PROGRAMOWANIE GRY AI

Edytory i inne narzędzia są ważne przy tworzeniu treści: tworzeniu danych, które nadają grze rozgrywkę. Od oczywistego projektu wizualnego geometrii poziomów, po konfigurację algorytmów AI, które tworzą zachowanie postaci. Jak widzieliśmy w poprzednim rozdziale, różne podejścia do sztucznej inteligencji będą wymagały różnych narzędzi: rozszerzeń do edycji poziomów lub modelowania 3D lub niestandardowych narzędzi do wizualizacji i konfiguracji procesu podejmowania decyzji. Dokładny wymóg zależy od zastosowanego podejścia. W przeciwieństwie do tego, jest jedno narzędzie, które jest zawsze potrzebne. Do tego stopnia, że można to łatwo pominąć w dyskusji o technologiach wspierających. Cała sztuczna inteligencja jest zaprogramowana. A język programowania ma duży wpływ na projektowanie AI. Większość tej książki skupia się na technikach, które nie są specyficzne dla gry. Na przykład odnajdywanie ścieżek lub drzewa decyzyjne to ogólne podejścia, które można zastosować w całej gamie gier. Techniki te należy wdrożyć. Dziesięć lat temu nie byłoby nierozsądne założyć, że zostaną one zaimplementowane w C++ jako część podstawowego kodu gry. Teraz zmieniło się to radykalnie. Chociaż silniki gier są nadal zwykle implementowane w C i C++, większość sztucznej inteligencji jest zazwyczaj implementowana w innym języku. Najpopularniejsze komercyjne silniki gier, Unity i Unreal Engine 4 (UE4), zapewniają system wyszukiwania ścieżek napisany w C++, ale jeśli potrzebujesz innej techniki sztucznej inteligencji (na przykład drzewa zachowań lub sterowania zachowaniami), należy ją napisać, lub licencjonowane od strony trzeciej. W UE4 może to nadal obejmować kodowanie w C++, ale w Unity prawdopodobnie oznacza to implementację w C#. W obu przypadkach nowy kod działa jak rodzaj wtyczki, na równi z kodem gry, a nie podstawowymi udogodnieniami, takimi jak sieć i renderowanie 3D. W tym samym czasie rozwinął się rozwój mobilny i online, gdzie języki implementacji są ograniczone przez platformę docelową. Swift (a wcześniej Objective-C) na iOS, Java (lub inne języki JVM, takie jak Kotlin) na Androida i JavaScript w sieci. Razem te trendy oznaczają, że techniki sztucznej inteligencji są wdrażane w znacznie szerszym zakresie języków. W następnej części omówiono różne wymagania, jakie różne języki nakładają na techniki opisane do tej pory w tej książce. Ale ogólne techniki AI to tylko jedno miejsce, w którym kod AI wymaga wsparcia programistycznego. Do sterowania zachowaniem znaków często używa się małych skryptów napisanych we wbudowanym języku programowania. Często jest to łatwiejsze niż użycie techniki ogólnego przeznaczenia i opracowanie narzędzi do prawidłowej konfiguracji. Zakładając, że osoba tworząca zachowanie postaci jest wystarczająco pewna w prostych zadaniach programistycznych, może być znacznie łatwiej stworzyć zachowanie oparte na stanach za pomocą serii instrukcji if, niż przeciągać bloki i przejścia w edytorze automatu stanów ogólnego przeznaczenia. Komercyjne silniki gier zawsze zapewniają jakąś formę obsługi skryptów. Jeśli tworzysz swój zestaw narzędzi od podstaw, może być konieczne dodanie własnego. Na szczęście istnieje kilka języków, które można łatwo zintegrować.

JĘZYK WDROŻENIA

Kiedyś gry pisane były prawie wyłącznie w C. Części kodu, które musiały działać tak szybko, jak to możliwe, były często ponownie implementowane w języku asemblerowym, gdy programiści czuli, że mogą wykonać lepszą pracę niż kompilator (a dla niektórych rodzajów kodu zdecydowanie mogli). Branża niechętnie migrowała do C++. Mówię niechętnie, ponieważ C++ jest znacznie większym językiem, z dość znaczącymi pułapkami wydajności dla nieostrożnych: toczyła się prawdziwa debata, czy C++ okaże się wystarczająco wydajny. Z tego okresu dostępnych jest wiele doskonałych baz kodu, w tym kod do niektórych z najlepiej sprzedających się tytułów. Na przykład źródło C++ do Quake 3 jest często cytowane przez programistów jako jedna z najpiękniejszych i najlepiej napisanych baz kodu C++. Dwie znaczące zmiany w branży w ciągu ostatnich 20 lat zmieniły sposób tworzenia gier. Pierwszym z nich jest wzrost rynku mobilnego jako głównego rynku. Chociaż możliwe jest tworzenie gier mobilnych

w C++ zarówno na iOS, jak i Androida, robi to tylko niewielka liczba twórców gier. Apple zachęca programistów iOS do używania Swift (a wcześniej Swift, Objective-C), podczas gdy Android opiera się na wirtualnej maszynie Java, którą można łatwiej zaprogramować w Javie lub języku JVM, takim jak Kotlin. Druga znacząca zmiana dotyczy wykorzystania silników gier. Większość programistów na komputery stacjonarne i konsole (i coraz częściej również na urządzenia mobilne) korzysta teraz z istniejących silników. W przypadku dużych studiów, takich jak EA, może to być ich własny wewnętrzny silnik. Dla większości programistów jest to Unity i Unreal Engine. C++ jest nadal głównym językiem używanym do tworzenia tych silników gier. A gry opracowane w UE4 często mają część rozgrywki zaimplementowaną w C++. Ale komputery są o wiele szybsze, wiele z tego, co zostało zaprogramowane w języku niskiego poziomu, można teraz zaimplementować w języku skryptowym silnika. Na przykład Blueprint w UE4 i C# w Unity. Eksplozja wykorzystania Unity w edukacji, wśród hobbystów, na urządzeniach mobilnych i w społeczności twórców niezależnych oznacza, że C# może teraz słusznie twierdzić, że jest podstawowym językiem do wdrażania gier. Trudno powiedzieć dokładnie, bo trudno to zmierzyć, ale przynajmniej sugestia nie jest absurdalna. Większość porad zawartych w tej książce jest neutralna językowo. W tej sekcji pokrótce przyjrzę się pięciu językom, które w chwili pisania tego tekstu są najważniejsze dla implementacji gier: C++, C#, Swift (dla iOS), Java (dla Androida) i JavaScript (dla gier internetowych i serwerów).

C++

C++ był pierwotnie rozszerzeniem języka C, aby dodać obsługę programowania obiektowego. Wciąż jest to w większości kompatybilny nadzbiór, chociaż czasami nowe funkcje pojawiają się w dwóch językach w różnym czasie. W ciągu ostatnich kilku lat Rust pozycjonował się jako nowy język do pisania niskopoziomowego kodu systemowego (rodzaj oprogramowania, które tworzy systemy operacyjne lub inne aplikacje o znaczeniu krytycznym dla zasobów i wydajności), ale poza kilkoma godnymi uwagi projektami Rust, C++ jest nadal najpopularniejszym wyborem do programowania niskopoziomowego. C++ to duży język. Od czasu pierwszego wydania w 1985 roku stale się rozwija. Nowe wersje standardu są obecnie wydawane co trzy lata, każda z własnym zestawem nowych funkcji i często nową składnią. Ten nadęty zestaw funkcji był konsekwentną krytyką C++ przez co najmniej ostatnie 15 lat. Tak bardzo, że skutecznie podzielił się na wiele języków. Nie, jak to zrobiły Scheme i Lisp, z powodu wielu wdrożeń, ale dlatego, że każda firma, która go adoptuje, ma tendencję do oswajania złożoności, adoptując tylko podzbiór całego języka. Ale rzadko ten sam podzbiór. Funkcje takie jak język szablonów, kompleksowa obsługa przeciążania operatorów, wielokrotne dziedziczenie klas, informacje o typie środowiska wykonawczego i przyjmowanie wyjątków są regularnie zabronione. Oprócz podstawowego języka C++ określa również standardową bibliotekę szablonów (STL). Jest to zestaw podstawowych typów danych zaimplementowanych przy użyciu szablonów C++. Podstawowe typy danych, które są częścią rdzenia w innych językach, takie jak tablice o zmiennym rozmiarze, mapy haszowe, stosy i kolejki, są dostarczane przez STL. Niestety implementacje STL są różne i jako całość ma reputację wśród twórców gier jako trudna w zarządzaniu, niska wydajność lub nieregularne wykorzystanie pamięci. Pracowałem dla więcej niż jednej firmy, która zakazuje STL w swoim kodzie, woląc tworzyć własne, nieszablonowe wersje tych podstawowych typów danych. Sytuacja jest jeszcze bardziej złożona, gdy do „poprawiania" niektórych problemów w języku podstawowym można użyć wspólnych bibliotek, takich jak Boost. W miarę rozwoju C++ czerpał dobre pomysły z Boosta, implementując je nieco inaczej i potencjalnie dając kolejną oś niezgodności. C++ jest to język, który najlepiej znam do implementacji gier. W rezultacie pseudokod powinien naturalnie zostać przetłumaczony na C++, w zależności od podzbioru funkcji, których Ty lub Twój zespół chcecie używać. Moim głównym założeniem jest dostępność struktur danych dobrej jakości.

C#

C# został stworzony przez Microsoft jako rywal dla Javy, mający na celu jego własne środowisko uruchomieniowe języka wspólnego (CLR). Początkowe wersje języka były dość podobne do Javy, ale z biegiem czasu chętniej zmieniał rdzeń, a języki się rozdzieliły. Ponieważ środowisko uruchomieniowe języka wspólnego jest zastrzeżone dla firmy Microsoft, inni programiści wdrożyli alternatywę typu open source o nazwie Mono. Ponieważ Microsoft open source język C#, zachowując tylko zastrzeżone środowisko uruchomieniowe, Mono zapewnia bezpłatny sposób używania języka w innym projekcie. Silnik Unity dokładnie to zrobił. Teoretycznie pozwala to na użycie dowolnego języka CLR z silnikiem Unity. Firma Unity obsługuje tylko C#, jednak po wycofaniu własnego języka JavaScript, takiego jak język skryptowy. Znam programistów używających F#, funkcjonalnego języka CLR, ale głównie eksperymentalnie. Zdecydowana większość kodu napisanego dla Unity to teraz C#. Częstą krytyką silnika Unity jest to, że pozostaje on daleko w tyle za obecną wersją C#. Częściowo dzieje się tak dlatego, że Mono niewiele ustępuje CLR Microsoftu, ale głównie dlatego, że Unity bardzo konserwatywnie podchodzi do aktualizacji wersji Mono, której używa w swoim silniku. Może to mieć wpływ na Ciebie jako programistę, ponieważ samouczki i zasoby online dla programistów C# są często niezgodne z wersją języka w silniku. Na szczęście jest to nieco łagodzone przez przytłaczającą liczbę samouczków i zasobów dotyczących Unity, chociaż często są one skierowane do bardziej początkujących programistów. Być może najbardziej znaczącą różnicą między C# a C++ jest sposób, w jaki Mono zarządza pamięcią. Zamiast jawnej alokacji i cofania alokacji, Mono śledzi pamięć używaną przez program i okresowo zwalnia całą pamięć, którą może znaleźć, a która nie jest już potrzebna. Niestety, znalezienie tej pamięci do cofnięcia alokacji (tzw. garbage collection) zajmuje trochę czasu, czasami dziesiątki milisekund. A co gorsza „okresowo” jest trudny do przewidzenia i niemożliwy do kontrolowania. Gry, które opierają się na tym zachowaniu, podlegają sporadycznym spowolnieniom, spadkom liczby klatek na sekundę lub zacinaniu się. Na szczęście usuwanie śmieci jest zwykle bardzo szybkie, jeśli nie ma nic do zebrania (istnieją skrajne przypadki, w których udowodnienie, że nic nie wymaga zbierania, zajmuje trochę czasu, ale są one zazwyczaj rzadkie w przypadku normalnych wzorców użytkowania). Zbieranie śmieci działa poprzez śledzenie, które obiekty wiedzą o istnieniu jakich innych obiektów. Zaczynając od zestawu znanych obiektów, przechodzi przez ten wykres i zaznacza wszystkie obiekty, do których można dotrzeć, w jednym lub wielu krokach. Jeśli na końcu tego procesu obiekt nie jest oznaczony, nic nie ma do niego dostępu i można go pobrać za darmo. Algorytm ten jest znany jako mark-and-sweep lub śledzenie zbierania śmieci. Zacinania się śmieci można uniknąć, jeśli twój kod alokuje wszystkie obiekty, których będzie potrzebował za jednym razem (zwykle po załadowaniu poziomu), a następnie zwolni je wszystkie za jednym razem, gdy gra się zakończy (przed załadowaniem następnego poziomu , na przykład). Nie ma sposobu jawnego przydzielania lub zwalniania pamięci w C#, więc pracujemy wokół algorytmu zaznaczania i wymiatania. Na początku poziomu tworzymy wszystkie obiekty, których możemy kiedykolwiek potrzebować i przechowujemy odniesienie do nich (jeśli są to podobne obiekty, możemy na przykład przechowywać je w tablicy). Gwarantuje to, że są zawsze dostępne, a więc nie będą zbierane śmieci. Podczas wykonywania algorytmu, gdy obiekt jest potrzebny, jest on pobierany z naszego wstępnie przydzielonego bloku. Na końcu poziomu usuwane są odniesienia do wszystkich obiektów, niezależnie od tego, czy z nich korzystaliśmy, czy nie. Odśmiecacz wykryje, że są one gotowe do zebrania. Problemy z szybkością zbierania śmieci w Mono są tak dotkliwe, że takie podejście jest praktycznie niezbędne. Nie ma to znaczącego wpływu na żaden z algorytmów przedstawionych w tej książce, ale należy o tym pamiętać przy każdej implementacji C#. Ostatnio Mono wdrożyło system zbierania śmieci, który jest znacznie mniej prawdopodobny ,powoduje opadanie klatek i zacinanie się. W chwili pisania jest to dostępne w Unity jako opcja, ale nie jest domyślnie włączone. Kiedy stanie się domyślnym wyrzucaniem elementów bezużytecznych, zapotrzebowanie na pule obiektów nieco się zmniejszy. Ale wyrzucanie śmieci zawsze będzie procesem czasochłonnym i przerywanym. W przypadku algorytmów takich jak A* liczba obiektów jest na tyle duża, że zawsze wskazana będzie jakaś forma wstępnej alokacji.

SWIFT

Swift to język Apple do wdrażania aplikacji na macOS i iOS. Chociaż jest to język skompilowany i teoretycznie można użyć dowolnego języka skompilowanego, takiego jak C++, interfejsy API systemu operacyjnego są ujawniane w Swift, a wszechstronne wsparcie zapewnia narzędzie programistyczne Xcode firmy Apple. O ile nie używasz zintegrowanego silnika gry, takiego jak Unity, praca z innym językiem na iOS jest trudnym wyzwaniem. Swift zastępuje starsze wsparcie Apple dla języka Objective-C. Objective-C jest, podobnie jak C++, rozszerzeniem języka C do obsługi programowania obiektowego. Te dwa języki różnią się dramatycznie w sposobie, w jaki wyobrażają sobie obiekty, co utrudnia współdziałanie między nimi. Objective-C nigdy nie doczekało się takiego samego rozpowszechnienia jak C++, a użycie tego języka stopniowo się zmniejszało. Zanim został zastąpiony przez Swift, programowanie dla urządzeń Apple było jedynym godnym uwagi przypadkiem użycia Objective-C. Początkowo Apple wprowadził Swift jako Objective-C bez C. Był to głównie chwyt marketingowy, aby złagodzić obawy programistów dotyczące uczenia się czegoś nowego, w rzeczywistości języki są bardzo różne. Swift kompiluje do kodu maszynowego dla konkretnego sprzętu, w przeciwieństwie do C# i Java, które kompilują do kodu bajtowego działającego na maszynie wirtualnej. Ale podobnie jak Java i C#, kod Swift jest wykonywany z wszechstronnym środowiskiem wykonawczym, obsługującym interfejsy systemu operacyjnego i udogodnieniami, takimi jak zarządzanie pamięcią i usuwanie elementów bezużytecznych. W praktyce różnica jest niewielka i SWIFT bardziej przypomina Java lub C# niż C lub C++. Podejście do zbierania śmieci, którego używa Swift, różni się od Mono. W chwili pisania tego tekstu używa automatycznego zliczania referencji (ARC). W tym podejściu każdy obiekt przechowuje licznik: odnosząc się do liczby miejsc w kodzie, które znają ten obiekt. Zmienna ustawiona tak, aby wskazywała na obiekt, zwiększa tę liczbę. Umieszczenie obiektu w tablicy również go zwiększa. Lub przechowywanie go w zmiennej składowej innego obiektu. Jeśli zmienna wykracza poza zakres, licznik jest zmniejszany. To samo dzieje się, jeśli tablica zostanie usunięta lub zawierający obiekt. Jeśli liczba osiągnie zero, obiekt nie może zostać osiągnięty przez żadną część kodu, więc można go bezpiecznie usunąć. Istnieją systemy zliczania odwołań, które są tak proste, ale proste podejście może spowodować wyciek pamięci: nigdy nie zbieranie niektórych nieużywanych cykli obiektów, do których istnieją odwołania, ale jako grupa, nigdy nie można osiągnąć. Na przykład, jeśli obiekt A jest jedyną rzeczą, która wie o obiekcie B, a obiekt B jest jedyną rzeczą, która wie o obiekcie A, zarówno A, jak i B mogą być zbierane, ale oba mają liczbę odwołań równą 1. Te cykle może stać się złożona, z wieloma zaangażowanymi obiektami. To właśnie te złożone cykle spowalniają usuwanie śmieci. Kompromisy między dwoma podejściami (liczenie referencji Swifta i markand-sweep Mono) są złożone i znacznie wykraczają poza zakres tego rozdziału. W praktyce wydaje się, że podejście Swifta jest mniej prawdopodobne, aby wstrzymywać lub zacinać kod gry. Ale mniej nie znaczy nigdy. Nadal warto unikać niechcianego wyrzucania śmieci. Aby to zrobić, możemy użyć dokładnie tego samego podejścia, które widzieliśmy w przypadku C#. Możemy wstępnie przydzielić wszystkie potrzebne nam obiekty i zwolnić ostatnie odniesienie do nich na końcu poziomu. Jeśli chodzi o ulepszony system zbierania śmieci Mono, odśmiecacz Swift zwykle działa wystarczająco szybko, aby problemy nie były zauważalne, chyba że wykonujesz wiele przydziałów. Ale problemy mogą często pojawiać się bez ostrzeżenia, po pewnym czasie gry. Możliwe, być może nawet najlepszą praktyką, jest zachowanie ostrożności i wstępne przydzielanie obiektów w kodzie AI, bez względu na to, czy jest to absolutnie konieczne, czy nie.

JAVA

Java jest wygodnie najpopularniejszym językiem programowania na świecie, Został zaprojektowany jako język ogólnego przeznaczenia ze szczegółową specyfikacją kodu bajtowego i maszyny wirtualnej (znanej jako wirtualna maszyna Java lub JVM). Blokując maszynę wirtualną, język ma na celu umożliwienie kompilacji kodu źródłowego na jednej maszynie, a wynikowy kod bajtowy może działać

na tej samej lub dowolnej innej maszynie. Jego slogan marketingowy brzmi „napisz raz, uruchom wszędzie”. W zasadzie to się udaje. Środowisko wykonawcze Java na dwóch bardzo różnych komputerach będzie musiało komunikować się z dwoma bardzo różnymi bazowymi systemami operacyjnymi, co może prowadzić do niewielkich różnic w zachowaniu, ale Java wykonuje dobrą robotę, próbując je załagodzić. To właśnie ta zdolność do jednorazowej kompilacji i uruchamiania w dowolnym miejscu czyni go szczególnie atrakcyjnym do tworzenia aplikacji mobilnych. W nowoczesnych smartfonach jest duży wybór sprzętu i kłopotliwe byłoby zdobycie odpowiednich definicji kompilatora i skompilowanie gry dla każdego z nich. Specyfika sprzętu nie eliminuje całkowicie potrzeby testowania na wielu urządzeniach, ale sytuacja byłaby znacznie gorsza w przypadku języka kompilującego się do kodu maszynowego, takiego jak C++. Kiedy Google opracował swój mobilny system operacyjny, Android, wybrał z tego powodu Javę jako podstawowy język programowania.

Jako język programowania ma reputację dość szczegółowego. I wiele z tej gadatliwości ma postać standardowego kodu: podobnych wzorców, które mają stosunkowo niewielką funkcjonalność, ale muszą być pisane w ten sam sposób raz za razem. Edytory i inne narzędzia ułatwiają to trochę, automatycznie generując niektóre z nich, ale nadal nadyma kod i wymaga konserwacji. Java ma również opinię tego, że wolno się zmienia. Z jednej strony jest to korzyść, co oznacza, że dobrze napisany kod pozostaje idiomatyczny przez długi czas. Ale oznacza to również, że ważne ulepszenia produktywności i dobre pomysły z innych języków są przyjmowane powoli. Bezpieczne dla typu struktury danych były wymagane przez ponad dekadę, zanim zostały ostatecznie wprowadzone, chociaż implementacja była szeroko krytykowana i ostatnio okazała się nieprawidłowa [3] (problem, który może rozciągać się na niektóre inne języki w tej sekcji, działa na JVM). Większość korzyści płynących z Javy zapewnia JVM. Jest to maszyna JVM zaprojektowana do uruchamiania tego samego skompilowanego kodu na dowolnej platformie. Gdy skompilowany kod bajtowy jest uruchomiony, maszyna wirtualna nie wie ani nie obchodzi, w jaki sposób ten kod został wygenerowany. Pod koniec lat 90. pracowałem krótko dla firmy niezwiązanej z grami, której produkt tworzył kod bajtowy z prostego dialogu z użytkownikiem. Jest częściej używany do obsługi innych języków, takich jak funkcjonalna Scala, dialekt Lisp Clojure, język skryptowy Groovy i dziesiątki innych, wszystkie kompilujące się do prawidłowego kodu bajtowego JVM. Ponieważ Java zapewnia bardzo dużą standardową bibliotekę, również skompilowaną do kodu bajtowego, funkcje te mogą być używane w dowolnym języku JVM. Podobnie system Android udostępnia swoje interfejsy w sposób, który można wywołać z dowolnego języka. Niedawno Google poparł Kotlin, język JVM opracowany przez JetBrains, który ma podobne funkcje i podobny etos do Javy, ale jest znacznie nowocześniejszy w stylu. Wydaje się prawdopodobne, że w nadchodzących latach coraz większa część tworzenia aplikacji na Androida przejdzie na nowy język. Jednocześnie twórcy gier mobilnych coraz częściej korzystają z wieloplatformowych silników gier. Który trend będzie rósł szybciej, trudno powiedzieć. Możliwe, że Kotlin nigdy nie stanie się znaczącym graczem w branży gier. Jeśli chodzi o porady dotyczące implementacji, pisanie dla JVM jest bardzo podobne do pisania dla Mono. JVM używa modułu odśmiecania pamięci typu mark-and-sweep, którego można ręcznie zażądać, wywołując System.gc(). Podobnie jak Mono, implementacja modułu odśmiecania pamięci nie jest wydajna, gdy przydzielono wiele małych obiektów. Rozsądnie jest zastosować to samo podejście do puli alokacji opisane w poprzedniej sekcji.

JAVASCRIPT

JavaScript został stworzony jako język skryptowy dla stron internetowych. Jest używany w tworzeniu gier w dwóch celach: implementacji gier przeznaczonych do grania w sieci oraz budowania serwerów dla gier wieloosobowych online za pomocą Node. Podobnie jak wszystkie języki skryptowe, krytyczny dla wydajności kod napisany w JavaScript może czasami działać wolno. Dlatego rzadko spotyka się

wyrafinowane algorytmy napisane w JavaScript. Bardzo możliwe jest zbudowanie maszyny stanów lub drzewa zachowań w JavaScript, ale planowanie, odnajdywanie ścieżek, minimaksowanie lub uczenie się prawdopodobnie wymagają zbyt wielu zasobów. Na serwerze te krytyczne algorytmy wydajności można zaimplementować w C++ i połączyć ze środowiskiem wykonawczym JavaScript. W przeglądarce coś podobnego było pierwotnie możliwe, ale teraz zostało usunięte przez dostawców przeglądarek ze względów bezpieczeństwa. JavaScript jest szerzej używany w tworzeniu gier jako język skryptowy, osadzony w grze lub silniku gry. Sekcja poniżej bardziej szczegółowo opisuje jego użycie jako języka skryptowego. Z punktu widzenia języka JavaScript wyróżnia się zastosowaniem prototypowego dziedziczenia. Dziedziczenie prototypowe różni się od bardziej znanego dziedziczenia opartego na klasach języków, takich jak C++ i Python. Ostatnie wersje JavaScriptu dodały słowo kluczowe class, aby uczynić je bardziej dostępnymi dla programistów innych języków, ale klasy są nadal implementowane pod maską przy użyciu prototypów. Prototypy różnią się od klas tym, że każdy obiekt może dziedziczyć z dowolnego innego obiektu. W języku opartym na klasach obiekty Wydajność występują w dwóch odmianach: klasach i instancjach. Instancje mogą dziedziczyć tylko po klasach, a klasy mają ograniczoną formę dziedziczenia po sobie, zwykle nazywaną podklasą. W językach prototypowych jest to znacznie prostsze. Istnieje tylko dziedziczenie, a każdy obiekt może dziedziczyć z dowolnego innego obiektu. Praktycznym rezultatem tego jest to, że nie jesteśmy ograniczeni do dwupoziomowej hierarchii klasy i instancji. W omawianie zachowania drzew, opisałem powszechną potrzebę trzech poziomów podczas definiowania i tworzenia instancji AI. To dobrze odwzorowuje się w JavaScript. Pozwala to programistom na stworzenie jednego obiektu głównego dla szerokiej klasy znaków, a następnie obiekt może dziedziczyć z tego, aby skonfigurować ustawienia dla konkretnego typu postaci (ten etap pośredni może wykonać projektant poziomów lub artysta techniczny) , a następnie w ostatnim kroku można zobaczyć, jak obiekt tworzy instancję typu znaku dla pojedynczej postaci na poziomie. W przeciwieństwie do innych języków w tej sekcji, JavaScript jest jednowątkowy. Jako język do rozszerzania stron internetowych, został zaprojektowany jako sterowany zdarzeniami: środowisko wykonawcze JavaScript monitoruje określone zdarzenia (takie jak dane wejściowe użytkownika, aktywność sieciowa lub zaplanowane limity czasu); gdy wystąpi zdarzenie, środowisko wykonawcze wywołuje kod, który został zarejestrowany jako zainteresowany; ten kod jest następnie uruchamiany sekwencyjnie tak długo, jak potrzebuje; a gdy nie ma już kodu do uruchomienia, sterowanie powraca do środowiska wykonawczego, które czeka na wystąpienie innego zdarzenia. Gwarantujemy, że kod nie zostanie przerwany podczas działania i żaden inny kod nie zostanie uruchomiony w tym samym czasie. Jest to idealne rozwiązanie dla prostych skryptów w przeglądarce, ale wykonanie niektórych czynności trwa znacznie dłużej. Nie chcemy, aby cały proces JavaScript zatrzymał się w oczekiwaniu na wynik. W przeglądarce internetowej może to być zapytanie o dane w sieci, które w niektórych przypadkach mogą być mierzone w sekundach. JavaScript używa do tego wywołań zwrotnych (w późniejszych wersjach JavaScript są one opakowane w łatwiejsze w użyciu struktury, takie jak obietnice lub „async”, chociaż podstawowe zachowanie jest takie samo). Wywołanie zwrotne wykorzystuje ten sam proces zdarzenia. Rejestrujesz kod, który ma być wywoływany po zakończeniu akcji (na przykład po otrzymaniu wyników z serwera), a następnie uruchamiasz akcję. Dzięki temu wiele wywołań zwrotnych może w dowolnym momencie oczekiwać na dane. Doskonale sprawdza się w wielu aplikacjach serwerowych, gdzie kod musi czekać na dane z bazy danych, z innych usług lub na odczyt plików. Niestety nie nadaje się do sytuacji, w których długotrwałe zadanie wymaga dużej ilości obliczeń. Do tej kategorii należy wiele gier. Jeśli silnik JavaScript wykonuje obliczenia, nie może kontynuować dystrybucji zdarzeń do innych części kodu. Nie możemy łatwo poprosić procedurę JavaScript, aby uruchomiła Pathfinder i powiadomić nas, kiedy odnajdywanie ścieżek zostanie zakończone. Oczywista implementacja zatrzymałaby się do czasu zakończenia wyszukiwania ścieżek. Są dwa rozwiązania. Kod można napisać w celu wykonania „współpracy wielozadaniowej”: każdy długoterminowy proces wykona trochę pracy, a następnie dobrowolnie zwróci kontrolę do środowiska wykonawczego,

oczekując, że wkrótce zostanie ponownie wywołany, aby wykonać więcej pracy. JavaScript może następnie wykorzystać te przerwy do wywołania innych części kodu. To działa, ale wymaga zaimplementowania algorytmów, które to obsługują. W rozdziale 10 widzieliśmy, dlaczego możemy chcieć to zrobić w dowolnym języku: aby umożliwić dzielenie długo działających algorytmów na wiele ramek renderowania. W JavaScript przechodzi od bycia „przyjemnym w posiadaniu" do praktycznie niezbędnego. Drugim rozwiązaniem jest użycie wielu interpreterów JavaScript i napisanie kodu komunikacyjnego, aby zachować ich synchronizację. Zamiast wywoływać funkcję pathfindingu, przekazujemy komunikat do innego procesu, który jest w stanie go odnaleźć. Po zakończeniu ten proces przekaże wiadomość z powrotem. Takie podejście jest stosowane w przeglądarce za pośrednictwem interfejsu API Web-Workers. Ponownie, jest to skuteczne, ale wymaga jeszcze więcej kodu koordynacyjnego niż kooperacyjna wielozadaniowość, a ponieważ różne procesy nie mogą nawzajem zmieniać danych, zwykle przekazywanie danych w przód i w tył w wiadomościach wiąże się z dużym obciążeniem (często poprzez konwersję i z tekstowego formatu JSON).

SKRYPTOWANE AI

Znaczna część procesu decyzyjnego w grach nie wykorzystuje żadnej z technik opisanych przez nas. Na początku i w połowie lat 90. większość sztucznej inteligencji była zakodowana na sztywno przy użyciu niestandardowego kodu pisanego do podejmowania decyzji. Jest to szybkie i dobrze sprawdza się w małych zespołach programistycznych, gdy programista prawdopodobnie również projektuje grę. Jest to nadal dominujący model dla platform o skromnych potrzebach programistycznych oraz zespołów niezależnych, w których programiści są również odpowiedzialni za wdrażanie rozgrywki. W miarę jak produkcja staje się bardziej złożona, prace programistyczne stają się bardziej niszowe. Duże gry z setkami pracowników mają tendencję do dzielenia zadań programistycznych na małe, ograniczone role. W studiu istnieje potrzeba oddzielenia treści (projektów zachowań) od silnika. Projektanci poziomów mogą być zobowiązani do zaprojektowania ogólnych zachowań postaci, bez możliwości edytowania ich kodu. Aby to ułatwić, wiele studiów używa ogólnych technik i niestandardowych edytorów, jak opisano w poprzednio. Rozwiązaniem pośrednim jest zaprogramowanie zestawu technik i łączenie ich przez kompetentnych technicznie projektantów poziomów za pomocą prostego języka programowania, niezależnego od głównego kodu gry. Są to często nazywane skryptami. Skrypty można traktować jako pliki danych, a jeśli język skryptowy jest wystarczająco prosty, projektanci poziomów lub artyści techniczni mogą tworzyć zachowania. Przypadkowym efektem ubocznym obsługi języka skryptowego jest możliwość tworzenia przez graczy własnego zachowania postaci i rozszerzania gry. Modyfikacja jest ważną siłą finansową w grach na PC (może wydłużyć okres ich przechowywania w pełnej cenie poza typowy dla innych tytułów osiem tygodni), do tego stopnia, że wiele tytułów triple-A zawiera jakiś system skryptów (często w ramach silnika, którego używają). Są jednak negatywy. Nie jestem przekonany, że powszechne użycie skryptów w projekcie jest szczególnie skalowalne, ani że nie jest łatwo wygenerować w ten sposób wyrafinowane zachowania. Z mojego doświadczenia wynika, że łatwo jest rozpocząć pisanie skryptów, ale ostatecznie stają się one trudne do rozszerzenia i debugowania. Oprócz kwestii skalowania podejście to mija się z celem ustalonych technik sztucznej inteligencji. Istnieją, ponieważ są eleganckimi rozwiązaniami problemów z zachowaniem, a nie dlatego, że programowanie w C++ jest niewygodne. Nawet jeśli przejdziesz do języka skryptowego, musisz pomyśleć o algorytmach używanych w skryptach postaci. Pisanie kodu ad hoc w skryptach jest tak trudne, jak pisanie go w C++ i prawie na pewno brakuje narzędzi do debugowania, które są tak dojrzałe. Kilku znanych mi programistów wpadło w tę pułapkę, zakładając, że język skryptowy oznacza, że nie muszą myśleć o sposobie implementacji postaci. Nawet jeśli używasz języka skryptowego, radzę pomyśleć o architekturze i algorytmach, których używasz w tych skryptach. Może być tak, że skrypt może zaimplementować jedną z innych technik opisanych w tej książce, lub może być tak, że oddzielna dedykowana implementacja byłaby bardziej praktyczna obok lub zamiast języka skryptowego. Ale

pomijając wszystkie te ograniczenia, niewątpliwie skrypty mają kilka ważnych zastosowań: do implementowania nietrywialnych wyzwalaczy i zachowania na poziomie gry (na przykład, które klawisze otwierają które drzwi), do iterowania mechaniki gry w coś fajnego do grania oraz do szybkiego prototypowania sztucznej inteligencji postaci . Ta sekcja zawiera krótkie wprowadzenie do obsługi języka skryptowego wystarczająco potężnego, aby uruchomić sztuczną inteligencję w twojej grze. Jest celowo płytki i zaprojektowany, aby zapewnić wystarczającą ilość informacji, aby zacząć lub zdecydować, że nie jest to warte wysiłku. Dostępnych jest kilka doskonałych stron internetowych porównujących istniejące języki, a kilka tekstów obejmuje implementację własnego języka od podstaw.

CO TO JEST SKRYPTOWA AI?

Wyrażenie „skryptowa sztuczna inteligencja" jest nieco niejednoznaczne. Na potrzeby tego rozdziału odnosi się do tych ręcznie zakodowanych programów napisanych w języku skryptowym, które kontrolują zachowanie postaci. to samo niezależnie od kontekstu, w sposób, który wydaje się nieinteligentny. Można powiedzieć, że postać, która zawsze próbuje podążać tą samą trasą patrolu, nawet gdy ta trasa jest zablokowana, ma oskryptowaną sztuczną inteligencję, nawet jeśli ta trasa jest generowana przez nietechnicznego projektanta w edytorze poziomów. Trzecie użycie tego terminu jest mniej pejoratywne: scenki można opisać jako oskryptowaną sztuczną inteligencję. Można powiedzieć, że postać, która ma mało zdrowia, chowa się za osłoną, wznosi barykadę i zaczyna się leczyć, ma oskryptowaną sztuczną inteligencję. Nawet jeśli ta sekwencja jest zaimplementowana jako drzewo zachowań. Na potrzeby tego rozdziału będę trzymać się pierwszej definicji: sztuczna inteligencja w skryptach to sztuczna inteligencja napisana w języku skryptowym.

CO SPRAWIA DOBRY JĘZYK SKRYPTOWY?

Jest kilka udogodnień, których gra zawsze będzie wymagała od swojego języka skryptowego. Wybór języka często sprowadza się do kompromisów między tymi obawami.

Prędkość

Języki skryptowe dla gier muszą działać tak szybko, jak to możliwe. Jeśli zamierzasz używać wielu skryptów do zachowania postaci i wydarzeń na poziomie gry, skrypty będą musiały zostać wykonane jako część głównej pętli gry. Oznacza to, że wolno działające skrypty zajmą czas potrzebny na renderowanie sceny, uruchomienie silnika fizyki lub przygotowanie dźwięku. Większość języków może być algorytmami w dowolnym momencie, działającymi w wielu ramkach (szczegóły w rozdziale 10). To w pewnym stopniu zmniejsza presję na prędkość, ale nie może rozwiązać problemu całkowicie.

Kompilacja i interpretacja

Języki skryptowe są szeroko interpretowane, kompilowane bajtowo lub w pełni kompilowane, chociaż istnieje wiele odmian każdej techniki. Tłumaczone języki są traktowane jako tekst. Interpreter patrzy na każdą linijkę, zastanawia się, co ona oznacza i wykonuje określone przez nią działanie. Języki skompilowane bajtowo są konwertowane z tekstu na format wewnętrzny, zwany kodem bajtowym. Ten kod bajtowy jest zwykle znacznie bardziej zwarty niż format tekstowy. Ponieważ kod bajtowy jest w formacie zoptymalizowanym do wykonywania, można go uruchomić znacznie szybciej. Języki kompilowane bajtowo wymagają etapu kompilacji; trwają dłużej, aby zacząć, ale potem działają szybciej. Droższy etap kompilacji można wykonać podczas ładowania poziomu, ale zwykle jest wykonywany przed wydaniem gry. Wszystkie najpopularniejsze języki skryptowe gier są kompilowane bajtami. Niektóre, jak Lua, oferują możliwość odłączenia kompilatora i nie rozpowszechniania go z finalną grą. W ten sposób wszystkie skrypty mogą zostać skompilowane przed przejściem gry do masteru, a tylko skompilowane wersje muszą być dołączone do gry. Uniemożliwia to jednak

użytkownikom pisanie własnego skryptu. W pełni skompilowane języki tworzą kod maszynowy. Zwykle musi to być połączone z głównym kodem gry, co może przekreślić sens posiadania oddzielnego języka skryptowego. Znam jednak jednego programistę z bardzo zgrabnym systemem łączenia w czasie wykonywania, który może kompilować i łączyć kod maszynowy ze skryptów w czasie wykonywania. Ogólnie jednak zakres ogromnych problemów związanych z tym podejściem jest ogromny. Radzę oszczędzać włosy i wybrać coś bardziej wypróbowanego i przetestowanego.

Rozszerzalność i integracja

Twój język skryptowy musi mieć dostęp do ważnych funkcji w grze. Skrypt, który kontroluje postać, na przykład, musi być w stanie przesłuchać grę, aby dowiedzieć się, co widzi, a następnie poinformować grę, co chce zrobić. Zestaw funkcji, do których musi uzyskać dostęp, jest rzadko znany, gdy język skryptowy jest zaimplementowany lub wybrany. Ważne jest, aby mieć język, który może łatwo wywoływać funkcje lub używać klas w głównym kodzie gry. Równie ważne jest, aby programiści mogli łatwo udostępniać nowe funkcje lub klasy, gdy autorzy skryptu tego zażądają. Niektóre języki (najlepszym przykładem jest Lua) nakładają bardzo cienką warstwę między skryptem a resztą programu. Dzięki temu bardzo łatwo jest manipulować danymi gry z poziomu skryptów, bez konieczności posiadania całego zestawu skomplikowanych tłumaczeń.

Ponowne wejście

Często przydatne jest ponowne wprowadzenie skryptów. Mogą biegać przez chwilę, a kiedy skończy się ich budżet czasowy, mogą zostać wstrzymani. Gdy skrypt ma trochę czasu na uruchomienie, może wznowić w miejscu, w którym został przerwany. Często pomocne jest pozwolenie skryptowi na kontrolowanie wydajności, gdy osiągnie naturalny spokój. Wtedy algorytm planowania może dać mu więcej czasu, jeśli ma go do dyspozycji, albo ruszy dalej. Na przykład skrypt kontrolujący postać może mieć pięć różnych etapów (badanie sytuacji, sprawdzanie stanu zdrowia, podejmowanie decyzji o ruchu, planowanie trasy i wykonywanie ruchu). Wszystko to można umieścić w jednym skrypcie, który umieszcza się między każdą sekcją. Następnie każdy będzie uruchamiany co pięć klatek, a obciążenie SI jest rozłożone. Nie wszystkie skrypty powinny być przerywane i wznawiane. Skrypt, który monitoruje szybko zmieniające się zdarzenie w grze, może wymagać uruchomienia od początku w każdej klatce (w przeciwnym razie może działać na nieprawidłowych informacjach). Bardziej wyrafinowane ponowne wejście powinno pozwolić autorowi scenariusza na oznaczenie sekcji jako nieprzerywalnych. Te subtelności nie są obecne w większości gotowych języków, ale mogą być ogromnym dobrodziejstwem, jeśli zdecydujesz się napisać własny.

UMIESZCZANIE

Osadzanie jest związane z rozszerzalnością. Język osadzony jest przeznaczony do włączenia do innego programu. Kiedy uruchamiasz język skryptowy ze swojej stacji roboczej, zwykle uruchamiasz dedykowany program do interpretacji pliku z kodem źródłowym. W grze system skryptowy musi być kontrolowany z poziomu głównego programu. Gra decyduje, które skrypty należy uruchomić i powinna być w stanie powiedzieć językowi skryptowemu, aby je przetworzył.

WYBÓR JĘZYKA OPEN SOURCE

Dostępnych jest wiele języków skryptowych, a wiele z nich jest wydawanych na licencjach odpowiednich do włączenia do gry. Zwykle jakaś odmiana open source. Chociaż podejmowano różne próby komercyjnego języka skryptowego dla branży gier, trudno jest zastrzeżonemu językowi konkurować z dużą liczbą wysokiej jakości alternatywnych rozwiązań typu open source. Oprogramowanie typu open source jest wydawane na podstawie licencjobiorcy, który daje

użytkownikom prawa do umieszczania go we własnym oprogramowaniu bez ponoszenia opłat. Niektóre licencje typu open source wymagają od użytkownika wydania nowo utworzonego produktu typu open source. Oczywiście nie nadają się one do gier komercyjnych. Oprogramowanie typu open source, jak sama nazwa wskazuje, umożliwia również dostęp do przeglądania i zmiany kodu źródłowego. Ułatwia to przyciągnięcie programistów do zgłaszania poprawek błędów i ulepszeń w bazie kodu. Jako programista daje pewność, że wiesz, że problemy można rozwiązać lub wprowadzić modyfikacje. Na przykład firma, z którą się konsultowałem, zleciła jednemu z autorów Lua wdrożenie niestandardowego rozszerzenia składni, zaledwie kilka dni pracy, które znacznie poprawiło wykorzystanie języka w ich projekcie. Jednak modyfikacje nie zawsze są wskazane. Niektóre licencje open-source (w szczególności generalna licencja publiczna, GPL), nawet te, które pozwalają na używanie języka w produktach komercyjnych, wymagają udostępnienia jakichkolwiek modyfikacji samego języka lub w najgorszym przypadku dowolnego kodu, do którego linkujesz do tej biblioteki. To prawie na pewno będzie problemem dla twojego projektu, chyba że zamierzasz udostępnić swoją grę jako open source. Niezależnie od tego, czy język skryptowy jest otwartym kodem źródłowym, istnieją prawne konsekwencje używania tego języka w projekcie. Przed użyciem jakiejkolwiek technologii zewnętrznej w produkcie, który zamierzasz dystrybuować, należy skonsultować się z prawnikiem zajmującym się własnością intelektualną. Ta książka nie może właściwie doradzić ci prawnych konsekwencji używania języka osób trzecich. Poniższe komentarze mają na celu wskazanie rodzajów rzeczy, które mogą budzić niepokój. Jest wiele innych. Ponieważ nikt nie sprzedaje Ci oprogramowania open source, nikt nie ponosi odpowiedzialności, jeśli oprogramowanie pójdzie nie tak. Może to być niewielką irytacją, jeśli podczas opracowywania pojawi się trudny do znalezienia błąd.

Może to być jednak poważny problem prawny, jeśli Twoje oprogramowanie spowoduje, że komputer klienta wyczyści jego dysk twardy. W przypadku większości oprogramowania typu open source jesteś odpowiedzialny za zachowanie produktu. Odwrotną stroną tego jest to, że dobrze ugruntowany język open source był prawdopodobnie używany miliony razy przez innych programistów, co sprawia, że nowe odkrycie patologicznych błędów jest mało prawdopodobne. Kiedy licencjonujesz technologię od firmy, firma zwykle działa jako warstwa izolująca między tobą a pozwem o naruszenie praw autorskich lub patentu. Na przykład naukowiec, który opracowuje i patentuje nową technikę, ma prawo do jej komercjalizacji. Jeśli ta sama technika zostanie zaimplementowana w oprogramowaniu, bez zgody badacza, mogą mieć powód do podjęcia kroków prawnych. Kupując oprogramowanie od firmy, bierze ona odpowiedzialność za zawartość oprogramowania. Jeśli więc badacz przyjdzie za tobą, firma, która sprzedała ci oprogramowanie, jest zwykle odpowiedzialna za naruszenie (zależy to od podpisanej umowy). Kiedy korzystasz z oprogramowania typu open source, nikt nie udziela Ci licencji na oprogramowanie, a ponieważ nie napisałeś go, nie wiesz, czy jego część została skradziona lub skopiowana. Jeśli nie będziesz bardzo ostrożny, nie będziesz wiedział, czy łamie jakiekolwiek patenty lub inne prawa własności intelektualnej. W rezultacie możesz być odpowiedzialny za naruszenie. Musisz upewnić się, że rozumiesz prawne konsekwencje korzystania z „wolnego" oprogramowania. Nie zawsze jest to najtańszy lub najlepszy wybór, mimo że koszty początkowe są bardzo niskie. Przed podjęciem zobowiązania skonsultuj się z prawnikiem. W przeszłości tego rodzaju kwestie prawne motywowały programistów do tworzenia własnych języków. Oprócz niestandardowych skryptów dla komercyjnych silników gier (takich jak język wizualny „planów" UE4), jest to teraz rzadkie.

WYBÓR JĘZYKA

Każdy ma swój ulubiony język, a próba wybrania najlepszego pojedynczego gotowego języka skryptowego jest niemożliwa. Przeczytaj dowolną grupę dyskusyjną dotyczącą języków programowania, aby znaleźć niekończące się wojny typu „mój język jest lepszy niż twój". Oczywiście są plusy i minusy, ale żaden nie jest potężniejszy niż twój zespół programistyczny, który dobrze zna

określony język lub składnię. Spośród niemal nieograniczonych możliwości wyboru jest kilka języków, które są wielokrotnie używane. Wybierając język do integracji z Twoją grą, warto mieć świadomość, które języki są zwykle podejrzanymi i jakie są ich mocne i słabe strony. Pamiętaj, że zwykle można zhakować, zrestrukturyzować lub przepisać istniejące języki, aby obejść ich oczywiste wady. Wielu (być może nawet większość) twórców gier komercyjnych używających języków skryptowych robi to w pomniejszy sposób; niektórzy kończą z językami prawie nierozpoznawalnymi od miejsca, w którym zaczęli. Opisane poniżej języki są omówione w gotowych formularzach. Przyjrzę się czterem językom w kolejności, w jakiej osobiście rozważyłbym je w nowym projekcie: Lua, Scheme, JavaScript i Python.

Lua

Lua to prosty język proceduralny zbudowany od podstaw jako język osadzania. Projekt języka był motywowany rozszerzalnością. W przeciwieństwie do większości języków osadzonych nie ogranicza się to do dodawania nowych funkcji lub typów danych w C lub C++. Sposób działania języka Lua można również poprawić. Lua ma niewielką liczbę podstawowych bibliotek, które zapewniają podstawową funkcjonalność. Jednak jego stosunkowo pozbawiony cech rdzeń jest częścią atrakcji. W grach prawdopodobnie nie będziesz potrzebować bibliotek do przetwarzania czegokolwiek poza matematyką i logiką. Mały rdzeń jest łatwy do nauczenia i bardzo elastyczny. Lua nie obsługuje funkcji ponownego wejścia. Cały interpreter (dokładnie obiekt „stanu", który zawiera stan interpretera) jest obiektem C++ i jest całkowicie reentrant. Używanie obiektów wielu stanów może zapewnić pewne wsparcie ponownego wejścia, kosztem pamięci i braku komunikacji między nimi. Lua ma pojęcie „zdarzeń" i „tagów". Zdarzenia Wydajność ują w pewnych momentach wykonywania skryptu, na przykład, gdy dwie wartości są dodawane do siebie, gdy wywoływana jest funkcja, gdy odpytywana jest tablica mieszająca lub gdy uruchamiany jest moduł odśmiecania pamięci. Wobec tych zdarzeń można zarejestrować procedury w C++ lub Lua. Te procedury „tag" są wywoływane, gdy wystąpi zdarzenie, umożliwiając zmianę domyślnego zachowania Lua. Ten głęboki poziom modyfikacji zachowania sprawia, że Lua jest jednym z najbardziej konfigurowalnych języków, jakie można znaleźć. Mechanizm zdarzeń i tagów służy do zapewnienia podstawowej obsługi zorientowanej obiektowo (Lua nie jest zorientowany ściśle obiektowo, ale możesz dostosować jego zachowanie, aby zbliżyć się do niego tak blisko, jak chcesz), ale może być również użyty do ujawnienia złożonego C++ typy do Lua lub do zwięzłego zaimplementowania zarządzania pamięcią. Kolejną cechą Lua uwielbianą przez programistów C++ jest typ danych „userdata". Lua obsługuje popularne typy danych, takie jak floaty, int i stringi. Ponadto obsługuje ogólne „dane użytkownika" z powiązanym podtypem („tag"). Domyślnie Lua nie wie, jak zrobić cokolwiek z danymi użytkownika, ale za pomocą metod tagów można dodać dowolne pożądane zachowanie. Dane użytkownika są powszechnie używane do przechowywania wskaźnika instancji C++. Ta natywna obsługa wskaźników może powodować problemy, ale często oznacza, że potrzeba znacznie mniej kodu interfejsu, aby Lua działała z obiektami gry. Jak na język skryptowy, Lua znajduje się na końcu skali. Ma bardzo prosty model wykonania, który w szczycie jest szybki. W połączeniu z możliwością wywoływania funkcji C lub C++ bez dużej ilości kodu interfejsu oznacza to, że wydajność w świecie rzeczywistym jest imponująca. Składnia Lua jest rozpoznawalna dla programistów C i Pascal. Nie jest to najłatwiejszy język do nauczenia się artystów i projektantów poziomów, ale jego względny brak funkcji składni oznacza, że jest on osiągalny dla zapalonych pracowników. Pomimo tego, że dokumentacja jest gorsza niż w przypadku pozostałych dwóch głównych języków, Lua jest najczęściej używanym gotowym językiem skryptowym w grach. Znane przejście Lucas Arts z wewnętrznego języka SCUMM na Lua zmotywowało wielu programistów do zbadania jego możliwości. Ani Unity, ani UE4 nie używają Lua, ale silnik open source Godot krótko go wspierał, zanim skupił się na własnym, niestandardowym języku. Aby dowiedzieć się więcej, najlepszym źródłem informacji jest książka Lua Programming in Lua , która jest również dostępna bezpłatnie online.

Schemat i wariacje

Scheme to język skryptowy wywodzący się z LISP, starego języka, który był używany do budowy większości klasycznych systemów AI przed latami 90. (i wielu później, ale bez tej samej dominacji). Pierwszą rzeczą, którą należy zauważyć w programie Scheme, jest jego składnia. Dla programistów nie przyzwyczajonych do LISP, Scheme może być trudny do zrozumienia. Nawiasy obejmują wywołania funkcji (i prawie wszystko jest wywołaniem funkcji) i wszystkie inne bloki kodu. Oznacza to, że mogą się bardzo zagnieździć. Dobre wcięcie kodu pomaga, ale edytor, który może sprawdzić zamykające nawiasy, jest niezbędny do poważnego programowania. Dla każdego zestawu nawiasów pierwszy element określa, co robi blok; może to być funkcja arytmetyczna:

(+ 0,5)

lub oświadczenie dotyczące kontroli przepływu:

(jeśli (> a 1.0) (ustaw! a 1.0))

Jest to łatwe do zrozumienia dla komputera, ale jest sprzeczne z naszym naturalnym językiem. Nie-programistom i osobom przyzwyczajonym do języków podobnych do C może być trudno przez jakiś czas myśleć w Scheme. W przeciwieństwie do innych języków w tej sekcji, istnieją dosłownie setki wersji Scheme, nie wspominając o innych wariantach LISP odpowiednich do użycia jako język osadzony. Każdy wariant ma swoje własne kompromisy, które utrudniają dokonywanie uogólnień na temat szybkości lub wykorzystania pamięci. Jednak w najlepszym wydaniu (przychodzą mi na myśl minischeme i tinyscheme) mogą być bardzo, bardzo małe (minischeme to mniej niż 2500 linijek kodu C dla całego systemu, chociaż brakuje mu niektórych bardziej egzotycznych funkcji pełnej implementacji schematu) i niezwykle łatwe do dostosowania. Najszybsze implementacje mogą być tak szybkie, jak każdy inny język skryptowy, a kompilacja może być zazwyczaj znacznie wydajniejsza niż inne języki (ponieważ składnia LISP została pierwotnie zaprojektowana do łatwego analizowania). Jednak tam, gdzie Scheme naprawdę błyszczy, jest jego elastyczność. W języku nie ma rozróżnienia między kodem a danymi, co ułatwia przenoszenie skryptów w ramach Scheme, modyfikowanie ich, a następnie wykonywanie ich później. To nie przypadek, że większość znaczących programów AI wykorzystujących techniki opisane w tej książce zostało pierwotnie napisane w LISP-ie. Dużo używałem Scheme, wystarczająco, aby móc przejrzeć jego niezręczną składnię (wielu z nas musiało nauczyć się LISP-a jako studenci AI - był uważany za główny język AI do przełomu XXI wieku). Zawodowo nigdy nie używałem Scheme w grze (chociaż znam przynajmniej jedno studio, które je posiada), ale zbudowałem więcej języków opartych na Scheme niż w jakimkolwiek innym języku (do tej pory siedem). Jeśli planujesz rozwijać swój własny język, gorąco polecam najpierw nauczyć się Scheme i przeczytać kilka prostych implementacji. Prawdopodobnie otworzy ci to oczy na to, jak łatwe może być stworzenie języka.

JavaScript

JavaScript to język skryptowy przeznaczony dla stron internetowych. Mimo że w nazwie jest Java, ma bardzo mały związek z tym językiem. Nazwa jest ciekawostką dawno nieistniejącej umowy marketingowej między Netscape, niegdyś producentem przeglądarek, a Sun Microsystems, ówczesnym właścicielem Javy. Ściśle mówiąc, język ten nazywa się ECMAScript, często w skrócie ES (szczególnie z przyrostkiem wskazującym wersję specyfikacji, taką jak ES6), chociaż poprzednia nazwa utknęła i jest mało prawdopodobne, aby została usunięta.

Nie ma jednej standardowej implementacji JavaScript. Wiele implementacji JavaScript w grach lub silnikach gier jest bliższych inspiracji JavaScriptem niż implementacją samego języka. Jednym z takich

przykładów jest UnityScript, przestarzały język skryptowy z silnika gry Unity. Te quasi-JavaScripty mogą używać tej samej składni (która jest stosunkowo podobna do C), ale w niektórych przypadkach nie obsługują nawet dziedziczenia prototypowego. Spośród pozostałych użytkowników JavaScript, większość to gry przeznaczone do uruchamiania w przeglądarce. W tym przypadku nie ma wbudowanego języka skryptowego. Skrypty są po prostu przekazywane do przeglądarki w celu ich uruchomienia. Wreszcie implementacja JavaScript V8, stworzona dla przeglądarki Chrome, jest szeroko osadzona. Jest to implementacja będąca sercem systemu Node JavaScript, na której zbudowano kilka kompletnych gier (i która służy do rozwijania strony serwerowej wielu innych). Może być również po prostu osadzony w istniejącym silniku. Obsługa pobierania i wyprowadzania danych oraz wystawiania podstawowych funkcji w JavaScript jest prosta i dobrze udokumentowana. V8 jest również w pełni ponownie wprowadzony i obsługuje wiele izolowanych interpreterów w tym samym kodzie (więc różne znaki mogą uruchamiać swoje skrypty w tym samym czasie). JavaScript jest silnym językiem do osadzania. Ale chyba najważniejszą jego zaletą jest znajomość. Jako język sieci jest znany wielu programistom. Dokumentacja jest obszerna i mnóstwo samouczków (choć trzeba powiedzieć, że niektóre są pisane przez samych nowicjuszy i zawierają złe rady). Jednak dzięki składni podobnej do języka C może on onieśmielać nie-programistów w sposób, którego unika bardziej naturalna i zawierająca słowa kluczowe składnia, taka jak Lua czy Python.

Python

Python to łatwy do nauczenia, zorientowany obiektowo język skryptowy z doskonałą rozszerzalnością i obsługą osadzania. Zapewnia doskonałe wsparcie dla programowania w językach mieszanych, w tym możliwość przezroczystego wywoływania C i C++ z Pythona. Python obsługuje funkcje re-entrant jako część podstawowego języka od wersji 2.2 (nazywanych generatorami). Python ma ogromny zakres dostępnych bibliotek i ma bardzo dużą bazę użytkowników. Użytkownicy Pythona mają opinię pomocnych, a grupa dyskusyjna comp.lang.python jest doskonałym źródłem rozwiązywania problemów i porad. Główne wady Pythona to szybkość i rozmiar. Chociaż w ciągu ostatnich kilku lat poczyniono znaczne postępy w szybkości realizacji, nadal może to być powolne. Python opiera się na wyszukiwaniu w tablicy mieszającej (według łańcucha) w przypadku wielu podstawowych operacji (wywołania funkcji, dostęp do zmiennych, programowanie obiektowe). To zwiększa koszty. Chociaż dobra praktyka programistyczna może w dużej mierze złagodzić problem z szybkością, Python ma również reputację dużego. Ponieważ ma znacznie więcej funkcji niż Lua, jest większy, gdy jest połączony z plikiem wykonywalnym gry. Python 2.X i kolejne wydania Pythona 2.3 dodały wiele funkcji do języka. Każde kolejne wydanie spełniało więcej obietnic Pythona jako narzędzia do inżynierii oprogramowania, ale tym samym czyniło go mniej atrakcyjnym jako język osadzony dla gier. Wcześniejsze wersje Pythona były pod tym względem znacznie lepsze, a programiści pracujący z Pythonem często preferują poprzednie wydania. Python często wydaje się dziwny programistom C lub C++, ponieważ używa wcięć do grupowania instrukcji, podobnie jak pseudokod w tej książce. Ta sama funkcja ułatwia naukę osobom niebędącym programistami, które nie mają nawiasów do zapomnienia i nie przechodzą przez normalną fazę uczenia się bez wcinania kodu. Python jest znany z tego, że jest bardzo czytelnym językiem. Nawet stosunkowo początkujący programiści mogą szybko zobaczyć, co robi skrypt. Nowsze dodatki do składni Pythona znacznie zaszkodziły tej reputacji, ale nadal wydaje się, że jest nieco wyższa od swoich konkurentów. Spośród języków skryptowych, z którymi pracowałem, Python był najłatwiejszy do nauczenia projektantów poziomów i artystów. W poprzednim projekcie chciałem skorzystać z tej korzyści, ale problemy z rozmiarem i szybkością sprawiły, że osadzanie języka było niepraktyczne. Rozwiązaniem było stworzenie nowego języka (patrz sekcja poniżej), ale użycie składni Pythona.

Inne opcje

Istnieje cała gama innych możliwych języków. Z mojego doświadczenia wynika, że każdy z nich jest albo zupełnie nieużywany w grach (według naszej najlepszej wiedzy), albo ma istotne słabości, które sprawiają, że jest trudnym wyborem w porównaniu z konkurencją. O ile mi wiadomo, żaden z języków w tej sekcji nie był wykorzystywany komercyjnie jako narzędzie do tworzenia skryptów w grze. Jak zwykle jednak zespół z konkretnymi uprzedzeniami i pasją do jednego konkretnego języka może obejść te ograniczenia i uzyskać użyteczny wynik.

Tcl

Tcl jest bardzo dobrze używanym językiem do osadzania. Został zaprojektowany jako język integracyjny, łączący wiele systemów napisanych w różnych językach. Tcl oznacza język sterowania narzędziami. Większość przetwarzania Tcl opiera się na ciągach, co może bardzo spowolnić wykonanie. Inną poważną wadą jest jego dziwaczna składnia, do której trzeba się trochę przyzwyczaić, i w przeciwieństwie do Scheme, ostatecznie nie obiecuje dodatkowej funkcjonalności. Niespójności w składni (takie jak przekazywanie argumentów według wartości lub nazwy) są poważniejszymi wadami przypadkowych uczniów.

Java

Java jest wszechobecna w wielu dziedzinach programowania. Ponieważ jest to język skompilowany, jego użycie jako języka skryptowego jest ograniczone. Z tego samego powodu może być jednak szybki. Używając kompilacji JIT (kod bajtowy zostaje zamieniony na natywny kod maszynowy przed wykonaniem), może zbliżyć się do C++ dla szybkości. Środowisko wykonawcze jest jednak bardzo duże i ma spory ślad pamięci. Najpoważniejsze są jednak kwestie integracyjne. Java Native Interface (łączący kod Java i C++) został zaprojektowany do rozszerzania Java, a nie do osadzania. W związku z tym zarządzanie może być trudne. Chociaż nie jako język skryptowy, Java jest używana do tworzenia gier. Minecraft, być może największa niezależna gra wszech czasów, jest w całości zaimplementowana w Javie, podobnie jak wiele jej modów i rozszerzeń.

Ruby

Ruby to nowoczesny język o tej samej elegancji, co w Pythonie, ale jego obsługa idiomów obiektowych jest bardziej zakorzeniona. Ma kilka fajnych funkcji, dzięki którym jest w stanie efektywnie manipulować własnym kodem. Może to być pomocne, gdy skrypty muszą wywoływać i modyfikować zachowanie innych skryptów. Nie jest to wysoce re-entrant od strony C++, ale bardzo łatwo jest stworzyć wyrafinowaną re-entrant z poziomu Ruby. Łatwo go zintegrować z kodem C (nie tak łatwym jak Lua, ale łatwiejszym niż na przykład Python). Wydaje się jednak, że Ruby może zanikać, nigdy nie docierając do odbiorców innych języków w tym rozdziale. Nie był używany (zmodyfikowany lub w inny sposób) w żadnej znanej mi grze. To słabość kurczaka i jajka: przy niewielu osobach, które dzielą się swoimi doświadczeniami z osadzania, trudno jest uzyskać dobre porady dotyczące pułapek i najlepszych praktyk. To może się szybko zmienić po jednym lub dwóch głośnych wdrożeniach.

TWORZENIE JĘZYKA

Aż do początku XXI wieku języki skryptowe używane w grach były tak samo prawdopodobne, że zostały stworzone przez programistów specjalnie na ich potrzeby, jak miały być wbudowanymi językami open source. W ciągu ostatnich 10 lat ta równowaga uległa zmianie, ale nadal istnieją sytuacje, w których język niestandardowy jest przydatny. Szczególnie dla stylów gier o bardzo niszowych wymaganiach. Platforma PuzzleScript autorstwa Stephena Lavelle'a [35] i system interaktywnej fikcji Inform 7 autorstwa Grahama Nelsona [44] to fascynujące języki oparte na regułach (a nie proceduralne, obiektowe czy funkcjonalne) i są mocno powiązane z ich silnikami gier. Język Inkle's Ink, opracowany

dla własnych gier, jest przeznaczony dla środowiska uruchomieniowego napisanego dla silnika gier Unity. Wszystkie są napisane z myślą o przypadkach użycia, w których istniejące języki byłyby znacznie bardziej uciążliwe. Komercyjne silniki gier obejmują obsługę języków skryptowych i w pewnym momencie były to niestandardowe języki, podobne do powszechnie dostępnych ofert open source. Nie są już aktywnie rozwijane. Unreal Engine miał UnrealScript, a Unity miał UnityScript. Firma Epic usunęła UnrealScript z UE4, a UnityScript jest obecnie przestarzały. Wydaje się, że nie ma powodu, aby rozwijać i utrzymywać od podstaw język, który jest tak podobny do języków dojrzałych. UE4 ma swój własny, niestandardowy język, Blueprint, który jest uzasadniony, ponieważ jest tak nietypowy: jest to język wizualny, który znajduje się gdzieś pomiędzy programowaniem a określaniem drzewa zachowań. Więc czy powinieneś inwestować we własny język? Rozważ dokładnie plusy i minusy.

Zalety

Kiedy tworzysz własny język skryptowy, możesz upewnić się, że robi dokładnie to, czego chcesz. Ponieważ gry są wrażliwe na ograniczenia pamięci i szybkości, w języku można umieścić tylko te funkcje, których potrzebujesz. Jak omówiono powyżej, na przykład istniejące języki często mają słabą obsługę ponownego wprowadzania: może to być kluczowa część twojego projektu. Możesz także dodać funkcje, które są specyficzne dla aplikacji do gier i które normalnie nie byłyby zawarte w języku ogólnego przeznaczenia. Lub, podobnie jak języki oparte na regułach, o których wspomniałem w poprzedniej sekcji, zbuduj swój język w radykalnie odmienny sposób. Ponieważ jest on tworzony wewnętrznie, gdy coś pójdzie nie tak z językiem, Ty lub Twój zespół znacie jego budowę i często możecie znaleźć błąd i szybciej utworzyć obejście. Za każdym razem, gdy dołączasz do gry kod innej firmy, tracisz nad nim kontrolę. W większości przypadków korzyści przeważają nad brakiem elastyczności, ale w przypadku niektórych projektów kontrola jest koniecznością.

Wady

Nowo utworzone języki mają zwykle bardziej podstawowe funkcje i są mniej niezawodne niż gotowe alternatywy. Jeśli wybierzesz dość dojrzały język, taki jak te opisane powyżej, zyskasz dużo czasu na tworzenie, debugowanie i optymalizację, które zostały wykonane przez innych ludzi. Każda osoba, która wcześniej używała tego języka, jest swego rodzaju testerem jakości. Język wewnętrzny wymaga gruntownego przetestowania, co jest dodatkowym kosztem. Gdy Twój zespół zbuduje podstawowy język i przejdzie do innych zadań związanych z kodowaniem, programowanie zatrzymuje się. Nie ma społeczności wykwalifikowanych programistów kontynuujących prace nad językiem, ulepszanie go i usuwanie błędów bez dodatkowych kosztów. Wiele języków open source udostępnia strony internetowe (często GitHub), na których można omawiać problemy, zgłaszać błędy i pobierać dokumentację. Wiele gier, zwłaszcza na komputery PC, jest pisanych z myślą o umożliwieniu konsumentom edytowania ich zachowań. Klienci budujący nowe obiekty, poziomy lub całe mody mogą przedłużyć okres trwałości gry. Korzystanie z niestandardowego języka skryptowego napisanego dla Twojej gry wymaga od użytkowników nauki tego języka. To z kolei może oznaczać, że będziesz musiał zapewnić samouczki, przykładowy kod i wsparcie dla programistów. W większości istniejących języków istnieją grupy dyskusyjne lub fora internetowe, na których klienci mogą uzyskać porady bez kontaktowania się z Tobą lub Twoim zespołem. Często na Stack Overflow będzie oddana grupa wykwalifikowanych programistów, którzy odpowiedzą na pytania. Trudno z tym konkurować, a jeśli spróbujesz, będzie to bardzo drogie. Jeśli jesteś hobbystą, polecam tworzenie własnego języka tylko jako ćwiczenie. Lub jeśli tworzysz swoje gry komercyjnie, tylko jeśli Twoja gra lub silnik gry mają bardzo specyficzne i specyficzne funkcje.

TOCZENIE WŁASNEGO

Niezależnie od wyglądu i możliwości ostatecznego języka, skrypty przechodzą przez ten sam proces na swojej drodze do wykonania: wszystkie języki skryptowe muszą zapewniać ten sam podstawowy zestaw elementów. Ponieważ te elementy są tak wszechobecne, narzędzia zostały opracowane i dopracowane, aby ułatwić ich budowanie. W tej książce nie mogę podać kompletnego przewodnika po budowaniu własnego języka skryptowego. Istnieje wiele książek na temat budowy języka (chociaż, o dziwo, nie znam żadnych dobrych książek o tworzeniu prostego języka skryptowego). W tej sekcji przyjrzymy się elementom budowy języka skryptowego z bardzo wysokiego poziomu, jako pomoc w zrozumieniu, a nie implementacji.

Etapy przetwarzania języka

Zaczynając jako tekst w pliku tekstowym, skrypt zazwyczaj przechodzi przez cztery etapy: tokenizację, parsowanie, kompilację i interpretację. Cztery etapy tworzą potok, z których każdy modyfikuje dane wejściowe, aby przekonwertować je na łatwiejszy do manipulowania format. Etapy nie mogą następować jeden po drugim. Wszystkie stopnie mogą być ze sobą połączone lub zestawy stopni mogą tworzyć oddzielne fazy. Skrypt może być tokenizowany, analizowany i kompilowany w trybie offline, na przykład w celu późniejszej interpretacji.

Tokenizacja

Tokenizacja identyfikuje elementy w tekście. Plik tekstowy to po prostu ciąg znaków (w sensie znaków ASCII!). Tokenizer ustala, które bajty należą do siebie i jaką grupę tworzą. Ciąg łańcuchów:

1a = 3,2;

można podzielić na sześć żetonów:

1 tekst „a”

2 <spacja> spacja

3 ’=’ operator równości

4 <spacja> spacja

5 3.2 liczba zmiennoprzecinkowa

6 ’;’ koniec identyfikatora instrukcji

Zauważ, że tokenizer nie sprawdza, w jaki sposób pasują one do siebie w znaczące porcje; to jest zadanie parsera. Dane wejściowe tokenizera to ciąg znaków. Dane wyjściowe to sekwencja tokenów.

Rozbiór gramatyczny zdania

Znaczenie programu jest bardzo hierarchiczne: nazwę zmiennej można znaleźć wewnątrz instrukcji przypisania, wewnątrz instrukcji IF, która znajduje się w treści funkcji, w definicji klasy, na przykład w deklaracji przestrzeni nazw. Parser pobiera sekwencję tokenów, identyfikuje rolę, jaką każdy z nich odgrywa w programie, oraz identyfikuje ogólną strukturę hierarchiczną programu.

Linia kodu:

1 jeśli (a < b) powrót;

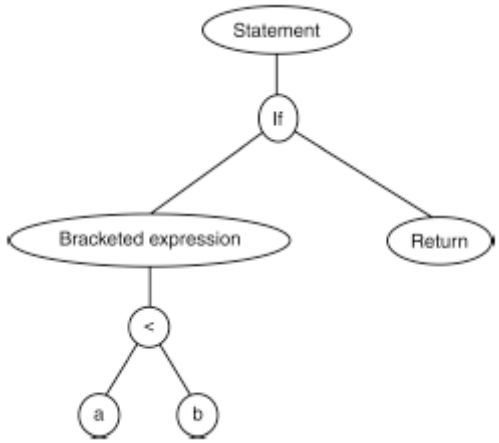
zamienione na sekwencję tokena:

1 słowo kluczowe(if), spacja, otwarte nawiasy, nazwa(a), operator(<),

2 imię (b), zamknięte nawiasy, spacja, słowo kluczowe (powrót),

3 end-of-statement

jest przekształcany przez parser w strukturę taką, jak pokazano na rysunku 13.1.



Ta hierarchiczna struktura jest znana jako drzewo analizy, a czasem drzewo składni lub abstrakcyjne drzewo składni (w skrócie AST). Drzewa parsowania w pełnym języku mogą być bardziej złożone, dodając dodatkowe warstwy dla różnych typów symboli lub grupując razem instrukcje. Zazwyczaj parser wypisuje wraz z drzewem dodatkowe dane, w szczególności tablicę symboli, która identyfikuje, jakie nazwy zmiennych lub funkcji zostały użyte w kodzie. To nie jest konieczne. Niektóre języki wyszukują nazwy zmiennych dynamicznie, gdy są uruchamiane w interpreterze (np. Python robi to). Błędy składniowe w kodzie pojawiają się podczas parsowania, ponieważ uniemożliwiają parserowi zbudowanie danych wyjściowych. Parser nie rozpracowuje, jak program powinien być uruchomiony; to jest zadanie kompilatora.

Kompilacja

Kompilator zamienia drzewo analizy w kod bajtowy, który może być uruchomiony przez interpreter. Kod bajtowy to zazwyczaj sekwencyjne dane binarne. Kompilatory nieoptymalizujące zazwyczaj wyświetlają kod bajtowy jako dosłowne tłumaczenie drzewa analizy. Czyli kod taki jak:

1 a = 3;

2 if (a < 0) return 1;

3 else return 0;

może zostać skompilowany w:

1 load 3

2 set-value-of a

3 get-value-of a

4 compare-with-zero

5 if-greater-jump-to LABEL

6 load 1

7 return

8 LABEL:

9 load 0

10 return

Kompilatory optymalizujące starają się zrozumieć program i wykorzystać wcześniejszą wiedzę, aby przyspieszyć wygenerowany kod. Kompilator optymalizujący może zauważyć, że a musi być równe 3, gdy wystąpi powyższa instrukcja IF. Może zatem generować:

1 load 3

2 set-value-of a

3 load 0

4 return

Stworzenie wydajnego kompilatora wykracza daleko poza zakres tej książki. Proste kompilatory nie są trudne do zbudowania, ale nie należy lekceważyć wysiłku i doświadczenia potrzebnego do zbudowania dobrego rozwiązania. Istnieje wiele setek domowych języków z żałosnymi kompilatorami. Tokenizacja, parsowanie i kompilacja są często wykonywane w trybie offline i są zwykle nazywane „kompilacją", mimo że proces ten obejmuje wszystkie trzy etapy. Wygenerowany kod bajtowy może być następnie przechowywany i interpretowany w czasie wykonywania. Parser i kompilator mogą być duże i nie ma sensu mieć narzutu tych modułów w ostatecznej wersji gry.

Interpretacja

Ostatni etap potoku uruchamia kod bajtowy. W kompilatorze dla języka takiego jak C lub C++ produktem końcowym będą instrukcje maszynowe, które mogą być uruchamiane bezpośrednio przez procesor. W języku skryptowym często trzeba zapewnić usługi (takie jak ponowne wejście i bezpieczne wykonanie), których nie można łatwo osiągnąć za pomocą języka maszynowego. Ostateczny kod bajtowy jest uruchamiany na „maszynie wirtualnej". Jest to skutecznie emulator maszyny, która nigdy nie istniała w sprzęcie. Ty decydujesz, jakie instrukcje może wykonać maszyna, a są to instrukcje kodu bajtowego. W poprzednim przykładzie

1 load <value>

2 set-value-of <variable>

3 get-value-of <variable>

4 compare-with-zero

5 if-greater-jump-to <location>

6 return

są kodem bajtowym. Twoje instrukcje kodu bajtowego nie muszą być również ograniczone do tych, które można zobaczyć w prawdziwym sprzęcie. Na przykład może istnieć kod bajtowy do „przekształcenia danych w zestaw współrzędnych gry": rodzaj instrukcji, która ułatwia tworzenie kompilatora, ale której żaden prawdziwy sprzęt nigdy nie będzie potrzebował. Większość maszyn wirtualnych składa się z dużej instrukcji switch w języku C: każdy kod bajtowy ma krótki bit kodu C, który jest wykonywany po osiągnięciu kodu bajtowego w interpreterze. Tak więc kod bajtowy „dodaj" zawiera fragment kodu C/C++, który wykonuje operację dodawania. Nasz przykład konwersji może

mieć dwie lub trzy linie C++, aby wykonać wymaganą konwersję i skopiować wyniki z powrotem do odpowiedniego miejsca.

Kompilacja na czas

Ze względu na wysoce sekwencyjny charakter kodu bajtowego, możliwe jest napisanie maszyny wirtualnej, która działa bardzo szybko. Mimo że nadal jest interpretowany, jest wielokrotnie szybszy niż interpretacja języka źródłowego wiersz po wierszu. Możliwe jest jednak całkowite usunięcie kroku interpretacji poprzez dodanie dodatkowego kroku kompilacji. Niektóre kody bajtowe można skompilować do języka maszynowego sprzętu docelowego. Gdy odbywa się to na maszynie wirtualnej, tuż przed wykonaniem, nazywa się to kompilacją just-intime (JIT). Nie jest to powszechne w językach skryptowych gier, ale jest podstawą języków takich jak Java i kod bajtowy Microsoft .NET.

Narzędzia: szybkie spojrzenie na Lex i Yacc

Lex i Yacc to dwa główne narzędzia używane odpowiednio do budowania tokenizerów i parserów. Każda ma wiele różnych implementacji i jest dostarczana z większością dystrybucji UNIX (wersje są również dostępne dla innych platform). Najczęściej używane przeze mnie warianty Linuksa to Flex i Bison. Aby stworzyć tokenizer z Lexem, mówisz mu, co składa się na różne tokeny w Twoim języku. Na przykład, co stanowi liczbę (nawet to różni się w zależności od języka - porównaj 0,4f do 1,2e-9). Tworzy kod C, który konwertuje strumień tekstowy z twojego programu na strumień kodów tokenów i danych tokenów. Oprogramowanie, które generuje, jest prawie na pewno lepsze i szybsze niż to, co możesz napisać sam. Yacc buduje parsery. Wymaga reprezentacji gramatyki twojego języka – jakie tokeny mają sens razem i jakie duże struktury można składać na przykład z mniejszych. Ta gramatyka jest podana w zestawie reguł, które pokazują, jak większe struktury są robione z prostszych lub z tokenów, na przykład:

1 przypisanie: NAZWA '=' wyrażenie;

2 wyrażenie: wyrażenie „+” wyrażenie;

3 wyrażenie: NAZWA

Pierwsza linia to reguła, która mówi Yaccowi, że gdy znajdzie token NAZWA, po którym następuje znak równości, po którym następuje struktura, którą zna jako wyrażenie (dla której pokazane są dwie z wielu reguł), to wie, że ma przypisanie. Pierwsza reguła wyrażenia jest definiowana rekurencyjnie. Yacc generuje również kod C. W większości przypadków powstałe oprogramowanie jest tak samo dobre lub lepsze niż tworzone ręcznie, chyba że masz doświadczenie w pisaniu parserów. W przeciwieństwie do Lexa, ostateczny kod można często zoptymalizować, jeśli szybkość jest absolutnie krytyczna. Na szczęście w przypadku skryptowania gier kod może być zwykle skompilowany, gdy gra nie jest uruchomiona, więc niewielka nieefektywność nie jest ważna. Zarówno Lex, jak i Yacc umożliwiają dodanie własnego kodu C do oprogramowania tokenizującego lub parsującego. Nie ma jednak de facto standardowego narzędzia do kompilacji. W zależności od sposobu, w jaki będzie się zachowywać język, będzie się to znacznie różnić. Jednak bardzo często zdarza się, że Yacc buduje AST, nad którym kompilator ma pracować, i istnieją różne narzędzia do tego, z których każde ma swój własny format wyjściowy. Wiele kompilatorów opartych na Yacc nie musi tworzyć drzewa składni. Mogą tworzyć dane wyjściowe kodu bajtowego z reguł za pomocą kodu C zapisanego w pliku Yacc. Gdy tylko zostanie znalezione przyporządkowanie, wyprowadzany jest jego kod bajtowy. Jednak tworzenie w ten sposób kompilatorów optymalizujących jest bardzo trudne. Jeśli więc zamierzasz stworzyć profesjonalne rozwiązanie, warto udać się bezpośrednio do jakiegoś drzewa parsowania.