Game Maker Language

Game Maker Language (w skrócie: "GML") to język skryptowy stworzony przez Mark Overmars na potrzeby programu Game Maker. Jest on używany do ustalania i wykonywania automatycznych akcji przez program w dowolnym momencie. W Game Makerze można go używać w czterech miejscach: Scripts, Execute a Piece of Code, Creation Code oraz przy ustawianiu tzw. klocków. Język ten ma składnię i strukturę przypominające dobrze znane języki programowania takie jak C++ czy Pascal.

Podstawowe zastrzeżenia

- Nazwy wszystkich zasobów gry (sprites, sounds, objects itd.) muszą być różne i mogą mieć na początku tylko literę, a mogą składać się tylko z liter, liczb i podkreśleń "_".
- Posiadając niezarejestrowaną wersję Game Makera, niektóre funkcje mogą nie działać m.in. system Particle, obsługa dodatkowych funkcji typu gradienty w tle itd.

Struktura

Pisząc skrypt używamy wielu poleceń, zwanych wyrażeniami (ang. statements). Początek skryptu powinno się zacząć symbolem "{", a kończyć "}". Mimo to, na samym początku i na samym końcu rzadko się używa tych znaków, a ich brak nie jest uznawany przez Game Maker jako błąd. Po każdym wyrażeniu, stawiamy średnik ";", na wzór innych języków programowania. Nie dając go, program może uznać, że podane wyrażenie trwa dalej, np. w następnej linijce. Tak więc, nasz skrypt będzie wyglądał mniej więcej tak:

```
{
<wyrażenie>;
<wyrażenie>;
...
}
```

Wyrażeń może być bardzo wiele. Kilka z nich zostanie wymienionych w dalszej części artykułu, ale większość z nich znajdziesz w pliku pomocy do Game Maker.

Zmienne

Podstawowe informacje

GML jak wiele innych języków programowania zawiera zmienne. Służą one do zapamiętywania przez program pewnych wartości, np. położenia obiektu w room. Zmienna może zawierać liczbę lub tekst (string). Zmienne w Game Makerze są deklarowane trochę inaczej niż w znanych językach programowania, ale też większość jest już wbudowana, np. *mouse_x* lub *mouse_y* określające położenie myszki. Nazwa zmiennej musi zaczynać się literą i może zawierać litery, liczby i podkreślenia "_" (maks. długość – 64 znaki).

Deklarowanie

Zmienne w GML deklarujemy w poniższy sposób:

```
<zmienna> = <wartość>;
```

Deklarować można w taki prosty sposób, jednak można też bardziej skomplikowanie. Jeżeli chcemy dodać jakąś wartość do obecnej wartości np. 2 do wartości 5, to zamiast = dajemy +=. Podobnie jest z odejmowaniem -=, mnożeniem *=, dzieleniem /= lub używając operatorów bitowych |=, &\ lub ^=.

Inne rodzaje zmiennych

W GML możemy używać poza tymi prostymi zmiennymi także innych. Jeżeli użyjemy normalnych, to będą one przechowywane tylko w jednym objekcie. Jeżeli chcemy utworzyć zmienną ogólnodostępną, trzeba posłużyć się tzw. globalnymi. Zmienne globalne od zwykłych różnią się przedrostkiem *global.* w nazwie. Zmienna globalna może wyglądać tak:

```
global.nazwa=1;
```

Czasami jednak, chcemy, żeby zmienna była tylko używana w jednym skrypcie, a nie do każdego skryptu w danym objekcie. W takim przypadku, zmienną musimy najpierw zadeklarować w *var*, a potem ustalamy jej wartość, jak w innych przypadkach. Może to wyglądać tak:

```
{
var abc,def;
abc=1;
def=2;
}
```

Adresowanie zmiennych

Czasami do skryptu potrzebujemy znać zmienną zadeklarowaną w innym objekcie. Po przeczytaniu powyższych informacji, każdy by to zrobił na podstawie stworzenia globalnej o wartości tamtej. Nic bardziej błędnego. W takich przypadkach wykorzystujemy adresowanie zmiennych. Przyjmijmy, że chcemy zmienić wartość zmiennej x obiektu pilka. Piszemy wtedy tak:

```
pilka.x=2;
```

Tutaj poznajemy pierwszą metodę adresowania – poprzez podanie nazwy obiektu. Należy pamiętać o kropce między nazwą obiektu i nazwą zmiennej. Możemy także adresować nie podając nazwy obiektu. Wtedy piszemy:

- *self* : obiekt w którym jest wykonywana akcja
- *other* : obiekt, który jest w trakcie kolizji z obiektem w którym wykonywana jest akcja
- *all* : wszystkie obiekty
- *noone:* żaden z obiektów (brzmi dziwnie, ale czasami się przydaje)
- *global* : żaden z obiektów, ale tworzy zmienną globalną (patrz poprzedni podroździał)

Zmienne możemy również adresować, podając id obiektu (w nawiasie), np.

```
(100032).x=250;
```

Jednak skąd brać id obiektu? Tworząc go, dostaniemy jego id, np.

```
obiekt=instance_create(250,546,pilka);
```

Wtedy, już nie podając id, możemy adresować w poniższy sposób:

```
{
obiekt=instance_create(250,546,pilka);
obiekt.x=267;
}
```

Dla ułatwienia można korzystać z numeracji zmiennych, głównie przydatne przy tworzeniu kilku obiektów (lecz można to także użyć do już gotowych obiektów) i wtedy razem z adresowaniem, będzie to wyglądać tak:

```
{
obj[0]=instance_create(250,546,pilka);
obj[1]=ludzik;
obj[0].speed=5;
obj[1].y=555;
}
```

Tablice

W GML można używać tablic jedno i dwuwymiarowych. Tworzenie ich jest proste – poprostu wpisujemy wartość w nawiasy kwadratowe (przy jednowymiarowych jedną liczbę, przy dwuwymiarowych – liczbę, przecinek i drugą liczbę). Tablice mogą wyglądać tak:

```
{
a[0]=1;
b[1,3]=2;
}
```

Instrukcje warunkowe, pętli, itd.

Instrukcja warunkowa

Instrukcja warunkowa w GML ma formy:

```
if (<warunek>) <wyrażenie>
```

lub

```
if (<warunek>) <wyrażenie> else <wyrażenie>
```

Warunek może składać się z wielu funkcji, wtedy piszemy:

```
if (<warunek>)
{
<wyrażenie>
}
else
{
<wyrażenie>
}
```

Poprawnie zapisany przykład instrukcji warunkowej wygląda tak:

```
if (x < pilka.x)
{
speed=5;
pilka.speed=15;
}
else
{
speed=15
pilka.speed=5
}
```

Instrukcja pętli *repeat*

Instrukcja pętli repeat wygląda tak:

```
repeat (<wartość>) <wyrażenie>
```

Wartość ustala ile razy pętla ma być wykonywana i musi być podana jako liczba naturalna.
Przykład:

```
{
repeat (2) instance_create(random(400),random(400),pilka);
}
```

Instrukcja pętli *while*

Instrukcja pętli while wygląda tak:

```
while (<warunek>) <wyrażenie>
```

Instrukcja ta polega na tym, że wyrażenie (nawet składające się z wielu funkcji) jest wykonywane kiedy pewien warunek jest spełniony. Używając jej trzeba uważać, ponieważ można zapętlić coś w nieskończoność przez co gra może się zawiesić.
Przykład:

```
{
while (!place_free(x+32,y+32)) instance_create(x+32,y+32,pilka);
}
```

Instrukcja pętli *do*

Instrukcja pętli do wygląda tak:

```
do <wyrażenie> until (<warunek>)
```

Wyrażenie zawarte w tej pętli (nawet składające się z wielu funkcji) jest wykonywane tak długo, aż warunek zawarty w until będzie wykonany. Używając jej trzeba uważać, ponieważ można zapętlić coś w nieskończoność przez co gra może się zawiesić.
Przykład:

```
{
do instance_create(random(600),random(600),pilka) until
instance_number(pilka)=100;
}
```

Instrukcja pętli-warunkowa *for*

Instrukcja pętli-warunkowa for wygląda tak:

```
for (<wyrażenie1>; <warunek>; <wyrażenie2>) <wyrażenie3>
```

Wygląda skomplikowanie, nieprawdaż? Jednak to bardzo proste. Wygląda to mniej więcej tak:

- wyrażenie1 jest wykonywane;
- warunek jest sprawdzany;
- jeżeli jest prawdziwy, wyrażenie3 jest wykonywane;
- potem wyrażenie2;
- potem znowu od początku, aż warunek będzie fałszywy.

Jeżeli nie rozumiesz, to pomyśl tak. Wyrażenie1 inicjuje pętle for. Warunek sprawdza, czy pętla ma być zakonczona. Wyrażenie2, to takie "przeciągnięcie" pętli, które jest sprawdzane za każdym następnym razem po wykonaniu wyrażenie1.
Najpopularniejszym przykładem wykorzystania for jest tworzenie licznika z pewnym przedziałem liczbowym.
Przykład:

```
{
for (i=0; i<=9; i+=1) list[i] = i+1;
}
```

Inne instrukcje i wyrażenia

Instrukcja *switch*

Instrukcja switch wygląda tak:

```
switch (<warunek>)
{
case <warunek1>: <wyrażenie1>; ... ; break;
case <warunek2>: <wyrażenie2>; ... ; break;
...
default: <wyrażenie>; ...
}
```

Działa to tak:

- warunek jest sprawdzany;
- sprawdzane są pozostałe warunki;
- jeżeli jeden z warunków jest spełniony, wyrażenia są wykonywane, aż do wystąpienia break;
- jeżeli żaden warunek nie jest spełniony, jest wykonywane wyrażenie w default (nie jest wymagany).

Można też korzystać z tzw. multiple case (wielokrotnych case). Wtedy kolejny case dajemy w miejsce wyrażenia. Także break nie jest potrzebny. Jeżeli nie ma break, to kod po prostu jest wykonywany dalej.
Przykład:

```
switch (keyboard_key)
{
case vk_left:
case vk_numpad4:
x-=4; break;
case vk_right:
case vk_numpad6:
x+=4; break;
}
```

Wyrażenie *break*

Wyrażenie break wygląda tak:

```
break
```

W przypadku użycia tego kodu z pętlami, bądź instrukcją for lub with, zakończy daną pętlę lub wyrażenie. Jeżeli jest użyty poza nimi, kończy działanie programu (nie kończy gry).

Wyrażenie *continue*

Wyrażenie continue wygląda tak:

```
continue
```

W przypadku użycia tego kodu z pętlami, bądź instrukcją with, będzie kontynuować działanie kodu z następną wartością dla pętli lub instrukcją with.

Wyrażenie *exit*

Wyrażenie exit wygląda tak:

```
exit
```

Wyrażenie to po prostu kończy działanie skryptu. (Nie kończy ono działania gry! Jak chcesz zakończyć grę, musisz użyć funkcji game_end();).

Funkcje

Funkcja składa się z nazwy funkcji po której są podane arguments w nawiasie, rozdzielane przecinkami.

```
<funkcja>(<argument1>,<argument2>, ...);
```

W Game Makerze mamy dwa typy funkcji. Pierwsze, to spora kolekcja wbudowanych funkcji, do kontrolowania wszystkiego co się dzieje w grze. Drugi typ to każdy skrypt zdefiniowany przez ciebie (w zakładce scripts). Możemy ich także używać jak funkcje.
Musisz pamiętać, że jak nie wpisujemy do funkcji arguments, to zostawiamy nawiasy! Niektóre funkcje zwracają wartości (np. variable_global_exists(nazwa);) i mogą być wtedy używane jako wyrażenia. Pozostałe wykonują po prostu polecenia.
Także musisz pamiętać, że funkcje nie mogą być adresem zmiennej. Jeżeli już, to zapisujemy funkcję w nawiasie np. (instance_nearest(x,y,obj)).speed=0;.

Arguments

Tworząc skrypty, możesz zaimplementować do nich arguments. Są one przechowywane w zmiennych argument0, argument1...argument15. W Game Maker możemy zaimplementować aż do 16 arguments (w przypadku korzystania z tzw. klocka, możemy zaimplementować tylko 5 argumentów). Skrypty z arguments uruchamiamy na wzór funkcji (patrz wyżej).

Zwracanie wartości przez skrypt

Wcześniej pisałem o tym, że funkcje mogą zwracać pewną wartość jak również, że skrypty zdefiniowane w zakładce scripts, to też funkcje. Więc teraz, jak tworzyć zwracanie wartości przez skrypt? To proste. Wykorzystujemy wtedy instrukcję return, która wygląda tak:

```
return <wyrażenie>
```

Trzeba pamiętać, że return automatycznie kończy działanie skryptu!
Przykład:

```
{
return (argument0*argument0)
}
```

Konstrukcja with

Jak już pisałem wcześniej, możliwe jest ustalanie lub sprawdzanie wartości zmiennej zawartej w innym objekcie, czyli tzw. adresowanie. Dla przykładu, chcesz żeby wszystkie obiekty pilka przesunęły się o 8 pikseli w dół. Zgodnie z tym co napisałem wcześniej, będziesz myślał, że można to zapisać tak:

```
pilka.y = pilka.y + 8;
```

Jednak jest to zapis nieprawidłowy. Dlaczego? Otóż, będzie pobrane położenie Y jednej z piłek i dodane do niego 8. Zajdzie taki proces, że w końcu, wszystkie piłki będą w tej samej lini. Też wyrażenie

```
pilka.y += 8;
```

doprowadzi do tego samego efektu. Więc co robić? Wtedy korzystamy z instrukcji with. Wygląda ona tak:

```
with (<wyrażenie>) <polecenie>
```

<wyrażenie> to obiekt na którym ma być wykonywane polecenie. Możesz tam dać id obiektu lub jego nazwę (jeżeli wszystkie mają zareagować). Można też użyć jeden ze "specjalnych" obiektów (all, self, other, noone). <polecenie> jest wykonywane dla wszystkich obiektów z osobna, co zapobiega takim błędom jak powyżej. Więc, jak chcesz przesunąć

piłki o 8 pikseli w dół, to możesz użyć

```
with (pilka) y+=8;
```

Możesz też korzystać z kilku poleceń na raz. Wtedy dla przykładu, żeby przesunąć każdą piłkę w losową pozycję damy:

```
with (pilka)
{
x=random(room_width);
y=random(room_height);
}
```

Pozostałe przykłady wykorzystania tej instrukcji znajdziesz w pliku pomocy Game Makera (dokumentacji) w: The Game Maker Language (GML) -> GML Language Overview -> With construction.

Komentarze

Teraz coś prostego i użytecznego, ale zrozumiałego tylko dla programisty :-) . Pomówmy teraz o komentarzach. Więc, wszystko w linii napisane po // nie będzie odczytywane przez program np.

```
x = 25 // x = 44 – to już nie jest odczytywane przez program
```

Jednak, jak przejdziemy do następnej linii, to komentarz już nie będzie działał. Co robić, jak chcemy mieć komentarze wieloliniowe? Jest to prosta rzecz. Dajemy wtedy tekst między /* i */. Przykład:

```
x+=44
/*
if x=0 {
show_message('tego program nie przeczyta');
}
*/
show_message('ale to już program przeczyta');
```

Nie wierzysz? Sprawdź.

Porady dotyczące dalszej nauki

- Nie bój się zaglądać do pliku pomocy. Został on napisany po to, żebyś mógł łatwo i szybko znaleźć odpowiednią funkcję. Wystarczy po odpaleniu tzw. helpa wejść w Wyszukaj lub Indeks i wpisać po angielsku co nas interesuje, np. gdy chcemy poszukać funkcji związanych z rysowaniem, wpisujemy w Indeks draw i pojawia nam się spis funkcji. Wtedy wystarczy kliknąć dwukrotnie i odpali nam się roździał gdzie mamy opisane to co nas interesuje. Jeżeli nie umiesz angielskiego na przynajmniej średnim poziomie, zajrzyj do słownika. Opisy zazwyczaj są krótkie, przez co ich tłumaczenie nie powinno zająć dużo czasu.
- Jeżeli nie umiesz zrobić jakiejś rzeczy, nie martw się. Osobiście polecam metodę prób i błędów, w końcu dotrzesz do rozwiązania problemu. Jak już na prawdę nie masz pomysłów, przeszukaj artykuły i przykłady na naszej stronie lub na oficjalnej stronie Game Makera. Jest to prawdziwa skarbnica wiedzy o tworzeniu gier. Zawsze też możesz przeszukać forum, ew. się na nim zapytać, ale pamiętaj: obowiązkiem forumowiczów nie jest odpowiadanie na pytania. Jeżeli odpowiadają, to tylko z własnej, dobrej woli, dlatego nie powinieneś być bardzo nachalny.
- Pisząc skrypty, zawsze możemy się posiłkować podpowiedziami. Pod miejscem gdzie piszemy kod, jest taki szary prostokąt. Pojawiają się tam funkcje, zmienne itd. które odpowiadają temu co zacząłeś pisać, np. pisząc draw_tex, pojawi się tam draw_text(x, y, string). Czasami jest to bardzo pomocne, szczególnie gdy piszemy skrypty z pamięci bez posiłkowania się plikiem pomocy.
- GMLa, tak jak każdego innego języka skryptowego/programowania nie da się nauczyć w jeden dzień i sądzić że się wszystko umie, tylko po przeczytaniu pomocy. Najważniejsza jest praktyka. Czasami nawet pisząc dużo skryptów i posiłkując się plikiem pomocy można się więcej nauczyć niż tylko czytaniem go. Dlatego, powtórzę jeszcze raz – korzystaj z metody prób i błędów. Jest to najlepszy sposób, żeby się nauczyć dobrze języka.

Spis przydatnych wiadomości przy pisaniu skryptów

- Kiedy chcemy ustawić alarm na konkretną ilość sekund, nie strzelaj nigdy. Alarmy są w tzw. stepach. Jest to zależne od szybkości rooma (room speed). Domyślnie wynosi ona 30, więc idąc drogą domysłu, 30 stepów to 1 sekunda. Jednak, jeżeli nie chce nam się liczyć albo nie mamy domyślnej wartości, warto skorzystać ze zmiennej room_speed. Przechowuje ona informacje o szybkości rooma. Wtedy możemy ustawić alarm wpisując tak: room_speed*<ilość sekund>. Przykładowo, po wpisaniu room_speed*5 alarm będzie wykonany po 5 sekundach.
- Staraj się używać jak najmniej zmiennych, jak najczęściej korzystać z arguments oraz ograniczać ilość

eventów do minimum. Na przykład, zamiast tworzyć eventy do poszczególnych klawiszy, lepiej napisać w step odpowiednią konstrukcję if.

- Zamiast tworzyć sprite i dać mu animację gdzie się obraca o 360 stopni, lepiej zastosować zmienną image_angle. Wtedy tworzymy jeden sprite zwrócony w prawo i obracamy go. Korzystanie z image_angle wygląda tak: image_angle=<stopnie>. Przykładowo, gdy chcemy obrócić sprite o 240 stopni wpisujemy: image_angle=240. Można też to mieszać z innymi zmiennymi/funkcjami w których dane są zapisane w stopniach, np. image_angle=direction.
- Jak chcesz sprawdzić czy skrypt nie zawiera błędów, nie odpalaj gry. Szybciej jest sprawdzić naciskając na Check the script for syntax errors. Oczywiście sprawdza on tylko składnię i poprawność napisania funkcji, zmiennych czy nazw, to warto z niego korzystać. Ewentualnie można uruchamiać grę w Debug Mode (czerwona ikonka obok Run Game).

Autor: Marmot (aka BP Ultimate)