tworzenia gier komputerowych, proces powstawania i rozwoju koncepcji oraz tworzenia gry.

Po powyższym wstępie teoretycznym, w rozdziale 6, opisane są także konkretne narzędzia wykorzystane przy tworzeniu gry będącej głównym celem pracy. W rozdziale 7 możemy znaleźć opis, jak od strony praktycznej wyglądało napisanie gry.

Na końcu znajduje się podsumowanie pracy wraz z wyciągniętymi wnioskami.

2. Implementacja Sztucznej Inteligencji w grach komputerowych

2.1 Sztuczna inteligencja

Gdy słyszymy pojęcie sztucznej inteligencji pierwsze, co przychodzi nam na myśl to wymyślne roboty - androidy. Często przedstawiane są w kulturze popularnej

jako twory, które są coraz trudniejsze do rozróżnienia z człowiekiem (np. film Ex Machina z roku 2015 [45]), są dla niego wsparciem, czy też narzędziem i same potrafią podejmować logiczne decyzje (np. C-3PO i R2-D2 z serii filmów Gwiezdne Wojny [46]). Zachowania, które je charakteryzują, gdyby były to rzeczywiste twory, są wynikiem realizacji przez nie właśnie zaimplementowanej w ich oprogramowaniu sztucznej inteligencji, mającej naśladować inteligencję istoty ludzkiej.

Sztuczna inteligencja (ang. *Artificial Intelligence* (*AI*)) to naśladowanie procesów myślowych i inteligencji ludzkiej realizowana przez maszyny [1]. Innymi słowy, opisywana jest jako "nauka o maszynach realizujących zadania, które wymagają inteligencji wówczas, gdy są wykonywane przez człowieka" [2]. Encyklopedia PWN natomiast podaje definicję: "sztuczna inteligencja to dziedzina nauki zajmująca się badaniem mechanizmów ludzkiej inteligencji oraz modelowaniem i konstruowaniem systemów, które są w stanie wspomagać lub zastępować inteligentne działania człowieka" [62]. W informatyce definiuje się ją jako badania na temat "inteligentnych agentów" (ang. *Intelligent Agent*) [1] - urządzeń monitorujących swoje środowisko i podejmujących działania w celu zmaksymalizowania szans na osiągnięcie określonego celu. W związku z tym zajmuje się tworzeniem modeli i algorytmów symulujących określone zachowania, a następnie implementacją programów realizujących postawione przez modele założenia.

SI zajmuje się logiką rozmytą [47], obliczeniami ewolucyjnymi [48], sieciami neuronowymi [49], sztucznym życiem [50] oraz robotyką [51] i bioniką [63]. SI obecnie jest wykorzystywane do rozpoznawania ludzkiej mowy [52], tworzenia zachowań przeciwników w grach strategicznych (np. szachach) [1], samojeżdżących samochodów [53], symulacji komputerowych [54], realizacji routingu w sieciach komputerowych [55] i interpretacji skomplikowanych danych [1]. SI znajduje zastosowanie w sytuacji, kiedy decyzja musi być podjęta w przypadku braku kompletnych danych, dzięki ich analizie i wyciąganiu wniosków [2]. SI pomaga nawet stawiać diagnozy lekarskie [62].

Kolejny rozdział przedstawia krótką historię rozwoju koncepcji sztucznej inteligencji.

2.2 Historia sztucznej inteligencji [2]

Pierwsze koncepcje sztucznej inteligencji zaczęły powstawać już między rokiem w latach 60 XX wieku. Przyczynił się do tego rozwój technologii komputerowych i upowszechnienie komputerów. W roku 1950 roku powstał "test Turinga" - jest to sposób określenia, na ile dana maszyna jest w stanie posługiwać się językiem naturalnym i przeprowadzać proces myślenia podobny do ludzkiego. Test polegał na tym, że grupa osób przeprowadza rozmowę w języku naturalnym z testowanym obiektem. Jeżeli ta osoby nie jest w stanie określić, czy rozmawia z innym człowiekiem, czy też z maszyną, to można uznać, że maszyna, z którą rozmawia przeszła test. Sama idea testu bardzo wiele mówi o sztucznej inteligencji, jej założeniach i celach, które ma zamiar osiągać. Uważano, że dość duża część maszyn (30%) o pojemności pamięci około 119MB będzie w stanie przejść ten test w roku 2000 [3]. Jak się okazuje programy zaczęły przechodzić ten test pomyślnie dopiero w roku 2014 [56].

Termin "sztuczna inteligencja" pojawił się około roku 1955 i został zaproponowany przez Johna McCarthy'ego [57]. Założył on drugie na świecie laboratorium SI - znajduje się ono w MIT (Massachusetts Institute of Technology) i jest ciągle czołowym ośrodkiem SI na świecie. McCarthy określił SI jako "konstruowanie maszyn, o których działaniu dałoby się powiedzieć, że są podobne do ludzkich przejawów inteligencji" [57].

W latach 1965-1970 opadł początkowy entuzjazm dotyczący SI. Zyskano świadomość ograniczoności ludzkich możliwości w konstruowaniu maszyn - zauważono, że pewne koncepcje są nad wyraz trudne do realizacji i w przy obecnych w tamtych latach ograniczeniach sprzętowych i algorytmicznych są nie do przeskoczenia. Problemy, które wtedy udało się dostrzec do dziś są wyzwaniem - mówimy tutaj o np. odczytywaniu pisma odręcznego, streszczeniach tekstów, tłumaczeniach automatycznych i rozpoznawaniu obiektów na niedokładnych zdjęciach. Kilka lat później natomiast postanowiono skupiać się na konkretnych, osiągalnych celach, co odniosło skutek i przyniosło narodziny pierwszym, użytecznym systemom eksperckim [58].

Po roku 1980tym rozpoczęła się komercjalizacja osiągnięć odniesionych na polu badań nad SI. Słowo “inteligentny” w tym czasie zaczęło się przyjmować w reklamach i w zasadzie ciągle jest tam obecne. Sieci neuronowe i logika rozmyta przestały być pojęciami stosowanymi tylko i wyłącznie w nauce, ale zaczęły być stosowane w sprzętach dostępnych przeciętnemu obywatelowi. SI zaczęła być wykorzystywana w sprzętach RTV i AGD. Dzięki rozwojowi technologicznemu zadania, których nie dało się zrealizować przy pomocy SI mogły być rozwiązane “na siłę” (przez sprawdzenie wszystkich możliwości rozwiązania i wyszukanie prawidłowego).

Wiele gier stanowi doskonałe pole do popisu dla SI. Chociaż podstawowe algorytmy często sprowadzają się do prostego analizy szans na wygraną po wykonaniu określonego ruchu, zagraniu odpowiednią kartą, itp., a następnie podjęciu na tej podstawie decyzji, to ich dalszy rozwój może dać wymierne rezultaty. Gra w szachy już po 8 posunięciach daje nam prawie 92 miliardy sposobów, w jaki bierki ułożone są na planszy. Pierwszy program szachowy zaczął powstawać w roku 1948, kiedy jeszcze nie było dość zaawansowanego komputera, na którym mógłby być zaimplementowany [44]. A. Turing w 1951 był przekonany, że “Nikt nie jest w stanie ułożyć programu lepszego od własnego poziomu gry” [2]. Kilkanaście lat po wypowiedzeniu tego zdania, w roku 1967 pierwszy komputer pokonał “profesjonalnego” szachistę podczas turnieju [2]. Dziesięć lat później komputer pokonał mistrza klasy międzynarodowej, a w roku 1997 specjalny, grający w szachy system komputerowy Deep Blue [4], stworzony przez IBM, wygrał partię szachów, ze światowym mistrzem - Garri Kasparowem. Był to pierwszy komputer, który wygrał mecz z regularną kontrolą czasu z aktualnym mistrzem świata. Deep Blue, który korzystał z aż 418 procesorów doczekał się następcy - Deep Juniora, który składał się z zaledwie 8 procesorów Intela 1.6Ghz. Chociaż Kasparow przed oficjalnym meczem długo trenował, rozpoznając słabe strony Deep Juniora, to zdołał zaledwie z nim zremisować. Warto przy tym powiedzieć, że o ile udało się opracować komputer

1. Komputer Deep Blue [4]

zdolny wygrywać z mistrzami szachowymi i algorytm wygrywający z mistrzami 1977 roku, to gra Go jest już dużo trudniejsza dla komputerów - plansza do Go posiada aż 361 pól, co wiążę się z dużo większym zapotrzebowaniem na moc obliczeniową i wymaga dużo lepszego programu. AlphaGo pokonał arcymistrza tej gry dopiero w roku 2016 [59].

W następnym rozdziale przedstawione zostały sposoby wykorzystania koncepcji SI w grach komputerowych.

2.3 Wykorzystanie koncepcji SI w grach komputerowych [5][12]

Jednymi z pierwszych prób zaimplementowania sztucznej inteligencji do gier było też związane z przeniesieniem istniejących już gier do świata komputerów (np. warcaby w 1952r, szachy 1948r [2]). W obecnych czasach tradycyjne gry planszowe przeniesione do wirtualnego świata jest to zaledwie ułamek wszystkich gier, które wykorzystują SI. Ciągle oczywiście tworzy się kolejne odsłony "szachów", ale także tworzy się coraz bardziej skomplikowane gry strategiczne - nie tylko takie, w których kolejne ruchy wykonuje się w turach, ale także czasu rzeczywistego. SI jest wykorzystywana jednak nie tylko w grach strategicznych, ale także w zasadzie każdym rodzaju gier - od prostych platformówek, aż po gry akcji, czy symulacje [5].

SI nie odpowiada tylko za ruchy naszego jedynego oponenta. Często jest podstawą zachowań każdego NPC (bohater niezależny, ang. *non-player character*), jakiego napotykamy na swojej drodze w grach RPG (zobacz: 4.2 Komputerowa gra fabularna), stosowania taktyk przez grupę przeciwników w grach akcji czy też ten "jeden oponent" jest zarządcą wielu pomniejszych jednostek. Chociaż w grach możemy doświadczyć właśnie takiej, "prawdziwej" sztucznej inteligencji, to jednak często obserwujemy po prostu drzewo decyzyjne [123] sprawdzające zdefiniowane, proste warunki i wykonujące odpowiednie akcje.

Sztuczna inteligencja nie jest wykorzystywana tylko do sterowania przeciwnikami, czy innymi, żywymi obiektami napotkanymi w grze. Z tym terminem wiąże się też generowanie kontekstu gry - np. ukształtowania terenu czy tworzenie obiektów i innej zawartości.

Kolejny rozdział opisuje, jak przebiega projektowanie i implementacji SI w grach komputerowych.

3. Implementacja SI w grach komputerowych

3.1 Projektowanie SI [7][9]

SI może zostać zaprojektowane od podstaw i chociaż nie istnieje jeden, uniwersalny schemat, to można skorzystać z gotowych szablonów. Istnieją różne sposoby i podejścia (typy) implementacji SI w grach komputerowych. Gdy stajemy przed wyborem, który z nich wybrać, musimy wziąć pod uwagę kilka czynników, z których najważniejszym wydają się ograniczenia techniczne - to bardzo istotne dla ilu osobnych jednostek będzie trzeba realizować algorytmy i jak skomplikowany on będzie. Inaczej będzie wyglądało podejście do SI w grach mobilnych, gdzie procesory mają relatywnie mniejszą moc, niż komputery PC najnowszej generacji. SI może być "wąskim gardłem", jeśli chodzi o wykorzystanie mocy obliczeniowej - algorytmy muszą być jak najmniej obciążające dla procesora. Trzeba wziąć pod uwagę jak różne algorytmy będziemy stosować - czy na przykład mamy do czynienia z armią przeciwników, z których każdy jest niezależnym tworem podejmującym decyzję na podstawie swojego otoczenia, czy też prezentujemy kilku bohaterów, ale każdy z nich ma swoje własne, unikalne właściwości i możliwości poznawcze. W tym drugim przypadku SI staje się także sposobem na urozmaicenie rozgrywki. Same algorytmy nie są jednak jedynym problemem. Projektując SI musimy odpowiedzieć na kilka pytań: Czym będzie SI w naszej grze?; Jakie cele ma realizować?; Jaką pulę zachowań oddajemy do jej dyspozycji?

Warto wspomnieć o pojęciu asymetrycznej i symetrycznej SI. Pierwszy rodzaj to SI, która ma w teorii dużą przewagę nad graczem. Jest świadoma większej ilości informacji o świecie gry (np. położenie zasobów, sposób poruszania się gracza). Dzięki temu możemy balansować rozgrywkę i dopasować ją do rodzaju gry. Sztuka często polega na tym, żeby gracz nie zauważył kiedy i w jaki sposób jego przeciwnik wykorzystuje te informacje. Symetryczna SI zakłada takie same możliwości komputer i gracza - projektant ma możliwość stworzenia iluzji gry przeciwko innej, żywej osobie,

która jest zmuszona do przestrzegania dokładnie tych samych zasad. Oba rodzaje sztucznej inteligencji mogą dawać graczowi satysfakcję z nauki jej działania i znajdowania sposobów na jej przewyższenie.

Przykładami sposobów implementacji SI są: zachowanie oparte o osiągnięcie celu (ang. *goal based behavior*) [60], drzewa zachowań (zobacz: 3.4 Drzewa zachowań) i skończona maszyna stanów (zobacz: 3.3 Skończone maszyny stanów).

Część z problemów, które mogą się pojawić podczas implementacji SI jest przedstawiona w kolejnym rozdziale.

3.2 Przykłady problemów i optymalizacja rozwiązań

Realizacja sztucznej inteligencji może wymagać mocy obliczeniowej, której nie znajdziemy w przeciętnym, domowym komputerze [7]. Pisząc algorytmy należy wziąć to pod uwagę. Poniżej przedstawione są niektóre ze sposobów i pomysłów, które pozwalają na zmniejszenie zapotrzebowania na zasoby komputera: dane dostarczane do algorytmu w określonych odstępach czasowych czy przykład podziału problemu na mniejsze i łatwiejsze do rozwiązania. Z niektórymi ograniczeniami trzeba sobie radzić przez zmianę sposobu myślania i zubożenie algorytmu - dzieje się tak w przypadku, gdy potrzebuje on zbyt dużej ilości danych wejściowych do obliczeń.

3.2.1 Aktualizacja danych

Zakładając wiele różnych obiektów monitorujących swoje zasoby (którym może być także wiedza na temat otoczenia), określamy jak często obiekt aktualizuje swoje informacje [7]. Naturalnym podejściem wydaje się aktualizacja danych co każdą klatkę (ang. *frame*), jednak takie podejście jest bardzo nieefektywne i jest zależne od tego, jak szybko zmieniają się klatki - nie ma powodu, dla którego na sprawniejszych komputerach (szybsze renderowanie klatek), także SI miała mieć większy wpływ na wykorzystanie procesora. Rozwiązaniem jest stosowanie innych interwałów między aktualizacjami danych. Przerwy liniowe (co stały, określony czas rzeczywisty) w grze działającej w 30 klatkach na sekundę (ang. *frame per second* (*FPS*)) ustalone na poziomie 1/3s zmniejszają wykorzystywaną moc obliczeniową prawie 10-krotnie. Można zastosować inny interwał osobny dla każdego obiektu, czy każdego rodzaju

akcji, a także zmiana okresu interwału w zależności od okoliczności. Aktualizować dane obiektu możemy też wywołać poprzez reakcję na nasze poczynania - dana akcja wykonana przez nas może być sygnałem do innego obiektu (ang. *trigger*), do kontrakcji, w związku z tym obliczenia nie będą wykonywane w międzyczasie. Dodatkowy nakład obliczeniowy związany z wprowadzeniem liczników jest zwykle niewielki, szczególnie biorąc pod uwagę oszczędności na mocy obliczeniowej. Wprowadzając wiele osobnych liczników dla każdego obiektu możemy także rozłożyć wykorzystanie procesora w czasie, nie dopuszczając do sytuacji, gdy wiele czynności jest zawsze wykonywanych w tym samym czasie. Dodatkowo wprowadzamy także wprowadzenie pola widzenia (monitorowany obszar) jako jeden z parametrów obiektów. Nieefektywne jest wykonywanie kodu, jeżeli nie będzie miał on wpływu na rozgrywkę? W grze Far Cry 3 sztuczna inteligencja NPC jest aktywowana, gdy gracz znajdzie się w odległości mniejszej, niż 500m (jednostek umownych, które w wirtualnym świecie są również nazywane metrami) od niego.

3.2.2 Wyszukiwanie ścieżki

Wyszukiwanie ścieżki to jeden z najbardziej powszechnych problemów, ale dzięki temu istnieją sprawdzone i optymalne metody jego rozwiązania [7]. Jest to też kłopot, który pojawił się dużo wcześniej, niż gry komputerowe, w związku z tym mogliśmy wykorzystać wymyślone już algorytmy i modele matematyczne dostosowując je do realiów gry. Rozwiązanie może różnić się w zależności od np. tego, czy realizujemy grę 2-wymiarową (2D), czy 3-wymiarową (3D). Wraz ze wzrostem rozmiaru obszaru wirtualnego świata dostępnego dla gracza - mapy (możliwości wykonanych ruchów) rośnie zapotrzebowanie na moc obliczeniową. Mapę dzielimy więc na system węzłów (graf) [61]. W przypadku, gdy obiekt ma dostać się do określonego punktu na mapie, wyszukiwany jest najbliższy mu węzeł grafu i wyznaczana jest ścieżka do niego, po której obiekt będzie się poruszał. O ile wydajność takiego rozwiązania jest duża, to do realizacji pomysłu wymagane jest naniesienie punktów grafu na mapę. Wykorzystywany jest, oprócz grafów, model hybrydowy, który łączy algorytmy wyszukiwania ścieżki i węzły dzieląc mapę na obszary. Obszary pełnią funkcję ograniczającą dla grafów, a przejścia z jednego obszaru do drugiego są zdefiniowane w konkretnych punktach. Jeżeli miejsce, do którego trzeba

przenieść obiekt jest w tym samym obszarze, to do obliczeń wykorzystywane są tylko węzły w tym właśnie obszarze.

3.2.3 Algorytmy pracujące na dużych ilościach danych

SI wymaga zaprowadzenia kompromisów [7]. Wymyślane algorytmy mogą być używane w przeprowadzaniu symulacji i wyciąganiu wniosków, a następnie podejmowaniu odpowiadającym im działań, jednak ceną za skuteczność algorytmów może być miejsce na dysku, którego algorytm będzie potrzebował, aby zapisać dane, na których będzie operował. Ilość pamięci potrzebna na przechowanie danych analizowanych przez SI okazuje się czasami za duża. W takim przypadku trzeba ograniczyć możliwości SI, albo dane, które będzie w stanie analizować. Prowadzi to do zmniejszenia możliwości elementów sterowanych przez komputer - zmniejszenie puli ruchów NPC, ilości jednocześnie sterowanych jednostek czy ograniczenie zbierania danych o poczynaniach gracza to przykłady radzenia sobie ze zmniejszeniem zapotrzebowania SI na pamięć.

3.2.4 Dalszy rozwój możliwości SI

Niewątpliwie duża część gier komputerowych opiera się na tworzeniu wyzywającej intelektualnie i dzięki temu satysfakcjonującej rozgrywki - można powiedzieć, że te gry opierają się na rozbudowanej SI. Warto jednak zauważyć, że, przedkładając znowu przykład gry w szachy, graczy nie zawsze interesuje idealnie inteligentny przeciwnik [12]. Spotykając się z takim przeciwnikiem gracz mógłby uznać, że jest z góry skazany na porażkę i w wielu przypadkach miałby rację. Wbrew pozorom tworzenie „gorszej" sztucznej inteligencji jest jest ulepszeniem. SI często mogłoby bez problemu wygrać z graczem, natomiast jest dostosowywane do jego umiejętności właśnie w ten sposób, żeby stanowić godnego przeciwnika, z którym jednak gracz jest w stanie wygrać. Algorytm musi celowo wykonywać akcje, które można uznać za błąd (symulacja błędu ludzkiego). Projektant musi wyważyć, w jaki sposób ma przebiegać pomyłka i jaka ma być jej częstotliwość - trzeba uważać, żeby gracz nie dostrzegł zbyt wielu błędów lub nie interpretował ich w ten sposób, gdyż

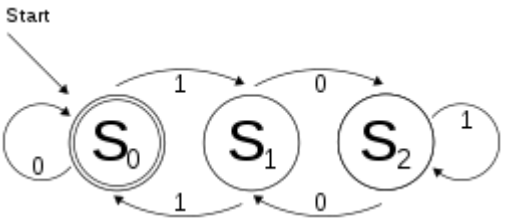
może przez to zniechęcić się do gry. Wychodząc z takiego założenia dochodzimy do pomysłu adaptacji możliwości SI dopasowującej się do umiejętności gracza.

Jak widać rozwój SI w grach komputerowych niekoniecznie oznacza osiągnięcie nieomylnego algorytmu - okazuje się też, że to właśnie odbieganie od takiego ideału może być celem projektanta i być dużo trudniejsze w realizacji i dzięki takiemu sposobowi stawiania wyzwań prowadzić do rozwoju SI w ogóle.

Kolejne rozdziały opisują najpopularniejsze metody implementacji SI w grach komputerowych.

3.3 Skończone maszyny stanów [6][68]

Jedna z metod implementacji SI to wykorzystanie skończonej maszyny stanów, nazywanej także automatem skończonym (ang. *finite state machine* (*FSM*)) [68]. FSM to abstrakcyjny, matematyczny model zachowań, który może jednocześnie przyjmować jeden z określonych stanów w danym czasie. Przejścia między stanami realizowane są przy spełnieniu zdefiniowanych warunków.



2. Przykład schematu automatu skończonego [68]

Zaprojektowane SI działa na obiekcie, który jest w jakimś stanie [6]. Komputer może wybrać do jakiego innego stanu przejść poprzez dostępne dla niego możliwości. Istotną rolą przy przechodzeniu między stanami jest priorytetyzacja zachowań - jeżeli komputer steruje NPC z podstawowym zadaniem eliminacji postaci gracza, to gdy NPC wyczerpie swoją własną energię, to jego cel może zmienić się na jak najszybsze dostanie się do miejsca, w którym ta energia zostanie przywrócona, dzięki czemu będzie mógł zrealizować swój cel.

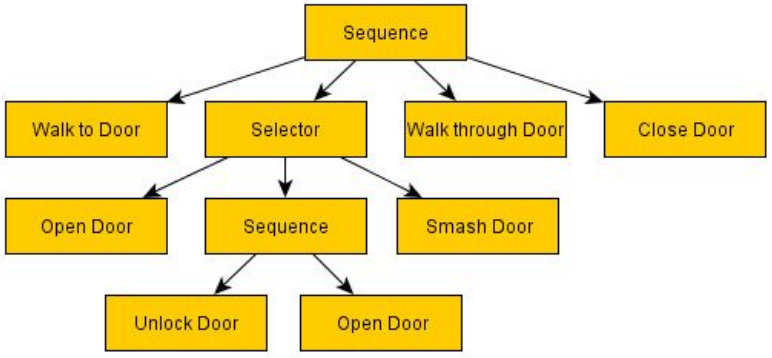
Skończone maszyny stanów to metoda swego czasu najpopularniejsza [6], dzięki temu znamy jej mocne i słabe strony. Do tych pierwszych zaliczamy oczywiście czytelność - sama koncepcja jest bardzo intuicyjna i wydaje się naturalnym podejściem do problemu. Dla projektanta tworzenie algorytmu używając skończonej maszyny stanów nie jest przez to dużym problemem. Jest przewidywalne w działaniu i łatwe do

prześledzenia. FSM jest szeroko stosowane gdy mowa obiektach, które mogą znajdować się niewielkiej liczby stanów.

Do wad tego rozwiązania możemy zaliczyć fakt, że przy zwiększaniu liczby stanów, kwadratowo rośnie ilość połączeń między nimi. Maszyny stanów są słabo skalowalne, a kod trudy w ponownym użyciu. Gdy projektowana jest SI np. wysokiego poziomu (decyzja czy np. atakować gracza, poruszać się do jakiegoś celu, oczekiwać na interakcję z graczem, itp.), to trudno jest jednocześnie realizować w ramach tej samej maszyny zachowania niskiego poziomu (np. jak dokładnie ma wyglądać stan atakowania gracza). W takiej sytuacji mogą pojawić się współpracujące ze sobą metody realizacji SI. Jak pokazuje przykład pierwszej gry Wiedźmin [69], wprowadzono tam, obok FSM, także kolejkę akcję atomowych, wywoływanych przez stany z maszyny stanów [6].

3.4 Drzewa zachowań [6]

Drzewo zachowań (ang. *behavior tree*), jak sama nazwa wskazuje, opiera się na strukturze drzewa (zobacz: rysunek 3 oraz 23). Korzeń-mózg odpowiada za podejmowanie wysokopoziomowych decyzji. Im bliżej znajdujemy się liści, tym niższego poziomu akcje wykonujemy. Wchodząc w dane rozgałęzienie zakładamy, że ostatecznie będziemy w stanie dojść do jakiegoś liścia odpowiadającego za pojedynczą, atomową akcję, która może być wykonana. Wybór rozgałęzień polega na rozpatrywaniu wcześniej zdefiniowanych warunków. Rozpatrywanie możliwości następuje w odstępach 1-nej czy 2-ch sekund, lub na żądanie. Odstępy czasowe nie tylko optymalizują program (oszczędność mocy obliczeniowej), ale też wprowadzają naturalne opóźnienie przy podejmowaniu decyzji, tak jak w przypadku istoty ludzkiej.



3. Część drzewa zachowań przedstawiająca sekwencję otwierania drzwi [70]

Często też różnica w czasie reakcji po prostu nie ma żadnego znaczenia. W związku ze stałym monitoringiem pewnych zmiennym, muszą być one zapisywane w jednym, centralnym miejscu.

Wywołanie węzła (decyzja algorytmu, aby wykonać akcję zdefiniowaną w węźle) jest związane z jego aktywacją i aktualizacją (np. wykonywanie akcji w pętli). Po zakończeniu wykonywania węzła wywołujemy funkcję odpowiedzialną za dezaktywację węzła. Przed naturalnym zakończeniem procesu wykonywania węzła (gdy akcja trwa dłużej, niż kolejne rozpatrzenie węzłów), algorytm może zmienić obecnie aktywny węzeł. W takiej sytuacji poprzedni proces jest przerywany.

Jak widać na rysunku nr. 3, węzły nie muszą być pojedynczą akcją, ale mogą być ich całą sekwencją (ang. *sequence*). Rysunek 3 prezentuje sekwencję otwierania drzwi, od akcji odpowiedzialnej za podejście do nich (*Walk to Door*), przez otwarcie (*Open Door*, sekwencja akcji *Unlock Door* (odblokowanie, np. kluczem) i *Open Door*, *Smash Door* (zniszczenie)) i przejście przez nie (*Walk through Door*), aż po zamknięcie (*Close Door*).

Problemami w drzewach zachowań jest określenie wykonywanej akcji w przypadku drzew niedeterministycznych. W takich drzewach 2 węzły mogą mieć równy priorytet wykonania oraz mieć dokładnie takie same warunki rozpoczęcia. Problemem jest też uzależnienie jednego węzła od wewnętrznego stanu innego węzła, czy wzrost priorytetu akcji po rozpoczęciu jej wykonywania - chcemy, aby akcja nie była zbyt szybko wywłaszczona, ale żeby taka możliwość ciągle istniała. Kolejny problem to sposób podejmowania decyzji i wywłaszczania akcji, czasami niezgadzający się z hierarchią drzewa. Część problemów można rozwiązać wprowadzając priorytety węzłów, jednak takie rozwiązanie prowadzi kosztowniejszego procesu decyzyjnego (trzeba rozpatrzyć cały szereg węzłów i je porównać) i utrudnia czytelność drzewa.

W kolejnym rozdziale opiszemy rodzaje gier oraz podamy ich wybrane przykłady.

4. Rodzaje gier komputerowych i przykłady zastosowania SI

Każda z przedstawionych poniżej typów gier charakteryzuje się unikalnym zestawem cech, od których często zależy w jaki sposób możemy w nich przedstawić SI.

Warto zauważyć, że w obecnych czasach bardzo często te gatunki są ze sobą mieszane aby sprostać coraz wyższym wymaganiom odbiorcy.

4.1 Gry strategiczne

Gry strategiczne wymagają od gracza analitycznego myślenia i planowania - możemy myśleć o nich jako o bardzo rozbudowanych szachach. Gracz musi obrać jakąś strategię na wygranie gry, podejmować decyzje taktyczne i logistyczne, także dotyczące gospodarki czy odkrywania nowych terenów.

4.1.1 Strategiczne gry turowe

TBS (ang. *turn-based strategy* (*TBS*)) to gatunek, do którego należą wspomniane już wielokrotnie szachy. Rozgrywka podzielona jest na tury, w których można poruszać się jednostkami, atakować przeciwnika, rozbudowywać swoje wojska i ekonomię. W tego typu grach SI reaguje na poczynania gracza - jeżeli wykonamy ruch zagrażający jednostkom sterowanym przez przeciwnika, gra odpowiednio zareaguje przegrupowywując jednostki. Przeciw naszym wystawi takie, które z większą skutecznością odeprą atak, czy nawet będą mogły przeprowadzić kontratak w kolejnej turze.

Tworząc grę taką jak np. Heroes of Might and Magic [74] umożliwiamy graczowi walkę z komputerem za pomocą wielokrotnie większej ilości bierek, niż w szachach. Do tego dostajemy do dyspozycji kolejne umiejętności: te dostępne dla niektórych z naszych pionów, ale także specjalne czary, którymi możemy zmienić losy partii. Po drugiej stronie stołu zasiada SI mająca równie wielkie pole do popisu.

4. Walka rozgrywana turowo w grze Heroes of Might and Magic III HD [74]

Zagłębiając się w tematykę TBS i wykorzystania w niej SI możemy napotkać na dostępne dla każdego konkursy projektowania SI [11]. Pisząc algorytm w wybranym przez siebie języku wysyłamy go potem, aby konkurował z innymi.

4.1.1 Strategiczna gra czasu rzeczywistego

RTS (ang. *real-time strategy* (*RTS*)) to gry w których (w przeciwieństwie do turowych gier strategicznych) akcje jednocześnie mogą podejmować wszyscy gracze, a skutki tych decyzji widoczne są natychmiast [71]. Nie trzeba czekać na ruch przeciwnika, przez co decyzje należy podejmować szybko, narażając się na błędy wynikające ze stresu i braku rozwagi. Gry RTS to pole do popisu dla SI. Znajdziemy tutaj rozwiązania dla problemów wyszukiwania ścieżek i zarządzania zasobami. Wykorzystuje się SI także w taki sposób, jak w strategicznych grach turowych (zobacz: 4.1.1 Strategiczne gry turowe).

Przykładem RTS jest seria Total War [72]. To cykl gier skupiających się na zarządzaniu państwem i podboju kolejnych terenów. Zobaczymy tutaj obszerną mapę, często wzorowaną na fragmencie mapy świata. Rozgrywka zwykle wprowadza nas w jakiś okres historii świata i pozwala zmienić jej bieg, oddając nam pod kontrolę całe

5. Walka rozgrywana w czasie rzeczywistym w grze Total War: ROME II - Empire Divided [75]

państwo, wraz z jego polityką, gospodarką i wojskowością. Komputer musi dostosować się do wykonywanych przez nas akcji, zbierając informacje na nasz temat przy pomocy szpiegów, proponować sojusze, wystawiać przeciw nam wojska kontrujące nasze jednostki. To walka w serii Total War jest częścią rozgrywki, nad którą warto się pochylić - bitwy można rozegrać automatycznie je rozstrzygając, albo wcielić się w dowódcę i walczyć z armią wroga. Każda z setek, lub nawet tysięcy jednostek biorących udział w walce ma swój własny system zachowań.

Kolejny wybrany przykład tego typu gry to także na grę Black & White [73]. Wcielamy się w boga mogącego za pomocą cudów zdobywać wyznawców, pomagać im, karać i walczyć z innymi bóstwami. Ciekawym aspektem tej gry jest to, że do naszej dyspozycji zostaje oddany reprezentant naszej boskiej władzy, tzw. chowaniec. Jest to zwierze ingerujące w świat gry na różne sposoby - to, jak zareagujemy na podejmowane przez niego akcje będzie miało wpływ na jego przyszłe zachowanie (determinowane przez zbierane przez SI dane). Jeżeli nie zareagujemy gniewem na zjadanie poddanych, to możemy mieć potem problem z populacją wyznawców. Z kolei chwaląc chowańca przy podejmowaniu przez niego dobrych zachowań doprowadzi do tego, że będzie on pomagał mieszkańcom dbając np. o to, aby skład drewna był zawsze pełny. A sami mieszkańcy zostawieni sami sobie będą robić to, czego najbardziej potrzebują - np. zbierać jedzenie, czy rozmnażać się.

4.1.3 Gra ekonomiczna

W grach RTS zazwyczaj występuje element ekonomiczny. Często kluczem do wygranej jest zdobycie kontroli nad ograniczonymi przez mapę zasobami, które pozwalają nam na produkcję odpowiednio zaawansowanego wojska. Gry ekonomiczne (ang. *business simulation game*) to jednak także tworzenie miast, a celem może być uzyskanie odpowiedniej ilości pieniędzy z podatków, wybudowanie prestiżowego budynku, itp. Za przykład można tutaj podać SimCity [76] i Caesar III [77]. W takich grach występuje wiele różnych modułów SI, poczynając od wyszukiwania dróg, aż do dogłębnych analiz ekonomicznych mających na celu dopasować warunki współpracy. Warto zwrócić uwagę na Banished [78] - zobaczymy tutaj rozwiązanie efektywnego znajdowania ścieżek w dużym mieście czy dynamicznego przydzielania pracowników do zadań.

Innym przykładem są gry symulujące pracę menadżerską, takie jak Football Manager [79], gdzie zarządzamy drużyną sportową nie tylko kierując finansami i składem zespołu, ale także taktyką, jaką obejmują gracze na boisku. Na podstawie naszych decyzji SI jest wykorzystana do symulacji wyników oraz do generowania graczy, którymi zarządzamy.

4.1.4 Obrona wież

TD (ang. *tower defense* (*TD*)) polegają na obronie punktów na mapie, do których próbują dostać się fale wrogów. W TD głównym sposobem obrony jest stawianie wież, ostrzeliwujących nadchodzące zastępy wrogów. Chociaż zwykle atakujące jednostki mają dość prostą SI, to czasami sposób wyznaczania przez nie ścieżki do celu uwzględnia zasięg naszych umocnień. Jednostki wroga dysponują też umiejętnościami, które wykorzystywane są w odpowiednim momencie. Przykładem takiej gry jest Sanctum [80]. Oprócz stawiania różnych wież sterujemy też zadającym obrażenia bohaterem. Napotkani wrogowie nie muszą po prostu iść do celu, ale priorytetem dla nich może stać się eliminacja gracza, lub stawianych przez niego wież.

4.2 Komputerowa gra fabularna

Komputerowa gra fabularna (ang. *computer role-playing game* (*cRPG*)) to gra, w której kluczowe jest przeżywanie przygody i rozwój bohatera, lub bohaterów. Na swojej drodze spotykamy wielu towarzyszy i przeciwników. Od SI zależy w jaki sposób będą się oni zachowywać i walczyć. W grach RPG ważna jest immersja - utożsamienie się z bohaterem i wtopienie się w świat. Druga kwestia to nie tylko zaawansowana grafika, ale także cykl życia napotykanych postaci. Nie jest to najważniejsza rzecz, ale zwracamy uwagę na to, jak zachowują się postaci - czy w nocy idą spać?, jak spędzają dzień?, mają rodziny?, jedzą? Nie muszą to być skomplikowane akcje, jednak planując, chociażby ogólnikowo SI tak, aby uwzględniało te aspekty w zauważalny sposób poprawia odbiór gry.

4.2.1 Fabularna gra akcji

Fabularna gra akcji (ang. *action role-playing game*, *action RPG*) to gra RPG z systemem walki opartym o refleks gracza. W takich grach, dzięki SI świat może monitorować nasze zachowania. W Oblivionie [81] została przedstawiona technologia Radiant AI [82] odpowiadająca za nieprzewidywalne i dynamiczne odpowiedzi świata na poczynania gracza. Do tego jak radzimy sobie w grze dopasowany jest poziom przeciwników, NPC obserwują poczynania gracza (komentarz odnoszący się do wykonanej misji; zastępowanie sklepikarza w jego sklepie, w przypadku jego śmierci) i jego umiejętności (jeśli jesteś wyszkolony w kradzieży kieszonkowej, strażnicy mogą to skomentować), a także zapamiętują w jaki sposób byli traktowani (nasłanie zabójców w odpowiedzi na zabójstwo członka gildii).

Pierwsza gra, która znacznie rozwinęła SI w grach z otwartym światem to Gothic [83]. Mieszkańcy rozmawiają losowymi kwestiami, które czasem nawet układają się w logiczną całość. NPC zapamiętują, że wcześniej zostali przez nas pobici i nie chcą z nami rozmawiać, a sprawę zgłosili do ratusza. Przed Gothic NPC byli tylko kiepskim tłem, wałęsającym się bez celu po miastach, a interakcja z nimi była bardzo ograniczona.

6. Sklepikarze oczekujący na klientów przy swoich straganach w grze Gothic II [83]

W grze Middle-Earth: Shadow of Mordor [84] zaproponowano zaawansowany mechanizm zwany systemem nemezis. Polegał on na tym, że ułożeni w hierarchii wodzowie orków współpracowali lub rywalizowali ze sobą, próbując zwiększyć swoją potęgę. Tworzeni oni byli za po mocą losowych, wpływających na siebie cech, które po interakcji z graczem mogły się zmieniać. Gdy przeciwnik nas pokonał mógł zyskać nowe moce. Gdy z kolei pokonujemy wrogów, nabierają oni do nas respektu.

Assassin's Creed: Unity [85] doskonale natomiast symuluje zachowania tłumów NPC, za co dostało nagrodę społeczności w kategorii "AI Technology in a Supporting Role" [86].

4.2.2 Hack'n'slash

W grach hack'n'slash (ang. *hack* – rąbać, *slash* – ciąć) przemierzamy mapę pozbawiając życia niezliczonej ilości wrogich jednostek, które wydają się nie mieć zbyt dużej inteligencji. Zauważamy, że po prostu widząc nas atakują, a jeżeli muszą do nas podejść, to zwykle wybierają najkrótszą drogę. Przykładem bardziej zaawansowanej SI

w tego typu grze jest Diablo III [87]. Niektórzy wrogowie atakujący z dystansu poruszają się sprawiając, że trudniej ich trafić. Zdolności specjalne używane są w taki sposób, że odcinają nam drogę ucieczki, co często potrafi skończyć się śmiercią naszego bohatera.

4.3 Strzelanka pierwszoosobowa

W grach FPS (ang. *first-person shooter* (*FPS*)) patrzymy z oczu naszego bohatera i używamy broni palnej w celu wykonania określonych działań zdefiniowanych np. przez scenariusz gry wojennej, gdzie wcielamy się w żołnierza. W nowoczesnych grach często wprowadzany jest element sterowania towarzyszami. Kierowany przez nas oddział wykonuje wydawane przez nas rozkazy, lub po prostu podąża za nami jak najlepiej wpasowując się w nasz styl gry.

Chociaż gra F.E.A.R [88] ma już ponad 10 lat ciągle jest czołowym reprezentantem realizacji SI w grach FPS [5]. Wprowadzała podejmowanie działań przez przeciwników na podstawie kontekstu sytuacji, w której się znajdują: wykorzystanie otoczenia, wyszukiwanie barykad i zasłon, wzajemne osłanianie się i flankowanie gracza [8]. W Halo [89] użyto SI wykreowanej przy użyciu drzewa zachowań - jedna akcja może prowadzić do kilku reakcji. Przeciwnicy prowadzą skuteczny ogień zaporowy, odrzucają granaty i współpracują z towarzyszami. Jeżeli natomiast rozbijemy oddział wroga, to niedobitki potrafią uciec i przegrupować się. Kontrprzykładem jest pierwsza część Just Cause [90] - jest to gra, w której nasi przeciwnicy często po prostu stoją w miejscu prowadząc ostrzał w naszą stronę.

4.4 Gry sportowe

4.4.1 Gry walki

SI w sportowych (ang. *sports game*) grach walki (ang. *fighting games*), to przykład SI symetrycznej. Komputer ma do dyspozycji postać dysponującą podobnym zasobem ruchów dostępnych dla bohatera. Zasypując się nawzajem gradem ciosów zauważamy, że komputer wie, kiedy wykonać blok, a kiedy może pozwolić sobie na wyprowadzenie skutecznego ataku.

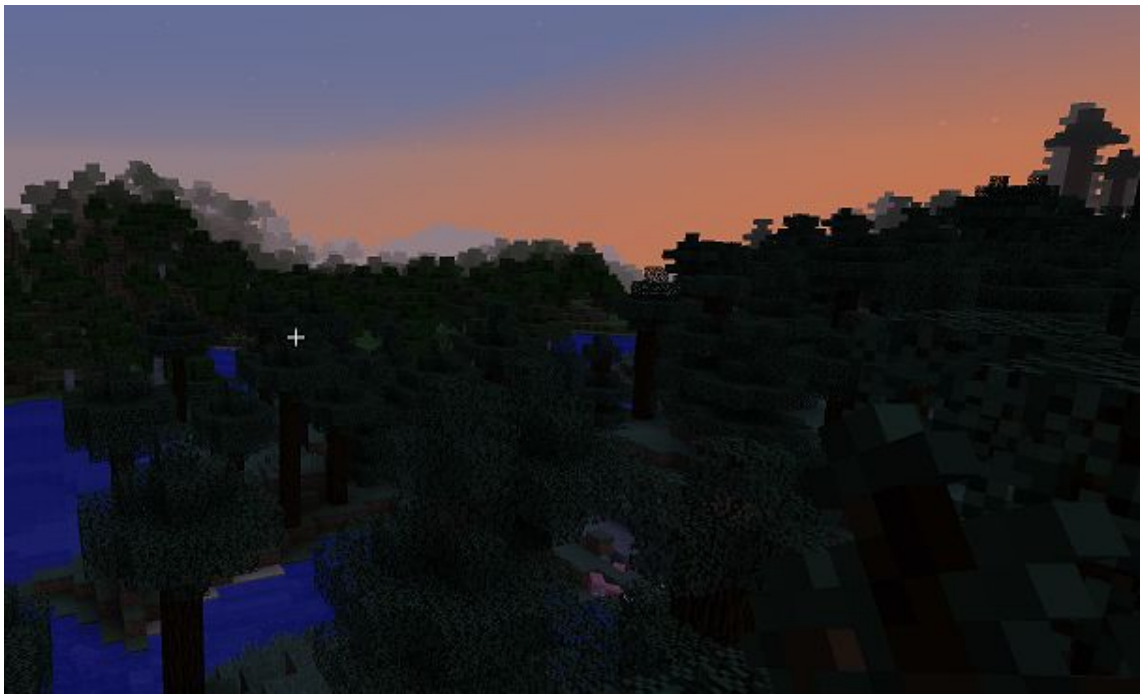
4.4.2 Komputerowe wyścigi

Gry wyścigowe (ang. *racing video game*), takie jak Forza Horizon 3 [91] potrafią analizować sposób, w jaki gramy i uczyć się go. Nasi przeciwnicy przewidują dzięki temu nasze ruchy i mogę np. zajechać nam drogę. Innym przykładem tego rodzaju gier jest Mario Kart [92] - chcąc być pierwsi na mecie możemy to osiągnąć także przez spowolnienie innych graczy. Tak samo gra, dzięki implementacji SI, mając do dyspozycji te same narzędzia, co my (miny i bomby, zamazywanie ekranu graczowi, czasowe przyspieszenia i osłony), decyduje w jaki sposób i kiedy z nich korzysta.

4.5 Piaskownice

Piaskownice (ang. *sandbox*) to gry, w który gracz jest niejako pozostawiony sobie samemu. Elementy tego typu gier występują w już wyżej opisywanych gatunkach. Piaskownice jednak (np. Minecraft [65], Terraria [66], No Man's Sky [67]) wykorzystują wspomnianemu już proceduralnemu generowaniu zawartości. Za pomocą algorytmów SI mogą być tworzone poziomy, tekstury, postacie, muzyka, zasady czy zadania. Tak tworzona zawartość może być podstawą dla danej gry, albo być odpowiedzialna zaledwie za np. dekoracje otoczenia. Proceduralna generacja to podejście pozwalające wykreować duży (nawet fizycznie niemożliwy do przemierzenia), niepowtarzalny świat bez potrzeby zatrudniania dużego zespołu projektantów. Kreowana zawartość może być dostosowywana do konkretnego gracza - oferować treść odpowiednią dla jego stylu gry lub poziomowi doświadczenia (odnajdywania się w grze), dzięki czemu gra jest ciągłym wyzwaniem.

Treść może być tworzona przy rozpoczynaniu nowej rozgrywki (offline generation), lub w czasie jej trwania (online generation). Tworzony świat może być rzeczywiście losowy, w związku z tym algorytmy muszą tworzyć treść wychodząc z losowego ziarna (ang. *seed*) lub, zgodnie z inną naturą - można zastosować algorytmy deterministyczne. Te dwa podejścia mogą również być ze sobą są łączone. Po wygenerowaniu spójnego świata czy planszy, SI może być odpowiedzialna także za jej dostosowanie do rozgrywki na podstawie dostarczanych danych (np. na temat wyników rozgrywanych przez ludzi gier) [64].



7. Świat wygenerowany w grze Minecraft. Widok na las [opracowanie własne]

Treść generowana proceduralnie może skutkować powstawaniem nieprzewidywalnych tworów, dlatego też ważne jest wprowadzenia testów i norm, które muszą spełniać twory. Normy te można też zmieniać, aby dostawać inne wyniki algorytmów.

4.6 Gry przygodowe

Gry przygodowe (ang. *adventure game*) przeszły ostatnimi laty pewną metamorfozę. Rozgrywka najczęściej była bardzo prosta i sprowadzała się do przemierzania zwykle niezbyt złożonego świata zbierając przedmioty i poszukując informacji, które poprowadzą nas dalej poprzez fabułę gry. Wraz z rozwojem technologicznym do przodu poszła oprawa graficzna, a co za tym idzie także sposób odwzorowywania bohaterów, wraz z symulacją odczuwania przez nich emocji. Pokazywanie emocji stało się uzupełnieniem prowadzonej narracji. Gdy widzimy, co odczuwa postać, którą gramy, możemy się z nią lepiej utożsamić. Tym lepszy efekt uzyskamy, jeżeli nasze działania sprawią, że to sterowany bohater będzie się "czuł" tak jak my. Przechodząc dalej - jeżeli nadamy NPC odpowiednio wyczulone możliwości monitorowania środowiska (nasze poczynania, inni NPC, świat, itp.), możemy

zaimplementować odpowiednie, dopasowane do okoliczności reakcje. Sposób, w jaki rozmawiają z nami napotykane postacie, może być budowany dzięki SI.

4.7 MMO

Gry MMO (ang. *massively multiplayer online*) skupiają całe masy żywych ludzi, mogących współpracować lub konkurować w jednej grze. Grami MMO mogą być zarówno gry RPG (np. World of Warcraft [94]), jak i chociażby gry ekonomiczne (np. przeglądarkowa gra Ogame [93]). Często sami gracze próbują "pomóc" sobie w grze

8. Grupa graczy współpracujących w walce w grze World of Warcraft [95]

tworzą swoją własną SI, lub częściej - wykorzystując już wykreowaną. Realizujący ją program potrafi zazwyczaj wykonywać proste czynności, takie jak zarządzanie jednostkami, czy sterowanie postacią w celu wykonania powtarzalnej czynności. Takie proste programy jednak mogą być punktem wyjścia do ciekawszego zagadnienia, jakim jest próba stworzenia programu, potrafiącego grać w różne gry bez poprzedniego dostosowania jego algorytmów. Zagadnienie to nazywane jest GGP (ang. *General Game Playing*) i rozwija ono uczenie się od człowieka sposobów rozgrywania różnych gier poprzez pojedynczy program [13].

4.8 Symulacje

Symulacje (ang. *simulation video game*) to bardzo szeroka rodzina gier (od symulatorów posiadania zwierzęcia, przez symulatory kierowania pojazdami, do symulacji życia) i SI znajduje tu bardzo zróżnicowane zastosowania.

9. Wskaźniki potrzeb dla postaci z gry The Sims [97]

Ciekawy pomysł przedstawia gra Sims [96], w której kontrola gracza nad podopiecznymi (simami) jest ograniczona. Prowadzą oni normalne, codzienne życie, a my dajemy im tylko wskazówki. Każdy sim ma własne zdolności, zainteresowania i cechy. Simy kierują się pragnieniami - jeżeli chcą jeść, to jedzą, jeżeli chcą spać, to śpią[13]. Simy mają możliwość wchodzenia w interakcje z różnorakimi przedmiotami, z których każdy posiada 2 informacje wykorzystywane przez SI: jaką korzyść przyniesie jego użycie, oraz w jaki sposób dochodzi do interakcji [5].

Opisane zostały gatunki gier komputerowych i sposób, w jaki korzystają ze SI. Poniższy rozdział opisuje proces powstawania gier komputerowych.

5. Tworzenie gier komputerowych

Tworzenie gier komputerowych obejmuje wiele aspektów: od samego pomysłu i koncepcji, przez projektowanie i programowanie, aż do publikacji i utrzymania.

5.1 Twórcy gier komputerowych

Tworzeniem gier komputerowych zajmuje się zespół ludzi: specjaliści w swoich dziedzinach, pracujący nad tytułami, jak i całe organizacje.

5.1.1 Zespół deweloperski [26]

Aby gra powstała istotny jest też oczywiście zespół ją tworzący. Każdy z członków zespołu ma przypisane konkretne role.

Projektant (ang. *designer*) to osoba zajmująca się projektowaniem zasad rozgrywki. Przy dużych projektach często zatrudnionych jest wielu projektantów, z których jeden przewodzi innym. Projektanci wymyślają jak gra ma wyglądać, działać i jak ma się w nią grać.

Artyści zajmują się wizualną stroną gry. Tworzą tła, modele postaci, animacje, tekstury, wygląd zjawisk środowiskowych (np. deszcz). Zajmują się także przerywnikami filmowymi. Ich praca musi być spójna, aby nie było widać różnic pomiędzy różnymi elementami widocznymi w grze.

Dźwiękowiec (ang. *sound engineer*) zajmuje się udźwiękowieniem gry. Tworzy, wyszukuje, dopasowuje dźwięki do elementów gry (postacie, zwierzęta, przedmioty, środowisko, muzyka).

Programiści (ang. *programmer*) są odpowiedzialni za pisanie kodu gry i dostarczenie narzędzi do projektanta poziomów (ang. *level designer*). Programiści wykorzystują zasoby dostarczane przez artystów, projektantów, dźwiękowców. Dzięki nim umieszczone w grze przez projektanta poziomów elementy będą zachowywały się w oczekiwany sposób. Takie elementy muszą być możliwe do zmodyfikowania w prosty sposób (przez parametry), gdyż projektanci poziomów niekoniecznie muszą znać się na programowaniu, a dzięki modyfikacjom będą oni potrafili ustalić jaka wersja danego zasobu będzie w danym miejscu gry najlepiej pasowała. Programiści odpowiedzialni są za symulację fizyki, SI, wykorzystanie grafiki, dźwięku, poprawną implementację zasad rozgrywki, stworzenie interfejsu użytkownika (ang. *UI - user interface*), sterowanie w grze, komunikację sieciową, itd.

Testerzy (ang. *tester, QA - quality assurance*) sprawdzają, czy stworzone elementy i końcowy produkt są zgodne z dokumentacją. Szukają błędów, odpowiednio je dokumentują, zwracają raport do osób za dany błąd odpowiedzialnych. Starają się przewidywać niestandardowe zachowania gracza i upewniają się, że gra jest w stanie poradzić sobie z taką sytuacją. Analizują też, czy zaimplementowane koncepcje się sprawdzają i zapewniają informację zwrotną (ang. *feedback*) projektantom.

Każda grupa wyżej wymienionych osób ma nad sobą **kierownika zespołu** (ang. *team leader*), który oprócz współpracowania z zespołem w procesie produkcji, ma także przydzielać zadania, kierując je do konkretnych pracowników.

W kolejnym rozdziale opisane są studia gier komputerowych, które zatrudniają osoby pełniące opisane wyżej role.

5.1.2 Studia gier komputerowych

Jako odpowiedzialnych za najpopularniejsze i często najbardziej rozpoznawalne tytuły to właśnie często duże studia gier komputerowych najczęściej kojarzymy, jako instytucje odpowiedzialne za tworzenie gier. Takie studia (producenci) zwykle wspierane są przez dużego wydawcę od strony finansowej i marketingowej. Są to zwykle relatywnie duże zespoły, składające się z od 20, do nawet 100 i więcej osób. W przeciwieństwie do małych, niezależnych producentów zwykle każdy z członków zespołu specjalizuje się w określonych aktywnościach, nie dzieląc swojej uwagi na wiele aspektów tworzonej gry [26].

Jako przykłady najbardziej popularnych producentów możemy zaliczyć Nintendo [98], Valve Corporation [99], Rockstar Games [100], Electronic Arts [101], Activision Blizzard [102][34]. Także polski rynek jest bogaty w odnoszących sukcesy producentów takich jak CD Projekt Red [103], Techland [104], 11 bit studios [105].

Następny rozdział opowiada o grach nie tworzonych przez duże studia.

5.1.3 Niezależna gra komputerowa [28][35]

Terminem "niezależna gra komputerowa" (ang. *independent video game*, potocznie gra indie, indyk) określamy grę komputerową, która została stworzona właśnie przez pojedynczą osobę, lub mały zespół, który nie jest finansowany przez wydawcę. W dzisiejszych czasach twórcą gier komputerowych może zostać każdy. Dzięki łatwo dostępnym, często darmowym, narzędziom developerskim, niezliczonej ilości poradników i gotowych do wykorzystania elementów coraz łatwiej jest stworzyć grę, którą można następnie udostępnić za darmo, a nawet sprzedawać za pomocą platform takich jak Steam (Steam Greenlight i jego następca - Steam Direct) [106], czy

Google Play [107] (aplikacje mobilne) - możliwość dystrybucji cyfrowej znacznie zmniejsza wymagany wkład własny niezależnego twórcy.

Chociaż niezależne tworzenie gier komputerowych jest często traktowane jako hobby, kiedy twórcy pracują nad danym tytułem w czasie wolnym, to produkcje rozwijają rynek często przedstawiając nowe pomysły, interesującą oryginalną oprawą graficzną, czy zaskakującą fabułą.

Brak wsparcia ze strony wydawnictwa rodzi finansowe problemy, które często przedłużają proces produkcji lub nawet są przyczyną zarzucenia developmentu. Kolejnym czynnikiem, który przyczynił się do zwiększenia ilości wydawanych gier *indie* są nowe formy finansowania. Crowdfunding to metoda dająca duże wsparcie niezależnym twórcom. Polega na ogłoszeniu przez twórcę zbiórki na dany cel: w przypadku gry komputerowej opisuje się koncepcję gry oraz ustala ile pieniędzy jest potrzebnych aby osiągnąć konkretne efekty. Określenie wielu progów kwotowych umożliwia stopniowanie zaawansowania gry - jeżeli ludzie z całego świata "zrzucą się" zbierając pewną kwotę dostaną grę w wersji podstawowej. Przy osiągnięciu kolejnego progu kwotowego, organizator zbiórki może się zobowiązać do dodania kolejnej zawartości: nowych przeciwników, wydanie gry na więcej platform, dodatkowe bonusy dla sponsorów, itp. Jako przykład można podać grę Divinity: Original Sin 2 [109], którą finansowano w ten sposób. Przy osiągnięciu progu 1 miliona dolarów zaproponowano wprowadzenie umiejętności rasowych (osobne dla ludzi, krasnoludów, itd.), a po zdobyciu kolejnych 650 tysięcy dolarów obiecano wsparcie dla modyfikacji [108]. Zwykle podczas przeprowadzania zbiórki za określoną, wpłaconą kwotę dostaje się określoną nagrodę: czy to samą grę, czy też np. możliwość nazwania jakieś postaci i wymyślenie jej historii.

W ciągu ostatnich lat można zauważyć bardzo szybki rozrost rynku niezależnych gier komputerowych - Jak podaje serwis Steamspy [110] ilość wydawanych gier wzrosła w połowie 2015 roku ponad 15-krotnie [35] w stosunku do początku roku 2013. W ciągu ostatnich lat można zauważyć bardzo szybki rozrost rynku niezależnych gier komputerowych. Jest to jednocześnie zjawisko pozytywne, jak i negatywne. Z jednej strony gracze mogą liczyć na dużo więcej ciekawych i tanich gier, a z drugiej narażeniu są na wiele kiepskich produkcji pełnych błędów i niedoskonałości, stworzonych tylko po to, żeby zarobić na reklamach lub kuszących niską ceną.

W związku z zalewem kiepskich produkcji ambitnym twórcom trudniej wybić się spośród konkurencji i dotrzeć do odbiorcy.

Poniższy rozdział opisuje wybrana narzędzia, które wspierają twórców w produkcji gier.

5.2 Przykładowe silniki służące tworzeniu gier komputerowych [17][23]

Silnik gry (ang. *game engine*) [19] to oprogramowanie wykorzystywane do produkcji gier komputerowych. Wykorzystywane są przy produkcji tytułów na różne platformy: komputery klasy PC, konsole oraz urządzenia mobilne. Silniki wspierają zazwyczaj wyświetlania grafiki 2D i 3D, obsługę kolizji, symulacji fizyki, odtwarzanie animacji, dźwięku, tworzenie sztucznej inteligencji, pisanie skryptów i zarządzanie pamięcią. Silniki znacznie ułatwiają portowanie (przenoszenie gier na inne platformy sprzętowe i programowe) gier - silniki gier zaliczamy do oprogramowania pośredniczącego (ang. *middleware*). Wraz z silnikiem możemy wykorzystywać także inne narzędzia (np. MS Visual Studio jako edytor kodu z silnikiem Unity, programy do tworzenia grafiki/obiektów 3D).

5.2.1 Unity 5

Unity [20] to silnik umożliwia pisanie gier na bardzo dużo platform, także eksport gry do wersji, którą można uruchomić przez przeglądarkę internetową. Wspiera także wirtualną rzeczywistość dla większości dostępnych na rynku urządzeń. Dla użytkowników udostępnia repozytoria i możliwość uzyskiwania certyfikatów. Kod gry możemy pisać obecnie w językach C# oraz JavaScript, a także Boo, który jest mniej popularny i posiada mniejsze możliwości. Obsługa silnika jest intuicyjna, a dużą część zasobów możemy pobrać za darmo ze sklepu. Jedna z najliczniejszych społeczności tworzy wiele poradników i pomaga sobie na forach. Silnik dostępny w wersji darmowej (Personal), Plus i Pro - wersje te pozwalają na

10. Logo Unity [20]

osiąganie różnej wielkości zysków z gier bez dodatkowych opłat, a także różne dodatki (np. dostęp do kodu źródłowego silnika, system analityki).

5.2.2 Unreal Engine 4

11. Logo Unreal Engine 4 [18]

Unreal Engine 4 [18] to silnik wykorzystywany zarówno przy produkcji gier, jak i animacji (projekty filmowe), czy też wizualizacji architektonicznych. Silnik jest napisany w języku C++. Aktualna wersja pochodzi z roku 2014. Wykonywane na tym silniku gry są wspierane na wszystkich większych systemach i konsolach: Windows, Linux, iOS, Android, XBOX ONE, PS4. Obsługuje także wirtualną rzeczywistość. Silnik ten wyposażony jest w system blueprintów - graficznego projektowania mechaniki gry lub kodu odpowiadającego za wydarzenia, co wyróżnia go na tle konkurencji. Kod pisany jest w C++. Dla użytkowników udostępniony jest na stronie producenta Market Place [18] - witryna internetowa, w którym możemy kupić, a następnie pobrać zasoby. W przypadku poszukiwania darmowych zasobów można także odwiedzić assetstore dla silnika Unity, należy się jednak liczyć z tym, że znalezione tam pakiety mogą wymagać dostosowania do innego środowiska. Społeczność oraz twórcy tworzą dużą ilość poradników pomagających zarówno początkującym, jak i zaawansowanym użytkownikom. Dokumentacja silnika dostępna jest na stronie producenta. W przypadku komercyjnego wykorzystania silnika, wymagana jest opłata zależna od zysków.

5.2.3 CryEngine 5

12. Logo CryEngine [21]

Silnik CryEngine 5 [21] jest dystrybuowana całkowicie za darmo, chociaż można składać dobrowolne dotacje. Silnik posiada bardzo zaawansowane możliwości graficzne, jednak jest trudniejszy w obsłudze, w związku z tym

korzystają z niego bardziej doświadczeni developerzy. Jest także warty polecenia ze względu na bardzo rozbudowane możliwości edycji audio (Fmod) oraz łatwość w implementacji SI. Gry pisze się w języku C#. Dokumentacja i poradniki dostępne są na stronie producenta. Wspiera platformy XBOX ONE, Windows, PS4, Linux i wirtualna rzeczywistość (urządzenie Oculus).

5.2.4 Game Maker

Silnik Game Maker [22] jest wspierany przez wiele platform, a wykorzystywany do produkcji gier 2D. Jest bardzo łatwy w obsłudze, przez co może być wykorzystywany przez początkujących twórców. Sprzedawany jest w różnych wersjach różniących się m. in. platformami, na które można stworzyć gry. W porównaniu do innych silników jest drogi. Posiada własny język skryptowy GameMaker Language. Tworząc przy użyciu tego silnika nie musimy martwić się o zarządzanie pamięcią. Możliwości silnika możemy rozszerzać przez pobieranie dodatków.

13. Logo Game Maker [22]

W kolejnym rozdziale przechodzimy do opisu procesu produkcji gier komputerowych.

5.3 Proces produkcji

Proces produkcji gry komputerowej dzielimy w zasadzie na 3 części: preprodukcję (przygotowanie koncepcji gry, zasad tworzenia i określenie, czy pomysł przyniesie zyski), produkcję (implementację i testowanie) oraz postprodukcję (utrzymanie, naprawianie błędów (ang. *bug fixing*), tworzenie dodatkowej zawartości).

Poniżej omówimy dokładniej te etapy.

5.3.1 Preprodukcja [26][27][33]

Preprodukcja jest to faza, która nie jest wymagana, jednak jej ominięcie w dużym stopniu utrudni proces twórczy. Małe studia, czy też osoby niezależne często pomijają ten etap produkcji, przeskakując od razu od pomysłu, do jego realizacji.

Rezultatem jest chociażby dopisywanie nowych, nieplanowanych wcześniej funkcjonalności, co może kłócić się z wstępną koncepcją gry, utrudniać utrzymanie kodu co przekłada się na ilość błędów i opóźniać zakończenie prac.

Podczas tego etapu ustalanych jest wiele koncepcji, które mają duży wpływ na cały proces tworzenia gry. Po pierwsze ustalane jest czym gra ma być, a także definiuje się za co odpowiedzialni będą grupy wchodzące w skład zespołu developerskiego. Pisane są dla nich odpowiednie wytyczne.

Należy odpowiedzieć na fundamentalne pytania: Jaka jest grupa odbiorców dla gry? Jak ma wyglądać rozgrywka (ang. *gameplay*)? Co ogranicza produkcję (czas, budżet) i w jakim stopniu? Na jakich zasadach będzie opierała się gra?

Od ustalenia płci i wieku odbiorców będzie zależeć wiele cech, których oczekują oni od produkcji. Do nich będzie też skierowana kampania marketingowa. Dane, na których będziemy się opierać będą podstawą do wyboru stylu i dynamiki rozgrywki. Młodsi odbiorcy będą oczekiwać bogatych efektów graficznych, podczas gdy starsi mogą cieszyć się bardziej stonowanym charakterem gry. Projektując grę pod konkretną grupę odbiorczą możemy w tej grupie liczyć na dużo większy sukces, co przełoży się na renomę naszego studia. Gry tworzone tak, były dostosowane do największej grupy docelowej, zwykle są przeciętne i nie wyróżniają się niczym spośród wielu innych produkcji.

Określenie stylu rozgrywki to rzecz, która nie może zmienić się w trakcie produkcji. Krótkie zdanie, jakim określimy produkcję musi pozostać aktualne aż do końca produkcji. Przykładem może być "buduj i rozwijaj miasto, broniąc się przed najeźdźcami" [77], "strzelaj do zombie szukając bezpiecznego schronienia" gdy mówimy o jednej z gier FPS (np. Left 4 Dead 2 [114]), albo "eksploruj fantastyczny świat odgrywając i rozwijaj swoją postać" (gra RPG - np. Oblivion [84]), kiedy planujemy stworzyć grę RPG. Zdefiniować należy jakie elementy naszej gry będą miały w zamierzeniu przyciągać gracza.

Dobrą praktyką jest określenie kilku filarów, na których będzie opierała się rozgrywka. Dla przykładu w powyżej zaproponowanej grze RPG będą to: otwarty świat fantasy, odgrywanie samodzielnie stworzonej postaci, rozwój oparty o interakcje z NPC i walkę. Jak widać są to zdania proste, jednak na realizację tych założeń pozostaje bardzo dużo swobody.

Dbając o pomyślne ukończenie projektu, należy zadbać o określenie czynników, które będą nas ograniczały w produkcji. Niezbędne są chociażby wstępne ustalenia dotyczące czasu, w który gra ma być stworzona oraz przeznaczonego na nią budżetu. Trzeba też zbadać, na co zespół może sobie pozwolić i jakimi dysponuje zasobami (ludzie i ich wynagrodzenie, oprogramowanie, sprzęt, miejsce, itp.). Wstępne estymaty powinny uwzględniać jak najwięcej czynników, jednak i tak nie wolno zapomnieć o nieprzewidzianych okolicznościach, biorąc na nie odpowiednią poprawkę. Po tych ustaleniach można przejść do dzielenia procesu produkcji na mniejsze części, z którymi będzie musiał sobie radzić zespół. Dzięki ustaleniu kamieni milowych (ang. *milestone*) możliwy jest monitoring przebiegu prac. Podział na części wymaga rozmów z odpowiednimi podzespołami produkcyjnymi i konsultowanie ich możliwości. Ustalenie konkretnych, ostatecznych terminów (ang. *deadline*) jest też motywacją dla pracowników - może to być motywacja pozytywna w przypadku, kiedy termin będzie osiągalnym wyzwaniem, lub negatywna, jeżeli termin będzie wydawał się już na pierwszy rzut oka nieosiągalny.

Kolejnym tematem, który występuje podczas tworzenia gry jest ścieżka przepływu pracy (ang. *pipeline*). Tak jak w większości projektów informatycznych, zadania (ang. *task*) muszą być najpierw dokładnie przeanalizowane, odpowiednio rozdzielone, a każda z osób związanych z tymi zadaniami musi wiedzieć, jaki powinien być dla danego taska kolejny krok. Przykładowa, krótka ścieżka dla zadania to: 1. zdefiniowanie > 2. dokumentacja > 3. zatwierdzenie > 4. implementacja > 5. QA (zapewnienie jakości, ang. *Quality Assurance*) > 6. wdrożenie do projektu. Dzięki zdefiniowaniu odpowiedniego przepływu pracy zadania nie będą "wisiały w powietrzu", tylko zawsze ktoś będzie za nie odpowiedzialny (zobacz: rola kierownika zespołu w rozdziale 5.1.1 Zespół deweloperski). Istnieją narzędzia pomagające w wprowadzeniu takiego przepływu prac, jak Jira [115] (oprogramowanie do śledzenia postępu projektów), czy chociażby prosta tablica kanban [116] (zadania przypisywane do statusów, w których się znajdują na jednej, przejrzystej tabeli), taka jak używane podczas implementacji gry związanej z tą pracą dyplomową tablica Trello (zobacz: 6.3 Trello).

Ostatnim pojęciem jest prototypowanie. Dzięki eksperymentowaniu z różnymi algorytmami i testowaniu ich w praktyce dokładniej przeprowadzimy analizę,

w związku z czym zminimalizujemy ilość problemów wynikłych z jej błędów. Ten krok nie musi nawet wymagać implementacji omawianych pomysłów, a może być wykonany chociażby na papierze. Prototypowanie nie jest zarezerwowane dla preprodukcji, ale także może być wykorzystywane w czasie produkcji.

Kolejną po preprodukcji fazą jest produkcja właściwa, opisana w kolejnym rozdziale.

5.3.2 Produkcja właściwa

Produkcja to część cyklu, w której tworzona jest treść gry (dźwięk, grafika, dialogi, przerywniki filmowe, itp.) i pisany jest kod źródłowy [26]. Podczas tego okresu nad grą pracuje największa ilość osób - tutaj wykazuje się każda z osób z zespołu deweloperskiego (zobacz: 5.1.1 Zespół deweloperski). Osoby te ściśle współpracują ze sobą, razem obmyślając drogi implementacji, dostarczając nowe zasoby, tworząc narzędzia i konsultując stan prac i napotykane problemy.

W trakcie produkcji tworzone są kolejne wersje oprogramowania [40]. Stworzenie kolejnej wersji czasami nazywa się osiągnięciem kamienia milowego. Na początku tworzy się wersje robocze (pre-alfa), do której dostęp mają osoby pracujące na tej wersji. Jest to wersja dostępna na repozytorium i zwykle dostępna przez wykorzystanie systemów kontroli wersji (jak np. Git, zobacz: 6.2 Unity Collaborate). Następnie gra przygotowywana jest dla szerszego grona odbiorców (wersja alfa), w której z programu w zasadzie można już w ograniczonym zakresie korzystać. W wersji alfa główna zawartość gry (większość mechanizmów rozgrywki) powinna być już ukończona - za podstawę do większości zaimplementowanych w tej wersji elementów jest krok prototypowania w fazie preprodukcji. Kolejną wersją jest beta (tutaj zaczynają pracę tzw. beta testerzy), a następnie, po poprawie zgłoszonych błędów tworzona jest jest wersja RC (ang. *Release Candidate* - kandydat do wydania). Po wydaniu wersji beta nie tworzy się już nowej zawartości, a tylko poprawia zgłaszane błędy. Ostatnia wspomniana wersja staje się wersją stabilną, który wysyła się do tłoczni (w przypadku sprzedaży fizycznych kopii) i wypuszcza na rynek.

Kolejnym etapem w cyklu życia gry jest postprodukcja.

5.3.3 Postprodukcja

Testowanie gry podczas produkcji nie wykryje wszystkich błędów [26]. Nawet podczas współpracy z beta testerami (większa ilość użytkowników, czasami np. ludzie, którzy kupili grę przed premierą) nie wszystkie błędy są wyłapywane. Aby gra była lepiej oceniana w środowisku graczy należy naprawiać zgłaszane przez graczy błędy. Po otrzymaniu i poprawieniu zgłoszeń (zależy od czasu zbierania raportów błędów, ilości zgłoszeń, itd.), tworzone są poprawki i wydawana jest łatka. O ile w przeszłości taka praktyka mogła nie mieć miejsca (prostsze gry, trudność z dostępem do łatek (ang. *patch*) ze względu na ograniczony dostęp do internetu), to w dzisiejszych czasach jest to rzecz obowiązkowa. Już od roku 2013 większość gier jest sprzedawanych jako kopie cyfrowe [41] - udostępnianie w internecie łatek nie jest już problemem.

Podczas opieki nad programem, badając zainteresowanie rynku danym tytułem i oceniając jego potencjał można dojść do wniosku, że produkt sprzedaje się na tyle dobrze, że można rozważyć wydanie dodatku do gry (zawartość do pobrania, ang. *downloadable content* (*DLC*)). Dodatek taki może być uznawany za osobny produkt, w związku z czym cykl produkcji rozpoczyna się od początku. W przypadku, gdy dodatek to po prostu kosmetyczne zmiany (co jest zjawiskiem coraz częstszym), nie trwa on oczywiście tak długo i często jest powtarzanym schematem (dużo podobnych, prostych, dodatków do gry).

5.4 Dopasowywanie trudności gry

Osobny akapit należy poświęcić dopasowaniu trudności rozgrywki, jako element bezpośrednio związany ze sztuczną inteligencją. Podczas tworzenia gier, przed procesem testowania trudno jest określić parametry, które definiują trudność gry [28]. Dopasowanie trudności rozgrywki jest istotne nie tylko dlatego, że gracz musi sprostać jakiemuś wyzwaniu, lecz również z powodu stałych oczekiwań kolejnych wyzwań. Producenci muszą stale zabiegać o uwagę gracza.

Gdy gracz zaczyna zabawę, zwykle zaopatruje się go w krótką instrukcję (ang. *tutorial*), która pomaga mu przyswoić sobie podstawowe zasady poruszania się po świecie gry. Niektóre gry mają na tyle intuicyjne sterowanie, że taki krok nie jest

wymagany. Po zapoznaniu gracza z podstawową wiedzą na temat tego, w jaki sposób może on ją eksplorować stopniowo, w trakcie całej rozgrywki powinno się prezentować mu kolejne, nowe mechanizmy, które przyciągną jego uwagę. Należy utrzymywać stan gracza pomiędzy zagubieniem i frustracją, a zwykłą nudą [28] - w przypadku zbyt dużego poziomu trudności gra nie sprawia radości ze względu na częste porażki, a z drugiej strony zbyt łatwa gra nie zapewnia dającego satysfakcji wyzwania. Przykładem jest przedstawianie graczowi nowych typów przeciwników (z bogatszą SI: unikający naszych ataków, zaskakujący nas obraną taktyką czy umiejętnościami; z innymi cechami: odporni na część z ataków, z większą ilością punktów życia) i wyposażanie go w nowe umiejętności radzenia sobie z nimi (narzędzia dające nam większą swobodę poruszania, bardziej zaawansowane typy broni).

%%%

Ten rozdział kończy teoretyczny przegląd sztucznej inteligencji, rynku gier komputerowych i metodyki ich produkcji. Kolejne rozdziały opisują proces tworzenia gry wykorzystującej opisane powyżej elementy oraz wykorzystane przy produkcji narzędzia.

6. Narzędzia wykorzystane do implementacji projektu gry

W kolejnych podrozdziałach opisane zostaną narzędzia wykorzystane do implementacji gry.

6.1 Unity (wersja 5.5.2f1)

Wybranym do zrealizowania gry silnikiem jest silnik Unity (zobacz: 5.2.1 Unity 5). Przy wyborze kierowałem się popularnością, dostępnością wysokiej jakości poradników oraz darmowych zasobów, a także personalnymi preferencjami i wcześniejszym, podstawowym poznaniem programu. Jako, że oprogramowanie to nie jest dostępne na systemie Linux, to wykorzystałem także system operacyjny Windows 10.

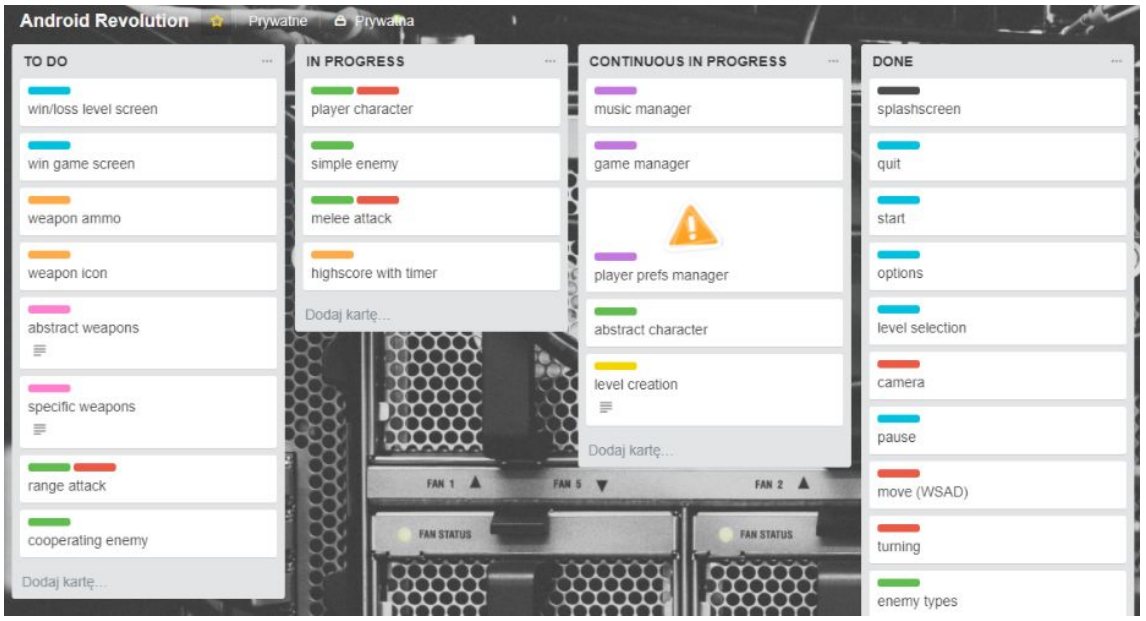
6.2 Unity Collaborate

Unity Collaborate, tak jak popularny Git czy Subversion, jest systemem kontroli wersji (ang. *version/revision control system*) [25]. Pozwala na zapisywanie stanu projektu w dowolnym momencie, dzięki czemu możemy zawsze wrócić właśnie do tego konkretnego miejsca. W przypadku pracy wieloosobowego zespołu każdy członek może pracować nad swoim zadaniem, a po skończeniu udostępnić go reszcie, łącząc swoje zmiany ze zmianami wprowadzonymi przez innych (ang. *merge*). Istnieje wiele nakładek i dodatków do programów, które ułatwiają korzystanie z takich systemów, a także wiele miejsc ułatwiających zarządzanie repozytoriami oraz udostępniające przestrzeń dyskową - przykłady to GitHub, BitBucket, czy GitLab.

Unity Collaborate jest narzędziem stworzony specjalnie na potrzeby silnika Unity. W tej pracy wykorzystywany jest nie jako narzędzie do pracy z zespołem (samodzielna implementacja), lecz po prostu do tworzenia kopii zapasowej (ang. *back-up*) przechowywanej w chmurze Unity.

6.3 Trello

Trello [24] jest narzędziem pomagającym rozdzielać zadania i uporządkować pracę. Głównym elementem jest tablica - składa się z list wypełnionych kartami używanymi przez zespół. Po zdefiniowaniu każdego zadania możemy zająć się nim



14. Tablica Trello wykorzystywana przy implementacji gry będącej przedmiotem pracy dyplomowej [opracowanie własne]

członek zespołu, zaczynając nad nim pracę. Zadaniom można dodawać etykiety, przez co można wykorzystać Trello do stworzenia tablicy podobnej do tych, wykorzystywanych przy korzystaniu z metody programowania zwinnego (ang. *agile software development*). Oprócz tego Trello wykorzystywane jest także w codziennym życiu, jak chociażby do stworzenia listy rzeczy do zrobienia w domu, czy w pracy biurowej.

6.4 Paint oraz Gimp

Paint [120] jest to prosty edytor graficzny zainstalowany domyślnie na systemach Windows. Przeznaczony jest do obróbki grafiki rastrowej. W przeciwieństwie do drugiego używanego programu, bardziej zaawansowanego Gimpa [117] nie obsługuje warstw. Wstępnie stworzone grafiki w programie Paint mogą być dalej obrabiane bardziej profesjonalnym narzędziem Gimp (np. usunięcie tła).

6.5 Microsoft Visual Studio 2015 Community Edition

Microsoft Visual Studio [37] to bezpłatne, w pełni wyposażone środowisko zintegrowane środowisko programistyczne (ang. *integrated development environment* (*IDE*)) dla uczniów i studentów, deweloperów oprogramowania typu open-source i dla indywidualnych deweloperów. Silnik Unity domyślnie wyposażony jest w IDE MonoDevelop, jednak MS Visual Studio jest narzędziem bardziej zaawansowanym i dzięki odpowiedniej konfiguracji, dużo wygodniejszym w użytku. Cały kod będący częścią pracy dyplomowej jest pisany w tym środowisku.

6.6 C# (C Sharp)

C# to obiektowy język programowania [38]. Po skompilowaniu kod zamieniany jest na język pośredni, wykorzystywany przez środowisko .NET Framework [118]. C# został stworzony opierając się gramatykę języków C++ oraz Java. C# to jeden z 3 języków wspieranych przez Unity (obok UnityScript oraz Boo).

Zdecydowałem się na wykorzystanie C# ze względu na podobną do języka Java (z którym mam większe doświadczenie) składnię oraz powszechność wykorzystywania tego języka w poradnikach. Jak podaje producent silnika Unity, ponad 80% skryptów wykorzystywanych w pracy z Unity jest napisane w tym języku [39].

7. Tworzenie gry

Postanowiłem zaprojektować grę, w której dostrzec będzie można użyteczność silnika Unity w kwestii sztucznej inteligencji (biblioteka UnityEngine.AI). Za inspirację

15. Zrzut ekranu z gry Hotline Miami [119]

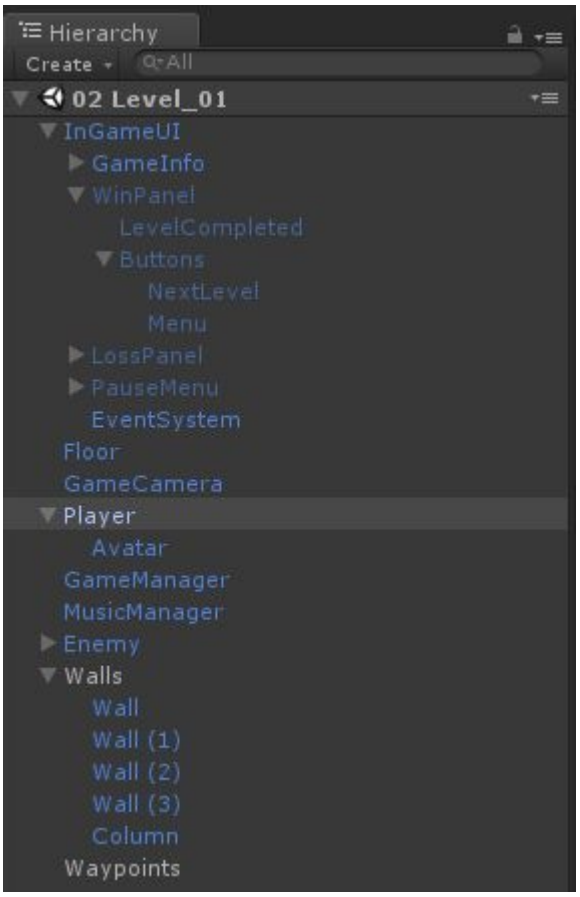
można uznać bardzo pozytywnie przyjętą przez środowisko grę Hotline Miami [119], w której widzimy z góry (grafika 2D) kierowaną przez nas postać, spotykającą na swojej drodze przeciwników, z którymi musi się szybko uporać. Do stworzenia Hotline Miami został użyty silnik Game Maker (zobacz: 5.2.4 Game Maker), a gra została stworzona przez zaledwie 2 osoby (studio Devolver Digital). Sztuczna inteligencja w grze sprawia, że przeciwnicy potrafią posługiwać się bronią białą i palną, oraz wyszukują źródło hałasu, które związane są z akcjami gracza (np. eliminacja jednego z przeciwników). Do zadań gracza zwykle należy zdobywanie broni i wykorzystanie jej, oraz rozkładu pomieszczeń, do pokonania przeciwników.

Swoją grę nazwałem "Android Revolution". Chociaż nie jest ona tak rozbudowana jak Hotline Miami, to prezentuję wbudowany w silnik Unity algorytm wyszukiwania ścieżek oraz implementację prostych zachowań przeciwników wraz z ich współpracą w celu znalezienia i pokonania gracza. Kod starałem się pisać w taki sposób, aby w przyszłości dodać kolejne cechy - ma być prosty w rozbudowie (nowe typy broni i przeciwników) i zachowywać jak najwyższy poziom abstrakcji.

7.1 Korzystanie z silnika Unity

Unity opiera się na koncepcji tworzenia obiektów (GameObject) różnego typu (obiekty 2D, 3D, źródła światła, kamera, źródła dźwięku, elementu interfejsu

użytkownika, itp.) i przypisywania do nich komponentów, których właściwości można dostosować do indywidualnych potrzeb twórcy. Komponentami przypisywanymi do obiektów mogą być już stworzone na potrzeby silnika elementy (komponent dzięki któremu możemy obsługiwać fizykę, kolizje, definiujący teksturę, pomagający w poruszaniu się i nawigacji), albo skrypty napisane przez dewelopera. Tworzenie pustych obiektów (posiada tylko dane nt. jego umiejscowienia) mogą służyć jako element porządkowy (budujący hierarchię obiektów - zobacz: rysunek 16).



16. Zakładka hierarchii obiektów w środowisku Unity

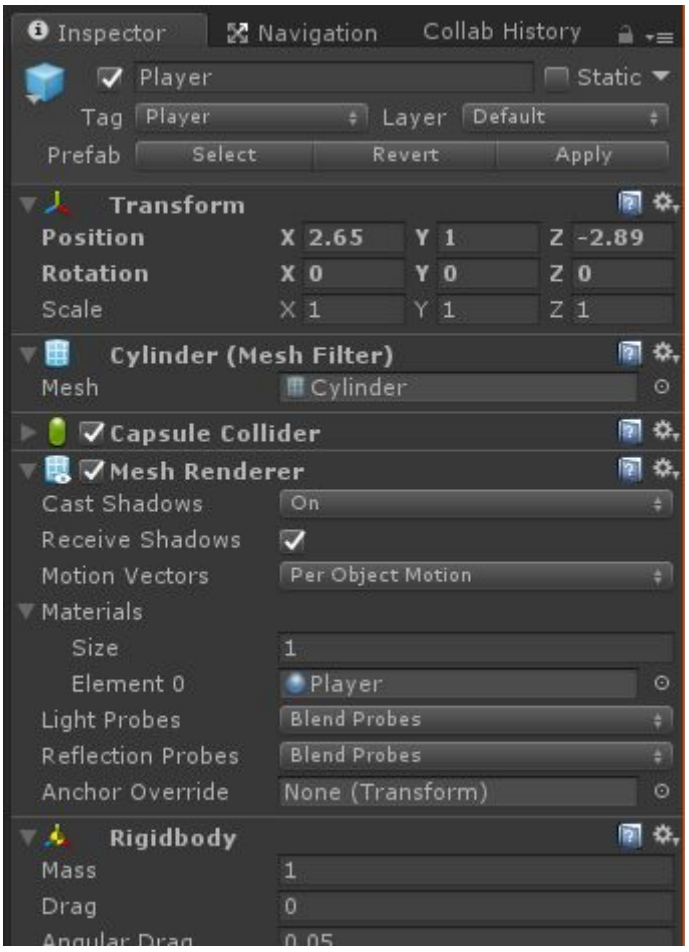
Obiekty można zmodyfikować, zmieniając ich parametry, za pomocą zakładki *Inspector* (zobacz: rysunek 17). W tym menu możemy też dodać nowe obiekty za pomocą przycisku *Add Component*. Umiejscowienie obiektów w scenie, ich rotację a także skalowanie możemy dodatkowo zmienić za pomocą specjalnych, ułatwiających te czynności narzędzi. Skrypty, które są przypisywane do obiektów także mogą umożliwiać parametryzowanie w prosty sposób poprzez przedstawienie zmiennych jako publiczne. Dzięki czemu w menu *Inspector* możemy ustawić daną zmienną, przypisując jej wartość lub istniejący element (obiekt ze sceny, prefabrykat, itd.).

Wszystkie dodane obiekty istnieją w tzw. scenie (zobacz: rysunek 22). Każda kolejna scena może mieć inną funkcję. W implementowanej grze są to np. ekran powitalny, menu (osobna scena dla menu głównego, opcji, itp.) oraz kolejne poziomy.

Każdy zasób (dotyczy także scen), którego używamy w grze, a który nie jest domyślnie zdefiniowany w Unity musi być dodany do folderu z zasobami (ang. *assets*, zobacz: rysunek 18). Możemy w tym folderze tworzyć podfoldery, aby uporządkować

wykorzystane elementy - do osobnego folderu wrzucone mogą być obrazy, do innego skrypty, do innego muzyka itd.

Obiekty już stworzone w scenie i odpowiednio sparametryzowane można zapisać w tym samym folderze, aby potem można było z nich łatwo korzystać. Taki obiekt nazywa się prefabrykatem (ang. *prefab*) i jest oznaczony na liście obiektów umieszczonych w scenie kolorem niebieskim (zobacz: rysunek 16). Jeżeli zmienimy prefabrykat, to zmiany zostaną wprowadzone także w każdej instancji obiektu, który bierze go za podstawę. Instancję prefabrykatu też możemy zmieniać, jednak nie ma to wpływu na inne kopie (chyba, że zaaplikujemy zmiany w instancji przyciskiem *Accept*). W tworzonej grze jako prefabrykaty zdefiniowałem np. kamerę (wykorzystywana będzie w każdym poziomie), przeciwnika, gracza, cały interfejs graficzny używany w poziomie, GameManagera i MusicManagera (puste obiekty z przypisanym skryptem, zobacz: 7.2.1 GameManager, 7.2.2 MusicManager), ściany, itd. Prefabrykaty umieściłem w osobnym folderze i podzieliłem na kategorie. Przykładem prefabrykatu, którego właściwości są zmieniane może być przeciwnik (każdy z inną listą punktów patrolowych), albo elementy ścian i kolumn (różne wymiary i umiejscowienie).

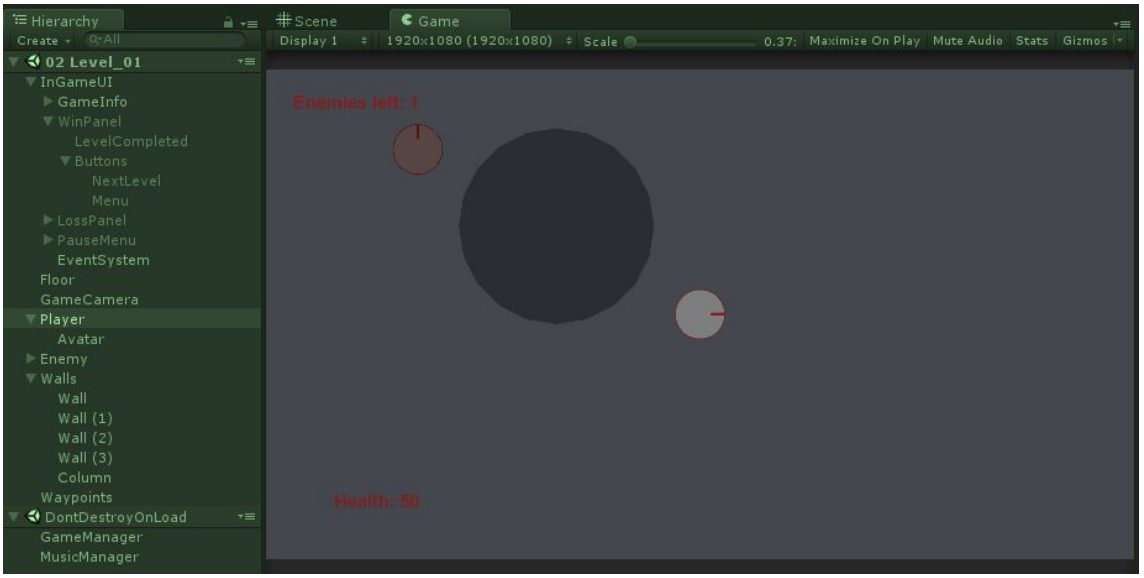


17. Zakładka *Inspector* z podglądem obiektu *Player*

Po stworzeniu sceny i umiejscowieniu w niej obiektów trzeba dodać ją do ustawień budowania projektu (ang. *build settings*, zobacz: 7.8 Budowanie gry), po czym można przetestować działanie za pomocą odpowiedniego przycisku. W zakładce *Game*

18. Zawartość folderu *Assets*

pojawia się okno gry (rysunek 19), w której widzimy efekty naszej pracy, na które patrzymy z perspektywy ustawionej w scenie kamery. Dzięki takiemu rozwiązaniu nie musimy budować całej gry za każdym razem, gdy chcemy zobaczyć działanie naszych zmian.



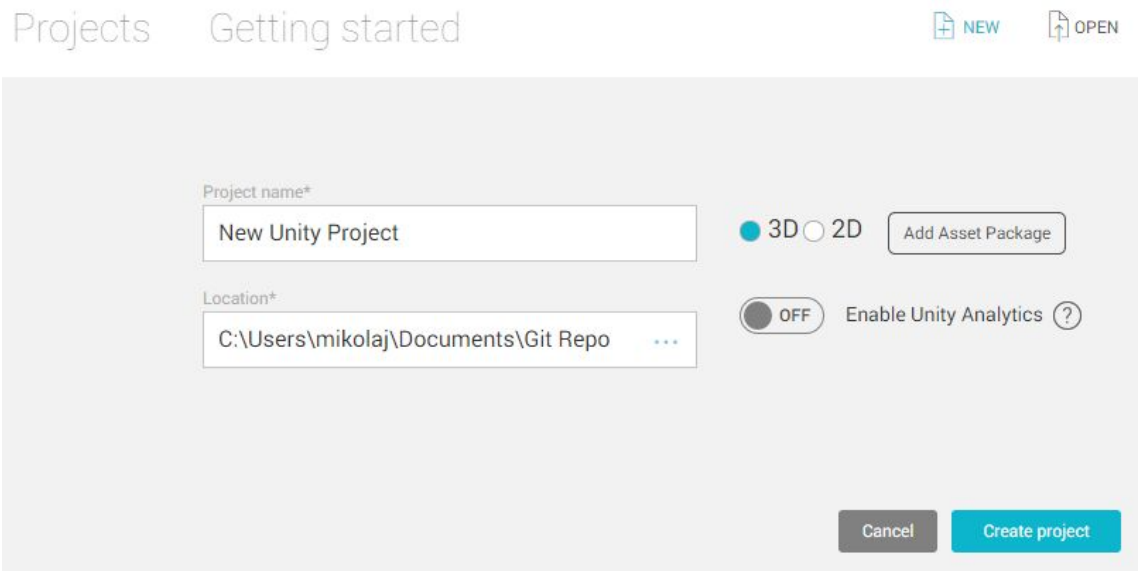
19. Uruchomiona w środowisku Unity (zakładka *Game*) gra. Po lewej stronie widać hierarchię obiektów

Aby nie stracić w przypadku awarii dysku wprowadzanych zmian w grze, wykorzystuje się Unity Collaborate, które posiada osobny przycisk *Collab*. Używając go dostajemy dostęp do menu, gdzie akceptujemy zmiany i wysyłamy je na zewnętrzny serwer z odpowiednim komentarzem. Przez to menu możemy także zaprosić do

współpracy innych deweloperów. Historię zmian możemy podglądnąć w zakładce *Collab History*.

7.2 Tworzenie nowego projektu

Stworzenie nowego projektu rozpoczynamy od uruchomienia silnika i kliknięciu przycisku *New*, a następnie nazwania projektu, oraz wyboru opcji 2D albo 3D (rysunek 20). Są to rzeczy, które można zmienić w czasie tworzenia gry.



20. Okno tworzenia projektu

Po założeniu projektu tworzymy nowe obiekty, które będą używane w większości scen. Część z nich będzie inicjalizowana już zaraz po otwarciu aplikacji i będą przechodziły ze sceny do sceny, zachowując swój stan. Nie ma potrzeby inicjalizowania ich w każdej scenia osobno, jako, że będą wykorzystywane w niemal każdej scenie. Należy w związku z tym wykorzystać wzorzec projektowy singleton [43]. Wzorca tego używamy na samym początku cyklu życia obiektu - podczas wywołania metody *Awake()* (zobacz: listing 1). Jak widać na rysunku 19, część z obiektów jest oznaczona jako przechodzące ze sceny do sceny (*DontDestroyOnLoad*).

```
static GameManager instance = null;

private void Awake() {
     InstanceSetup();

     [...]
```

```
}

private void InstanceSetup() {
      if (instance != null) {
            Destroy(gameObject);
      } else {
            instance = this;
            DontDestroyOnLoad(gameObject);
      }
}
```

Listing 1. Implementacja wzorca projektowego “singleton” (tutaj: w klasie GameManager). Metoda *Object.DontDestroyOnLoad(Object target)* sprawia, że obiekt nie jest niszczony przy wczytywaniu kolejnej sceny.

7.2.1 GameManager

GameManager to odseparowany obiekt, dzięki któremu będziemy mogli w prosty sposób poruszać się między scenami i zarządzać grą w przypadku, gdy nie stworzymy nowego skryptu do obsługi danej funkcji gry. Dobrą praktyką jest tworzenie osobnego skryptu do zarządzania każdą unikalną funkcją. Do GameManagera będą odwoływać się chociażby przyciski w menu.

Chociaż poruszać się między scenami można odwołując się do ich nazw, to dzięki silnikowi Unity, oraz odpowiedniemu ustawieniu scen przy budowaniu projektu będziemy w stanie przechodzić po prostu do kolejnych poziomów w grze za pomocą bezargumentowej metody.

```
public void LoadLevel(string sceneName) {
      SceneManager.LoadScene(sceneName);
}

public void LoadNextLevel() {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildInd
ex + 1);
}

public void Quit() {
      Application.Quit();
}
```

Listing 2. 3 podstawowe funkcje nawigacyjne zaimplementowane w klasie GameManager. Pozwalają one: wczytać sceną o podanej nazwie, wczytać scenę będącą kolejną w kolejce budowania, zakończyć działanie aplikacji.

7.2.2 MusicManager

MusicManager zostanie zainicjalizowany już w pierwszej scenie i będzie odpowiadał za odtwarzanie plików z muzyką tła. Za uruchamianie odtwarzania innych plików dźwiękowych w grze odpowiadać będą konkretne obiekty, których ten dźwięk będzie dotyczył.

```
public AudioClip[] levelMusicArray;
private AudioSource currentPlaying;

void Awake() {
      [...]

      currentPlaying = GetComponent<AudioSource>();
      currentPlaying.enabled = true;
      currentPlaying.volume =
PlayerPrefsManager.GetMasterVolume();
}

void OnSceneLoaded(Scene scene, LoadSceneMode mode) {
      int level = scene.buildIndex;
      if (levelMusicArray.Length > level) {
            AudioClip newAudioClip = levelMusicArray[level];

            [...]

            if (newAudioClip && currentPlaying.clip !=
newAudioClip) {
                  currentPlaying.clip = newAudioClip;
                  currentPlaying.loop = true;

                  [...]

                  currentPlaying.Play();
            }
      }

      [...]
}
```

Listing 3. Część klasy MusicManager. Publiczna tablica *levelMusicArray* pozwala nam na przypisanie do kolejnych scen różnych utworów muzycznych (jako muzyka tła) bez ingerencji w kod skryptu. W funkcji *Awake()* ustawiamy referencję do obiektu AudioSource pozwalającego na odtwarzanie muzyki oraz ustawiamy głośność odwołując się do klasy PlayerPrefsManager (zobacz: 7.2.3 PlayerPrefsManager, 7.3.2 Menu Opcji). Metoda *OnSceneLoaded(Scene scene, LoadSceneMode mode)* odpowiada za wczytanie odpowiedniego pliku z muzyką oraz odtworzenie go. Jeżeli do kolejno wczytanej sceny przypiszemy taki sam plik, to nie zostanie on przerwany, ale będzie dalej odtwarzany.

7.2.3 PlayerPrefsManager

Za pomocą zapewnionej przez Unity klasy *PlayerPrefs* możemy zapisywać proste dane (liczbę całkowitą, zmiennoprzecinkową oraz tekst) do pliku. Ten plik wykorzystywany jest m.in. do zapisywania ustawień użytkownika (ustawienia grafiki, audio, rozgrywki, sterowania itp.). Nie wykorzystuje się go do innych celów, jako, że dostęp do tych danych jest relatywnie wolny. Zapisane dane utrzymują swoją wartość po wyłączeniu gry i są dostępne po jej ponownym włączeniu. Napisanie klasy *PlayerPrefsManager* ma ułatwić korzystanie z *PlayerPrefs*. W *PlayerPrefsManager* umieściłem metody, dzięki którym mogę korzystając z innych skryptów łatwo zmienić ustawienia gry. *PlayerPrefsManger* to klasa statyczna, dzięki czemu nie muszę tworzyć instancji tej klasy przez odwołaniem się do jednej z jej metod.

Do zapisywanych wartości odwołujemy się poprzez zdefiniowany samodzielnie klucz. Zapisane dane to pary klucza i wartości (słownik).

```
public const string DIFFICULTY_KEY = "difficulty";
public const string MUSIC_VOLUME_KEY = "music_volume";
[...]
const string LEVEL_KEY = "level_unlocked_";
const string LAST_UNLOCKED_LEVEL = "last_level_unlocked";

public const float DIFFICULTY_DEFAULT = 2f;
public const float MUSIC_VOLUME_DEFAULT = 0.8f;
[...]

public static void SetDifficulty(float difficulty) {
     if (difficulty >= 1f && difficulty <= 3f) {
          PlayerPrefs.SetFloat(DIFFICULTY_KEY, difficulty);
     } else {
          Debug.LogError("Difficulty value out of range");
     }
}

public static float GetDifficulty() {
     return PlayerPrefs.GetFloat(DIFFICULTY_KEY);
}

public static void SetMusicVolume(float volume) {
     if (volume >= 0f && volume <= 1f) {
          PlayerPrefs.SetFloat(MUSIC_VOLUME_KEY, volume);
     } else {
          Debug.LogError("Music volume value out of range");
     }
```

```
}

public static float GetMusicVolume() {
     return PlayerPrefs.GetFloat(MUSIC_VOLUME_KEY);
}

public static void UnlockLevel(int level) {
     if (level >= 0 && level <=
SceneManager.sceneCountInBuildSettings - 1) {
          SetPlayerPrefsInt(LEVEL_KEY + level.ToString(),
1);
          if (level != 1) {
               SetPlayerPrefsInt(LAST_UNLOCKED_LEVEL,
level);
          }
     }
}

[...]

public static bool IsLevelUnlocked(int level) {
     int? levelUnlocked = PlayerPrefs.GetInt(LEVEL_KEY +
level.ToString());
     return levelUnlocked != null && levelUnlocked == 1;
}

[...]
```

Listing 4. Na początku klasy zdefiniowane są klucze, którymi będziemy odwoływali się do zapisanych wartości. Poniżej można zobaczyć kod odpowiedzialny za ustawianie i czytanie konkretnych danych (poziom trudności gry, głośność muzyki). Na samym dole zaprezentowany jest kod odpowiedzialny za odblokowywanie kolejnych poziomów graczowi oraz za sprawdzenie, czy dany poziom jest dla niego dostępny. Wykorzystanie obiektu *Debug* pozwala na wypisywanie informacji do konsoli w trakcie działania gry.

7.3 Ekran powitalny

Bezpłatna wersja Unity wymusza na aplikacji, żeby zostało wyświetlone logo silnika przy uruchamianiu programu. Żeby dać znać użytkownikowi kto stworzył grę (jakie studio) można dodać kolejną scenę, odpowiedzialną za powitanie gracza (ekran powitalny, ang. *splash screen*). Taka scena posłuży do budowy marki - gracz będzie kojarzył logo (oraz ewentualnie krótki dźwięk) z producentem. Scena będzie widoczna kilka sekund. Długością wyświetlania sterujemy za pomocą zmiennej zdefiniowanej w grze (nadanie jej odpowiedniej wartości przy tworzeniu) i załączonym do obiektu obsługującego ekran powitalny plikiem dźwiękowym o określonej długości.

W tej scenie potrzebujemy tła (nowy obiekt UI>Image), teksu z nazwą studia (UI>Text) oraz obiektu (Create Empty) przechowującego nasz skrypt *SplashScreenController*.

```
public GameManager gameManager;

public float splashScreenMinSecondsLength = 2.5f;

private void Start() {
      SplashScreenSoundHandler();
}

private void SplashScreenSoundHandler() {
      [...]

      AudioSource slashScreenAudioSource =
GetComponent<AudioSource>();

      [...]

      float splashScreenDuration = slashScreenAudioSource.clip
!= null ? Mathf.Max(slashScreenAudioSource.clip.length,
splashScreenMinSecondsLength) : splashScreenMinSecondsLength;
      slashScreenAudioSource.Play();
      gameManager.Invoke("LoadNextLevel",
splashScreenDuration);

      [...]
}
```

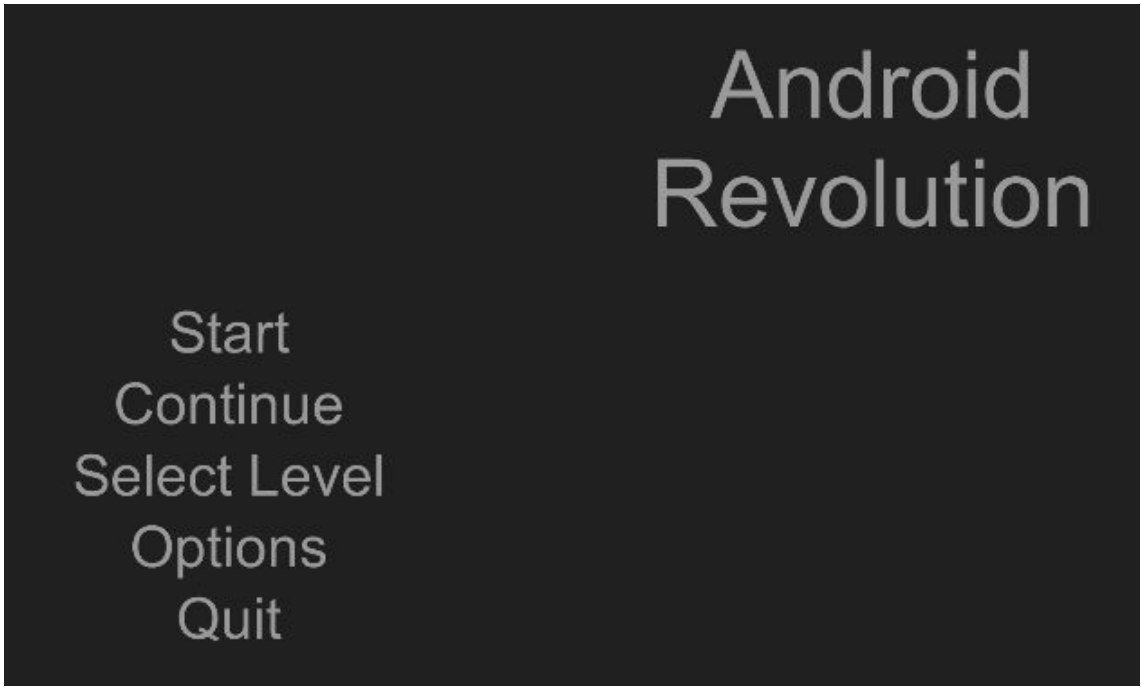
Listing 5. Część klasy SplashScreen odpowiadająca za odgrywanie dźwięku i automatyczne wczytanie kolejnej sceny. Zmienna *splashScreenDuration* określa jak długo (w sekundach) ma być wyświetlana scena. Bierze ona większą wartość spośród ustalonej przez twórcę wartości (publiczny dostęp do zmiennej pozwala na jej edycję bez zmian w kodzie) i długości pliku dźwiękowego (jeżeli ten został zdefiniowany). Metoda *Invoke(string methodName, float time)* wywołana na obiekcie gameManager wywołuje w nim podaną funkcję *LoadNextLevel()* odpowiadającą za załadowanie kolejnej sceny z ustalonym wcześniej opóźnieniem.

7.4 Menu

Chociaż wspomniane wyżej obiekty GameManager i MusicManager muszą być umieszczone tylko w pierwszej scenie (jako, że nie są niszczone przy przechodzeniu między scenami), to będziemy je dodawać do każdej sceny, aby ułatwić sobie testowanie. Dzięki takiemu zabiegowi nie musimy za każdym razem zaczynać testowania od ekranu powitalnego, który te obiekty ładuje.

7.4.1 Menu główne

Menu główne to miejsce, w którym będziemy mogli zacząć nową grę (lub wybrać konkretny, odblokowany wcześniej poziom), przejść do opcji gry i zakończyć ją naciskając na odpowiednie przyciski. Oprócz przycisków umieścimy tam oczywiście tło (UI>Image) oraz tytuł gry (obiekt UI>Text). Przyciski (ang. *button*) to elementy tekstowe (UI>Text), do których dodajemy komponent *Button* (Add Component>Button), dzięki któremu definiujemy zachowanie przycisku (kolor po najechaniu i naciśnięciu, wykonywana akcja). Akcje wykonywane przez przyciski są zdefiniowane w skrypcie *GameManager*. Przyciski zostały przypisane w hierarchii osobnemu, pustemu obiektowi o nazwie *Buttons*.



21. Ekran menu głównego

7.4.2 Menu opcji

Menu opcji jest bardzo podobne w tworzeniu, jak menu główne. Kolejnym wykorzystanym tutaj elementem są suwaki (UI>Slider), dzięki którym gracz może ustawić trudność gry i poziomy głośności. Zachowanie *sliderów* ustawia się podobnie jak zachowanie przycisków. Aby określić zachowanie przycisków stworzony został obiekt *OptionsController* wspomagający się klasą *PlayerPrefsManager*.

```
public void SetDefaults() {
     difficultySlider.value =
PlayerPrefsManager.DIFFICULTY_DEFAULT;
     musicVolumeSlider.value =
PlayerPrefsManager.MUSIC_VOLUME_DEFAULT;
     soundEffectsVolumeSlider.value =
PlayerPrefsManager.SOUND_EFFECTS_VOLUME_DEFAULT;
     UpdatePlayerPrefs();
}

private void UpdatePlayerPrefs() {

PlayerPrefsManager.SetDifficulty(difficultySlider.value);

PlayerPrefsManager.SetMusicVolume(musicVolumeSlider.value);

PlayerPrefsManager.SetSoundEffectsVolume(musicVolumeSlider.va
lue);
}

public void SetDifficulty() {

PlayerPrefsManager.SetDifficulty(difficultySlider.value);
}

public void SetMusicVolume() {
     musicManager.GetComponent<AudioSource>().volume =
musicVolumeSlider.value;

PlayerPrefsManager.SetMusicVolume(musicVolumeSlider.value);
}

public void SetSoundEffectsVolume() {

PlayerPrefsManager.SetSoundEffectsVolume(soundEffectsVolumeSl
ider.value);
}
```

Listing 6. *OptionsController* zawiera metody ustawiające początkowe wartości *sliderów*, zmieniającą wpisy w *PlayerPrefs* przy manipulowaniu *sliderami* i dającą możliwość przywrócenia opcji do wartości domyślnych

7.4.3 Menu wyboru poziomu

Poziomy (ang. *level*) gracz będzie mógł wybierać, po ich odblokowaniu. Jak w większości gier z taką opcją, poziomy będzie trzeba najpierw odblokować. Kolejne poziomy będą możliwe do odblokowania, po pomyślnym ukończeniu poprzedniego. Pierwszy poziom jest domyślnie odblokowany za pomocą obiektu *GameManager*.

Przyciski w menu wyboru poziomu tworzone są dynamicznie w kodzie - jedynym parametrem jaki musi zostać podany przez dewelopera jest ich ilość. Odpowiedzialnym za budowę przycisków wyboru poziomów jest pusty obiekt *LevelSelection*, do którego przypisałem komponent skryptu o nazwie *LevelSelectionController*.

```
void Start() {
      [...]

      for (int levelNumber = 1; levelNumber <=
levelTotalCount; levelNumber++) {
            Button levelButton =
Instantiate(LevelButtonPrefab);
            SetLevelButtonName(levelButton, levelNumber);
            SetLevelButtonBehaviour(levelButton, levelNumber);
            SetLevelButtonPosition(levelButton,
levelTilesParent, tilesPlacedInRow, columnCounter);

            [...]
      }
}
```

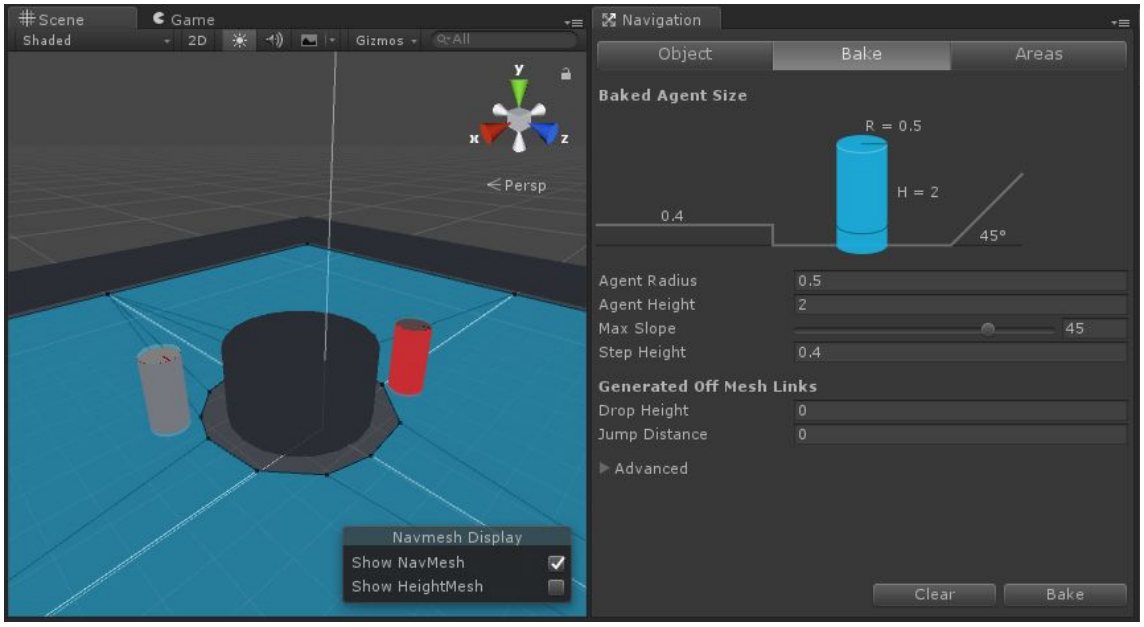
Listing 7. W metodzie *Start()* tworzone są przyciski wyboru poziomu. Na początku tworzony jest nowy obiekt z prefabrykatu, następnie ustawia się jego nazwa i przypisywany jest do odpowiedniego rodzica w hierarchii, dodawana jest funkcja odpowiadająca za zachowanie po naciśnięciu oraz na sam koniec ustawiany jest on w odpowiednim miejscu w scenie.

7.5 Tworzenie świata gry

Jak napisano powyżej (rozdział 7.4.3 Menu wyboru poziomu), dajemy graczowi możliwość wyboru poziomu. Poziomy to rdzeń naszej gry. Są to sceny w których w końcu przed graczem stawiamy jakieś wyzwanie. Tutaj gracz zmierzy się z przeciwnikami. Każdy kolejny poziom będzie różnił się ilością przeciwników, ich rozkładem lub typem. Urozmaiceniem będzie także obszar gry, w którym będziemy się poruszać - w tym przypadku rozkład pokojów, ścian, kolumn czy innych przeszkód, które będą odgradzać nas i przeciwników.

Unity wspiera wyszukiwanie ścieżki tylko w świecie 3D, dlatego postanowiłem stworzyć świat 3D i ustawić kamerę, która będzie go przedstawiała jako świat 2D (rzut z góry, zobacz: rysunek 19 - zakładka *Game*). Poziom tworzymy poprzez ustawienie terenu, po którym będzie się poruszał gracz (3D Object>Plane). Zmieniamy jego wielkość tak, aby sprostała oczekiwaniom, jakie stawiamy przed tworzonym poziomem.

Kolejnym krokiem jest dodanie do poziomu przeszkód, które mają odgradzać postać gracza i jego przeciwników (ograniczenie pola widzenia dla przeciwników). Będą to 2 rodzaje przeszkód: kolumny (3D Object>Cylinder) i ściany (3D Object>Cube). Obiekty te wyposażone są w odpowiednie komponenty *Collider*, dzięki którym nie będą wchodzić w kolizję z graczem, który przez to nie będzie mógł przez nie przechodzić. Ustawiamy je taki sposób, aby dotykały podłoża, ustawiamy je jako obiekty statyczne (checkbox *Static* w zakładce *Inspector*) i przypisujemy do nich odpowiednią, nowo utworzoną warstwę *Walls* (Layers>Add Layer...). Dzięki temu oznaczeniu będziemy mogli sprawdzić, czy przeciwnik widzi gracza (zobacz: 7.6 Tworzenie gracza i przeciwników).



22. Zakładka *Scene* oraz zakładka *Navigation*. W pierwszej zakładce możemy zobaczyć umieszczone obiekty oraz zaznaczoną na niebiesko wygenerowaną siatkę nawigacyjną

Po dodaniu wszystkich przeszkód na scenie, należy cały obiekt terenu osłonić ścianami tak, aby gracz nie był w stanie z niego zejść. Następnie wybieramy obiekt terenu oraz otwieramy zakładkę *Navigation* (Window>Navigation), aby wygenerować siatkę (zobacz: rysunek 22), po której będzie mógł poruszać się przeciwnik. Obiekty statyczne (ściany i kolumny) zostaną zaznaczone jako miejsca, których przeciwnik będzie musiał unikać. W zakładce *Navigation>Bake* klikamy przycisk *Bake* generujący siatkę. Jeżeli nie jesteśmy zadowoleni z efektów widocznych w podglądzie sceny, możemy zmienić dostępne parametry generowania siatki.

Następnie dodajemy także puste obiekty, które będą używane jako punkty, w kierunku których będzie poruszał się przeciwnik podczas patrolu (ang. *waypoint*). Należy uważać, żeby punkt był na takiej samej wysokości, jak tworzony później przeciwnik. Te punkty będziemy mogli potem przypisywać do przeciwnika tak, żeby każdy z nich miał zdefiniowaną trasą patrolu.

"Podłogę", przeszkody i punkt patrolowy zdefiniować należy jako prefabrykaty, gdyż dzięki temu mogą być ponownie użyte.

7.6 Tworzenie gracza i przeciwników

7.6.1 Elementy wspólne

Gracz będzie poruszał się postacią, oraz obracał za pomocą ruchów myszką. Sterowanie będzie możliwe za pomocą klawiszy W, S, A oraz D (jak zwykle w tego typu grach), które będą odpowiedzialne za poruszanie się w górę, dół, lewo i prawo. Obracanie się postacią będzie realizowane przez obrót avatara (płaski obraz) znajdującego się nad graczem (obiektem przestrzennym) w kierunku kursora myszy. Kliknięcie lewego przycisku myszy odpowiada za atak. Chociaż kod przygotowany jest już do rozszerzenia o inne typy broni (w kolejnych wersjach gry), to obecnie dostępnym i jednocześnie domyślnym orężem jest broń biała (ang. *melee*). Gdy atakujemy, przed atakującym pojawi się obiekt odpowiedzialny za wykrycie kolizji z przeciwnikiem - jeżeli kolizja zostanie wykryta, przeciwnik otrzyma obrażenia. Po sprawdzeniu, czy następuje kolizja, obiekt zniknie.

Na początek tworzymy wyżej wspomniany prefabrykat ataku. Będzie to obiekt pojawiający się podczas ataku i wyzwalający kolizję z graczem lub przeciwnikiem (w zależności, kto wykonuje atak). Jeżeli kolizja zostanie wykryta, to od celu kolizji będą odejmowane punkty zdrowia. Obiekt będzie nazwany *MeleeDetector* i będzie prostopadłościanem (3D Object>Cube). Ustawiamy checkbox *isTrigger* komponentu *Box Collider* na *true*. Ustawimy pozycję Y tak, aby obiekt znajdował się nad planszą (dzięki czemu kontakt z podłożem nie wywoła kolizji), dodamy komponent dzięki któremu będziemy mogli z nim łatwo kolidować (Add Component>Rigidbody) oraz zablokujemy w nim możliwość poruszania się (wszystkie checkboxy *Freeze Position* oraz *Freeze Rotation* komponentu *Rigidbody*). Dodamy też do niego skrypt

przechowujący informację o zadawanych obrażeniach oraz obsługujący kolizje. Na chwilę obecną można zostawić domyślny wygląd obiektu, aby atak był widoczny.

```
public float damage = 10f;

private void OnTriggerEnter(Collider collider) {
     if (!transform.parent) {
          Debug.LogError("MeleeDetector instance has no
parent attached");
          Destroy(gameObject);
     } else if ((transform.parent.tag == "Player" &&
collider.CompareTag("Enemy")) ||
          (transform.parent.tag == "Enemy" &&
collider.CompareTag("Player"))) {


collider.gameObject.GetComponent<Character>().TakeDamage((int
) damage);
          Destroy(gameObject);
     }

     Destroy(gameObject, 0.5f);
}
```

Listing 8. Publicznie widoczna w skrypcie *MeleeDetector* zmienna *damage* przechowuje domyślną wartość 10. Przez jej dostępność może być łatwo zmieniona podczas tworzenia instancji. Metoda *OnCollisionEnter(Collision collision)* sprawdza, kto był wykonawcą ataku, a kto celem (zabezpieczenie przed uderzeniem samego siebie) i zadaje obrażenia celowi. Zadanie ułatwia nam fakt, że gracz i jego oponenci dziedziczą po klasie *Character*, dzięki temu nie musimy operować na 2 wyrażeniach *if*, gdyż każda z postaci posiada poniekąd komponent *Character*. Jeżeli przez 0.5 sekundy nie nastąpi kolizja, to obiekt niszczy sam siebie.

Jako, że przeciwnik i gracz będą w pewien sposób podobni, skrypt odpowiedzialny za ich zachowanie będzie miał wspólnego rodzica, opisującego np. ilość punktów życia, czy posiadaną broń. Chociaż te rzeczy będą mogły być zmienione w specyficznych skryptach, to w skrypcie *Character* są one już zadeklarowane. Poniżej przedstawione skrypty *PlayerController* oraz *Enemy* dziedziczą po tej klasie.

```
public int health = 50;
public float movementSpeed = 1f;
public Weapon weaponEquipped;
public GameObject meleeDetectorPrefab;
protected new Rigidbody rigidbody;
```

```
protected void Start() {
      rigidbody = GetComponent<Rigidbody>();
      weaponEquipped = new Weapon(20f, 2f);
}
```

Listing 9. *Character* przechowuje wspólne dla gracza i jego oponentów zmienne: poziom życia, szybkość, broń oraz odniesienie do komponentu *Rigidbody*. Do tego ustawia domyślną broń. W zakładce *Inspector* ustawiamy parametr *meleeDetectorPrefab* przeciągając w jego miejsce prefabrykat *meleeDetector*.

Jak widzimy w listingu powyżej, gracz i przeciwnik wyposażeni są w broń. Opisuje ją skrypt *Weapon*, który jest już przygotowany na rozszerzenie gry o nowy typ broni - broń dystansową.

```
public class Weapon {
      public enum weaponType {
            Melee,
            Ranged
      }

      public weaponType type;
      public float damage;
      public float range;
      public int magazineSize = 0;

      public Weapon(float damage, float range) {
            type = weaponType.Melee;
            this.damage = damage;
            this.range = range;
      }

      public Weapon(float damage, int magazineSize) {
            type = weaponType.Ranged;
            this.damage = damage;
            this.magazineSize = magazineSize;
      }
}
```

Listing 10. *Weapon* opisuje parametry broni oraz udostępnia 2 konstruktory: dla broni białej i dystansowej.

Kolejną wspólną cechą gracza i przeciwnika jest bryła (3D Object>Cylinder), która ma ich imitować oraz przykrywający ją awatar (2D Object>Sprite, utworzony jako obiekt potomny ww. bryły). Obraz awatara (parametr *Sprite* komponentu *Sprite Renderer*) to w tym przypadku koło w formacie .png, utworzone za pomocą programów

Paint (wykonanie grafiki) i Gimp (usuwanie tła). Unity podczas tworzenie obiektu cylindra domyślnie uposaża go w komponent odpowiedzialny za kolizje - można go też oczywiście dodać manualnie (Add Component>Capsule Collider).

7.6.2 Gracz

Gracz różnił się będzie od przeciwnika oczywiście skryptem. Do tego do gracza dodany jest komponent odpowiedzialny za obsługę fizyki (Add Component>Rigidbody). Żeby gracz nie obracał się, możemy zablokować jego zmiany pozycji i rotacji w opcjach komponentu. Blokujemy rotację we wszystkich kierunkach zaznaczając checkboxy oznaczone jako *Freeze Rotation* w komponencie *Rigidbody* oraz upewniamy się, że obiekt nie "wypadnie z planszy" podczas interakcji z innymi obiektami, blokując mu możliwość poruszania się po osi Y (*Freeze Position*, checkbox Y).

```
public float attackCooldown = 1f;
private bool isAttackPossible = true;

private void Update () {
      Move();
      Attack();
}

private void Move() {
      float axisHorizontal = Input.GetAxis("Horizontal");
      float axisVertical = Input.GetAxis("Vertical");

      transform.Translate(new Vector3(axisHorizontal, 0f,
axisVertical) * Time.deltaTime * movementSpeed);
      rigidbody.velocity = new Vector3();
}

private new void Attack() {
      if (Input.GetMouseButtonDown(0)) {
            if (isAttackPossible) {
                  isAttackPossible = false;
                  Invoke("SetAttackPossible", attackCooldown);
                  base.Attack();
            }
      }
}

private void SetAttackPossible() {
      isAttackPossible = true;
```

```
}
```

Listing 11. Skrypt *PlayerController* zaczynamy od definicji częstotliwości ataku. Zmienna *attackCooldown* to ilość sekund, co które gracz może wykonać atak. W określeniu, czy gracz może wykonać atak pomaga także zmienna *isAttackPossible*. W metodzie *Update()* wykonujemy ruch oraz nasłuchujemy kliknięcia lewego przycisku myszy w metodzie *Attack()*.

Do tego inaczej będzie zachowywał się awatar, który ma być skierowany w kierunku myszki. Aby tego dokonać do awatara gracza dodajemy skrypt *RotateTowardsObject*. Gdybyśmy obracali się całym obiektem gracza, to sterowanie byłoby relatywne do całego obiektu i np. klawisz W (ruch do góry) odpowiadałby za poruszanie się w lewo przy obrocie obiektu o 90 stopni w lewo. To byłoby niewygodne dla gracza. Można byłoby też obejść ten problem zmieniając sposób sterowania na bardziej prymitywny.

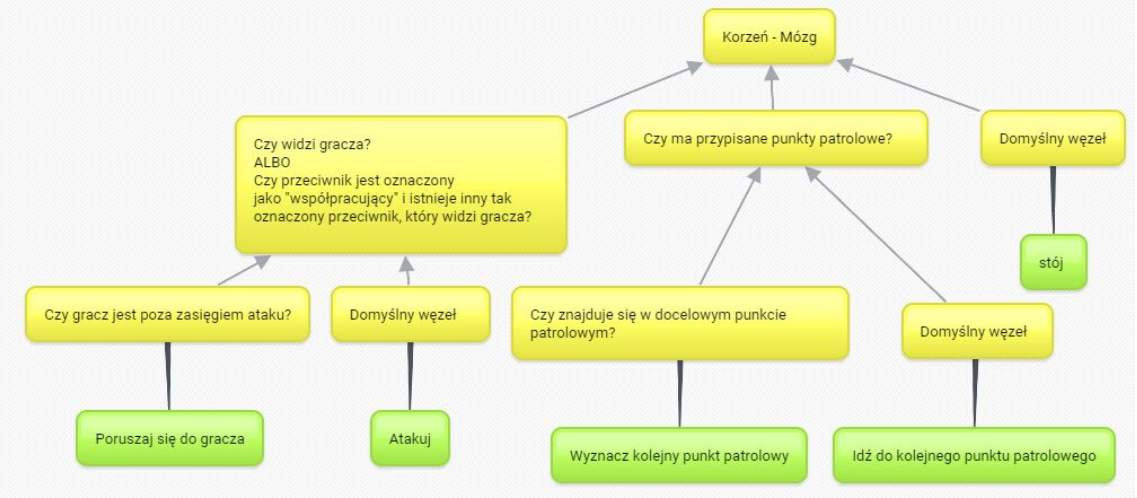
```
private void Update() {
     if (!PauseMenuController.isGamePaused) {
          Vector3 targetPosition =
Camera.main.ScreenToWorldPoint(new
Vector3(Input.mousePosition.x, Input.mousePosition.y, 0));
          transform.LookAt(targetPosition);
          transform.rotation = Quaternion.Euler(new
Vector3(90, transform.rotation.eulerAngles.y, 0));
     }
}
```

Listing 12. Metoda update w skrypcie *RotateTowardsObject* sprawdza, czy gra nie jest zatrzymana (zobacz: 7.7 Interfejs użytkownika wewnątrz gry), a następnie obraca awatarem gracza w kierunku myszy. Ostatnia linia kodu obraca obraz awatara tak, aby odpowiadał naszemu punktowi widzenia.

7.6.3 Przeciwnik, implementacja sztucznej inteligencji

Biorąc stworzoną bryłę za podstawę, dodajemy komponenty specyficzne dla przeciwnika. Nie pomijamy elementu *Rigidbody* (aby łatwo wchodzić w kolizję podczas ataku), jednak nie blokujemy rotacji. Zaznaczamy checkbox *isKinematic*, aby przeciwnik nie mógł być "przesuwany" przez gracza, gdy ten idzie w jego stronę. Dodajemy komponent, który będzie służył do przeciwnikowi do **wyznaczania trasy** i poruszania się (Add Component>Nav Mesh Agent). Dzięki komponentowi Nav Mesh

Agent realizujemy część sztucznej inteligencji (wyszukiwanie ścieżki) przeciwnika - jest on częścią biblioteki *UnityEngine.AI*. Parametry tego komponentu będziemy zmieniać poprzez skrypt *Enemy*. Skrypt *Enemy* wraz z komponentem Nav Mesh Agent odpowiada za całość zachowania przeciwnika (realizacja SI w grze). Nie musimy obsługiwać rotacji awatara, jako, że przy poruszaniu się, przeciwnika obraca z nim (awatar jest obiektem potomnym bryły przeciwnika).



23. Drzewo przedstawiające zachowanie przeciwnika (działanie SI) w grze.

Podstawowa wersja przeciwnika potrafi **poruszać się po zdefiniowanych punktach, patrolując teren** (odpowiednie punkty z rozgrywanego poziomu dodane ("przeciągnięte") do publicznej listy punktów patrolowych przeciwnika) i **gonić przeciwnika**, jeżeli ten nie jest przysłonięty przez ściany. Jeżeli gracz jest w zasięgu ataku, **przeciwnik atakuje gracza**. Kooperacja z innymi oponentami gracza będzie się odbywać w projekcie gry poprzez **wspólne pole widzenia (informacje dzielone między przeciwnikami)** - jeżeli kooperujący przeciwnik będzie widział gracza oraz w grze będzie istniał inny kooperujący przeciwnik, zostanie on poinformowany o pozycji gracza. Zachowanie przeciwnika obrazuje drzewo przedstawione na rysunku 23. Całość zachowań przeciwnika realizuje temat pracy, czyli wykorzystanie metod sztucznej inteligencji w środowisku Unity. Jako rozwinięcie jednej z cech przeciwnika - współpracy, w przyszłości można zaimplementować także mechanizm, dzięki któremu ataki przeciwników będą synchronizowane, a przy dodaniu nowych typów broni

- chowanie się oponentów za ścianami i podchodzenie do gracza z różnych stron (otaczanie).

```
public enum enemyType {
      Basic,
      Armored,
      Fast
}
public enemyType type = enemyType.Basic;
public bool cooperating = false;
public static bool cooperatingVisibility = false;

private GameObject player;
NavMeshAgent navMeshAgent;

private int wallLayer;
private bool playerVisible;
public GameObject[] waypoints;
private int currentWaypoint;

private double nextAttackTime;
private double attackPeriod;

protected new void Start() {
      base.Start();
      player = GameObject.FindGameObjectWithTag("Player");
      setUpType();
      setUpNavMeshAgent();

      wallLayer = 1 << LayerMask.NameToLayer("Walls");
      currentWaypoint = waypoints.Length > 0 ? 0 : -1;

      attackPeriod = 1 / (PlayerPrefsManager.GetDifficulty() /
7 + .334);
}

[...]

private void Update() {
      CheckWaypoint();
      SetDestination();
      Attack();
}

private void CheckWaypoint() {
      if (currentWaypoint >= 0) {
            bool currentWaypointChecked =
Vector3.Distance(transform.position,
```

```
waypoints[currentWaypoint].transform.position) <=
navMeshAgent.stoppingDistance * 1.1f;
            if (currentWaypointChecked) {
                  if (currentWaypoint + 1 == waypoints.Length)
{
                        currentWaypoint = 0;
                  } else {
                        currentWaypoint++;
                  }
            }
      }
}

private void SetDestination() {
      playerVisible = player != null ?
!Physics.Linecast(transform.position,
player.transform.position, wallLayer) : false;
      Vector3 destination;
      if (playerVisible && cooperating) {
            cooperatingVisibility = true;
            Invoke("ResetCooperatingVisibility", 3f);
      }
      if (playerVisible || (cooperatingVisibility &&
cooperating)) {
            destination = player.transform.position;
      } else if (currentWaypoint >= 0) {
            destination =
waypoints[currentWaypoint].transform.position;
      } else {
            destination = transform.position;
      }
      navMeshAgent.destination = destination;
}

private void ResetCooperatingVisibility() {
      cooperatingVisibility = false;
}

private new void Attack() {
      if (Time.time > nextAttackTime) {
            nextAttackTime += attackPeriod;
            if (player && Vector3.Distance(transform.position,
player.transform.position) < 1.5f *
navMeshAgent.stoppingDistance) {
                  base.Attack();
            }
      }
}
```

Listing 13. Początek skryptu *Enemy* to definicja typów przeciwników. Następnie deklarujemy

używane w niej zmienne, dzięki którym kontrolujemy współpracę przeciwników, sposób poruszania się i częstotliwość ataków. W metodzie *Start()* zmieniamy domyślne wartości punktów życia i szybkości przeciwników za pomocą metody *SetUpType()*. Ustawiamy też początkowe wartości dla komponentu *NavMeshAgent* - np. to, w jakiej minimalnej odległości od gracza przeciwnika ma przestać się poruszać. W metodzie *Update()* następuje sprawdzenie punktów patrolowych, a następnie ustalenie punktu, do którego ma poruszać się przeciwnik. Używając biblioteki *Physics* tworzymy linię łączącą przeciwnika i gracza, a następnie sprawdzamy, czy nie przecina ona żadnych ścian - dzięki temu wiemy, czy przeciwnik widzi gracza. Statyczna zmienna *cooperatingVisibility* jest ustawiana na *true*, jeżeli przeciwnik współpracuje z innymi. Dzięki temu inni przeciwnicy będą znać pozycję gracza mimo, że będą dzielić ich od niego ściany. Jeżeli żaden z przeciwników współpracujących nie widzi gracza przez jedną sekundę, to widoczność gracza zostaje wyłączona. Na podstawie tych informacji (widoczność gracza, sprawdzenie punktów patrolowych) możemy ustawić cel, w kierunku którego porusza się przeciwnik. Na sam koniec sprawdzane są warunki metody *Attack()*. Jeżeli przeciwnik jest w odpowiedniej odległości od gracza, wykonuje on atak z wyliczoną na podstawie parametru *difficulty* czytanego z *PlayerPrefsManagera* częstotliwością.

7.7 Interfejs użytkownika wewnątrz gry

Tworzenie UI rozpoczynamy od stworzenia nowego obiektu, na którym będziemy umieszczać kolejne elementy (UI>Panel). Dodajemy do niego komponenty - skrypty: jeden odpowiadający za aktualizowanie danych widocznych na ekranie oraz monitorowanie warunków zwycięstwa i porażki, drugi obsługujący menu pauzy. Następnie, jako obiekty potomne ww. obiektu, dodajemy elementy (UI>Text), które będą pokazywać interesujące gracza informacje: ilość punktów jaką zdobył, liczbę pozostałych do pokonania przeciwników, wskaźnik zdrowia gracza, typ używanej broni oraz wskaźnik amunicji. Odpowiednio je nazywamy.

```
public GameObject winPanel;
public GameObject lossPanel;
public GameObject gameInfo;

private Text health;
private const string healthPrefix = "Health: ";
private PlayerController playerController;
[...]

void Start () {
     health =
GameObject.Find("HealthLabel").GetComponent<Text>();
     playerController =
GameObject.FindGameObjectWithTag("Player").GetComponent<Playe
rController>();
```

```
      [...]
}

void Update () {
      health.text = healthPrefix + playerController.health;

      [...]

      if (playerController.health <= 0) {
            PauseMenuController.isGamePaused = true;
            lossPanel.SetActive(true);
      } else if (enemies <= 0) {
            PauseMenuController.isGamePaused = true;
            Time.timeScale = 0f;
            if (!winPanel.activeSelf) {
                  winPanel.SetActive(true);
                  int levelNumber = GetLevelNumber();
                  PlayerPrefsManager.UnlockLevel(levelNumber +
1);

PlayerPrefsManager.SetLastLevelUnlocked(levelNumber + 1);
            }
      }
}
```

Listing 14. Metoda *Update()* w skrypcie *InGameUIController* odpowiada za aktualizację wyświetlanych przez grę danych dotyczących bieżącego poziomu. Przed jej wywołaniem odpowiednie elementy są znajdowane, aby móc łatwo się do nich odwołać (*score*, *enemyCounter*, itp.) oraz ustawiane są odpowiednie prefiksy do wyświetlanego tekstu.
Kod skryptu odpowiada także za monitorowanie warunków zwycięstwa i wyświetlanie odpowiedniego panelu (wygranej, przegranej). Odnośniki do paneli są zdefiniowane na samym początku skryptu i przypisuje się je w zakładce *Inspector*.

Kolejnym krokiem jest stworzenie menu pauzy. W głównym panelu UI tworzymy obraz (UI>Image) do którego dodajemy pusty element porządkujący o nazwie *Buttons*, do którego z kolei dodajemy przyciski (UI>Text oraz dodanie do nich komponentu *Button*). Przyciski będą dawały możliwość kontynuowania gry, albo powrotu do menu. Za te zachowania będzie odpowiedzialny skrypt *PauseMenuController*.

```
public static bool isGamePaused = false;

void Update () {
      if (Input.GetKeyDown(KeyCode.Escape)) {
            if (isGamePaused) {
                  Resume();
```

```
            } else {
                  Pause();
            }
      }
}

public void Resume() {
      pauseMenuUI.SetActive(false);
      Time.timeScale = 1f;
      isGamePaused = false;
}

private void Pause() {
      pauseMenuUI.SetActive(true);
      Time.timeScale = 0f;
      isGamePaused = true;
}

public void ReturnToMenu() {
      Resume();
      FindObjectOfType<GameManager>().LoadLevel("01a
MainMenu");
}
```

Listing 15. W metodzie *Update()* nasłuchujemy naciśnięcia przycisku Escape przez gracza. Odpowiada on za zatrzymanie i wznowienie gry. Metoda *Resume()* wznawia grę przez modyfikację płynącego czasu ustawiając w nim wartość domyślną (1) oraz dezaktywuje okno pauzy. Metoda *Pause()* ma zadanie przeciwne. Dzięki metodzie *ReturnToMenu()* możemy powrócić do menu - znajduje ona obiekt *GameManager* w scenie i wywołuje w nim metodę *LoadLevel(string sceneName)* podając jako argument nazwę sceny głównego menu. Zmienna *isGamePaused* jest statyczna, gdyż będzie wykorzystywana w innych skryptach, gdy będziemy chcieli się dowiedzieć, czy dane operacje nierozciągnięte w czasie mogą być wykonane.

Na podobnej zasadzie tworzymy panele wygranej i przegranej, z których korzystamy w wyżej zaprezentowanym skrypcie *InGameUIController*. Składają się one z 2 przycisków, jednego odpowiedzialnego (w zależności od wyniku rozgrywki) za powtórzenie poziomu lub przejście do kolejnego i drugiego, przenoszącego nas do menu głównego.

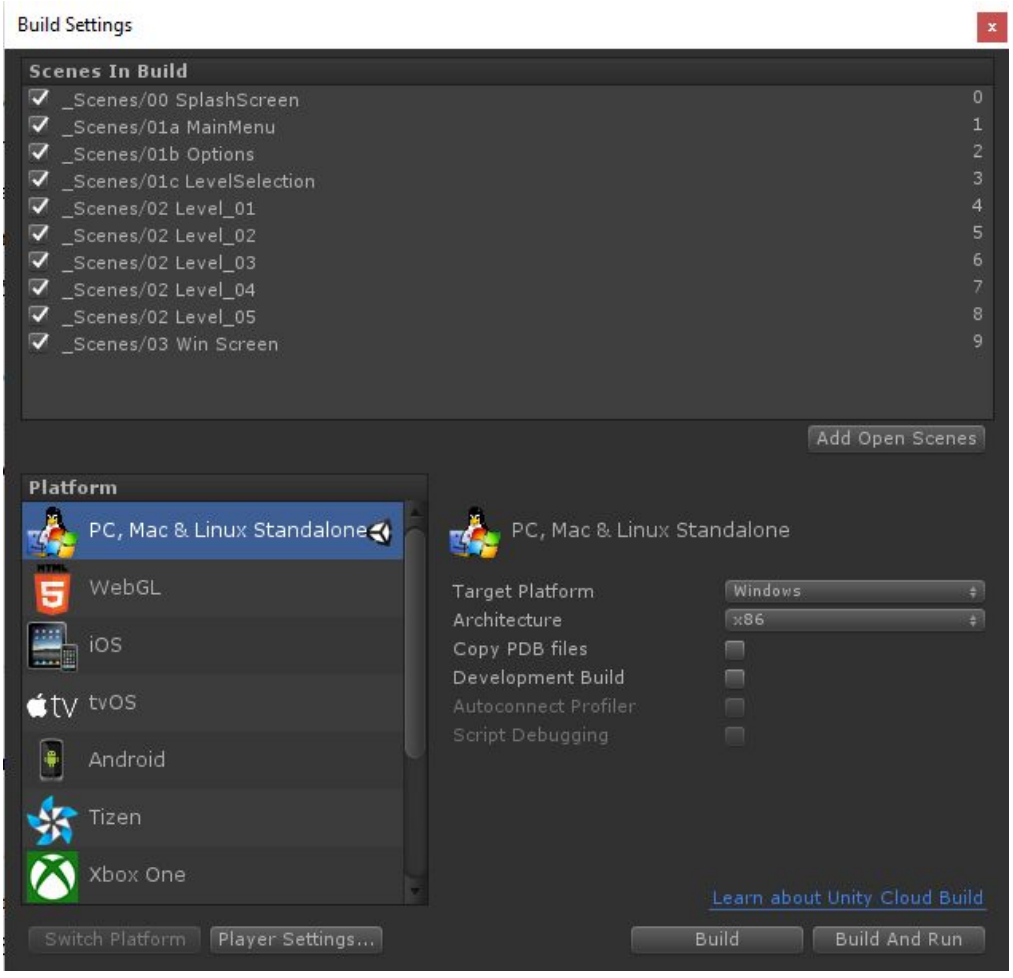
7.8 Budowanie gry

Ostatnim elementem gry jest wdrożenie (ang. *deployment*). Kompilacja gry do pliku wykonywalnego odbywa się przez wybranie opcji File>Build Settings… Pojawia

się menu (zobacz: rysunek 25) pozwalające nam wybrać, które sceny chcemy zbudować, na jakiej platformie gra ma działać (np. Windows, Linux, Android, iOS, przeglądarka internetowa) oraz ustawiamy klikając przycisk *Player Settings* takie rzeczy jak nazwa naszej firmy, nazwa produktu (gry), ustawiamy ikonę pliku, manipulujemy rozdzielczością itp.

Po ustawieniu interesujących nas opcji należy nacisnąć przycisk *Build*. Wybieramy miejsce, gdzie ma pojawić się zbudowana gra i nazywamy plik. Pliki źródłowe oraz wersja binarna, którą można uruchomić na systemie Windows w architekturze x86, są dostępne na portalu GitHub [121] (zobacz: rysunek 24).

24. Kod QR [122] wskazujący repozytorium z plikami źródłowymi gry i jej wersją binarną [121]



25. Menu *Build Settings*

8. Podsumowanie

Sztuczna inteligencja jest szeroko stosowana przy tworzeniu gier komputerowych i często jest kluczowym elementem rozgrywki. Wbrew pozorom, gry komputerowe angażują wiele zaawansowanych pojęć informatyki teoretycznej i ich implementacje, a ich tworzenie może mieć znaczący wpływ na rozwój nauki, także na polu SI.

Tworzenie gier komputerowych to doskonałe ćwiczenie do poznania podstaw sztucznej inteligencji. Pozwala ono zaznajomić się nie tylko z prostymi algorytmami, ale także w przypadku tworzenia bardziej skomplikowanych tytułów może przybliżyć temat chociażby sieci neuronowych.

Cel pracy został osiągnięty. Niektóre z postawionych wstępnych założeń dotyczących realizacji sztucznej inteligencji stawiały wyzwanie (np. wyszukiwanie ścieżki), jednak bliższe poznanie silnika Unity umożliwiło ich łatwą implementację, dzięki gotowym bibliotekom. Część z zachowań SI wymaga dłuższego pochylenia się nad tematem i stworzenia bardziej skomplikowanego systemu (np. otaczanie gracza, korzystanie z broni dystansowej). Gra w zadowalający sposób pokazuje w działaniu, jak silnik Unity wspiera realizowanie SI. Wykonana gra jest na tyle elastycznie zaprojektowana, że może być bez problemów rozwijana.

Bibliografia

[1] Artificial Intelligence - Wikipedia, https://en.wikipedia.org/wiki/Artificial_intelligence, 14 V 2017r

[2] Sztuczna inteligencja - podstawy, http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad1/w1.htm, 14 V 2017r

[3] Test Turinga - Wikipedia, https://pl.wikipedia.org/wiki/Test_Turinga, 14 V 2017r

[4] Deep Blue - Wikipedia, https://pl.wikipedia.org/wiki/Deep_Blue, 14 V 2017r

[5] What is most advanced artifical intelligence in a video game in history - Quora, https://www.quora.com/What-is-the-most-advanced-artificial-intelligence-in-a-video-game-in-history, 14 V 2017r

[6] WGK 2012 - Architektura AI: Drzewa Zachowań, Michał Słapa, https://www.youtube.com/watch?v=aXSWYDFOaYk, 14 V 2017r

[7] WGK 2012 - Projektowanie wydajnego AI, Łukasz Purcelewski, https://www.youtube.com/watch?v=1KXKPictKpo, 14 V 2017r

[8] Dlaczego w grach rozwija się tylko grafika?, Kacper Pitala, https://www.youtube.com/watch?v=Y3OLTJAzQbc&t=9s, 14 V 2017r

[9] Ask a Game Dev - Game Development Myths: Players Want Smart Artificial Intelligence, http://askagamedev.tumblr.com/post/76972636953/game-development-myths-players-want-smart, 15 V 2017r

[10] Ask a Game Dev, http://askagamedev.tumblr.com/post/120055940576/in-one-of-your-past-posts-specifically-the-one-on, 15 V 2017r

[11] The AI Games, http://theaigames.com/, 15 IV 2017r

[12] Precz z idiotami! Najlepsze AI w grach wideo [tvgry.pl], https://www.youtube.com/watch?v=3q1ZOo-GG3c&t=503s, 16 IV 2017r

[13] Games of the Year: The 2014 AiGameDev.com Awards for Game AI | AiGameDev.com, http://aigamedev.com/open/editorial/2014-awards/, 16 V 2017r

[14] Mastering Unity 2D Game Development - Simon Jackson, ISBN 978-1-84969-734-7, Packt Publishing Ltd., sierpień 2014

[15] The Art of Game Design - Jesse Schell, ISBN: 978-0-12-369496-6, Elsevier Inc., 2008

[16] Unity - Manual: Unity User Manual, https://docs.unity3d.com/Manual/index.html

[17] Przegląd silników gier komputerowych (2016), kubas1129, https://www.youtube.com/watch?v=7dMNDZAhYY8

[18] Game Engine Technology aby Unreal, https://www.unrealengine.com/en-US/features, 14 IX 2017r

[19] Game engine - Wikipedia, https://en.wikipedia.org/wiki/Game_engine, 14 IX 2017r

[20] Unity - Game Engine, https://unity3d.com/, 14 IX 2017r

[21] CRYENGINE | The complete solution for next generation game development aby Crytek, https://www.cryengine.com/, 14 IX 2017r

[22] GameMaker | YoYo Games, https://www.yoyogames.com/gamemaker, 14 IX 2017r

[23] The top 10 engines that can help you make your game | GamesBeat | Games | by Jason Wilson, https://venturebeat.com/2014/08/20/the-top-10-engines-that-can-help-you-make-your-game/view-all/, 14 IX 2017r

[24] Trello, https://trello.com/, 14 IX 2017r

[25] System kontroli wersji - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/System_kontroli_wersji, 28 XII 2017r

[26] Video game development - Wikipedia, https://en.wikipedia.org/wiki/Video_game_development, 14 IX 2017r

[27] What is Video Game Development | Study Video Game Development in the US, https://www.internationalstudent.com/study-video-game-development/what-is-video-game-development/, 14 IX 2017r

[28] Game Theory Applied: The Flow Channel - Indie Dev StoriesIndie Dev Stories, http://indiedevstories.com/2011/08/10/game-theory-applied-the-flow-channel/, 1 X 2017r

[30] Unity Community, https://forum.unity.com/, 1 X 2017r - 4 I 2018r

[31] Unity - Manual: Unity User Manual (2017.2), https://docs.unity3d.com/Manual/index.html, 1 X 2017r - 4 I 2018r

[32] Unity AI Game Programming - Second Edition, Ray Barrera, Packt Publishing

[33] Game Pre-Production? How is it done? : gamedev, https://www.reddit.com/r/gamedev/comments/2be1lt/game_preproduction_how_is_it_done/, 12 XII 2017r

[34] The 50 Top Video Game Design Companies in The Worls, https://www.gamedesigning.org/game-development-studios/, 29 XII 2017r

[35] Gamasutra: Sam Coster's Blog - Indiepocalypse? More like INDIESCHMOCALYPSE!, https://www.gamasutra.com/blogs/SamCoster/20151002/255185/Indiepocalypse_More_like_INDIESCHMOCALYPSE.php, 29 XII 2017r

[36] Hotline Miami - Wikipedia, https://en.wikipedia.org/wiki/Hotline_Miami, 29 XII 2017r

[37] Visual Studio IDE, Code Editor, Team Services i Mobile Center, https://www.visualstudio.com/pl/, 29 XII 2017r

[38] Podstawy języka :: 4programmers.net, https://4programmers.net/C_sharp/Podstawy_j%C4%99zyka, 29 XII 2017r

[39] Documentation, Unity scripting languages and you - Unity Blog, https://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/, 29 XII 2017r

[40] Cykl życia programu - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Cykl_%C5%BCycia_programu, 2 I 2018r

[41] U.S. computer and video game sales - digital vs. physical 2016 | Statistic, https://www.statista.com/statistics/190225/digital-and-physical-game-sales-in-the-us-since-2009/, 2 I 2018r

[42] 20 years ago, a computer first beat a chess world champion, http://mashable.com/2016/02/10/kasparov-deep-blue/#zSliyrDexEq3, 4 I 2018r

[43] Singleton pattern - Wikipedia, https://en.wikipedia.org/wiki/Singleton_pattern, 4 I 2018r

[44] Alan Turing - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Alan_Turing, 5 I 2018r

[45] Ex Machina (2015), http://www.filmweb.pl/film/Ex+Machina-2015-686419, 5 I 2018r

[46] Gwiezdne wojny - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Gwiezdne_wojny, 5 I 2018r

[47] Prawie wszystko o Logice Rozmytej, http://www.isep.pw.edu.pl/ZakladNapedu/dyplomy/fuzzy/index.htm, 5 I 2018r

[48] Algorytm ewolucyjny - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Algorytm_ewolucyjny, 5 I 2018r

[49] Co to jest sieć neuronowa i jak się uczy :: PCLab.pl, http://pclab.pl/art71255-2.html, 5 I 2018r

[50] Sztuczne Życie i algorytmy inspirowane biologicznie | Portal Sztucznego Życia, alfie.pl, http://www.alife.pl/czym-jest-sztuczne-zycie, 5 I 2018r

[51] Robotyka - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Robotyka, 5 I 2018r

[52] Natural language understanding - Wikipedia, https://en.wikipedia.org/wiki/Natural_language_understanding, 5 I 2018r

[53] Autonomous car - Wikipedia, https://en.wikipedia.org/wiki/Autonomous_car, 5 I 2018r

[54] Computer simulation - Wikipedia, https://en.wikipedia.org/wiki/Computer_simulation, 5 I 2018r

[55] How AI Will Transform Networking, https://www.networkcomputing.com/network-security/how-ai-will-transform-networking/401941015, 5 I 2018r

[56] Computer AI passes Turing test in ‘world first’ - BCC News, http://www.bbc.com/news/technology-27762088, 5 I 2018r

[57] Sztuczna inteligencja - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Sztuczna_inteligencja, 5 I 2018r

[58] System ekspercki - Encyklopedia Zarządzania, https://mfiles.pl/pl/index.php/System_ekspercki, 5 I 2018r

[59] AlphaGo Zero - sztuczna inteligencja gra w go, https://businessinsider.com.pl/technologie/nowe-technologie/alphago-zero-sztuczna-inteligencja-gra-w-go/v6s3hds, 5 I 2018r

[60] Goal Oriented Action Planning for Smarter AI, https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793, 5 I 2018r
[61] A* Pathfinding Project: Graph Types, https://arongranberg.com/astar/docs/graph_types.php, 5 I 2018r
[62] sztuczna inteligencja - Encyklopedia PWN - źródło wiarygodnej i rzetelnej wiedzy, https://encyklopedia.pwn.pl/haslo/sztuczna-inteligencja;3983490.html, 7 I 2018r
[63] Bionika - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Bionika, 7 I 2018r
[64] Game Artificial Intelligence: Challenges for the Scientific Community, Raúl Lara-Cabrera, Mariela Nogueira-Collazo, Carlos Cotta and Antonio J. Fernández-Leiva, http://ceur-ws.org/Vol-1394/paper_1.pdf, 15 V 2017r
[65] Oficjalna strona | Minecraft, https://minecraft.net/pl-pl/, 7 I 2018r
[66] Terraria, https://terraria.org/, 7 I 2018r
[67] No Man's Sky, https://www.nomanssky.com/, 7 I 2018r
[68] Automat skończony - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Automat_sko%C5%84czony, 7 I 2018r
[69] Wiedźmin 3: dziki gon - Oficjalna strona, http://thewitcher.com/pl/witcher1/, 7 I 2018r
[70] Gamasutra: Chris Simpson's Blog - Behavior Trees for AI: How they work, https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php, 7 I 2018r
[71] Strategiczna gra czasu rzeczywistego - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Strategiczna_gra_czasu_rzeczywistego, 7 I 2018r
[72] Home - Total War, https://www.totalwar.com/, 7 I 2018r
[73] Black & White (PC) | GRYOnline.pl, https://www.gry-online.pl/S016.asp?ID=32, 7 I 2018r
[74] Ubisoft - Heroes of Might and Magic III HD, https://www.ubisoft.com/en-gb/game/heroes-of-might-and-magic-3-hd/, 7 I 2018r
[75] Total War: Rome II - Empire Divided w serwisie Steam, http://store.steampowered.com/app/694880/Total_War_ROME_II__Empire_Divided/, 7 I 2018r

[76] SimCity, https://www.ea.com/pl-pl/games/simcity/simcity, 7 I 2018r
[77] Caesar III - Wikipedia, https://en.wikipedia.org/wiki/Caesar_III, 7 I 2018r
[78] Shining Rock Software, http://www.shiningrocksoftware.com/, 7 I 2018r
[79] Football Manager: The world's favourite football menagement game, http://www.footballmanager.com/, 7 I 2018r
[80] Sanctum - Coffee Stain Studios, https://www.coffeestainstudios.com/games/sanctum/, 7 I 2018r
[81] The Elder Scrolls Official Site | Oblivion, https://elderscrolls.bethesda.net/en/oblivion, 7 I 2018r
[82] Radiant AI - Wikipedia, https://en.wikipedia.org/wiki/Radiant_AI, 7 I 2018r
[83] Gothicpedia | FANDOM powered by Wikia, http://pl.gothic.wikia.com/wiki/Gothic_i_ArcaniA, 7 I 2018r
[84] Shadow of War, https://www.shadowofwar.com/, 7 I 2018r
[85] Ubisoft - Assassin's Creed Unity, https://www.ubisoft.com/en-us/game/assassins-creed-unity, 7 I 2018r
[86] Games of the Year: The 2014 AiGameDev.com Awards for Game AI | AiGameDev.com, http://aigamedev.com/open/editorial/2014-awards/#support, 7 I 2018r
[87] Diablo III - oficjalna strona gry, https://eu.battle.net/d3/pl/, 7 I 2018r
[88] F.E.A.R - Moholith, https://www.lith.com/games/fear, 7 I 2018r
[89] Halo - Oficjalna Strona, https://www.halowaypoint.com/pl-pl, 7 I 2018r
[90] Community | Just Cause 3, https://justcause.com/, 7 I 2018r
[91] Forza Horizon 3, https://forzamotorsport.net/en-US/games/fh3, 7 I 2018r
[92] Mario Kart™ 8 Deluxe for Nintendo Switch™ – Official Site, https://mariokart8.nintendo.com/, 7 I 2018r
[93] Strona startowa OGame, https://pl.ogame.gameforge.com/, 7 I 2018r
[94] World of Warcraft, https://worldofwarcraft.com/en-us/, 7 I 2018r
[95] Revisiting the World of Warcraft, nine years after I left | Ars Technica, https://arstechnica.com/gaming/2016/09/revisiting-the-world-of-warcraft-nine-years-after-i-left/, 7 I 2018r
[96] Gry wideo The Sims - Oficjalna strona EA, https://www.ea.com/pl-pl/games/the-sims, 7 I 2018r

[97] The Sims Wiki | Fandom powered by Wikia, http://sims.wikia.com/wiki/The_Sims_Wiki, 7 I 2018r
[98] Nintendo - Official Site - Video Game Consoles, Games, https://www.nintendo.com/, 7 I 2018r
[99] Valve, http://www.valvesoftware.com/, 7 I 2018r
[100] Rockstar Games, https://www.rockstargames.com/, 7 I 2018r
[101] Strona główna Electronic Arts - Oficjalna strona EA, https://www.ea.com/pl-pl, 7 I 2018r
[102] Activision Blizzars, https://www.activisionblizzard.com/, 7 I 2018r
[103] Home - CD PROJEKT, https://www.cdprojekt.com/pl/, 7 I 2018r
[104] Strona główna • Techland, http://techland.pl/?lang=pl, 7 I 2018r
[105] 11 bit studios - official site, http://www.11bitstudios.com/pl/, 7 I 2018r
[106] Witamy w serwisie Steam, http://store.steampowered.com/, 7 I 2018r
[107] Google Play, https://play.google.com/store, 7 I 2018r
[108] Divinity: Original Sin 2 by Larian Studios LLC - Kickstarter, https://www.kickstarter.com/projects/larianstudios/divinity-original-sin-2, 7 I 2018r
[109] Divinity: Original Sin 2, https://divinity.game/, 7 I 2018r
[110] SteamSpy - All the data and stats about Steam games, https://steamspy.com/, 7 I 2018r
[111] Game Engine Technology by Unreal, https://www.unrealengine.com/en-US/what-is-unreal-engine-4, 7 I 2018r
[112] CRYENGINE | The complete solution for next generation game development by CryTech, https://www.cryengine.com/, 7 I 2018r
[113] GameMaker | YoYo Games, https://www.yoyogames.com/gamemaker, 7 I 2018r
[114] Left 4 Dead Blog, http://www.l4d.com/blog/, 7 I 2018r
[115] Jira | Oprogramowanie do śledzenia postępu projektów i statusu zgłoszeń | Atlassian, https://pl.atlassian.com/software/jira, 7 I 2018r
[116] Kanban - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Kanban, 7 I 2018r
[117] GIMP - GNU Image Manipulation Program, https://www.gimp.org/, 7 I 2018r
[118] .Net Framework - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/.Net_Framework, 7 I 2018r

[119] Hotline Miami | Games | Devolver Digital, https://www.devolverdigital.com/games/view/hotline-miami, 7 I 2018r

[120] Microsoft Paint - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Microsoft_Paint, 10 I 2018r

[121] GitHub - mikolajrazny/android_revolution: project of unity game, https://github.com/mikolajrazny/android_revolution, 10 I 2018r

[122] Kod QR - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Kod_QR, 10 I 2018r

[123] Drzewo decyzyjne - Wikipedia, wolna encyklopedia, https://pl.wikipedia.org/wiki/Drzewo_decyzyjne, 10 I 2018r