ZARZĄDZANIE WYKONANIEM

W grze dostępne są tylko ograniczone zasoby procesora. Tradycyjnie większość z nich była wykorzystywana do tworzenia świetnej grafiki: główna siła napędowa gier na rynek masowy. Większość ciężkiego przetwarzania graficznego jest teraz wykonywana na GPU, ale nie w całości. Szybsze procesory pozwoliły również na stały wzrost budżetu procesora przeznaczonego dla twórców sztucznej inteligencji, co oznacza, że techniki zbyt kosztowne w pewnym momencie mogą być teraz wdrażane na nawet skromnym sprzęcie mobilnym. Nie jest rzadkością, że sztuczna inteligencja ma więcej niż 50% czasu procesora, chociaż 5 do 25% jest bardziej powszechnym zakresem. Nawet przy dłuższym dostępnym czasie wykonywania, czas procesora może łatwo zostać pochłonięty przez odnajdywanie ścieżek, złożone podejmowanie decyzji i analizę taktyczną. Sztuczna inteligencja jest również z natury niespójna. Czasem potrzeba dużo czasu na podjęcie decyzji (np. zaplanowanie trasy), a czasem wystarczy niewielki budżet (poruszanie się po trasie). Wszystkie twoje postacie mogą potrzebować pathfindingu w tym samym czasie, a wtedy możesz mieć setki klatek, w których nic się nie dzieje z AI. Dobry system sztucznej inteligencji wymaga udogodnień, które pozwolą najlepiej wykorzystać ograniczony czas przetwarzania. Są na to trzy główne elementy: podzielenie czasu wykonania między sztuczną inteligencję, która tego potrzebuje, posiadanie algorytmów, które mogą działać po trochu w kilku klatkach, oraz, gdy zasoby są ograniczone, preferencyjne traktowanie ważnych postaci. W tym rozdziale przyjrzymy się tym kwestiom zarządzania wydajnością, aby stworzyć kompleksowe narzędzie do planowania SI. Rozwiązanie jest motywowane przez sztuczną inteligencję, a bez złożonej sztucznej inteligencji rzadko jest potrzebne. Ale programiści z dobrym systemem planowania AI zwykle używają go również do wielu innych celów. Widziałem szereg aplikacji dla dobrej infrastruktury harmonogramowania: przyrostowe ładowanie nowych obszarów poziomu, zarządzanie teksturami, logikę gry, harmonogramowanie dźwięku i aktualizacje fizyki kontrolowane przez systemy planowania pierwotnie zaprojektowane z myślą o sztucznej inteligencji.

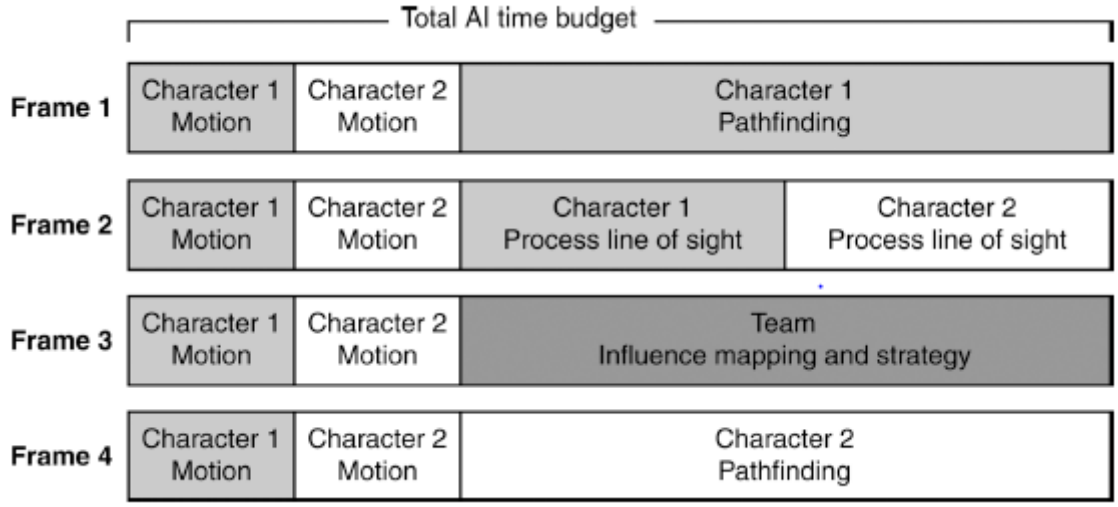
PLANOWANIE

Wiele elementów gry zmienia się błyskawicznie i trzeba je przetwarzać w każdej klatce. Znaki na ekranie są zwykle animowane, co wymaga aktualizacji geometrii w celu wyświetlenia każdej klatki. Położenie i ruch obiektów na świecie są przetwarzane przez system fizyki. Wymaga to częstych aktualizacji, aby prawidłowo przenosić obiekty w przestrzeni oraz aby odpowiednio odbijały się i wchodziły w interakcje. Aby rozgrywka była płynna, dane wejściowe użytkownika muszą być szybko przetwarzane, a informacje zwrotne dostarczane na ekranie. Natomiast AI kontrolująca niektóre postacie zmienia się znacznie rzadziej. Jeśli jednostka wojskowa porusza się po całej mapie gry, jej trasę można obliczyć raz, a następnie przebyć trasę aż do osiągnięcia celu. Podczas walki powietrznej samolot AI może być zmuszony do wykonywania skomplikowanych obliczeń ruchu w każdej klatce, aby pozostać w kontakcie ze swoją ofiarą. Ale kiedy samolot zdecyduje, kogo lecieć, nie musi tak często myśleć taktycznie i strategicznie. System planowania zarządza, które zadania i kiedy mają zostać uruchomione. Radzi sobie z różną częstotliwością wykonywania i różnym czasem trwania zadań. Powinno to pomóc w wygładzeniu profilu wykonywania gry, aby nie Wydajność owały duże szczyty przetwarzania. System planowania w tej sekcji będzie wystarczająco ogólny dla większości aplikacji do gier, AI i innych. Kluczową cechą przy projektowaniu harmonogramu jest szybkość. Nie chcemy spędzać dużo czasu na przetwarzaniu kodu harmonogramu, zwłaszcza, że jest on stale uruchamiany, wykonując dziesiątki, jeśli nie setki, a nawet tysiące zadań zarządzania w każdej klatce.

HARMONOGRAM

Harmonogramy działają, przydzielając pulę czasu wykonania do różnych zadań, w oparciu o to, które z nich potrzebują czasu. Różne zadania AI mogą i powinny być wykonywane z różną częstotliwością.

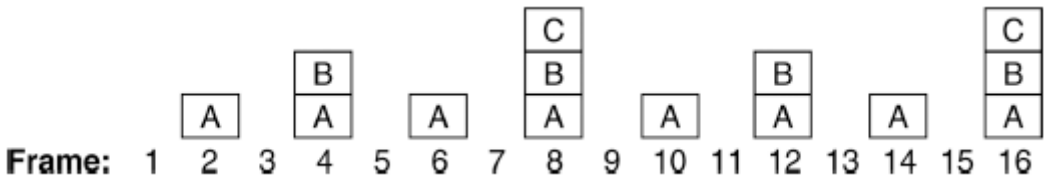
Możemy po prostu zaplanować, aby niektóre zadania były uruchamiane co kilka klatek, a inne, aby były uruchamiane częściej. Sztuczna inteligencja jest dzielona i rozłożona na wiele ramek. Jest to potężna technika zapewniająca, że gra nie zajmuje zbyt dużo czasu AI i że bardziej złożone zadania mogą być wykonywane rzadko. Pokazano to schematycznie na rysunku



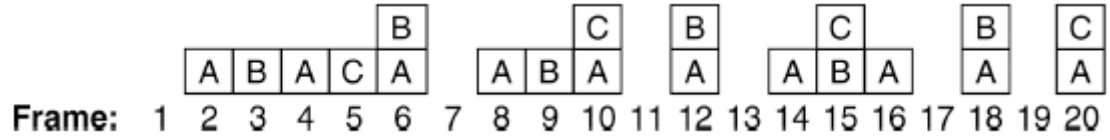
Jest to zgodne z tym, czego zwykle oczekiwalibyśmy od inteligentnych postaci. Prawdziwi ludzie przez cały czas podejmują proste decyzje w ułamku sekundy, takie jak podstawowa kontrola ruchu. Przetwarzanie informacji sensorycznych zajmuje nam trochę więcej czasu (na przykład reakcja na nadlatujący pocisk), ale przetwarzanie to trwa trochę dłużej. Podobnie rzadko podejmujemy decyzje taktyczne i strategiczne na dużą skalę: najwyżej co kilka sekund. Te decyzje na dużą skalę są zazwyczaj najbardziej czasochłonne. Kiedy jest wiele postaci, każda z własną sztuczną inteligencją, możemy użyć tej samej techniki krojenia, aby wykonać tylko kilka znaków na każdej klatce. Jeśli 100 znaków ma aktualizować swój stan co 30 klatek (raz na sekundę), to możemy przetworzyć 3 lub 4 znaki na każdą ramkę.

Częstotliwości

Harmonogram przyjmuje zadania, z których każde ma powiązaną częstotliwość, która określa, kiedy należy je uruchomić. W każdym przedziale czasowym harmonogram jest wywoływany, aby zarządzać całym budżetem AI. Decyduje, jakie zachowania należy uruchomić i wywołuje je. Odbywa się to poprzez zliczanie liczby przesłanych ramek. Wartość ta jest zwiększana za każdym razem, gdy wywoływany jest program planujący. Łatwo jest przetestować, czy każde zachowanie powinno być uruchamiane, sprawdzając, czy liczba klatek jest równomiernie podzielna przez częstotliwość. Operacja dzielenia modulo na liczbach całkowitych (%) jest wprawdzie jedną z najwolniejszych operacji matematycznych na liczbach całkowitych na większości sprzętu, ale ponieważ jest dostarczana na sprzęcie i ponieważ potrzebujemy ich stosunkowo niewiele, będzie wystarczająco szybka do tego zastosowania. To podejście samo w sobie cierpi na zlepianie się: niektóre ramki bez uruchomionych zadań i inne ramki z kilkoma zadaniami dzielącymi budżet. Na rysunku widzimy z tym problem



Istnieją trzy zachowania o częstotliwościach 2, 4 i 8. Za każdym razem, gdy pojawia się zachowanie B, A zawsze działa. Podobnie za każdym razem, gdy działa zachowanie C, uruchamiane są zarówno B, jak i A. Jeśli celem jest rozłożenie obciążenia, to jest to słabe rozwiązanie. W tym przypadku częstotliwości kolidują ze sobą, ponieważ mają wspólny dzielnik (dzielnik to liczba, którą można podzielić na inną całą liczbę razy). Tak więc 1, 2 i 3 są jedynymi dzielnikami 6. Wspólny dzielnik to taki, który dzieli się na zbiór liczb. Tak więc 8 i 12 mają trzy wspólne dzielniki: 1, 2 i 4. Wszystkie liczby mają 1 jako dzielnik, ale tutaj nie ma to znaczenia. To wyższe liczby powodują problemy. Pierwszym krokiem do rozwiązania problemu jest próba wybrania częstotliwości, które są względnie pierwsze: te, które nie mają liczby, która dzieli się na wszystkie (oprócz 1, oczywiście). Na rysunku 10.3 uczyniliśmy oba zachowania B i C częstszymi, ale mamy mniej problemów ze starciem, ponieważ są one względnie pierwsze.



Faza

Jednak nawet stosunkowo pierwsze częstotliwości nadal się ze sobą kolidują. Przykład pokazuje trzy zachowania o częstotliwościach 2, 3 i 5. Co 6 klatek, zachowania A i B są sprzeczne, a co 30 klatek, wszystkie są sprzeczne. Uczynienie częstotliwości względnie pierwszorzędnymi sprawia, że punkty kolizji są rzadsze, ale ich nie eliminuje. Aby rozwiązać problem, do każdego zachowania dodajemy dodatkowy parametr. Ten parametr, zwany fazą, nie zmienia częstotliwości, ale przesuwa przesunięcie w momencie wywołania zachowania. Wyobraź sobie trzy zachowania, wszystkie z częstotliwością 3. Zgodnie z pierwotnym harmonogramem, wszystkie będą działać w tym samym czasie – co trzy klatki. Gdybyśmy mogli je przesunąć, mogłyby one działać w kolejnych klatkach, więc każda klatka miałaby uruchomione jedno zachowanie, ale wszystkie zachowania byłyby uruchamiane co trzy klatki.

Pseudo kod

Podstawowy harmonogram możemy wdrożyć w następujący sposób:

```
1 class FrequencyScheduler:
2 # The data per behavior to schedule.
3 class BehaviorRecord:
4 thingToRun: function
5 frequency: int
6 phase: int
7
8 # The list of behavior records.
9 behaviors: BehaviorRecord[]
10
11 # The current frame number.
12 frame: int
```

```
13

14 # Adds a behavior to the list.

15 function addBehavior(func: function, frequency: int, phase: int):

16 # Compile and add the record.

17 record = new Record()

18 record.functionToCall = func

19 record.frequency = frequency

20 record.phase = phase

21 behaviors += record

22

23 # Called once per frame.

24 function run():

25 # Increment the frame number.

26 frame += 1

27

28 # Go through each behavior.

29 for behavior in behaviors:

30 # If it is due, run it.

31 if behavior.frequency % (frame + behavior.phase):

32 behavior.thingToRun()
```

Uwagi dotyczące implementacji

Wartość fazy jest dodawana do wartości czasu bezpośrednio przed wykonaniem podziału modularnego. Jest to najskuteczniejszy sposób włączenia fazy. Może wydawać się jaśniejsze sprawdzenie czegoś takiego:

```
time % frequency == phase
```

Dodanie fazy pozwala jednak na użycie wartości fazy większych niż częstotliwość. Jeśli chcesz zaplanować uruchamianie 100 agentów co 10 klatek, możesz wykonać następujące czynności:

```
1 for i in 1..100:

2 behavior[i].frequency = 10

3 behavior[i].phase = i
```

Jest to mniej podatne na błędy; jeśli programista zmieni częstotliwość, ale nie fazę, zachowanie nie przestanie nagle być wykonywane.

Wydajność

Harmonogram to O(1) w pamięci i O(n) w czasie, gdzie n to liczba zarządzanych zachowań.

Dostęp bezpośredni

Ten algorytm jest odpowiedni w sytuacjach, w których Wydajność uje rozsądna liczba zachowań (dziesiątki lub setki) i gdy częstotliwości są dość małe. Sprawdzenie jest konieczne, aby upewnić się, że każde zachowanie musi zostać uruchomione. Może się zdarzyć, że kilka zachowań zawsze działa razem (jak w przykładzie 100 agentów w poprzednich uwagach dotyczących implementacji). W takim przypadku sprawdzanie każdego ze 100 jest prawdopodobnie marnotrawstwem. Jeśli twoja gra ma tylko ustaloną liczbę znaków i wszystkie mają tę samą częstotliwość, możesz po prostu skonfigurować tablicę ze wszystkimi zachowaniami, które będą uruchamiane razem, przechowywanymi na liście w jednym elemencie tablicy. Ze stałą częstotliwością do elementu można uzyskać bezpośredni dostęp, a wszystkie zachowania są uruchamiane. Będzie to wtedy O(m) w czasie, co oznacza liczbę zachowań do uruchomienia.

Pseudo kod

Może to wyglądać następująco

```
1 class DirectAccessFrequencyScheduler:

2 # The data for a set of behaviors with one

3 # frequency.

4 class BehaviorSet:

5 functionLists: function[]

6 frequency: int

7

8 # The multiple sets, one for each frequency needed.

9 sets: BehaviorSet[]

10

11 # The current frame number.

12 frame: int

13

14 # Add a behavior to the list.

15 function addBehavior(func: function, frequency: int, phase: int):

16 # Find the correct set.

17 set: BehaviorSet = sets[frequency]

18 # Add the function to the list.

19 set.functionLists[phase] += func
```

```
20

21 # Called once per frame.

22 function run():

23 # Increment the frame number.

24 frame += 1

25

26 # Go through each frequency set.

27 for set in sets:

28 # Calculate the phase for this frequency.

29 phase: int = set.frequency % frame

30

31 # Run the behaviors in the appropriate location

32 # of the array.

33 for func in functionLists[phase]:

34 func()
```

Struktury danych i interfejsy

Element danych zestawu zawiera instancje BehaviorSet. W oryginalnej implementacji wykorzystałem operację for...in..., aby uzyskać elementy zestawu w dowolnej kolejności. W tej implementacji zestaw jest również używany jako tablica mieszająca, wyszukująca wpis według jego wartości częstotliwości. Jeśli istnieje pełny zestaw częstotliwości do maksimum (np. jeśli maksymalna częstotliwość wynosi 5, a istnieją instancje BehaviorSet dla częstotliwości 4, 3 i 2), wtedy może użyć wyszukiwania tablicowego według częstotliwości, a nie tablicy mieszającej.

Wydajność

Jest to O(fp) w czasie, gdzie f to liczba różnych częstotliwości, a p to liczba zachowań na wartość fazy. Jeśli wszystkie elementy tablicy mają pewną zawartość (tj. wszystkie fazy mają odpowiadające sobie zachowania), to zgodnie z obietnicą będzie to równe O(m). Przechowywanie to O(fFp), gdzie F jest średnią używaną częstotliwością. W przypadku stałej liczby zachowań może to być dobre rozwiązanie, ale jest głodne pamięci i nie zapewnia dobrego wzrostu wydajności, gdy używanych jest wiele różnych częstotliwości i wartości fazowych. W tym przypadku oryginalna implementacja, z pewnego rodzaju hierarchicznym harmonogramowaniem (omówionym w dalszej części sekcji), jest prawdopodobnie optymalna.

Jakość fazy

Obliczenie dobrych wartości faz w celu uniknięcia skoków może być trudne. Nie jest intuicyjnie jasne, czy określony zestaw wartości częstotliwości i fazy doprowadzi do regularnego skoku, czy nie. Naiwnością jest oczekiwać, że deweloper, który integruje komponenty gry, będzie w stanie ustawić optymalne wartości faz. Deweloper generalnie będzie miał jednak lepsze wyobrażenie o tym, jakie powinny być względne częstotliwości. Możemy stworzyć metrykę, która mierzy ilość akumulacji, która

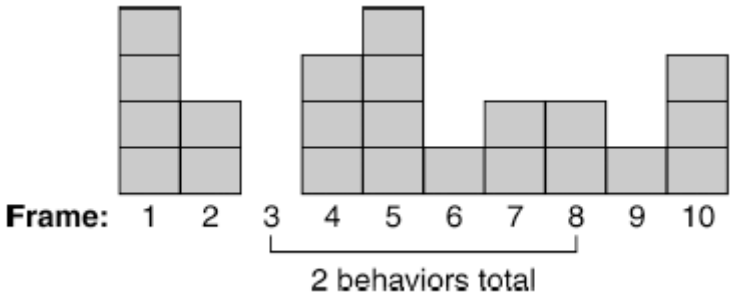
wystąpi w implementacji częstotliwości i fazy. Daje to informację zwrotną dotyczącą oczekiwanej jakości programu planującego. Po prostu próbkujemy dużą liczbę różnych losowych wartości czasu i gromadzimy statystyki dotyczące liczby uruchamianych zachowań. Próbkowanie milionów ramek dla dziesiątek zadań zajmie tylko kilka sekund. Otrzymujemy statystyki minimalne, maksymalne, średnie i dystrybucyjne. Optymalne planowanie będzie miało mały rozkład, z wartościami minimalnymi i maksymalnymi zbliżonymi do średniej.

Automatyczne fazowanie

Nawet przy dobrej jakości sprzężeniu zwrotnym zmiana wartości faz nie jest intuicyjna. Lepiej byłoby przejąć ciężar ustawiania faz od dewelopera. Możliwe jest obliczenie dobrego zestawu faz dla zestawu zadań o różnych częstotliwościach. Dzięki temu program planujący może ujawnić oryginalną implementację, przyjmując częstotliwość tylko dla każdego zadania.

Metoda Wrighta

Wright i Marshall, pierwsi, którzy dogłębnie napisali o harmonogramowaniu gier przez sztuczną inteligencję (chociaż było ono powszechnie używane przez wielu programistów w podobnej formie), dostarczyli prosty i potężny algorytm stopniowania. Kiedy nowe zachowanie jest dodawane do harmonogramu, z częstotliwością f, wykonujemy próbny przebieg harmonogramu dla ustalonej liczby ramek w przyszłości. Zamiast wykonywać zachowania w tym próbnym przebiegu, po prostu liczymy, ile zostałoby wykonanych. Znajdujemy klatkę z najmniejszą liczbą zachowań związanych z bieganiem. Wartość fazy dla zachowania jest ustawiona na liczbę klatek do przodu, przy której Wydajność uje to minimum. Stała liczba klatek to zwykle ręcznie ustawiana liczba, którą można znaleźć eksperymentalnie. W idealnym przypadku byłaby to najmniejsza wspólna wielokrotność (LCM) ze wszystkich wartości częstotliwości używanych w harmonogramie. Zazwyczaj jednak jest to duża liczba i niepotrzebnie spowalnia algorytm (dla częstotliwości 2, 3, 5, 7 i 11 mamy LCM 2310). Rysunek pokazuje to w działaniu.



Zachowanie jest dodawane z częstotliwością 5 klatek. Widzimy, że w ciągu następnych 10 klatek (łącznie z obecną) klatki 3 i 8 mają najmniej połączonych zachowań. Możemy zatem użyć wartości fazy równej 3. To podejście jest doskonałe w praktyce. Ma teoretyczną szansę, że nadal będzie wytwarzał ciężkie kolce, jeśli lookahead nie jest przynajmniej tak duży, jak rozmiar LCM.

Kolce pojedynczego zadania

Używając względnie pierwszych częstotliwości i obliczonych przesunięć fazowych, możesz zminimalizować liczbę klatek, które mają skoki w czasie AI, rozprowadzając ciężką pracę. W większości przypadków takie podejście jest wystarczające do zaplanowania AI i może być bardzo przydatne w przypadku innych elementów gry, które trzeba wykonywać tylko okazjonalnie. Jednak w niektórych przypadkach uruchomienie fragmentu kodu jest tak kosztowne, że samoczynnie spowoduje skok, jeśli

zostanie uruchomiony w ramce. Bardziej zaawansowani planiści muszą umożliwiać uruchamianie procesów w wielu ramkach. Są to procesy przerywalne.

PROCESY PRZERWANIA

Proces przerywalny to taki, który można wstrzymać i wznowić w razie potrzeby. Złożone algorytmy, takie jak odnajdywanie ścieżek, powinny być uruchamiane tylko przez krótki czas w każdej klatce. Po upływie wystarczającego czasu wynik będzie dostępny do użycia, ale nie zakończy się na tej samej klatce, w której się rozpoczął. W przypadku wielu algorytmów całkowity czas, jaki wykorzystuje algorytm, jest zdecydowanie za duży dla jednej klatki, ale w przypadku małych bitów nie zagraża to budżetowi.

Wątki

Istnieje już ogólne narzędzie programistyczne do realizacji dowolnego procesu przerywanego. Wątki są dostępne na wszystkich maszynach do gier (z wyjątkiem niektórych odmian wbudowanych procesorów o tak ograniczonych możliwościach, że i tak byłoby mało prawdopodobne, abyś był w stanie uruchomić złożoną sztuczną inteligencję). Wątki umożliwiają wstrzymywanie fragmentów kodu i powrót do nich w późniejszym czasie. Większość systemów wątków przełącza się między wątkami automatycznie za pomocą mechanizmu zwanego wielozadaniowością z wywłaszczaniem. Jest to mechanizm, w którym kod jest wstrzymywany, niezależnie od tego, co robi. Wszystkie jego ustawienia są zapisywane, a następnie w jego miejsce do procesora ładowany jest kolejny kod. Ta funkcja jest realizowana na poziomie sprzętowym i często jest zarządzana przez system operacyjny. Mogliśmy skorzystać z wątków, umieszczając czasochłonne zadania we własnym wątku. W ten sposób uniknęlibyśmy stosowania specjalnego systemu planowania. Niestety, mimo że jest proste w realizacji, często nie jest to sensowne rozwiązanie. Przełączanie między wątkami polega na rozładowaniu wszystkich danych dla wychodzącego wątku i ponownym załadowaniu wszystkich danych dla nowego wątku. To dodaje dużo czasu. Każdy przełącznik wiąże się z opróżnianiem pamięci podręcznej i wykonywaniem wielu czynności porządkowych. Wielu programistów, słusznie, unikało używania wielu wątków. Chociaż kilkadziesiąt wątków może nie powodować zauważalnych spadków wydajności na komputerze, użycie wątku w grze strategicznej czasu rzeczywistego (RTS) dla algorytmu odnajdywania ścieżki każdej postaci byłoby nadmierne.

Wątki oprogramowania

W przypadku większej liczby jednoczesnych zachowań najczęstszym rozwiązaniem jest ręczny harmonogram. Wymaga to napisania zachowań, aby zwracały kontrolę po przetworzeniu przez krótki czas. Podczas gdy sprzęt może ręcznie włączać i uruchamiać proces wątkowy, planista polega na tym, że zachowuje się ładnie i oddaje kontrolę po krótkim okresie przetwarzania. Ma to tę zaletę, że planista nie musi zarządzać czyszczeniem i porządkowaniem w celu zmiany wątku; zakłada, że zadanie zapisało wszystkie potrzebne dane (i tylko te, których potrzebował) przed zwróceniem kontroli. Takie podejście do planowania jest znane jako „wątki programowe” lub „wątki lekkie” (chociaż to ostatnie jest również używane w znaczeniu mikro-wątków; patrz poniżej), a ogólne podejście nazywa się „wielozadaniowością kooperacyjną”. System harmonogramowania, któremu się przyglądaliśmy do tej pory, radzi sobie z procesami przerywanymi bez modyfikacji. Trudność polega na zapisaniu zachowań do zaplanowania. Zachowania zaplanowane z częstotliwością 1 będą wywoływane w każdej klatce. Jeśli kod jest napisany w taki sposób, że wykonanie nieco większego przetwarzania zajmuje tylko chwilę, a następnie zwraca, ponowne wywołanie w końcu zapewni mu czas na ukończenie.

Mikro-wątki

Chociaż systemy operacyjne obsługują wątki, często dodają dużo dodatkowego przetwarzania i narzutów. To obciążenie pozwala im lepiej zarządzać przełączaniem wątków — w celu śledzenia błędów lub obsługi zaawansowanego zarządzania pamięcią. Ten narzut może być niepotrzebny w grze, a wielu programistów eksperymentowało z pisaniem własnego kodu przełączania wątków, czasami nazywanego mikro-wątkami (lub błędnie lekkimi wątkami). Zmniejszając narzut wątków, można osiągnąć stosunkowo szybką implementację wątków. Jeśli kod w każdym wątku jest świadomy sposobu przełączania wątków, może uniknąć operacji, które ujawniają wykonane skróty. Takie podejście daje bardzo szybki kod, ale może być również niezwykle trudne do debugowania i rozwijania. Chociaż może być odpowiedni do obsługi niewielkiej liczby kluczowych systemów, tworzenie całej gry w ten sposób może być koszmarem. Osobiście zawsze trzymałem się od tego z daleka, ale wiem, że są programiści, którzy nie mają nic przeciwko mieszaniu tego podejścia z niektórymi innymi technikami planowania w tej sekcji.

Hiperwątki i wiele rdzeni

Na komputerach PC i ostatnich generacjach konsol możliwe jest nowe podejście do wątków. Nowoczesne procesory mają kilka oddzielnych potoków przetwarzania, z których każdy działa w tym samym czasie. Najnowsze komputery PC i bieżąca generacja maszyn do gier mają wiele rdzeni: wiele kompletnych procesorów na jednym kawałku krzemu. Podczas normalnej pracy procesor dzieli swoje zadanie wykonania na porcje i wysyła porcję do każdego potoku. Następnie pobiera wyniki i łączy je ze sobą (czasami zdając sobie sprawę, że musi wrócić i zrobić coś ponownie, ponieważ wyniki jednego potoku są sprzeczne z wynikami innego).

Hyper-threading to technologia, w której te potoki mają własny wątek do przetworzenia; różne wątki działają dosłownie w tym samym czasie. Na maszynach wielordzeniowych każdemu procesorowi można nadać własny wątek. Wydaje się jasne, że ta równoległa architektura będzie coraz bardziej wszechobecna na komputerach PC, konsolach i przenośnych urządzeniach do gier. Jest potencjalnie bardzo szybki. Wątki są nadal przełączane w normalny sposób, więc dla dużej liczby wątków nadal nie jest to najbardziej wydajne rozwiązanie.

Jakość usługi

Producenci konsol mają rygorystyczne wymagania, które muszą zostać spełnione, zanim gra zostanie wydana na ich platformę. Liczba klatek na sekundę jest oczywistą oznaką jakości dla graczy, a wszyscy producenci konsol określają, że liczba klatek na sekundę powinna być stała.1 Częstotliwości klatek 30, 60 i (z nadejściem VR) 90 Hz są najbardziej powszechne i wymagają przetwarzania wszystkich gier być zakończone w 33, 16 lub 11 milisekund.

Przy 60 Hz, jeśli całe przetwarzanie zajmuje 16 milisekund, wszystko jest w porządku. Jeśli zostanie to zrobione w 15 milisekund, to też w porządku, ale konsola czeka na dodatkową milisekundę, nie robiąc nic. To czas, który można by wykorzystać, by uczynić grę bardziej imponującą - dodatkowy efekt wizualny, symulację tkaniny czy kilka dodatkowych kości w szkielecie postaci. Z tego powodu budżety czasowe są zwykle przesuwane tak blisko limitu, jak to tylko możliwe. Aby mieć pewność, że liczba klatek na sekundę nie spadnie, bardzo ważne jest, aby ograniczyć czas, w jakim zajmie grafika, fizyka i sztuczna inteligencja. Często bardziej akceptowalne jest posiadanie długoterminowego komponentu niż komponentu, który podlega gwałtownym wahaniom. System planowania, który opisałem do tej pory, oczekuje, że zachowania będą działać przez krótki czas. Ufa, że fluktuacje czasu działania uśrednią wszelkie różnice, aby zapewnić stały czas AI. W wielu przypadkach to po prostu nie wystarczy i konieczna jest większa kontrola. Synchronizacja wątków może być trudna. Jeśli zachowanie jest zawsze przerywane (tj. przez przełączenie wątku), zanim może zwrócić wynik, to jego postać może po prostu stać w miejscu i nic nie robić. Niewielka zmiana w ilości przetwarzania może często powodować tego

rodzaju problem, który jest bardzo trudny do debugowania, a może być jeszcze trudniejszy do naprawienia. Idealnie chcielibyśmy system to pozwala nam kontrolować całkowity czas wykonania, jednocześnie będąc w stanie zagwarantować, że zachowania zostaną uruchomione. Chcielibyśmy również mieć dostęp do statystyk, które pomogą nam zrozumieć, gdzie wykorzystywany jest czas przetwarzania i w jaki sposób zachowania biorą udział w torcie.

HARMONOGRAM RÓWNOWAŻENIA OBCIĄŻENIA

Harmonogram równoważenia obciążenia rozumie czas, jaki ma na uruchomienie, i rozdziela ten czas na zachowania, które muszą zostać uruchomione. Możemy zmienić nasz istniejący program planujący w program do równoważenia obciążenia, dodając proste dane czasowe. Harmonogram dzieli podany czas zgodnie z liczbą zachowań, które muszą być uruchomione w tej ramce. Zachowania, które są wywoływane, otrzymują informacje o czasie, dzięki czemu mogą zdecydować, kiedy przestać działać i powrócić. Ponieważ nadal jest to model wątkowości oprogramowania, nic nie może powstrzymać działania działania tak długo, jak chce. Planista ufa, że będą dobrze się zachowywać. Aby skorygować drobne błędy w czasie działania zachowań, program planujący ponownie oblicza czas pozostały po uruchomieniu każdego zachowania. W ten sposób zachowanie przekroczenia zmniejszy czas, który inni będą uruchamiać w tej samej klatce.

Pseudo kod

```
1 class LoadBalancingScheduler:
2 # The data per behavior to schedule.
3 class BehaviorRecord:
4 thingToRun: function
5 frequency: int
6 phase: int
7
8 # The list of behavior records.
9 behaviors: BehaviorRecord[]
10
11 # The current frame number.
12 frame: int
13
14 # Adds a behavior to the list.
15 function addBehavior(func: function, frequency: int, phase: int):
16 # Compile and add the record.
17 record = new Record()
18 record.functionToCall = func
19 record.frequency = frequency
```

```
20 record.phase = phase
21 behaviors += record
22
23 # Called once per frame.
24 function run(timeToRun: int):
25 frame += 1
26 runThese: BehaviorRecord[] = []
27
28 # Go through each behavior.
29 for behavior in behaviors:
30 # If it is due, schedule it.
31 if behavior.frequency % (frame + behavior.phase):
32 runThese.append(behavior)
33
34 # Keep track of the current time.
35 currentTime: int = time()
36
37 # Go through the behaviors to run.
38 numToRun: int = runThese.length()
39 for i in 0..numToRun:
40 # Find the time used on the previous iteration.
41 lastTime = currentTime
42 currentTime = time()
43 timeUsed = currentTime - lastTime
44
45 # Distribute remaining time to remaining behaviors.
46 timeToRun -= timeUsed
47 availableTime = timeToRun / (numToRun - i)
48
49 # Run the behavior.
50 runThese[i].thingToRun(availableTime)
```

Struktury danych

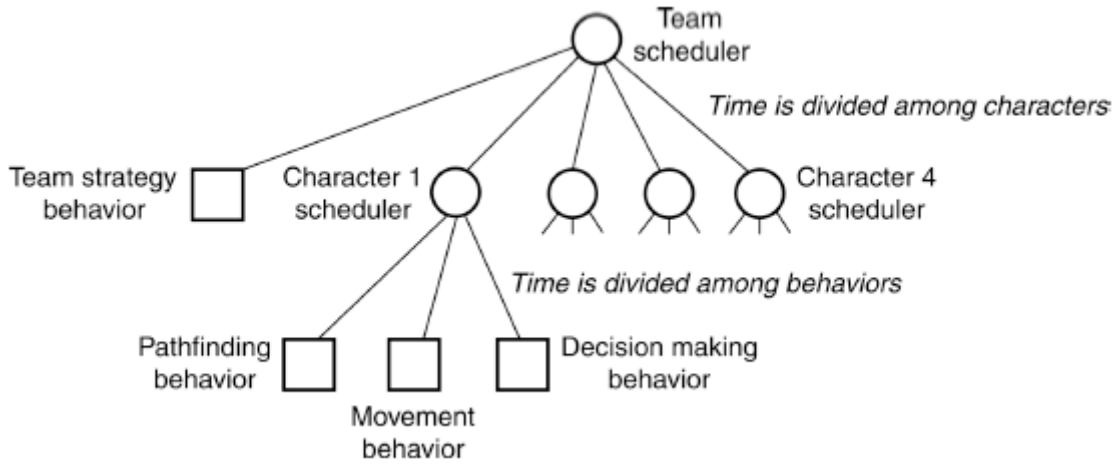
Funkcje, które rejestrujemy, powinny teraz przyjmować wartość czasu, wskazującą maksymalny czas ich działania. Przyjęliśmy, że lista funkcji, które chcemy uruchomić, ma metodę length, która pobiera liczbę elementów.

Wydajność

Algorytm pozostaje O(n) w czasie (n to całkowita liczba zachowań w harmonogramie), ale teraz ma O(m) w pamięci, gdzie m to liczba zachowań, które zostaną uruchomione. Nie możemy połączyć tych dwóch pętli, aby uzyskać pamięć O(1), ponieważ musimy wiedzieć, ile zachowań będziemy wykonywać, zanim będziemy mogli obliczyć dozwolony czas. Te wartości wykluczają czas przetwarzania i pamięć zachowań. Naszym jedynym celem w przypadku tego algorytmu jest to, aby zasoby przetwarzania wykorzystywane przez zaplanowane zachowania były znacznie większe niż te poświęcone na ich planowanie.

HIERARCHICZNY HARMONOGRAM

Chociaż pojedynczy system planowania może kontrolować dowolną liczbę zachowań, często wygodnie jest używać wielu systemów planowania. Postać może mieć wiele różnych zachowań do wykonania - na przykład odnajdywanie trasy, aktualizowanie swojego stanu emocjonalnego i podejmowanie lokalnych decyzji dotyczących sterowania. Byłoby wygodnie, gdybyśmy mogli uruchomić postać jako całość i mieć zaplanowane i przydzielone czasy jej poszczególnych elementów. Wtedy możemy mieć jeden planer najwyższego poziomu, który przydziela czas każdej postaci, a czas jest następnie dzielony zgodnie z kompozycją postaci. Harmonogramowanie hierarchiczne umożliwia uruchamianie systemu planowania jako zachowania przez inny program planujący. Harmonogram można przypisać do uruchamiania wszystkich zachowań dla jednego znaku, tak jak w poprzednim przykładzie. Tak jak na rysunku, inny planista może przydzielić czas na podstawie znaku.



Dzięki temu bardzo łatwo ulepszyć sztuczną inteligencję postaci bez utraty równowagi w czasie całej gry. Przy podejściu hierarchicznym nie ma powodu, dla którego planiści na różnych poziomach powinien być tego samego rodzaju. Możliwe jest użycie harmonogramu opartego na częstotliwości dla całej gry i harmonogramu opartego na priorytetach (opisane później) dla poszczególnych postaci.

Struktury danych i interfejsy

Aby to obsłużyć, możemy przejść od programu planującego wywołującego funkcje do ogólnego interfejsu dla wszystkich zachowań:

```
1 class Behavior:

2 function run(time)
```

Wszystko, co można zaplanować, powinno udostępniać ten interfejs. Jeśli chcemy hierarchicznego planowania, sami planiści również muszą je ujawnić (powyższy planista równoważenia obciążenia ma odpowiednią metodę, po prostu musi jawnie wywodzić się z zachowania). Możemy sprawić, by nasze harmonogramy działały, modyfikując klasę LoadBalancingScheduler w następujący sposób:

```
1 class LoadBalancingScheduler (Behavior):

2 # ... All contents as before ...
```

Ponieważ zachowania są teraz klasami, a nie funkcjami, musimy również zmienić sposób ich wywoływania. Wcześniej używaliśmy wywołania funkcji, a teraz musimy użyć wywołania metody, więc:

```
entry(availableTime)
```

staje się:

```
entry.run(availableTime)
```

w klasie LoadBalancingScheduler.

Wybór zachowania

Samo w sobie nie ma niczego, czego nie zapewnia harmonogramowanie hierarchiczne, czego nie może obsłużyć pojedynczy program planujący. Wychodzi na jaw, gdy jest używany w połączeniu z systemami poziomu szczegółowości, opisanymi w dalszej części. Systemy poziomu szczegółowości to selektory zachowania; wybierają tylko jedno zachowanie do uruchomienia. W strukturze hierarchicznej oznacza to, że planiści uruchamiający całą grę nie muszą wiedzieć, jakie zachowanie działa każda postać. Płaska struktura oznaczałaby usuwanie i rejestrowanie zachowań w głównym harmonogramie za każdym razem, gdy zmienia się wybór. Jest to podatne na błędy w czasie wykonywania, wycieki pamięci i trudne do wyśledzenia błędy.

HARMONOGRAMOWANIE PRIORYTETOWE

Istnieje wiele możliwych udoskonaleń systemu planowania opartego na częstotliwości. Najbardziej oczywistym jest umożliwienie różnym zachowaniom uzyskania innego udziału w dostępnym czasie. Przypisywanie priorytetu każdemu zachowaniu i przydzielanie na tej podstawie czasu jest dobrym podejściem.

W praktyce to nastawienie (zwykle nazywane priorytetem) jest tylko jedną z wielu zasad alokacji czasu, które można wdrożyć. Jeśli pójdziemy trochę dalej z priorytetami, możemy całkowicie wyeliminować potrzebę częstotliwości. Każde zachowanie otrzymuje część czasu AI zgodnie z jego priorytetem.

Pseudo kod

```
1 class PriorityScheduler:

2 # The data per behavior to schedule.

3 class BehaviorRecord:

4 thingToRun: function

5 frequency: int
```

```
6 phase: int
7 priority: float
8
9 # The list of behavior records.
10 behaviors: BehaviorRecord[]
11
12 # The current frame number.
13 frame: int
14
15 # Adds a behavior to the list.
16 function addBehavior(func: function,
17 frequency: int,
18 phase: int,
19 priority: float):
20 # Compile and add the record.
21 record = new Record()
22 record.functionToCall = func
23 record.frequency = frequency
24 record.phase = phase
25 record.priority = priority
26 behaviors += record
27
28 # Called once per frame.
29 function run(timeToRun: int):
30 # Increment the frame number.
31 frame += 1
32
33 # Keep a list of behaviors to run, and their total
34 # priority.
35 runThese: BehaviorRecord[] = []
36 totalPriority: float = 0
```

```
37
38 # Go through each behavior.
39 for behavior in behaviors:
40 # If it is due, schedule it.
41 if behavior.frequency % (frame + behavior.phase):
42 runThese.append(behavior)
43 totalPriority += behavior.priority
44
45 # Keep track of the current time.
46 currentTime: int = time()
47
48 # Go through the behaviors to run.
49 numToRun: int = runThese.length()
50 for i in 0..numToRun:
51 # Find the time used on the previous iteration.
52 lastTime = currentTime
53 currentTime = time()
54 timeUsed = currentTime - lastTime
55
56 # Distribute remaining time to remaining behaviors
57 # based on priority.
58 timeToRun -= timeUsed
59 availableTime =
60 timeToRun * behavior.priority / totalPriority
61
62 # Run the behavior.
63 runThese[i].thingToRun(availableTime)
```

Wydajność

Algorytm ten ma te same cechy, co harmonogram równoważenia obciążenia: O(n) w czasie i O(m) w pamięci, z wyłączeniem czasu przetwarzania i pamięci używanej przez zaplanowane zachowania.

Inne zasady

Jeden harmonogram oparty na priorytetach, z którym pracowałem, nie miał w ogóle danych dotyczących częstotliwości. Używał tylko priorytetów do podziału czasu, a wszystkie zachowania były zaplanowane tak, aby były uruchamiane w każdej klatce. Program planujący zakładał, że każde zachowanie jest przerywalne i będzie kontynuowało przetwarzanie w kolejnej ramce, jeśli się nie zakończy. W tym przypadku uruchomienie wszystkich zachowań, nawet przez krótki czas, miało sens. Alternatywnie moglibyśmy również użyć zasady, w której każde zachowanie prosi o określoną ilość czasu, a harmonogram dzieli dostępny czas, aby zachowania otrzymały to, o co proszą. Jeśli zachowanie prosi o więcej czasu niż jest dostępne, być może będzie musiało poczekać na kolejną ramkę przed otrzymaniem żądania. Zwykle łączy się to z pewnego rodzaju porządkiem pierwszeństwa, więc przy przydzielaniu budżetu preferowane będą zachowania, które są ważniejsze. Alternatywnie moglibyśmy rozłożyć czas zgodnie z nastawieniem, a następnie obliczyć rzeczywisty czas trwania zachowań i zmienić ich nastawienie. Na przykład zachowanie, które zawsze przekracza, może mieć mniej czasu na próbę upewnienia się, że nie uciska innych. Bez wątpienia niebo jest granicą, ale są też kwestie praktyczne. Jeśli twoja gra jest mocno obciążona, znalezienie idealnej strategii podziału czasu może zająć trochę czasu. Nie widziałem złożonej gry, w której sztuczna inteligencja nie skorzystałaby z pewnego rodzaju planowania (z wyjątkiem gier, w których sztuczna inteligencja jest tak prosta, że zawsze uruchamia wszystko w ramce). Mechanizm zwykle wymaga pewnych poprawek.

Problemy priorytetowe

Istnieją subtelne problemy z podejściami opartymi na priorytetach. Niektóre zachowania muszą działać regularnie, podczas gdy inne nie; niektóre zachowania można podzielić na małe odcinki czasu, podczas gdy inne wymagają całego czasu na raz; niektóre zachowania mogą skorzystać na wolnym czasie, podczas gdy inne nie ulegną poprawie. Hybrydowe podejście oparte na harmonogramowaniu priorytetów i częstotliwości może rozwiązać niektóre z tych problemów, ale nie wszystkie. Te same problemy pojawiają się w przypadku programistów sprzętu i systemów operacyjnych, którzy wdrażają wątki. Wątki mogą mieć priorytety, różne zasady alokacji i różne częstotliwości. Poszukaj informacji na temat wdrażania gwintowania, jeśli potrzebujesz naprawdę prostego podejścia do planowania. Planowanie gier musi być znacznie mniej skomplikowane niż system operacyjny, więc większość gier nie wymaga tak dokładnego dostrajania. Proste podejście, takie jak implementacja częstotliwości we wcześniejszej części tej sekcji, jest wystarczająco skuteczne.

ALGORYTMY W DOWOLNYM CZASIE

Problem z algorytmami przerywanymi polega na tym, że ich ukończenie może zająć dużo czasu. Wyobraź sobie postać, która próbuje zaplanować trasę na bardzo dużym poziomie gry. W tempie kilkuset mikrosekund na klatkę może to zająć kilka sekund. Gracz zobaczy, jak postać stoi nieruchomo, nie robiąc nic przez kilka sekund, zanim odejdzie z wielkim celem. Jeśli okienko percepcji nie jest zbyt duże, natychmiast zaalarmuje to gracza, a postać będzie wyglądać na nieinteligentną. Jak na ironię, im bardziej złożone przetwarzanie i im bardziej wyrafinowana sztuczna inteligencja, tym dłużej to zajmie i tym bardziej prawdopodobne jest, że postać będzie wyglądać głupio. Kiedy wykonujemy ten sam proces, często zaczynamy działać, zanim skończymy myśleć. To przeplatanie się działania i myślenia opiera się na naszej zdolności do generowania słabych, ale szybkich rozwiązań i ulepszania ich w czasie, aby uzyskać lepsze rozwiązania. Możemy na przykład ruszyć w przybliżonym kierunku naszego celu. W ciągu kilku sekund początkowego ruchu opracowaliśmy całą trasę. Są szanse, że wstępne przypuszczenie będzie mniej więcej w porządku, więc nic nie będzie nie na miejscu, ale od czasu do czasu przypomnimy sobie coś z kluczyka i będziemy musieli cofnąć się (będziemy w połowie drogi do samochodu i zdamy sobie sprawę, że zapomnieliśmy kluczyków, na przykład). Algorytmy AI, które mają tę samą właściwość, nazywane są „algorytmami w dowolnym momencie”. W każdej chwili możesz poprosić o najlepszy do tej pory pomysł, ale zostaw system, aby działał dłużej, a wynik się poprawi.

Umieszczenie dowolnego algorytmu w naszym istniejącym harmonogramie nie wymaga żadnych modyfikacji. Zachowanie musi być napisane w taki sposób, aby zawsze udostępniało najlepsze odgadnięcie przed zwróceniem kontroli do programu planującego. W ten sposób inne zachowanie może zacząć działać na podstawie domysłu, podczas gdy algorytm w dowolnym momencie udoskonala swoje rozwiązanie. Najczęstszym zastosowaniem algorytmów w dowolnym momencie jest ruch lub odnajdywanie ścieżki. Jest to zwykle najbardziej czasochłonny proces AI. Pewne odmiany popularnych technik odnajdywania ścieżek można łatwo przekształcić w algorytmy w dowolnym momencie. Innymi odpowiednimi kandydatami są turowa sztuczna inteligencja, uczenie się, tłumacze języka skryptowego i analiza taktyczna.

POZIOM SZCZEGÓŁÓW

W Części 2 opisałem okno percepcji: uwagę gracza, która wędruje selektywnie podczas rozgrywki. W dowolnym momencie gracz prawdopodobnie będzie skoncentrowany tylko na małym obszarze poziomu gry. Warto zadbać o to, aby ten obszar wyglądał dobrze i zawierał realistyczne postacie, nawet kosztem reszty poziomu.

GRAFIKA POZIOM SZCZEGÓŁÓW

Algorytmy poziomu szczegółowości (LOD) są stosowane od lat w programowaniu graficznym. Chodzi o to, aby poświęcić najwięcej wysiłku obliczeniowego na obszary gry, które są dla gracza najważniejsze. Z bliska obiekt jest rysowany z większą ilością szczegółów niż z daleka. W większości graficznych technik LOD szczegółowość jest funkcją złożoności geometrycznej liczby wielokątów narysowanych w modelu. Na odległość nawet kilka wielokątów może sprawiać wrażenie obiektu; zbliżenie, ten sam obiekt może wymagać tysięcy wielokątów. Innym powszechnym podejściem jest użycie LOD do szczegółów tekstur. Jest to obsługiwane w sprzęcie lub przynajmniej w każdym renderującym API. Tekstury są mipmapowane; są one przechowywane w wielu LOD, a odległe obiekty używają wersji o niższej rozdzielczości. Oprócz tekstury i geometrii można uprościć inne artefakty wizualne: efekty specjalne i animacje są zwykle zmniejszane lub usuwane w przypadku obiektów znajdujących się na odległość. Poziomy szczegółowości są zwykle oparte na odległości, ale nie tylko. W wielu algorytmach renderowania terenu, na przykład, sylwetki wzgórz w pewnej odległości są rysowane bardziej szczegółowo niż kawałek płaskiego terenu bezpośrednio obok gracza. Odległość to tylko heurystyka, ważna jest zauważalność. Wszystko, co jest bardziej zauważalne dla gracza, wymaga więcej szczegółów. Półkulisty reflektor na starym motocyklu, na przykład, rzuca się w oczy, jeśli jest zbudowany z kilku wielokątów (ludzkie oczy z łatwością wykrywają kąty). Może w końcu stanowić 15% wielokątów w modelu o małej liczbie wielokątów, po prostu dlatego, że nie spodziewamy się zobaczyć narożników obiektu sferycznego. W trzewiach roweru jednak tam, gdzie w rzeczywistości jest więcej szczegółów, możemy użyć mniej wielokątów, ponieważ oko spodziewa się zobaczyć narożniki i brak płynności. Istnieją tutaj dwie ogólne zasady. Po pierwsze, poświęć najwięcej wysiłku na rzeczy, które zostaną zauważone, a po drugie, poświęć wysiłek na te rzeczy, których nie można łatwo przybliżyć.

AI LOD

Algorytmy poziomu szczegółowości w sztucznej inteligencji nie różnią się od tych w grafice: przydzielają czas komputera przed tymi postaciami, które są najważniejsze lub najbardziej wrażliwe na błąd z punktu widzenia gracza. Na przykład samochody znajdujące się w pewnej odległości wzdłuż drogi nie muszą prawidłowo przestrzegać przepisów ruchu drogowego; gracze raczej nie zauważą losowej zmiany pasów. Z bardzo dużej odległości gracze raczej nie zauważą, że wiele samochodów przejeżdża przez siebie. Podobnie, jeśli postać w oddali potrzebuje 10 sekund, aby zdecydować, gdzie się przenieść, będzie to mniej zauważalne, niż gdyby pobliska postać nagle zatrzymała się na ten sam czas. Pomimo tych przykładów, AI LOD nie zależy przede wszystkim od odległości. Możemy obserwować

postać z dystansu i nadal mieć dobry pomysł na to, co robi. Nawet jeśli nie możemy ich oglądać, oczekujemy, że postacie będą cały czas działać. Gdyby sztuczna inteligencja działała tylko wtedy, gdy postać była na ekranie, wyglądałoby to dziwnie, gdy odwrócimy się na chwilę i zawrócimy, aby znaleźć tę samą postać w dokładnie tym samym miejscu, w połowie spaceru. Oprócz odległości musimy wziąć pod uwagę, jak prawdopodobne jest, że gracz obserwowałby postać lub spojrzałby, czy się poruszyła. To zależy od roli, jaką postać odgrywa w grze.

Znaczenie w AI jest często podyktowane fabułą gry. Wiele postaci w grze jest dodawanych dla smaku; nie ma znaczenia, czy zawsze chodzą po mieście według ustalonego schematu, ponieważ tylko niewielu graczy to zauważy. Możesz skończyć z zapalonymi graczami na forach, mówiącymi: „Podążałem za kowalem po mieście, a oni podążają tą samą trasą i nigdy nie kładą się spać ani nie sikają". Ale to nie ma większego znaczenia dla większości graczy i prawdopodobnie nie wpłynie na sprzedaż. Jeśli postać, która jest kluczowa dla fabuły gry, chodzi w kółko po głównym placu, większość graczy to zauważy. Warto, aby postać była nieco bardziej urozmaicona. Oczywiście należy to zrównoważyć z obawami związanymi z rozgrywką. Jeśli dana postać ma ważne informacje do zadania gracza, nie chcemy, aby gracz musiał przeszukiwać całe miasto, aby wyśledzić postać i zadać jeszcze jedno pytanie.

Znaczenie wartości

W tej sekcji zakładam, że ważność to pojedyncza wartość liczbowa, która odnosi się do każdej postaci w grze. Jak widzieliśmy, wiele czynników można połączyć, aby stworzyć wartość ważności. Początkowa implementacja może zwykle wystarczyć na początek odległości, aby upewnić się, że wszystko działa.

PLANOWANIE LOD

Prosty i skuteczny algorytm LOD oparty jest na omówionych wcześniej systemach harmonogramowania. Po prostu użycie częstotliwości planowania opartej na ważności postaci zapewnia system LOD. Ważne postacie mogą otrzymać więcej czasu przetwarzania niż inne, ponieważ są zaplanowane częściej. Jeśli używasz systemu planowania opartego na priorytetach, zarówno częstotliwość, jak i priorytet mogą zależeć od ważności. Ta zależność może być za pomocą funkcji, w której wraz ze wzrostem ważności maleje wartość częstotliwości, lub może być podzielona na kategorie, gdzie zakres wartości ważności daje jedną częstotliwość, a inny zakres mapuje się na inną częstotliwość. Częstotliwości, ponieważ są liczbami całkowitymi, efektywnie wykorzystują to drugie podejście (chociaż jeśli istnieją setki możliwych wartości częstotliwości, bardziej sensowne jest myślenie o tym jako o funkcji). Z drugiej strony priorytety mogą działać w obie strony. W ramach tego schematu postacie zachowują się tak samo, niezależnie od tego, czy ich wartość jest wysoka, czy niska. Skrócony dostępny czas ma różny wpływ na postać w zależności od tego, czy używany jest harmonogram oparty na częstotliwości, czy na priorytetach.

Harmonogramy częstotliwości

W implementacji opartej na częstotliwości mniej ważne postacie rzadziej podejmują decyzje. Na przykład postacie poruszające się po mieście mogą chodzić w linii prostej między wezwaniami do swojej sztucznej inteligencji. Jeśli sztuczna inteligencja jest wywoływana rzadko, mogą przestrzelić swój cel i od czasu do czasu muszą się cofnąć. Ewentualnie mogą nie być w stanie zareagować na czas na zderzenie z innym pieszym.

Harmonogramy priorytetowe

Implementacje oparte na priorytetach dają więcej czasu na ważne zachowania. Wszystkie zachowania mogą być uruchamiane w każdej klatce, ale ważne mogą działać dłużej. Zakładamy, że algorytmy są używane w dowolnym momencie, więc postać może zacząć działać przed zakończeniem przetwarzania

przez AI. Postacie o niskim znaczeniu będą miały tendencję do podejmowania gorszych decyzji niż te o dużym znaczeniu. Postacie powyżej, na przykład, nie przestrzelą swojego celu, ale mogą wybrać dziwaczną trasę do celu, zamiast pozornie oczywistego skrótu (tj. ich algorytm odnajdywania ścieżki może nie mieć czasu, aby uzyskać najlepszy wynik). Ewentualnie, unikając innego przechodnia, zachowanie może nie mieć czasu na sprawdzenie, czy nowa ścieżka jest wolna, powodując zderzenie postaci z kimś innym.

Planowanie połączone

Połączenie częstotliwości i planowania priorytetów może zmniejszyć problemy związane z planowaniem LOD. Harmonogramowanie priorytetów umożliwia częstsze uruchamianie sztucznej inteligencji (zmniejszając blokowanie zachowań, takie jak przeregulowanie), podczas gdy harmonogramowanie częstotliwości pozwala na dłuższe działanie sztucznej inteligencji (zapewniając lepszą jakość decyzji). Nie jest to jednak srebrna kula. W obu przykładach postać mało istotna może częściej kolidować z innymi postaciami. Łączenie podejść nie obejdzie faktu, że unikanie kolizji, niezbędne dla pobliskich postaci, wymaga dużej mocy obliczeniowej. Często lepiej jest całkowicie zmienić zachowanie postaci, gdy jej znaczenie spadnie.

ZADANIE BEHAWIORALNE

Behawioralny LOD pozwala, aby wybór zachowania postaci zależeć od jego ważności. Postać wybiera jedno zachowanie na raz na podstawie jego aktualnego znaczenia. Gdy zmienia się jego znaczenie, zachowanie może zostać zmienione na inne. Celem jest, aby zachowania o mniejszym znaczeniu wymagały mniej zasobów. Z każdą możliwą wartością ważności jest powiązane zachowanie. Na każdym kroku zachowanie jest wybierane na podstawie wartości ważności. Na przykład pieszy w grze RPG może mieć dość złożone wykrywanie kolizji, unikanie przeszkód i kierowanie ścieżką, gdy jest to ważne. Piesi na obrzeżach akcji (na przykład na odległym chodniku lub widziani z mostu) mogą mieć całkowicie wyłączone wykrywanie kolizji. Swobodne przechodzenie między sobą nie jest tak zauważalne, jak można by się spodziewać. Jest to z pewnością mniej zauważalne niż częste kolizje w stylu pinballa. Dzieje się tak, ponieważ nasz aparat optyczny jest dostrojony do wykrywania zmian w ruchu bardziej niż płynnego ruchu.

Przetwarzanie wejścia i wyjścia

Zachowania mają wymagania dotyczące pamięci, a także obciążenie procesora. W przypadku gier z wieloma postaciami (takich jak RPG lub gry strategiczne) niemożliwe jest przechowywanie w pamięci danych wszystkich możliwych zachowań wszystkich postaci jednocześnie. Chcemy, aby mechanizm LOD utrzymywał pamięć oraz czas wykonania na jak najniższym poziomie. Aby umożliwić prawidłowe tworzenie i niszczenie danych, kod jest wykonywany po wejściu do zachowania i po jego zakończeniu. Kod wychodzący może wyczyścić pamięć używaną w poprzednim LOD, a kod wejściowy może poprawnie skonfigurować dane w nowym LOD gotowe do przetworzenia. Aby wesprzeć ten dodatkowy krok, system LOD musi śledzić zachowanie, które uruchomił ostatnim razem. Jeśli zachowanie, które zamierza uruchomić, jest takie samo, nie są potrzebne żadne procesy wejścia ani wyjścia. Jeśli zachowanie jest inne, wywoływana jest procedura wyjścia bieżącego zachowania, a następnie procedura wejścia nowego zachowania.

Kompresja zachowania

Zachowania o niskiej szczegółowości są często przybliżeniem zachowań o dużej szczegółowości. Na przykład system odnajdywania ścieżki może ustąpić miejsca prostemu zachowaniu „szukania”. Informacje przechowywane w zachowaniu o dużej szczegółowości mogą być przydatne dla zachowania

o małej szczegółowości. Aby upewnić się, że sztuczna inteligencja jest wydajna w zakresie pamięci, zwykle wyrzucamy dane związane z zachowaniem, gdy jest ono wyłączone. Na etapie wejścia lub wyjścia kompresja zachowania może pobrać dane, które mogą być przydatne w nowym poziomie szczegółowości, przekonwertować je do prawidłowego formatu i przekazać dalej. Wyobraź sobie postacie RPG na rynku ze złożonymi systemami podejmowania decyzji nastawionych na cel. Kiedy są ważne, rozważają swoje potrzeby i planują działania, aby im sprostać. Kiedy są mniej ważne, poruszają się między przypadkowymi straganami. Używając kompresji zachowań, zauważalne sprzężenie między zachowaniami można zmniejszyć. Kiedy postacie przechodzą od niskiego do bardzo ważnego, ich plan jest ustawiony tak, że stragan, do którego zmierzali, stał się pierwszym punktem planu (aby uniknąć skręcania się w pół kroku i kierowania do innego celu). Kiedy przechodzą od wysokiego do niskiego znaczenia, nie dokonują od razu losowego wyboru; ich cel jest ustalany od pierwszej pozycji na planie. Kompresja zachowań zapewnia mało istotne zachowania z dużo większą wiarygodnością. Zachowania o wysokim znaczeniu mogą być uruchamiane rzadziej i mogą mieć mniejszy zakres wartości ważności, dla których są aktywne.

Wadą jest wysiłek programistyczny: niestandardowe procedury muszą być napisane dla każdej pary zachowań, które prawdopodobnie będą używane sekwencyjnie. O ile nie możesz zagwarantować, że znaczenie nigdy nie zmieni się szybko, procedury pojedynczego wejścia i wyjścia nie wystarczą; Procedury przejścia są wymagane dla każdej pary zachowań.

Histereza

Wyobraź sobie postać, która przełączała się między zachowaniami w odległości 10 metrów od gracza. Bliżej tej wartości postać ma złożone zachowanie, podczas gdy jest głupsza, gdy jest bardziej odległa. Jeśli gracz zdarzy się, że idzie za postacią, może ona stale przesuwać się przez granicę 10 metrów. Przełączanie się między zachowaniami, które może być niezauważalne, jeśli zdarza się sporadycznie, będzie wyróżniać się, jeśli będzie się szybko zmieniać. Jeśli jedno z zachowań wykorzystuje algorytm w dowolnym momencie, możliwe jest, że algorytm nigdy nie zdobędzie wystarczająco dużo czasu na wygenerowanie sensownych wyników; będzie stale wyłączany. Jeśli przełącznik zachowania ma skojarzony etap przetwarzania wejścia lub wyjścia, fluktuacja może spowodować, że postać będzie miała jeszcze mniej czasu, niż gdyby wybrała jeden lub drugi poziom. Podobnie jak w przypadku każdego procesu zmiany zachowania, dobrym pomysłem jest wprowadzenie histerezy: granic, które różnią się w zależności od tego, czy podstawowa wartość (w naszym przypadku znaczenie) rośnie czy maleje. W przypadku LOD każdemu zachowaniu przypisywany jest nakładający się zakres wartości ważności, tam gdzie jest on prawidłowy. Za każdym razem, gdy postać jest uruchamiana, sprawdza, czy aktualna ważność mieści się w zakresie bieżącego zachowania. Jeśli tak, to zachowanie jest uruchamiane. Jeśli tak nie jest, to zachowanie ulega zmianie. Jeśli dostępne jest tylko jedno zachowanie, można je wybrać. Jeśli dostępne jest więcej niż jedno zachowanie, potrzebujemy mechanizmu arbitrażowego, aby wybrać między nimi. Omówiono tutaj najpopularniejsze techniki arbitrażowe.

Wybierz dowolne dostępne zachowanie

To najskuteczniejszy mechanizm selekcji. Możemy znaleźć dowolne dostępne zachowanie, upewniając się, że każde z nich jest uporządkowane według jego zakresu i wykonując wyszukiwanie binarne. Zakres jest kontrolowany przez dwie wartości (maksymalną i minimalną), ale kolejność nie może tego kontrolować, więc wyszukiwanie binarne może nie dać poprawnego wyniku. Jeśli początkowe zachowanie nie jest dostępne, musimy przyjrzeć się pobliskim zakresom. Porządkowanie jest najczęściej wykonywane przez sortowanie w kolejności od środka zakresu.

Wybierz pierwsze dostępne zachowanie z listy

Jest to skuteczny sposób wyboru zachowania, ponieważ nie musimy sprawdzać, ile zachowań jest prawidłowych. Jak tylko ją znajdziemy, używamy jej. Jak widzieliśmy w rozdziale 5, może zapewnić podstawową kontrolę priorytetów. Układając możliwe zachowania w kolejności priorytetów, zostanie wybrane zachowanie o najwyższym priorytecie. To podejście jest również najprostsze do zaimplementowania i będzie stanowić podstawę poniższego pseudokodu.

Wybierz najbardziej centralne zachowanie

Wybieramy dostępne zachowanie, w którym wartość ważności znajduje się najbliżej środka jego zakresu. Ta heurystyka sprawia, że nowe zachowanie trwa najdłużej, zanim zostanie zamienione. Jest to przydatne, gdy przetwarzanie wejścia i wyjścia jest kosztowne.

Wybierz dostępne zachowanie z najmniejszym zakresem

Ta heurystyka preferuje najbardziej specyficzne zachowanie. Zakłada się, że jeśli zachowanie może działać tylko w małym zakresie, powinno być uruchamiane wtedy, gdy może, ponieważ jest dostrojone do tego małego zestawu wartości ważności.

Zachowania awaryjne

Druga i czwarta metoda wyboru pozwalają na zachowanie awaryjne, które jest uruchamiane tylko wtedy, gdy żadna inna nie jest dostępna. Zachowania zastępcze powinny mieć zakresy, które obejmują wszystkie możliwe znaczenie wartości. W metodzie drugiej ostatnie zachowanie na liście nigdy nie zostanie uruchomione, jeśli inne będzie dostępne. W metodzie czwartej ogromny zakres funkcji zastępczych oznacza, że zachowanie zawsze będzie zdominowane przez inne zachowania.

Pseudo kod

Behawioralny system LOD można wdrożyć w następujący sposób:

```
class BehavioralLOD extends Behavior:

2 # The list of behavior records.

3 records: BehaviorRecord[]

4

5 # The current behavior.

6 current: Behavior = null

7

8 # The current importance.

9 importance: float

10

11 # Finds the right record to run, and runs it.

12 function run(time: int):

13 # Check if we need to find a new behavior.

14 if not (current and current.isValid(importance)):
```

```
15
16 # Find a new behavior, by checking each in turn.
17 next: BehaviorRecord = null
18 for record in records:
19 # Check if the record is valid.
20 if record.isValid(importance):
21 # If so, use it.
22 next = record
23 break
24
25 # We're leaving the current behavior, so notify
26 # it where we're going.
27 if current and current.exit:
28 current.exit(next.behavior)
29
30 # Likewise, notify our new behavior where we're
31 # coming from.
32 if next and next.enter:
33 next.enter(current.behavior)
34
35 # Set our current behavior to be that found.
36 current = next
37
38 # We should have either decided to use the previous
39 # behavior, or else we have found a new one, either
40 # way it is stored in the current variable, so run it.
41 current.behavior.run(time)
```

Struktury danych i interfejsy

Założyłem, że zachowania mają następującą strukturę:

```
1 class Behavior:
2 function run(time: int)
```

dokładnie tak jak poprzednio.

Algorytm zarządza rekordami zachowań, które dodają dodatkowe informacje do podstawowego zachowania. Rekordy zachowań mają następującą strukturę:

```
1 # The data for one possible behavior.
2 class BehaviorRecord:
3 behavior: Behavior
4 minImportance: float
5 maxImportance: float
6 enter: function
7 exit: function
8
9 # Check if the importance is in the correct range.
10 function isValid(importance: float) -> bool:
11 return minImportance >= importance >= maxImportance
```

Elementy enter i exit zawierają odniesienie do funkcji (mogłyby być również zaimplementowane jako metody do przeciążenia, ale wtedy mielibyśmy do czynienia z wieloma podklasami zapisu zachowań). Jeśli nie jest potrzebne żadne ustawienie ani awaria, można je pozostawić nieuzbrojone. Te dwie funkcje są wywoływane, gdy odpowiednie zachowanie zostanie odpowiednio wprowadzone lub zakończone. Powinny mieć następującą formę:

```
1 function enterFunction(previous: Behavior)
2 function exitFunction(next: Behavior)
```

Przyjmują następne lub poprzednie zachowanie jako parametr, aby umożliwić im obsługę kompresji zachowań. W metodzie wyjścia zachowania może przekazać odpowiednie dane do następnego zachowania, jakie otrzymał. Jest to preferowane podejście, ponieważ pozwala wychodzącemu zachowaniu wyczyścić wszystkie swoje dane. Jeśli funkcja enter jest używana do próby zbadania poprzedniego zachowania danych, oznacza to, że dane mogły zostać już wyczyszczone. Moglibyśmy oczywiście zamienić kolejność dwóch wywołań tak, aby enter było wywoływane przed wyjściem. Niestety oznacza to, że pamięć dla obu zachowań jest aktywna w tym samym czasie, co może powodować skoki pamięci. Na platformach z czasochłonnym usuwaniem śmieci, takich jak Unity, zdecydowanie chcemy, aby pamięć była stabilna, a wstępnie przydzielona pamięć powinna być ponownie używana. Zwykle bezpiecznie jest błądzić po tej stronie i mieć krótki czas, gdy żadne z zachowań nie jest w pełni skonfigurowane.

Uwagi dotyczące implementacji

Powyższy pseudokod został zaprojektowany tak, aby zachowanie LOD mogło funkcjonować jako zachowanie samo w sobie. To pozwala nam używać go jako części hierarchicznego systemu planowania, jak omówiono w poprzedniej sekcji. W pełnej implementacji powinniśmy również śledzić czas potrzebny do podjęcia decyzji, które zachowanie powinno zostać uruchomione, a następnie odjąć

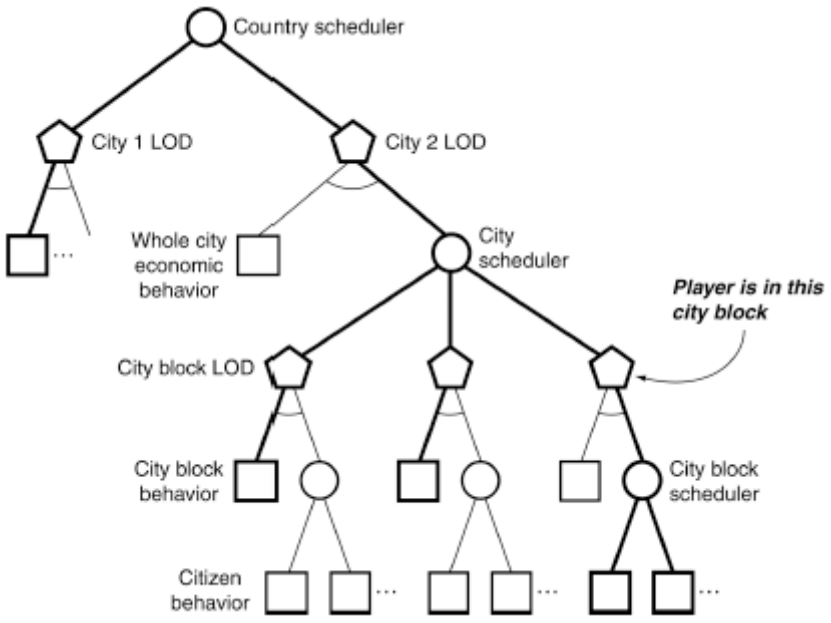
ten czas od czasu przejścia do zachowania. Chociaż wybór LOD jest szybki, najlepiej byłoby, aby czas był jak najdokładniejszy.

Wydajność

Algorytm to O(1) w pamięci i O(n) w czasie, gdzie n to liczba zachowań zarządzanych przez LOD. Jest to funkcja wybranego przez nas schematu arbitrażowego. Użycie schematu „wybierz dowolne dostępne zachowanie" może umożliwić algorytmowi zbliżenie się do O(log n) w czasie. Ponieważ zazwyczaj mamy do czynienia z bardzo małą liczbą LOD na postać (zwykle z naszego doświadczenia cztery to absolutne maksimum), nie ma potrzeby martwić się o czas O(n).

ZŁOŻENIE GRUP

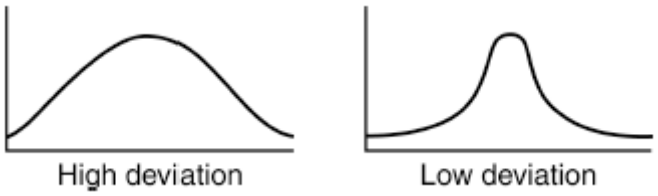
Nawet przy najprostszych zachowaniach duża liczba znaków wymaga dużej mocy obliczeniowej. W świecie gry z tysiącami postaci, nawet proste zachowania ruchowe będą zbyt trudne do przetworzenia. Możliwe jest wyłączenie postaci, gdy nie są one ważne, ale gracz łatwo to zauważy. Lepszym rozwiązaniem jest dodanie niskiego poziomu szczegółowości, gdy grupy znaków są przetwarzane jako całość, a nie jako pojedyncze osoby. Na przykład w grze fabularnej rozgrywającej się w czterech miastach wszystkie postacie w odległym mieście można zaktualizować za pomocą jednego zachowania: zmiany bogactwa jednostki, tworzenia dzieci, zabijania różnych obywateli i przenoszenia miejsc schronienia. Utracone zostają szczegóły codziennych zajęć każdego mieszkańca, takie jak spacer na targ, aby wydawać pieniądze, kupować przedmioty, zabierać je do domu, płacić podatki, łapać plagi i tak dalej. Ale ogólne poczucie rozwijającej się społeczności pozostaje. Takie właśnie podejście zastosowano w Republice: The Revolution . Przełączanie na grupę jest łatwe do wdrożenia przy użyciu hierarchicznego systemu planowania. Na najwyższym poziomie komponent LOD zachowania wybiera sposób przetwarzania całego miasta. Może wykorzystywać pojedyncze zachowanie „ekonomiczne" lub symulować poszczególne bloki miasta. Jeśli wybierze podejście blokowe, daje kontrolę nad systemem planowania, który rozdziela czas procesora na zestaw algorytmów zachowania LOD dla każdego bloku miejskiego. Te z kolei mogą przekazać swój czas systemom harmonogramowania, które kontrolują każdy znak z osobna, prawdopodobnie przy użyciu innego algorytmu LOD. Przypadek ten ilustruje rysunek 10.6. Jeśli gracz znajduje się obecnie w jednym bloku miasta, indywidualne zachowania dla tego bloku będą działać, zachowanie „blok" będzie działać dla innych bloków w tym samym mieście, a zachowanie „ekonomiczne" będzie działać dla innych miast. Pokazano to na rysunku .



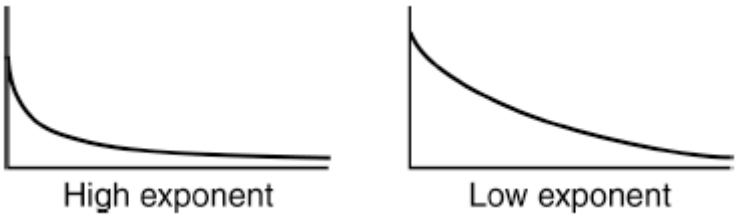
Łączy się to płynnie z innymi podejściami LOD lub planowania. Na najniższym poziomie hierarchii w naszym przykładzie moglibyśmy dodać priorytetowy algorytm LOD, który przypisuje czas procesora do osób w bieżącym bloku miejskim, w zależności od tego, jak blisko gracza znajdują się te osoby.

Rozkłady prawdopodobieństwa

Dotychczasowe podejście do grupowego LOD wymaga zachowania pewnych danych szkieletowych dla każdej postaci w grze. Może to być tak proste, jak wiek, bogactwo i wartości zdrowotne, lub może zawierać listę rzeczy, miejsca zamieszkania i pracy oraz motywy. Przy bardzo dużej liczbie znaków nawet ta skromna pamięć staje się zbyt duża. Ostatnio gry zaczęły używać grupowego LOD, które łączy ze sobą dane postaci. Zamiast przechowywać zestaw wartości dla każdego znaku, przechowuje liczbę znaków i rozkłady dla każdej wartości. Na rysunku 10.8 każdy zestaw znaków ma wartość bogactwa. Kiedy się łączą, ich indywidualne wartości bogactwa są tracone, ale ich dystrybucja jest zachowana. Gdy potrzebny jest bardzo ważny LOD, procedura kompresji może utworzyć odpowiednią liczbę nowych znaków przy użyciu tej samej dystrybucji. Utracona zostaje indywidualność każdej postaci, ale ogólna struktura społeczności jest taka sama. Wiele rzeczywistych wielkości jest rozłożonych na krzywej dzwonowej: krzywa rozkładu normalnego.



Można to przedstawić za pomocą dwóch wielkości: średniej (wartości średniej — w najwyższym punkcie krzywej) i odchylenia standardowego (oznaczającego, jak płaska jest krzywa). Spośród tych wielkości, które nie są normalnie rozłożone, rozkład mocy jest zwykle najbardziej dopasowany. Rozkład mocy jest używany dla ilości, w których wiele osób uzyskuje niskie wyniki, a niektóre wysokie. Na przykład podział pieniędzy wśród ludzi podlega prawu władzy.



Rozkład prawa potęgowego można przedstawić za pomocą jednej wartości: wykładnika (który również przedstawia, jak płaska jest krzywa). Tak więc przy jednym lub dwóch elementach danych możliwe jest wygenerowanie realistycznego rozkładu wartości dla całego zestawu znaków. Aby zdekompresować zachowanie, zwykle wystarczy utworzyć pojedyncze znaki z ich właściwościami niezależnie próbkowanymi z rozkładów prawdopodobieństwa. Zakłada to jednak, że te właściwości są niezależne. Na przykład wiek dorosłej postaci i jej wzrost mogą być prawdopodobnie niezależne. Wiek postaci i jej stan zdrowia mogą być ściślej powiązane. Możliwe są dystrybucje łączone, śledząc, jak jedna właściwość zmienia się wraz ze zmianą drugiej, ale obliczanie i przechowywanie tych danych może szybko stać się niewykonalne. Rozkłady prawdopodobieństwa prowadzą tylko do tej pory. Bardziej elastycznym podejściem byłoby wykorzystanie proceduralnych technik generowania treści z rozdziału 8 do tworzenia postaci w razie potrzeby. Z oczywistym minusem, że te techniki same w sobie mogą być

czasochłonne. Nie znam programistów używających w tym kontekście niczego poza najprostszymi technikami generowania proceduralnego.

PODSUMOWANIE

Opisałem systemy planowania, które wykonują zachowania z różnymi częstotliwościami lub przypisują każdemu z nich inne zasoby procesora. Przyjrzeliśmy się mechanizmom zmiany częstotliwości, priorytetu lub całego zachowania w zależności od tego, jak ważna jest postać dla gracza. W większości gier wymagania dotyczące planowania są dość skromne. Gra akcji może mieć 200 postaci na poziomie gry i często są one „wyłączone" lub „włączone". Nie potrzebujemy skomplikowanego planowania, aby poradzić sobie z tą sytuacją. Możemy po prostu użyć harmonogramu opartego na częstotliwości dla aktualnie „włączonych" znaków. Na nieco trudniejszym poziomie symulacje miast, takie jak Grand Theft Auto 3 [104], wymagają symulacji niewielkiej liczby postaci z teoretycznej populacji tysięcy. Postacie, których nie ma na ekranie, nie mają tożsamości (poza garstką postaci charakterystycznych dla fabuły). Gdy gracz się porusza, pojawiają się nowe postacie w oparciu o ogólne właściwości obszaru miasta i porę dnia. Jest to dość podstawowe zastosowanie techniki grupowego LOD. Ogólnokrajowe gry strategiczne, takie jak Republika, idą dalej, wymagając postaci o odrębnych tożsamościach. Grupowe algorytmy LOD, którym przyjrzeliśmy się w tym rozdziale, zostały w dużej mierze opracowane przez Elixir Studios, aby poradzić sobie z ogromną skalowalnością tej gry. Od tego czasu są używane z różnymi odmianami w wielu innych grach strategicznych.