helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Perełki programowania gier. Vademecum profesjonalisty. Tom 6

Autor: Mike Dickheiser
Tłumaczenie: Andrzej Grażyński (wstęp, części I-III, VI-VII); Mikołaj Szczepaniak (części IV-V)
ISBN: 978-83-246-1087-7
Tytuł oryginału: Game Programming Gems 6 (Book & CD-ROM)
Format: B5, stron: około 700
oprawa twarda
Zawiera CD-ROM

Wyczerpujący przegląd nowoczesnych technik tworzenia gier komputerowych

- Zaawansowane algorytmy sztucznej inteligencji
- Realistyczne odwzorowywanie zjawisk fizycznych, efektów dźwiękowych i oświetlenia
- Języki skryptowe i sterowanie danymi
- Techniki zabezpieczania gier w wersji beta

Przemysł gier komputerowych jest jedną z najszybciej rozwijających się branż informatyki. Gry, które jeszcze niedawno zapierały dech w piersiach, dziś wydają się proste i mało realistyczne. Współczesne symulatory i „strzelanki" to arcydzieła, których produkcja angażuje środki porównywalne z budżetami hollywoodzkich superprodukcji. Rosnące w ogromnym tempie możliwości komputerów i konsoli wpływają jednak nie tylko na jakość gier, ale także na proces ich tworzenia i wykorzystywane podczas niego narzędzia. Programiści tworzący gry implementują zaawansowane algorytmy sztucznej inteligencji, wykorzystują niezwykłe możliwości układów wyświetlających grafikę i skomplikowany aparat matematyczny.

Książka „Perełki programowania gier. Vademecum profesjonalisty. Tom 6" to doskonałe kompendium wiedzy dla wszystkich, którzy tworzą gry komputerowe lub zamierzają to robić. W każdym rozdziale, napisanym przez cenionego eksperta z tej branży, znajdziesz niezbędne informacje dotyczące różnych aspektów pisania gier. Przeczytasz o sztucznej inteligencji, symulacji zjawisk fizycznych oraz odwzorowywaniu oświetlenia i dźwięków. Poznasz nowoczesne techniki programowania współbieżnego, optymalizacji kodu pod kątem maszyn wieloprocesorowych, stosowania języków skryptowych i wykorzystywania możliwości procesorów graficznych.

- Programowanie pod kątem procesorów wielordzeniowych
- Siatkowa reprezentacja obiektów gry
- Testowanie pojedynczych modułów gry
- Optymalizacja korzystania z zasobów
- Rachunek wektorowy i macierzowy
- Symulacja zachowania cieczy
- Tworzenie algorytmów sztucznej inteligencji
- Korzystanie z reguł logiki rozmytej
- Programowanie skryptowe w językach Lua i Python
- Optymalizacja wyświetlania obiektów gry
- Wykorzystywanie możliwości procesorów graficznych
- Generowanie efektów dźwiękowych w czasie rzeczywistym
- Tworzenie gier sieciowych

Przeczytaj i stwórz grę, w którą zagrają miliony graczy

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.1

Zastosowanie modelowanego podejścia do implementacji AI na przykładzie Locust AI Engine w QA3

Armand Prieditis, Lookahead Decisions Inc.
prieditis@lookaheaddecisions.com

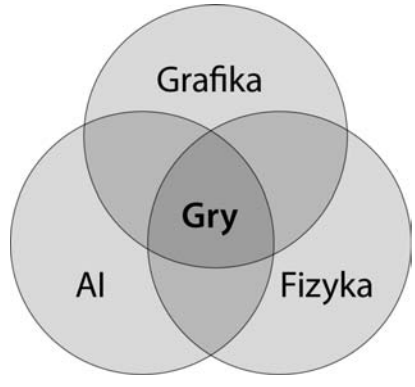
Mukesh Dalal, Lookahead Decisions Inc.
mukesh@lookaheaddecisions.com

Tworzenie dobrych gier w oparciu o AI jest naprawdę trudną sztuką. Tradycyjna filozofia tej sztuki opera się na regułach (ang. *rules*), które — niestety — są często nieprecyzyjne, a ich definiowanie, implementowanie, debugowanie i modyfikowanie — dość kosztowne. W dodatku często bywa i tak, iż ostateczny rezultat zastosowania owych reguł zdaje się mieć z jakąkolwiek inteligencją niewiele wspólnego. W niniejszym artykule opisujemy alternatywne podejście do sztucznej inteligencji — podejście oparte na modelowaniu realnego świata, w którym rozgrywa się scenariusz, czyli świata akcji, ich efektów i obserwacji. Jest to podejście bardziej niezawodne, elastyczne i skutkujące inteligentniejszym zachowaniem aktorów gry; w charakterze przykładu przedstawimy dotychczasowe rezultaty jego zastosowania w postaci silnika Locust AI Engine na potrzeby gry *Quake III*. Skoncentrujemy się głównie na korzyściach, jakie użytkownik gry zyskuje dzięki sztucznej inteligencji, i z tej perspektywy przedstawimy wizję przyszłych jej zastosowań.

Wprowadzenie

Z rysunku 3.1.1 wynika, że współczesne gry komputerowe osiągają swój realizm, czerpiąc z trzech dziedzin wiedzy: grafiki, fizyki i sztucznej inteligencji. Co prawda, samo pojęcie „realizmu" trudno jednoznacznie zdefiniować w sposób formalny, niemniej jednak intuicyjnie przekłada się ono w pierwszym rzędzie na zaangażowanie, jakie staje się udziałem gracza, i inteligencję przejawianą w zachowaniu tzw. bohaterów niezależnych (BN, z angielskiego *Non-Player Characters — NPC*)[1]: wygląd i zachowanie obiektów zbliżone jest mocno do rzeczywistości, a działania bohaterów niezależnych wydają się racjonalne. Te właśnie względy są nawet ważniejsze niż kreowanie „niezwyciężonych" przeciwników.

Rysunek 3.1.1. *Gry komputerowe jako kombinacja zastosowań trzech technologii*



Bo do tworzenia „niepokonanych" wcale nie jest konieczna sztuczna inteligencja — często wystarcza zwyczajne oszustwo czy nieuczciwość. Nietrudno przecież stworzyć BN, który bezbłędnie, *natychmiast* odnajduje kryjówkę przeciwnika (w którego wciela się gracz); zachowaniem bardziej realistycznym (mimo iż, oczywiście, mniej skutecznym) jest jednak przeszukiwanie terenu, budynku itp. przypominające racjonalne postępowanie. W ten oto sposób interes gracza postawiony zostaje ponad interesem jego przeciwnika; tego właśnie gracze od nas oczekują i to powinno stanowić nasz cel, cel projektantów.

W dwóch cytowanych na rysunku 3.1.1 dziedzinach — grafice komputerowej i obliczeniowych zastosowaniach fizyki —na przestrzeni kilku ostatnich lat poczyniono olbrzymi postęp. Oszałamiające możliwości takich narzędzi jak Maya czy 3D Max są obecnie raczej normą niż wyjątkiem, a pakiety fizyczne w rodzaju Havok ułatwiają programowanie realistycznych scen. Z tego powodu ani grafika, ani realistyczne symulacje zjawisk fizycznych nie są już dziś tymi czynnikami, które decydują o powodzeniu gier w kręgach ich użytkowników. Jeśli natomiast chodzi o sztuczną inteligencję, to daje ona większą okazję do twórczej inwencji i nowatorstwa, bo nie był jej udziałem żywiołowy rozwój, jakiego doświadczyli dwaj jej partnerzy. Pozostająca niejako w ich cieniu nie doczekała się też (na razie) żadnego wsparcia sprzętowego, więc siłą rzeczy związane z nią obliczenia obciążają bezpośrednio procesory centralne (CPU), co stawia przed programistami nowe wyzwania, ale też daje im szerokie pole do popisu. Z drugiej jed-

[1] Patrz *http://pl.wikipedia.org/wiki/Bohater_niezale%C5%BCny — przyp. tłum.*

nak strony, powierzenie wielu skomplikowanych funkcji koprocesorom graficznym zmniejsza obciążenie CPU, których moc przeznaczyć można tym samym w większym stopniu na potrzeby AI. W niniejszym artykule przedstawimy kilka przykładów wykorzystania tej sposobności.

Stan obecny — zbiory reguł deterministycznych

W większości współczesnych gier pierwiastek sztucznej inteligencji manifestuje się poprzez reguły. Każda reguła jest parą o postaci „sytuacja → akcja" — gdy stwierdzone zostanie wystąpienie (zaistnienie) określonej sytuacji, uruchamia się właściwą dla niej akcję, którą może być pojedynczy ruch obiektu, sekwencja takich ruchów określona przez skrypt czy też zachowanie bardziej złożone, reprezentowane przez klip wideo.

I tak np. widoczność wroga i słaba kondycja gracza to sytuacja nakazująca odwrót tego ostatniego, podobnie jak stwierdzenie wyraźnej przewagi wroga pod względem uzbrojenia. Animacja prezentowana w związku z akcją może być wyświetlana przez ustalony odcinek czasu bądź też do momentu kolejnego „dopasowania" aktualnej sytuacji do jednej z reguł. „Sytuacja" (jako składnik reguły) może mieć niekiedy postać skomplikowanej formuły boole'owskiej — w przedstawionym przykładzie jest to alternatywa niewyłączająca (OR) dwóch okoliczności.

Inną popularną regułą, związaną z poszukiwaniem ścieżek, jest wybór przez BN najkrótszej drogi prowadzącej do określonego celu. Przy braku dynamicznych przeszkód i innych agentów AI reguła ta sprawdza się całkiem dobrze.

Sposoby dopasowywania aktualnej sytuacji do konkretnej reguły mogą być zróżnicowane. Niekiedy proces dopasowywania może być sterowany za pomocą drzewa decyzyjnego, czego przykład przedstawiliśmy na rysunku 3.1.2: węzły drzewa reprezentują wówczas pewne zdefiniowane a priori okoliczności elementarne, zaś każdy z liści — jedną ze zdefiniowanych akcji. Na wspomnianym rysunku wszystko zaczyna się od sprawdzenia, czy BN ma sprawną broń; jeśli tak, sprawdza się, czy znajduje się on dostatecznie blisko przeciwnika. Spełnienie obu tych reguł wyzwala akcję, którą jest atak; jak łatwo zauważyć, atak może być również podjęty w jeszcze jednej sytuacji — BN nie ma broni, ale stwierdza, że przeciwnik, jako niższy wzrostem, jest słabszy fizycznie i można go pokonać w walce wręcz.

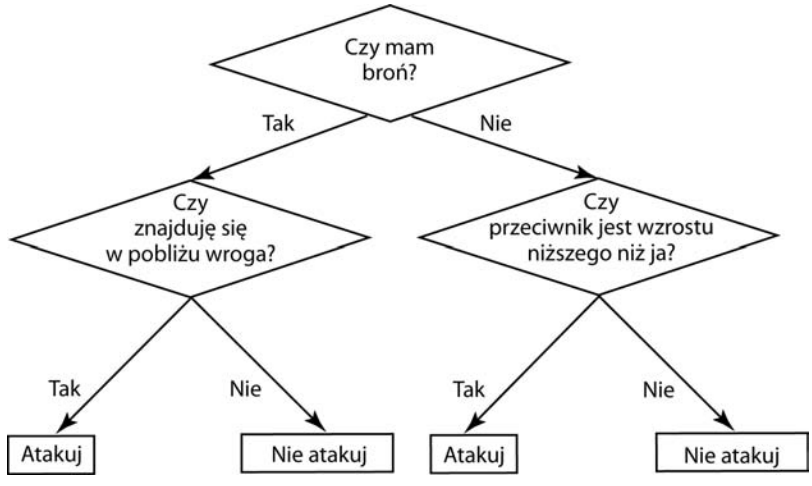
Każda ze ścieżek, prowadząca w drzewie decyzyjnym od jego korzenia do jednego z liści, reprezentuje więc pojedynczą regułę „sytuacja → akcja", przy czym „sytuacja" zdefiniowana jest przez ciąg odwiedzanych po drodze węzłów, zaś „akcja" reprezentowana jest przez liść kończący ścieżkę. Mimo iż drzewo decyzyjne stanowi bardzo oszczędną formę kodowania reguł, to pod względem zawartości informacyjnej jest ono równoważne klasycznemu zbiorowi par „sytuacja → akcja".

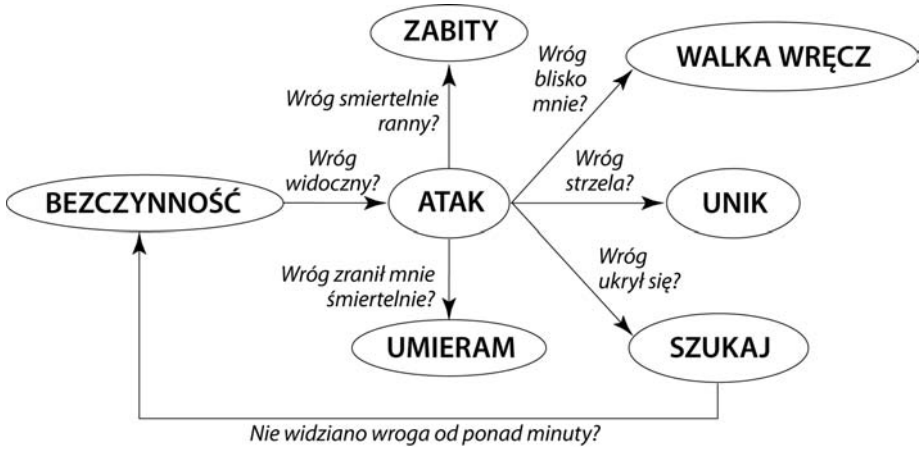
Innym środkiem kodowania reguł są maszyny skończenie stanowe (ang. *Finite State Machines — FSM*). Maszyna taka składa się ze zbioru (zdefiniowanych a priori) stanów[2], z których każdy odpowiada określonej akcji oraz (reprezentowanych w formie łuków)

[2] Z wyróżnionym stanem początkowym — *przyp. tłum.*

przejść między stanami, odpowiadających testowaniu sytuacji [Rabin02]. W maszynie FSM widocznej na rysunku 3.1.3 stanem początkowym może być BEZCZYNNOŚĆ, odpowiadająca bezczynności BN, aż do momentu ujrzenia wroga, kiedy to następuje przejście do stanu ATAK (który reprezentuje akcję, np. w formie wystrzałów z karabinu). Gdy w stanie ATAK BN traci wroga z pola widzenia, następuje przejście do stanu jego poszukiwania (SZUKAJ), czemu odpowiadać może inna stosowna animacja.

Rysunek 3.1.2. *Proste drzewo decyzyjne*





Rysunek 3.1.3. *Przykładowa maszyna skończenie stanowa*

Podobnie jak drzewo decyzyjne, FSM równoważna jest informacyjnie zwykłemu zbiorowi reguł; mimo to jest łatwiejsza w obsłudze od drzewa decyzyjnego, nie wymaga bowiem jawnego manipulowania flagami boole'owskimi czy stanami. Z tego względu maszyny skończenie stanowe wykorzystywane są w wielu komercyjnych grach, m.in. w *Age of Empires*, *Enemy Nations*, *Half-Life*, *Doom* i *Quake*.

Odrębnym zagadnieniem jest kodowanie przechowywanych reguł. Stosując logikę rozmytą, nadajemy regułom postać umożliwiającą ich dopasowywanie za pomocą metod probabilistycznych; z kolei sieci neuronowe zapewniają realizację bardziej złożonych dopasowań — oba te sposoby umożliwiają formułowanie złożonych warunków prościej i łatwiej niż w standardowych zbiorach reguł, stwarzają też możliwość „uczenia się" nowych reguł na podstawie obserwowanych przykładów. Systemy eksperckie łączą natomiast serie reguł z wewnętrznymi interpreterami, wprowadzającymi swoje własne symbole (takie, które nie są związane bezpośrednio z zaistniałymi sytuacjami); mimo iż może to przydać dość dużej elastyczności procesowi dopasowywania reguł, jest obliczeniowo bardziej kosztowne w porównaniu z dwiema wcześniej opisywanymi metodami i dlatego systemy eksperckie rzadko używane są podczas tworzenia gier. Zwróćmy uwagę, że system ekspercki równoważny jest standardowemu zbiorowi reguł w tym sensie, że najsłabsze warunki uzasadniające stosowalność reguły mogą być utożsamiane z „sytuacją", a wykonywane przez ten system działanie — z „akcją".

Reasumując, reguły mogą występować w wielu formach, o różnym stopniu elastyczności i czytelności, zawsze jednak będą one równoważne zbiorowi par „sytuacja → akcja".

Problemy związane z regułami

Mimo iż reguły mogą wydawać się w pierwszej chwili atrakcyjnym mechanizmem implementacji sztucznej inteligencji w tworzonych grach, wiąże się z nimi kilka problemów. Oto najważniejsze z nich.

Reguły często prowadzą do nieinteligentnych zachowań. Trudno za ich pomocą ogarnąć długofalowe rozgałęzienie akcji: po wykonaniu specyficznych działań przez kilku BN może się okazać, że pierwotnie wykonane działania nie były dobrym pomysłem. Przykładowo, gdy wszyscy BN zdążają najkrótszą ścieżką do tego samego celu, łatwo może na tej ścieżce powstać tłok.

Reguły okazują się kosztowne w implementowaniu i realizacji. Projektant definiujący reguły musi wykazywać się obszerną wiedzą, by jego praca dała zadowalające wyniki.

Zachowanie oparte na regułach uwarunkowane jest starannością projektantów. W zachowaniu postaci nie zaobserwujemy żadnego inteligentnego zachowania, które wcześniej nie zostało ściśle zaprogramowane. W obliczu napiętych terminów trudno oczekiwać od projektanta, że znajdzie wystarczająco dużo czasu na wyposażenie każdego BN w odpowiedni zasób inteligencji. Nie należy zapominać, iż dobrzy programiści gier nie zawsze są dobrymi graczami.

Zachowanie oparte na regułach może rozczarowywać graczy i wzbudzać ich nieufność. Jednym z najważniejszych czynników stanowiących o powodzeniu gry jest wywołanie pozytywnego wrażenia u graczy. Twarda, konsekwentna ręka projektanta może przynieść odczucie wręcz przeciwne — nieufność: nieracjonalne, sztuczne zachowanie powoduje, że po pozytywnym nastroju nie pozostaje ani śladu, zbyt inteligentne zachowania postaci odbierane zostają jako przejaw prymitywizmu.

Reguły bywają kruche — zawodzą w przypadku nawet drobnych odstępstw od przewidzianego a priori scenariusza. Jeżeli np. wszyscy BN wybierają zawsze najkrótszą ścieżkę do celu, to chwilowe nawet zablokowanie takiej ścieżki regułę tę dyskwalifikuje. Reguły okazują się też wątpliwe w obliczu informacji niepełnej, niepewnej i nieprecyzyjnej — nieznanego lub niedokładnie określonego położenia obiektów, nakładających się akcji, akcji uzależnionych czasowo itp.

Reguły okazują się trudne w modyfikowaniu, debugowaniu i generalnie skutkują nieefektywnymi iteracjami „generuj i poprawiaj". Typowym sposobem poprawiania „kruchej" reguły jest definiowanie od niej wyjątków. W efekcie projektant dodawać musi wciąż nowe wyjątki w związku z kolejnymi nietypowymi sytuacjami. Jako przykład rozpatrzmy następującą regułę: jeśli kondycja BN spadnie poniżej 50% normalnej wartości, powinien się on wycofać z walki i poszukać możliwości „doładowania". Jeżeli jednak przeciwnik BN jest już na skraju wyczerpania i można by go zabić jednym strzałem, wycofanie się z walki przy 50% kondycji wydaje się postępowaniem ewidentnie nieracjonalnym — i to jest właśnie jedna z sytuacji, gdy cytowana reguła okazuje się nieadekwatna. Definiujemy więc pierwszy wyjątek: jeśli kondycja BN jest mniejsza niż 50%, a kondycja jego przeciwnika mniejsza niż 5%, BN najpierw uśmierca przeciwnika, dopiero potem szuka możliwości „doładowania". Rozwiązujemy w ten sposób jeden problem, ale — niestety — zaraz pojawia się następny: co się stanie, gdy amunicja BN będzie już na wyczerpaniu i nie wystarczy jej na zabicie (wyczerpanego już skądinąd) przeciwnika? Nie ma innej rady, jak zdefiniowanie wyjątku od wyjątku: jeśli kondycja BN jest mniejsza niż 50%, a kondycja jego przeciwnika mniejsza niż 5% i jednocześnie amunicja BN nie jest mniejsza niż 5% maksimum, BN najpierw uśmierca przeciwnika, dopiero potem szuka możliwości „doładowania". Takie „łatanie logiki" jest procesem nie tylko nieefektywnym, ale prowadzi do przysłowiowego „kodu spaghetti"[3] — trudnego w utrzymaniu i nieodpornego na dodawanie do gry nowych cech. Debugowanie takiego kodu staje się koszmarem, bo wobec skomplikowanego kontekstu i złożonej informacji o stanie przyczyna błędu może znajdować się wiele ramek wcześniej od tej ramki, w której widoczne stają się jego konsekwencje.

Reguły nigdy nie są kompletne. Opisany proces „łatania" reguł — zdefiniuj nowy wyjątek i testuj regułę aż do napotkania kolejnego przypadku, w którym okaże się nieadekwatna — prowadzi nieuchronnie do niekompletnego produktu AI. Proces ten można bowiem ciągnąć potencjalnie w nieskończoność i zazwyczaj kończy się, gdy upływa termin ukończenia kolejnego etapu projektu. I wtedy pozostaje tylko nadzieja, że użytkownicy nie odkryją luk w logice AI — luk, o których często nie wiedzą sami projektanci. Użytkownicy są niezwykle dumni z każdej odkrytej luki tego rodzaju, dają temu wyraz na grupach dyskusyjnych i w blogach, ścierając w proch wszelkie pozytywne recenzje i reklamowy szum wokół „wspaniałości i niepowtarzalności gry".

Reguły są nieprzewidywalne i niestabilne. Opisywane już łatanie reguł skutkuje nie tylko trudnością w ich utrzymaniu, ale także ich wysoką nieprzewidywalnością. Jakaś nieprzewidziana sytuacja może załamać lub zapętlić całą grę.

[3] Nazwa ta wywodzi się stąd, że ingerencja w jednym miejscu kodu ma swe konsekwencje w jakimś odległym jego fragmencie — jak na talerzu ze spaghetti: poruszysz coś w jednym miejscu i natychmiast coś się rusza z drugiej strony — *przyp. tłum.*

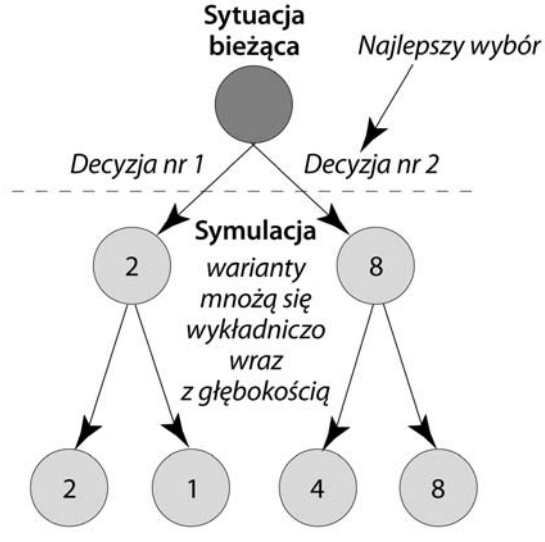
Reguły nie są skalowalne ze względu na liczbę BN. Kontrolowanie zachowania jednego tylko BN za pomocą zbioru reguł już jest sprawą trudną, a przy zwiększaniu liczby BN trudność ta (i zarazem kruchość reguł) rośnie w tempie wykładniczym. Sprawy komplikują się dodatkowo, gdy postaci te działają w korelacji lub w opozycji do siebie.

Wziąwszy pod uwagę powyższe okoliczności, związane — było nie było — z wciąż dominującym podejściem do projektowania gier, trudno się dziwić, iż zastosowanie AI w tej dziedzinie nie dorównuje tempem rozwoju grafice i symulacji zjawisk fizycznych. Staje się coraz bardziej oczywiste, że implementowanie AI w postaci zbiorów reguł sięgnęło już kresu swych możliwości.

Modelowane podejście do implementacji AI

Zgodnie z racjonalnym modelem teorii decyzji, najlepszą decyzją jest ta, która maksymalizuje zysk przewidywany w oparciu o analizę sekwencji przyszłych decyzji. Na rysunku 3.1.4 zilustrowany został dylemat: dla każdej z dwóch alternatyw możliwy jest kolejny wybór między dwiema możliwościami, a dla każdego z czterech wynikających stąd wariantów prognozowane są określone korzyści. Każdy z węzłów widocznego drzewa reprezentuje określoną sytuację (lub stan), zaś krawędzie łączące węzły reprezentują decyzje skutkujące zmianą sytuacji (stanu). Jak łatwo zauważyć, pierwszym krokiem na ścieżce do wariantu najkorzystniejszego jest podjęcie decyzji nr 2.

Rysunek 3.1.4. *Dylemat i mierzalne konsekwencje jego rozstrzygnięcia*



Opisane podejście implementowania AI w odniesieniu do tworzenia aplikacji, szczególnie gier, nazywane jest podejściem *modelowanym*, bowiem opiera się ono o „wysopoziomową dynamikę" systemu — zbiór dopuszczalnych akcji (decyzji) i konsekwencje ich podejmowania. Ogólnie rzecz biorąc, podejście to realizowane jest w trzech następujących krokach.

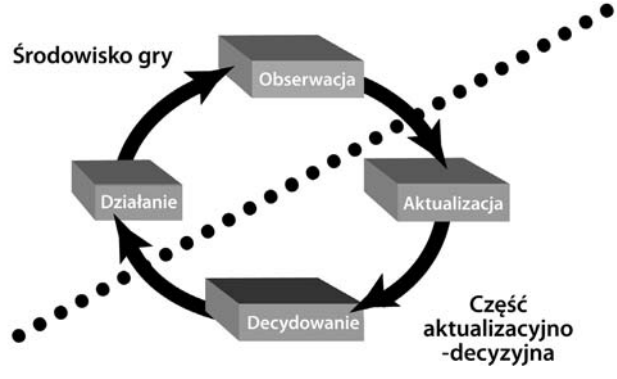
1. **Symulowanie** konsekwencji podejmowania każdej z możliwych decyzji i budowanie na tej podstawie drzewa decyzyjnego.
2. **Szacowanie** spodziewanych zysków w wyniku podejmowania każdej decyzji.
3. **Wybór** tej decyzji, która prowadzi do najkorzystniejszego z szacowanych wariantów.

W klasycznych grach „planszowych", takich jak szachy czy warcaby, na model postępowania składa się opis każdego z (potencjalnie ogromnej liczby) dopuszczalnych ruchów, kryterium zakończenia gry oraz zasady orzekania o zwycięstwie (albo remisie). W grach wideo analogiczny model zawiera opis wszystkich dopuszczalnych akcji, ich efektów i innych zdarzeń, jakie (w symulowanym świecie) mogą być konsekwencjami podejmowania tychże akcji. Zauważmy, że *reguły* gry są czymś innym niż *podejście oparte na regułach*: to ostatnie opiera się na *definiowaniu a priori* akcji podejmowanych w konkretnych sytuacjach, zamiast akcje te *opisywać*, jak ma to miejsce w podejściu modelowanym.

Interfejs

Szczegóły interfejsu łączącego implementację modelowanego podejmowania decyzji z resztą aplikacji (gry) bywają zróżnicowane, lecz zasady współdziałania tych dwóch elementów pozostają z grubsza takie same. Przykładowo w silniku Locust interfejs ten opiera się na czterech procesach: obserwacji, aktualizacji, decydowaniu i działaniu (patrz rysunek 3.1.5). Obserwowanie to zbieranie informacji o otaczającym świecie; aktualizacja polega na wykorzystywaniu wyników obserwacji do zmiany przewidywań, decydowanie to wybór określonej akcji na podstawie tychże przewidywań, zaś działanie to fizyczne realizowanie wybranej akcji w celu zmiany otaczającego środowiska. Zakłada się, że maszynie decyzyjnej znany jest bieżący stan tego środowiska, np. lokalizacje wszystkich BN; każda zmiana stanu środowiska odnotowywana jest przez proces aktualizacji.

Rysunek 3.1.5.
Interfejs łączący grę z maszyną decyzyjną



Przedstawiony na rysunku 3.1.5 cykl „obserwacja→aktualizacja→decydowanie→działanie" kręci się w nieskończoność. Kropkowana linia symbolizuje wspomniany interfejs oddzielający silnik Locust od świata gry, w której odbywa się obserwowanie i działanie; aktualizacja i podejmowanie decyzji są więc od tego świata oddzielone. Faktycznie jednak struktura procesu aktualizacyjnego jest przybliżeniem świata gry i jest od złożoności tego świata ściśle zależna.

Jednym z najważniejszych aspektów opisywanego modelu jest podejmowanie decyzji w czasie rzeczywistym. W skomplikowanym świecie rzadko daje się w całości realizować zakładane scenariusze, bowiem świat ten jest z natury nieprzewidywalny. Przykładowo ścieżka wyznaczona dla określonego BN, jako najkrótsza wiodąca do celu, może okazać się w pewnym momencie nieaktualna wobec obstrukcyjnych działań innych BN.

Locust uwzględnia uwarunkowania tego rodzaju, bowiem podejmowanie decyzji przeplata się w nim z obserwowaniem środowiska i jest stosownie do wyników obserwacji zmieniane. Pozwala to na dynamiczne modyfikowanie przyjętego scenariusza akcji. Co prawda, nie jest wykluczone planowanie scenariuszy niemodyfikowalnych, te jednak okazują się zwykle nierealizowalne ze względu na wspomnianą nieprzewidywalność otaczającego świata. Zamiast tego część aktualizacyjno-decyzyjna wykorzystuje sparametryzowany model przewidywań, którego parametry zmieniane są stosownie do wyników obserwacji.

Zalety i korzyści dla projektantów

Modelowane podejście do implementowania AI — przynajmniej w wersji obecnej w Locust Engine — oferuje projektantom kilka zalet w porównaniu z podejściem opartym na zbiorach reguł. Oto one.

- **Skutkuje bardziej inteligentnym zachowaniem postaci**. Dzieje się tak dlatego, że w stosunku do każdej podejmowanej decyzji analizowane są jej spodziewane konsekwencje. Zachowanie BN wspomaganych przez Locust Engine podporządkowane jest logicznie spójnym celom, a ich działania weryfikowane są co chwila pod kątem zgodności z owymi celami. Schemat ten w naturalny sposób godzi ze sobą długoterminowe plany strategiczne i doraźne decyzje o charakterze taktycznym poprzez zmianę priorytetów wspomnianych celów, stosownie do zmian zachodzących w otaczającym środowisku.
- **Redukuje czas i koszty implementacji**. Zrealizowanie gry w założonym terminie z zachowaniem przewidzianego budżetu jest oczywistym dążeniem projektantów i programistów. Programiści i projektanci doskonale jednak wiedzą, że programowanie sztucznej inteligencji może załamać harmonogram: co więcej, mimo przekroczenia terminu, zachowanie się postaci jest niedopuszczalnie mało inteligentne i wówczas dąży się jedynie do tego, by stało się ono (powiedzmy) akceptowalne przez graczy. Locust umożliwia szybką i mało kosztowną realizację modelu i to bez wymagania szczególnie głębokiej wiedzy projektantów. Konieczny jest jedynie precyzyjny opis akcji i ich efektów oraz powiązanie wyników obserwacji ze stanami modelu.
- **Wprowadza niekiedy niespodziewane efekty do zachowania postaci**. Postaci mogą wykazywać przejawy inteligencji nieprzewidziane oryginalnie przez projektantów. Słynny Deep Blue wykazał się umiejętnością gry w szachy na poziomie mistrzowskim, pokonując samego mistrza świata Garrego Kasparowa; o tak zaawansowane umiejętności trudno jednak podejrzewać twórców programu rozgrywającego. To jeden z dowodów na to, że inteligencja programu nie jest limitowana inteligencją jego twórców, zatem zachowanie się BN nie jest

ograniczone inteligencją autorów gry. Fakt ten pozwala programistom zająć się „kreatywnymi" aspektami BN — ich osobowościami, uzbrojeniem, czujnikami i celami — zamiast drobiazgowego troszczenia się o reguły.

- **Stwarza większy komfort użytkownikom gier**. Bardziej inteligentne zachowanie postaci to większe poczucie (iluzja) realizmu u gracza. Szczególnie cenne są te aspekty zachowania BN, które gracz identyfikuje z zachowaniem typowym dla siebie.
- **Jest bardziej niezawodne**. Przewidywanie prowadzi do większej niezawodności niż sztywne reguły, ma bowiem charakter adaptacyjny — wymusza dostosowanie do bieżącej sytuacji, pozostając jednocześnie techniką ogólnego przeznaczenia. W procesie podejmowania decyzji Locust bierze pod uwagę wiele czynników, takich jak kondycja poszczególnych BN, stan ich uzbrojenia i amunicji, bliskość źródeł zaopatrzenia („doładowania"), kondycję i odległość wroga, itp.; na ich podstawie podejmuje decyzję najlepszą w ramach narzuconych ograniczeń czasowych — im więcej czasu na podjęcie decyzji, tym więcej implikacji podjęcia danej decyzji można wziąć pod uwagę. Locust także znakomicie radzi sobie ze skomplikowaną topografią terenu, automatycznie tworząc hierarchiczne struktury punktów orientacyjnych.
- **Jest skalowalne pod względem liczby BN**. Rozproszenie procesów podejmowania decyzji redukuje złożoność obliczeniową, wykładniczą względem liczby BN w przypadku tradycyjnego podejścia opartego za zdeterminowanym zbiorze reguł. Obliczenia te mogą być także rozproszone między kilka procesorów lub kilka komputerów w sieci
- **Ułatwia debugowanie i modyfikowanie reguł**. Gdy okaże się, że zachowanie postaci ma być uzależnione od jakiegoś nowego czynnika, można ów czynnik uwzględnić w stosunkowo prosty sposób. Przy podejściu tradycyjnym, gdy czynnikiem tym ma być np. charakter (uosobienie) BN, można dla każdego BN ustalić różną hierarchię celów — i tak, nieśmiały i bojaźliwy BN szczególnie cenić sobie będzie możliwość częstego uzupełniania zapasów, podczas gdy śmiały i odważny dążył będzie do częstych starć z wrogiem, mniej obsesyjnie ceniąc sobie swoje bezpieczeństwo. Leniwy BN może sztucznie komplikować wszelkie ruchy lub wykonywać je bardziej ociężale, co pozwoli mu pozostawać jak najbliżej (bezpiecznej) pozycji początkowej. Podejście przyjęte w konstrukcji Locust Engine jest znacznie prostsze i prawie bezkontekstowe, co pozwala łatwiej i szybciej wynajdywać błędy oraz bardziej bezpiecznie modyfikować kod. Zmiana lub wzbogacenie zachowania BN jest kwestią zmiany priorytetów celów i innych parametrów symulacji; unika się w ten sposób produkowania „kodu spaghetti", typowego dla często modyfikowanych zbiorów reguł. Krotko mówiąc, gdy zmienia się model, zmieniają się podejmowane na jego bazie decyzje — samoistnie, bez żadnych dodatkowych zabiegów.
- **Umożliwia kreowanie przenośnych postaci**. Gracze lubią odnajdywać podobieństwa między grami, szczególnie ceniąc sobie podobieństwo postaci. Model zachowania się konkretnego BN w określonej grze może być łatwo zaimplementowany na gruncie innej gry. Locust Engine umożliwia nawet tworzenie całych bibliotek postaci w oparciu o opis ich zachowania zamiast jego fizycznej realizacji.

- **Stwarza okazję do większego urozmaicenia**. Jak już wspominaliśmy, sztuczna inteligencja jest najważniejszym z czynników decydujących o zróżnicowaniu gier. Zdeterminowane zbiory reguł z natury nie ułatwiają takiego różnicowania, w przeciwieństwie do podejścia modelowanego, szczególnie tego zastosowanego w Locust Engine. To ostatnie stwarza szerokie możliwości pod względem animowania postaci, zarówno wtedy, gdy mała liczba autonomicznych BN wykonuje jakąś „wysokopoziomową" akcję przez długi okres czasu, jak i wtedy, gdy duża liczba prostych postaci wspólnie przejmuje kontrolę nad akcją gry przez krótką chwilę. Możliwe jest także sterowanie „niskopoziomowym" przemieszczaniem się postaci — symulowanie tłoku i tłumu to przykłady prostszych funkcji Locust Engine, w dodatku całkowicie wbudowanych w specyfikacje celów, a nie sterowanych określonymi regułami. W efekcie zatłoczenie wydaje się spontaniczne, a nie zaprogramowane.

Podejście modelowane jest jednak gorsze pod dwoma względami w stosunku do sztywnych zbiorów reguł. Po pierwsze, wymaga zbudowania modelu — ten jednak tworzy się łatwiej niż zbiór reguł, poza tym można go zbudować na podstawie interakcji między postaciami. Po drugie, tworzenie i utrzymywanie drzewa reprezentującego przewidywane konsekwencje jest bardziej kosztowne obliczeniowo od przetwarzania zbioru reguł; w większości skomplikowanych aplikacji konsekwencje podjęcia optymalnej decyzji warte są jednak poświęcenia dodatkowego czasu. Locust Engine potrafi podejmować decyzje dotyczące wielu BN w ciągu kilku milisekund, co umożliwia reagowanie w czasie rzeczywistym na zachodzące zmiany, przy jednoczesnym eksponowaniu inteligentnego zachowania.

Locust AI Engine w Quake III Arena

Opiszemy teraz logiczne założenia *Quake III Arena* (*Q3A*) — wieloosobowej „strzelanki" FPP, produkcji Id Software. Jak za chwilę zobaczymy, podejmowane w czasie rzeczywistym decyzje w związku z przebiegiem gry są skomplikowane. Masz zaledwie ułamek sekundy, by reagować na ataki zawziętych BN, nieustannie próbujących Cię zabić; Ty masz do wyboru broń — różne rodzaje, każdy przydatny w innej sytuacji — ucieczkę i ukrywanie się.

Gracze przemieszczają się w ramach mapy, zwanej *areną*, a ich celem jest (wirtualne) zabijanie (*frag*[4] w terminologii *Quake*) wojowników wroga i zbieranie za to kolejnych punktów, stosownie do trybu gry. Jeśli „licznik życia" postaci, w którą wciela się gracz, osiągnie zero, postać ta umiera, lecz natychmiast, w określonym miejscu areny, pojawia się jej sobowtór (w terminologii *Quake* nazywa się to *respawn*); niestety, „dorobek" zmarłej postaci bezpowrotnie przepada. Gra kończy się w przypadku osiągnięcia przez któregoś z graczy (lub przez zespół) określonej liczby punktów lub upłynięcia limitu czasu.

System uzbrojenia został tak zaprojektowany, że dla każdej sytuacji istnieje odpowiednia broń; jedną z najważniejszych decyzji jest właśnie wybór właściwej broni. Broń także (podobnie jak postać reprezentująca gracza) odtwarza się w regularnych odstępach czasu

[4] Na czym polega wirtualny charakter owego zabijania albo czym różni się zabijanie od „fragowania" — o tym można przeczytać np. w Wikipedii (*http://pl.wikipedia.org/wiki/Frag*) — *przyp. tłum.*

— w określonych lokalizacjach, wystarczy tylko schylić się po nią. Każdy egzemplarz broni wyposażony jest w odpowiedni zasób amunicji. Skrzynki z dodatkową amunicją, niektóre ukryte, rozmieszczone są w określonych punktach areny.

Gdy „licznik życia" gracza zbliża się do zera, można go doładować w jednej z czterech kategorii: żółtej (25), pomarańczowej (50), zielonej (5, lecz może przekroczyć 100) i niebieskiej (100 i więcej). Łączna wartość doładowania nie może przekroczyć 200.

Poruszanie się postaci jest w *Q3A* nieskomplikowane: mogą one biegać, spacerować, skakać lub poruszać się w przysiadzie. Bieg jest ruchem domyślnym, choć bieganie po arenie nie jest specjalnie atrakcyjne; spacer (chód) jest bezgłośny, można więc skradać się od tyłu do przeciwnika. Mimo iż podskoki w przysiadzie też są bezgłośne, z natury rzeczy postać porusza się wtedy wolniej, ale za to jest trudniej zauważalna i może przemieszczać się pod niskimi konstrukcjami.

Silnik Locust kontroluje w *QA3* zachowanie wszystkich BN — zachowanie żadnej nie jest określone przez deterministyczne reguły. Model zachowania przewiduje następujące akcje: ruch do przodu pod określonym kątem, na określony dystans, ruch do tyłu pod określonym kątem, na określony dystans, kontynuowanie poprzedniej akcji, użycie aktualnie posiadanej broni oraz bezczynność. Co do przewidywań skutków podejmowanych decyzji, to po licznych eksperymentach doszliśmy do wniosku, że najlepsze wyniki daje analiza skutków decyzji od trzech do ośmiu poziomów w głąb, przy czym powyżej piątego poziomu optymalizowanie wydajności staje się coraz mniej skuteczne. Locust analizuje zawartość 20 – 30 ramek (pełnoekranowych) na sekundę, dokonując, po każdej aktualizacji ramki, prognozy zachowania dwóch-trzech BN; podjęcie decyzji o akcji wykonywanej przez jednego BN trwa nie dłużej niż 13 milisekund.

W oryginalnej wersji *QA3* opracowanie kompletnego zestawu reguł zajęło pół roku, a implementacja tego zestawu liczyła 50 000 wierszy kodu. Gdy zrezygnowaliśmy z deterministycznych reguł na rzecz silnika Locust, uporaliśmy się z implementacją modelu w ciągu dwóch tygodni, a związany z tym kod liczył 2 000 wierszy. Jak widać, kolosalna oszczędność czasu i zasobów. Najważniejsze jest jednak to, że w nowej wersji BN wykazują zdecydowanie większą inteligencję, podejmując bardziej racjonalne decyzje i stając się tym samym bardziej przebiegłymi przeciwnikami. To zaś przekłada się w pierwszym rzędzie na większe zadowolenie graczy.

Soar

Soar ([Laird00] i [LairdDuchi00]) jest ściśle powiązany z tematyką i duchem niniejszego artykułu. Jest on rozwiązaniem unikalnym w tym sensie, że decyzje podejmowane są na bazie reguł, lecz jednocześnie stosowane jest wnioskowanie wsteczne (ang. *backward chaining*) i tworzenie podcelów (ang. *subgoaling*) w celu zapobiegania niefortunnym konsekwencjom tych decyzji i wyboru między konkurencyjnymi regułami. Przykładowo zachowanie postaci sterowane jest za pomocą 715 reguł, 100 operatorów zmiany stanu wewnętrznego i 20 podstanów. Jako że Soar jest reprezentantem tzw. architektury poznawczej (ang. *cognitive architecture*), zawiera także modele ludzkiej pamięci, takie jak ograniczona pamięć operacyjna (ang. *working memory*), przetwarzanie symboli czy porcjowanie (ang. *chunking*). Opisywane przez nas podejście jest jednak ukierunkowane bardziej na uzyskanie odpowiedniej efektywności niż na modelowanie procesów poznawczych.

Konkluzja

Wizja przyszłości to tworzenie gier, w których poziom (sztucznej) inteligencji dorównuje wspaniałym efektom graficznym. Badania skłoniły nas do stwierdzenia, że powszechnie praktykowane implementowanie AI na bazie zbioru reguł deterministycznych nie jest drogą do tego celu. Wiedzieli już o tym w 1980 roku twórcy programu szachowego, który dziesięć lat później pokonał mistrza świata Garrego Kasparowa: zarzucili oni deterministyczne reguły na rzecz podejścia modelowanego. Uświadamia sobie ten fakt także coraz więcej twórców gier. W niniejszym artykule opisaliśmy modelowane podejście do procesu podejmowania decyzji na przykładzie silnika Locust Engine. Daje on nie tylko możliwość drastycznych redukcji kosztów i czasu implementacji AI, lecz także lepsze efekty końcowe (bardziej inteligentne zachowanie postaci), w porównaniu z przetwarzaniem zbiorów reguł.

Literatura cytowana

[Laird00] J. Laird, „It Knows What You're Going to Do: Adding Aticipation to a Quakebot". AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertaiment. AAAI Technical Report SS-00-02.

[LairdDuchi00] J. Laird i J. Duchi, „Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot". AAAI 2000 Fall Symposium Series: Simulating Human Agents.

[Rabin02] Steve Rabin, „Implementing a State Machine Language". *AI Programming Wisdom*, Charles River Media, 2002, str. 314 – 320.