Programowanie gier

Programowanie gier – specyficzna dziedzina pracy programistów. Łączy inżynierię oprogramowania, grafikę i multimedia, sztuczną inteligencję, fizykę, matematykę, optymalizację, algorytmikę i wiele innych dziedzin w jednym wspólnym celu. Wymaga współpracy z wieloma specjalistami z innych dziedzin, między innymi grafikami, muzykami, autorami tekstów.

Powstało wiele bibliotek, które wspomagają proces programowania gier, jedne z nich są przeznaczone dla początkujących programistów, jak na przykład Allegro czy SDL, inne przeznaczone są dla bardziej zaawansowanych (OpenGL). Istnieją biblioteki poświęcone tylko jednemu zagadnieniu, np. grafice (OpenGL) czy dźwiękowi (OpenAL, FMOD, BASS), inne natomiast są bardzo rozbudowane i zapewniają kompleksowe mechanizmy umożliwiające programiście skupienie się na samym pisaniu gier zamiast zmuszać go do spędzania czasu na rozwiązywaniu problemów technicznych (DirectX).

Grafika

W bardziej rozbudowanych grach komputerowych za przetwarzanie grafiki odpowiada tzw. silnik graficzny. Można go podzielić na dwa podstawowe rodzaje: silnik 2D i silnik 3D. W niektórych przypadkach silnik 2D może prezentować ładniejszą grafikę niż silnik 3D, jednak wraz ze wzrostem możliwości kart graficznych w dziedzinie przetwarzania grafiki silnik 2D wychodzi z użycia. Ponadto aktualne możliwości techniczne sprawiają, że pisanie dedykowanych silników 2D wychodzi z użycia, w wielu przypadkach dużo lepszym rozwiązaniem wydaje się zaangażowanie podsystemu grafiki 3D do generowania grafiki dwuwymiarowej – osiąga się to poprzez ustawienie specjalnego trybu rzutowania zwanego ortogonalnym, oraz renderowanie na ekranie prostokątów z nałożonymi teksturami.

Silnik 2D

Silnik 2D tworzy scenę w grze zazwyczaj poprzez nakładanie gotowych dwuwymiarowych obrazów w odpowiedniej kolejności. Czasem są one jeszcze modyfikowane, np. poprzez zmianę jasności, kolor-keying (maskowanie barw) albo przeźroczystości. Elementy sceny mogły zostać stworzone przez twórców poprzez wyrenderowanie obiektów trójwymiarowych.

Silnik 2D to taki, który przetwarza jedynie grafikę dwuwymiarową, jednak zazwyczaj twórcom gier zależy, by utworzona w ten sposób scena sprawiała wrażenie trójwymiarowej. Większość silników 2D wykorzystuje rzut izometryczny. Świat w ten sposób przedstawiony prezentuje się ładnie, często bardziej szczegółowo niż w przypadku grafiki 3D. Podstawową wadą tego typu wyświetlania grafiki jest brak perspektywy (obiekty, które w świecie gry mają sprawiać wrażenie bardziej oddalonych, są tej samej wielkości co obiekty bliższe).

Silniki 2D są zazwyczaj proste w tworzeniu. Głównymi problemami podczas ich tworzenia jest określenie miejsca wyświetlania poszczególnych elementów sceny oraz kolejności ich wyświetlania. Łatwo jest również z góry określić, które elementy sceny nie są widoczne, bo np. znajdują się poza ekranem. Powoduje to, że optymalizacje silników 2D są proste. Podstawową wadą jest brak

możliwości swobodnego obracania kamery w grze. Dotychczas silniki 2D były bardzo popularne w grach strategicznych, jednak wraz ze wzrostem jakości grafiki 3D zaprzestaje się tworzenia gier 2D.

Silnik 3D

Silnik 3D ma na celu przygotowanie oraz wyrenderowanie sceny 3D. Grafika oparta na silniku 3D jest bardzo atrakcyjna dla gracza, jednak stworzenie dobrego silnika 3D jest niezwykle trudnym i pracochłonnym zajęciem. Sam rendering jest procesem tak skomplikowanym, że wszystkie tworzone obecnie gry wykorzystują gotowe biblioteki, jak np. DirectX czy OpenGL. Dzięki temu programista nie musi tworzyć kodu odpowiedzialnego za sam rendering oraz nie musi zagłębiać się w szczegóły działania kart graficznych.

Podstawowymi elementami, z których tworzony jest świat gry, są obiekty trójwymiarowe, zazwyczaj tworzone w programach takich jak 3D Studio, oraz tekstury nakładane na te obiekty. Oprócz tego we współczesnych grach występują różne efekty, jak np. światła, odblaski, cieniowanie, cząsteczki. Efekty te wpływają na realizm grafiki oraz na prędkość tworzenia poszczególnych klatek sceny. Z upływem lat w grach komputerowych efektów jest coraz więcej, co sprawia, że utworzenie silnika 3D staje się procesem coraz bardziej pracochłonnym. Tworzenie takich efektów jak kurz, falowanie wody, refleksy świetlne, pochłania sporo pracy. Wymaga również wprowadzania coraz bardziej zaawansowanych technologii do kart graficznych, jak np. Vertex Shader i Pixel Shader.

Ograniczona prędkość przetwarzania grafiki sprawia, że twórcy gier poświęcają dużo czasu na optymalizację silników 3D. Jest to zagadnienie bardzo obszerne, np. jednym z najistotniejszym elementów optymalizacji jest określenie z góry, które elementy świata gry nie będą widoczne. Trudno jest określić np. czy dom, na który patrzymy, na pewno zasłania w całości dom znajdujący się dalej. W przypadku gier, w których gracz porusza się po pomieszczeniach przechodząc z pokoi do pokoi, określa się je jako odseparowane elementy. Wszelkie miejsca, przez które jeden pokój może być widziany z innego pokoju, określa się mianem portali. Dzięki temu, jeśli gracz znajduje się w którymś z pokoi, można pominąć wyświetlanie innych pokoi, chyba że w zasięgu kamery znajduje się portal. Ten sposób optymalizacji ma ogromny wpływ na szybkość działania gry. Jednym z prostszych sposobów optymalizacji, dzięki któremu można zredukować ilość wyświetlanych elementów, jest mgła.

Kolejną optymalizacją jest tworzenie tekstur o różnych wielkościach i nakładanie ich w zależności od odległości od kamery. Czasem stosuje się również obiekty o różnej liczbie wielokątów w zależności od odległości od kamery (Level of detail). O ile przeskalowanie tekstur jest proste, o tyle zmniejszenie ilości wielokątów w obiekcie 3D przy zachowaniu podobnego wyglądu z nałożoną teksturą jest bardzo trudne. W silnikach 3D stosuje się jeszcze cały szereg innych optymalizacji, jak np. renderowanie niektórych elementów sceny rzadziej niż przy każdej następnej klatce itp.

Ciągły wzrost wydajności kart graficznych oraz nowe jednostki i funkcje sprawiają, że silnik 3D szybko staje się przestarzały. Jednocześnie nakłady pracy podczas tworzenia nowego silnika są ogromne. Powoduje to, że większość gier korzysta z silników 3D innych producentów, zwłaszcza że są one dosyć uniwersalne i można je stosować w różnych grach (w przeciwieństwie do systemu sztucznej inteligencji).

Rozwiązania nietypowe

Czasem stosuje się rozwiązania na pograniczu silnika 3D, próbując osiągnąć zadowalającą grafikę niskim kosztem i jednocześnie szybko działającą. Takie rozwiązania można znaleźć np. w grach GTA 2 oraz Diablo 2. W Diablo 2 umożliwiło to wykorzystanie tych samych grafik co w trybie 2D, jednak dzięki zastosowaniu renderingu możliwe było uzyskanie perspektywy. Rozwiązanie wykorzystane w grze polegało na tym, że w scenie 3D zamiast pełnych, trójwymiarowych obiektów w odpowiednich miejscach wyświetlane były dwuwymiarowe grafiki 2D zwrócone w stronę kamery. Jednak takie rozwiązanie sprawiało, że nie było możliwości obracania kamery. Gdyby była taka możliwość, całe „oszustwo" wyszłoby na jaw i gracz zauważyłby, że wszystkie obiekty w grze są tak naprawdę płaskie.

Dźwięk

W przeciwieństwie do grafiki, dźwięk nie jest niezbędnym elementem gry, jednak wszystkie nowoczesne gry generują dźwięki. W większości gier w tle gra muzyka. Ma ona na celu wytworzenie klimatu i jest uzależniona od rodzaju gry. Dawniej stosowało się dźwięki z głośniczka systemowego, później MIDI, jednak w obecnych grach korzysta się częściej z innych formatów jak np. MP3/OGG. Utworzenie odpowiednich kompozycji zleca się zazwyczaj zewnętrznym firmom, zespołom lub orkiestrom. W wielu grach muzyka zależy od tego, co aktualnie dzieje się w grze.

Poza muzyką w grze występują często teksty mówione, co wymusza nagranie odpowiednich tekstów. Czasami na taką współpracę decydują się znani aktorzy, np. Cezary Pazura. Oprócz tego w grze występuje wiele krótkich dźwięków, jak np. stąpnięcie po ziemi, przeładowanie broni itp. Tego typu dźwięki bywają często nagrane w formacie WAV.

Gra może obsługiwać dźwięki mono, stereo lub wielokanałowe. W zależności od tego, gdzie oraz w jakiej odległości dany dźwięk się pojawi, gracz usłyszy go z odpowiedniej strony i odpowiednio głośno. By nadać realizmu grze stosuje się czasem różne inne efekty (np. niektóre gry samochodowe symulują efekt Dopplera). Pomimo tego dźwięk jest jednym z prostszych zagadnień pojawiających się podczas tworzenia gier.

Sztuczna inteligencja

W grach komputerowych często występują postacie kierowane przez sam komputer. Zazwyczaj zamierzeniem autorów jest, by zachowanie takich postaci przypominało w możliwie jak największym stopniu zachowanie ludzkie. Ponadto algorytmy sztucznej inteligencji są ściśle uzależnione od rodzaju gry. Są zupełnie inne w przypadku gier samochodowych, gier typu FPS, czy gier strategicznych. W większości tych gier pojawia się zagadnienie przemieszczania bota (czyli postaci kierowanej przez komputer) po świecie gry. Służą do tego specjalnie określone punkty w świecie oraz algorytmy wyszukiwania drogi, np. AStar. Zagadnienia związane z ruchem są obszerne i zależą w dużej mierze od rodzaju gry.

Zarządzanie zasobami

Współczesne gry wykorzystują ogromne ilości dodatkowych danych, takich jak tekstury, pliki dźwiękowe i muzyczne, pliki poziomów, modele postaci, pliki konfiguracyjne i skryptowe itp.

Wielkość tych zasobów często liczy się w setkach megabajtów, logiczne jest więc, że przechowywanie ich wszystkich jednocześnie w pamięci jest niemożliwe. Dlatego kolejnym ważnym aspektem programowania gier jest zaprojektowanie elastycznego, uniwersalnego i wydajnego zarządcy zasobów, który zautomatyzuje wczytywanie, zarządzanie priorytetami, używanie, zwalnianie i obsługę błędów wszystkich zasobów używanych przez grę.

Wiele gier komputerowych nie przechowuje swoich plików danych bezpośrednio na dysku, lecz stosuje technikę zwaną wirtualnym systemem plików, która polega na tym, że pliki zasobów są łączone w jeden wielki plik, z którego potrzebne zasoby są odczytywane na żądanie. Dzięki temu dużo trudniejsze staje się nielegalne użycie zasobów przez osoby trzecie, poza tym można stosować inne techniki, np. kompresowanie czy szyfrowanie.

Pętla gry

Właściwie cała gra działa na podstawie jednej pętli – pętli gry. W niej są wywoływane kolejno warstwy wejścia, logiki, grafiki oraz dźwięku.

Najczęstsza kolejność poleceń w pętli:

```
pętla( użytkownik nie wyłączył gry )
  sprawdzenie wejścia gry
  wykonanie warstwy logicznej
  sprawdzenie kolizji
  narysowanie grafiki
  odegranie dźwięków
koniec pętli
```

Warstwowa budowa

Gra zazwyczaj jest zbudowana z warstw: grafiki, logiki, wejścia, dźwięku. Każda z warstw przechowuje ważne tylko dla siebie dane – przykładowo warstwa wejścia stan wciśnięcia klawisza myszki – oraz odpowiednio na nie reaguje. W grze zazwyczaj występuje szeroko pojęta komunikacja między wszystkimi warstwami. Idąc dalej tym samym przykładem warstwa wejścia otrzymała komunikat, że gracz przycisnął lewą strzałkę na klawiaturze. W związku z tym, warstwa wejścia wywołuje warstwę logiczną, aby ta odpowiedziała – i warstwa logiki przesuwa pozycję gracza o n pikseli w lewo. Natomiast procedura rysująca z warstwy graficznej, chce narysować gracza, więc odwołuje się do warstwy logiki, aby ta zwróciła pozycję gracza – aby móc narysować go na ekranie według jego pozycji. Każda z warstw ma swoją funkcję, w której jest jej pętla, czyli na przykład rysowanie grafiki w warstwie graficznej. W pętli gry wszystkie pętle warstw są wywoływane według wyżej podanej kolejności.

Środowisko GameMaker

Autor:	YoYo Games
Pierwsze wydanie:	lato 1999
Aktualna wersja stabilna:	1.4.1773 – 14 października 2017; ponad 4 lata temu
Aktualna wersja testowa:	1.99.460 – 15 października 2015; ponad 6 lat temu
Język programowania:	GML
Platforma sprzętowa:	PC, Macintosh, PlayStation Vita, PlayStation 4, Xbox One
System operacyjny:	Windows, Windows Phone 8, macOS, iOS, Android, Tizen
Rodzaj:	Tworzenie gier
Licencja:	komercyjna

GameMaker – środowisko do projektowania gier i programów komputerowych, pozwalające na pracę zarówno osobom bez znajomości zasad programowania, jak i zaawansowanym programistom. Środowisko projektowe dostępne jest obecnie w wersjach na Windows (GameMaker 8.1, GameMaker:Studio) i macOS (GameMakerMac). Najbardziej rozbudowana wersja programu, Studio, pozwala na eksport gier na platformy: Windows, Windows Phone, Ubuntu (oraz kompatybilne z nim systemy linuxowe), OS X, iOS, Android, Tizen, MIPS, HTML5[1], PlayStation 3[2], PlayStation 4, PlayStation Vita. Pozostałe – GM:8.1 i GM:Mac pozwalają jedynie na eksport gier pod dany system operacyjny, dodatkowo są dostępne w wersjach Lite, umożliwiających przetestowanie programu przed zakupem.

GameMaker oferuje duży zestaw gotowych funkcji, definiujących zachowania w grze. Jako wsparcie dla początkujących programistów, program oferuje tworzenie zachowań poprzez przeciąganie odpowiednich ikon z akcjami reprezentującymi najprostsze funkcje (tzw. klocki), do obiektów, na zasadzie "przeciągnij i upuść", stanowią one jednak bardzo mały wycinek wszystkich możliwości i część z nich wciąż wymaga wpisania np. nazw zmiennych. Największym atutem programu jest własny język skryptowy GML (GameMaker Language), którego funkcje stworzono z naciskiem na tworzenie gier. Umożliwia on tworzenie bardziej rozbudowanych algorytmów, modyfikację zasobów, odczyt plików, połączenia z siecią, efekty cząsteczkowe, prosty tryb wizualizacji trójwymiarowej, a także rozszerzanie programu poprzez biblioteki DLL, co sprawia, że ma praktycznie nieograniczony potencjał.
Wersje

Game Maker Language

Game Maker Language (w skrócie: GML) – język skryptowy stworzony na potrzeby programu do tworzenia gier Game Maker. Został stworzony przez Marka Overmarsa jako suplement systemu "przeciągnij i upuść" dostępnego od pierwszych wersji programu. W późniejszych wersjach system "przeciągnij i upuść" jest już tłumaczony bezpośrednio do GML, jest więc jego graficzną formą. Poza kilkoma czynnikami zarządzanymi bezpośrednio przez silnik GameMakera (kolejność wykonywania i wywoływanie zdarzeń, kolejność plansz) pokrywa wszystkie możliwości programu i jest sporym rozszerzeniem systemu przeciągnij i upuść, który pokrywa tylko nieznaczną część dostępnych funkcji.

GML może być używany w: akcjach obiektów, skryptach, liniach czasu, kodzie tworzenia planszy, rozszerzeniach oraz w kodzie tworzenia instancji na planszy.
Spis treści

Biblioteki

W GameMakerze, zestaw akcji (klocków) systemu przeciągnij i upuść nazywany jest biblioteką akcji. Akcje są wyświetlane w oknie edycji obiektów, w zakładkach, jako kwadratowe ikony. GameMaker posiada dostępnych kilka standardowych bibliotek wystarczających początkującemu użytkownikowi, można jednak tworzyć własne zestawy - w praktyce jest to jednak mało praktyczne, gdyż bardziej rozbudowane gry stają się trudne do zarządzania, gdy akcji jest dużo. W praktyce, średnio zaawansowany użytkownik korzysta głównie z akcji "wykonaj kod" w której może bezpośrednio wpisywać funkcje. Bibliotek nie należy mylić z bibliotekami DLL.
Składnia

Składnia GML przypomina C++ oraz Pascal (GameMaker jest napisany w Delphi). W wielu miejscach jednak różni się od tych języków, lub łączy ich zachowania (np. można zamiennie używać begin i {, oraz } i end). Przypisywanie wartości nie jest wyrażeniem zwracającym wartości, dlatego nie może być używane w instrukcjach warunkowych.

Poza przypisywaniem za pomocą znaku równości i operacjami matematycznymi jakie mogą za tym znakiem zajść, GM dopuszcza składnię w postaci +=, -=, *=, oraz /=. Nie można jednak używać operatora trójargumentowego (?:). Znaki średnika można używać do rozgraniczania kolejnych wyrażeń, ale ich brak nie jest uznawany za błąd.

Funkcje

GameMaker posiada kilkaset wbudowanych funkcji. Dodatkowo, użytkownik może tworzyć własne funkcje (skrypty). Funkcje używane do rysowania korzystają z API Direct3D.

Język pozwala na korzystanie z zewnętrznych funkcji oferowanych przez biblioteki DLL.

Zmienne

Zmiennych w GameMakerze nie trzeba definiować, są one tworzone poprzez pierwsze przepisanie na zasadzie zmienna = wartosc. GML posiada dwa typy zmiennych - łańcuch tekstowy (string) oraz liczby rzeczywiste (real). Typ zmiennej zostaje ustalony przy przypisaniu wartości (typ może się więc zmieniać w trakcie), nie można jednak porównywać dwóch typów zmiennych - jest więc to typowanie silne. Język nie posiada typów boole'owskich (true/false), tak więc za prawdę przyjmuje się każdą wartość >= 0.5, za fałsz wartości < 0.5;

Duża liczba zmiennych i stałych jest w język wbudowana. Część zmiennych jest lokalna dla każdej instancji obiektów (jak "x" czy "speed"), część jest wspólna i globalna (jak "score"), a część jedynie do odczytu (jak "fps"). Stałe są zawsze dostępne globalnie.

Użytkownik może definiować własne stałe, zmienne globalne (poprzedzając je przy użyciu prefiksem global., lub bez niego jeśli wcześniej zadeklaruje listę po wyrażeniu globalvar), zmienne lokalne (w obiektach, po prostu przypisując im wartość), oraz zmienne tymczasowe (w każdym fragmencie kodu, o zasięgu danego bloku, definiując listę za słowem var). Jeśli z bloku wykonywany jest kod dla innego obiektu, to ma on dostęp do zmiennych tymczasowych, ich zasięg jest więc globalny (GM ma tylko jeden wątek, więc nie powoduje to konfliktów).

Istnieją dwa typy tablic - jedno i dwu wymiarowe. Tablice mogą zawierać mieszankę tekstu i liczb, ale nie mogą zawierać kolejnych tablic. Tablice nie mogą być też przekazywane jako referencje do funkcji. Maksymalny indeks tablicy to 31999, a maksymalna pojemność to 1.000.000 elementów. Poza tradycyjnymi tablicami istnieją też struktury danych: listy, kolejki, stosy, mapy, siatki, kolejki priorytetowe.

Zasięg

Kod może być wykonywany względem aktualnej instancji, lub innej instancji. Cały skrypt może mieć narzucone wykonywanie względem innej instancji za pomocą specjalnego przełącznika w edytorze danego skryptu, lub za pomocą wyrażenia with. Od tego zależy z którego zasięgu brane będą zmienne. GameMaker automatycznie przypisuje do wyrażenia other referencję na obcą instancję w przypadku kolizji, oraz gdy wykonywany kod jest wywoływany z innej instancji:

/* zwrot obiektów innyObiekt zostanie ustawiony na taki sam, jak instancji z której kod wywołano */
with (innyObiekt) { direction = other.direction }

Zarządzanie pamięcią

GameMaker sam zarządza pamięcią i usuwa zmienne które zostały ustawione w danej instancji w momencie jej niszczenia. Nie są jednak usuwane zasoby które dostępne były przez referencję, tzn. jeśli dodaliśmy do gry grafikę i jej referencja została przypisana do zmiennej w instancji obiektu, to jego zniszczenie powoduje jedynie usunięcie zmiennej, a nie grafiki przypisanej przez tę referencję. To samo dotyczy struktur danych - jeśli zgubimy referencję, struktura nadal istnieje w bazie. Referencje to kolejne liczby naturalne (licznik nie cofa się w przypadku usuwania zasobów, taka referencja po prostu zwróci błąd nieistniejącego zasobu), dla każdego typu zasobu/struktury danych istnieje osobny licznik.
Przykłady kodu

Napis Hello World w wyskakującym oknie z przyciskiem OK:

```
show_message('Hello World');
```

Przesunięcie instancji obiektu którego referencją jest zmienna pilka o 4 piksele w prawo:

```
pilka.x+=4;
```

Wykonanie kroku do punktu 250,100 (względem planszy) o rozmiarze 4 pikseli:

```
move_towards_point(250,100,4);
```

Ta sama funkcja, omijająca przeszkody oznaczone jako obiekty nieprzenikalne:

```
mp_potential_step(250,100,4,0);
```

Przykład komentarza jednoliniowego:

```
// komentarz jednoliniowy
```

Przykład komentarza wieloliniowego:

```
/*
 komentarz
 wieloliniowy
 */
```

Przezroczystość instancji obiektu:

```
image_alpha=0.5;
```

Pętla "for":

```
for (i=0; i<30; i+=1;)
{
<wyrażenie>
}
```

tworzenie instancji obiektu "ludzik"

```
instance_create(x,y,ludzik)
```

przeskoczenie obiektu o 24 pixele

```
if keyboard_check_pressed(vk_left)x-=24
```