Poradnik do RGSS. Część II

Zaawansowany kurs języka RGSS

:: Autor ::
Dawid „Pajper" Pieper

:: email
dawidpieper@o2.pl

www.rpgmaker.pl

Poradnik do RGSS. Część II

Zaawansowany kurs języka RGSS

Ruby to interpretacyjny język programowania, zorientowany obiektowo. Dzięki tym cechom jest zdecydowanie prostszy w obsłudze, niż na przykład język C++.
Kurs ten jest poświęcony tematyce głównie WINAPI, czyli bibliotekom dołączonym do systemu operacyjnego Windows.
Prócz tego pojawią się również informacje o zaawansowanych funkcjach samego Ruby'ego, a dokładniej RGSS – RUBY GAME SCRIPTING SYSTEM, który to język jest implementowany w programie RPG MAKER firmy Enterbrain.
Kurs będzie skupiony na tak zwanej pierwszej edycji RGSS, zaimplementowanej w programie RPG MAKER XP.
Minimalna biblioteka wymagana, by działały wszystkie skrypty przedstawione w tym kursie, to RGSS102E.dll.
Mimo, że kurs jest poświęcony RMXP, większość skryptów zadziała również w innych wersjach rpg makera, gdyż jest w nich wykorzystany ten sam silnik RGSS.
Uwaga! Większość skryptów przedstawionych w tym kursie będzie wymagała systemu windows XP lub nowszego. Nie przewiduję bowiem, by ktoś jeszcze używał systemu Win2000.

Jest to kontynuacja poprzedniego kursu: „Wstęp do programowania w RGSS".
Jak poprzednio – poradnik piszę dla Twierdzy RPG Makera. Bardzo dziękuję Reptile'owi za możliwość wrzucenia go na stronę.

Oczywiście, jak zawsze, zapraszam wszystkich czytelników na Twierdzę RPG Makera (http://rpgmaker.pl). Można tam odnaleźć wiele skryptów, materiałów graficznych i dźwiękowych, scenariuszy, zadań i innych materiałów, które na pewno przydadzą się podczas tworzenia gry komputerowej i nie tylko komputerowej.
Kurs ten można swobodnie wrzucać na dowolne strony. Pozostaje tylko jeden warunek, a właściwie dwa:
1. Nie zmienisz jego zawartości.
2. Zaznaczysz, kto jest jego autorem i że pochodzi on z Twierdzy RPG Makera.

Życzę miłej lektury!

Dawid Pieper.
dawidpieper@o2.pl
www.rpgmaker.pl

~ SPIS TREŚCI ~

~ ROZDZIAŁ 1 ~
Podstawy WINAPI

~ ROZDZIAŁ 2 ~
Domena Windowsa
OKIENKA

~ ROZDZIAŁ 3 ~
Grafika

~ ROZDZIAŁ 4 ~
Zasoby

~ ROZDZIAŁ 5 ~
Zarzadzanie procesorem i pamięcią

~ ROZDZIAŁ 6 ~
O obsłudze różnych urządzeń i okien jeszcze kilka słów

ROZDZIAŁ 1
Podstawy WINAPI

W tym kursie poruszę zaawansowane tematy, dotyczące programowania zarówno w RUBY/RGSS jak i w językach, w których RGSS napisano. Mam tu na myśli język DELPHI oraz C++.

Prosiłbym o przeczytanie poprzedniej części kursu przed tą częścią, w celu pełnego zrozumienia.

Będę tutaj oznaczał - w celu większej przejrzystości dokumentu – najważniejsze rzeczy kolorem czerwonym (składnie poleceń zielonym) dodając przy tym ikonkę, która podpowie funkcję kolorowego tekstu.

IKONA	FUNKCJA
	**Ostrzeżenia**
	**Większe fragmenty kodu**
	**Składnie poleceń**
	**Skrypty**

Co to jest biblioteka .DLL?

W poprzednim kursie używaliśmy wielokrotnie bibliotek *.DLL. Nie wyjaśniłem jednak dokładnie, czym one są, dlatego teraz to objaśnię.

Biblioteka *.DLL (DYNAMIC LINK LIBRARY) to rodzaj pliku zawierającego dane skrypty. Przypominają one już poznane moduły, ale nie są plikami tekstowymi, a kodem maszynowym ze strukturami kodu dwójkowego. Oznacza to, że nie można ich edytować np. w notatniku. Mogą też być użyte w dowolnym języku programowania. Na przykład biblioteka napisana w C++ może być bez problemów użyta w RGSS.

Przykładowo takimi bibliotekami są biblioteki RGSS. Zostały one napisane w C++ przez twórców RUBY, a następnie edytowane w DELPHI przez firmę ENTERBRAIN.
Biblioteki te mogą występować w kilku rodzajach:

Rodzaj	Rozszerzenia plików	Opis
Biblioteka dynamicznie linkowana	*.DLL	Najpopularniejszy rodzaj tych bibliotek. Jest obsługiwany przez prawie każdy język programowania. Taka biblioteka jest uruchamiana w trakcie działania programu. Jest uruchamiana dopiero po wydaniu jej polecenia i wyłączana zaraz po jego zakończeniu. W razie jej braku możemy poczynić odpowiednie kroki przez obsługę wyjątków.
Biblioteka nagłówkowa (.a) statycznie linkowana	Składa się z dwóch plików: LIB*.A oraz *.H	Biblioteka ta jest linkowana statycznie. Oznacza to, że jest podłączona do programu od samego początku jego działania. W razie jej braku nie jesteśmy w stanie obsłużyć wyjątku, gdyż program się nie uruchomi. Ponadto wymagany jest plik *.H, w którym zapiszemy wszystkie funkcje biblioteki. Gdy ktoś go wyedytuje, cały program przestanie działać, a dzięki niemu łatwo zmienić działanie programu, zmieniając funkcje biblioteki. Dlatego nie zalecam jego używania i nie będę się więcej nim zajmował.
Biblioteka statyczna .LIB	*.LIB	To, co u góry. Jedyną różnicę stanowi fakt, iż w przypadku takich bibliotek nie zawsze jest potrzebny plik *.H, a są one dostarczane wraz z systemem (WINAPI), przez co są bezpieczniejsze.
Biblioteka *.IDL	*.IDL	Rodzaj bibliotek używanych na przykład w pytonie. Nie będę o nich pisał, gdyż RGSS ich nie obsługuje.
Biblioteka TLB	*.TLB	Inny format *.DLL lub *.LIB. Nigdy go nie używałem, więc nie wiem, czy jest kompatybilny z RGSS. Na pewno jest kompatybilny z C++.

Skoro już wiesz, czym jest biblioteka, możemy zająć się jej wdrażaniem.

Wdrażanie bibliotek

W poprzedniej części poradnika popełniłem pewien błąd. Teraz przeanalizuję od nowa moduł Win32API, który jest używany do wdrażania bibliotek DLL w RGSS.

Składnia

```
Uchwyt = WIN32Api.new(lokalizacja_biblioteki,
adres_funkcji_biblioteki, rodzaje_parametrów, parametry)
Uchwyt.call(parametry)
```

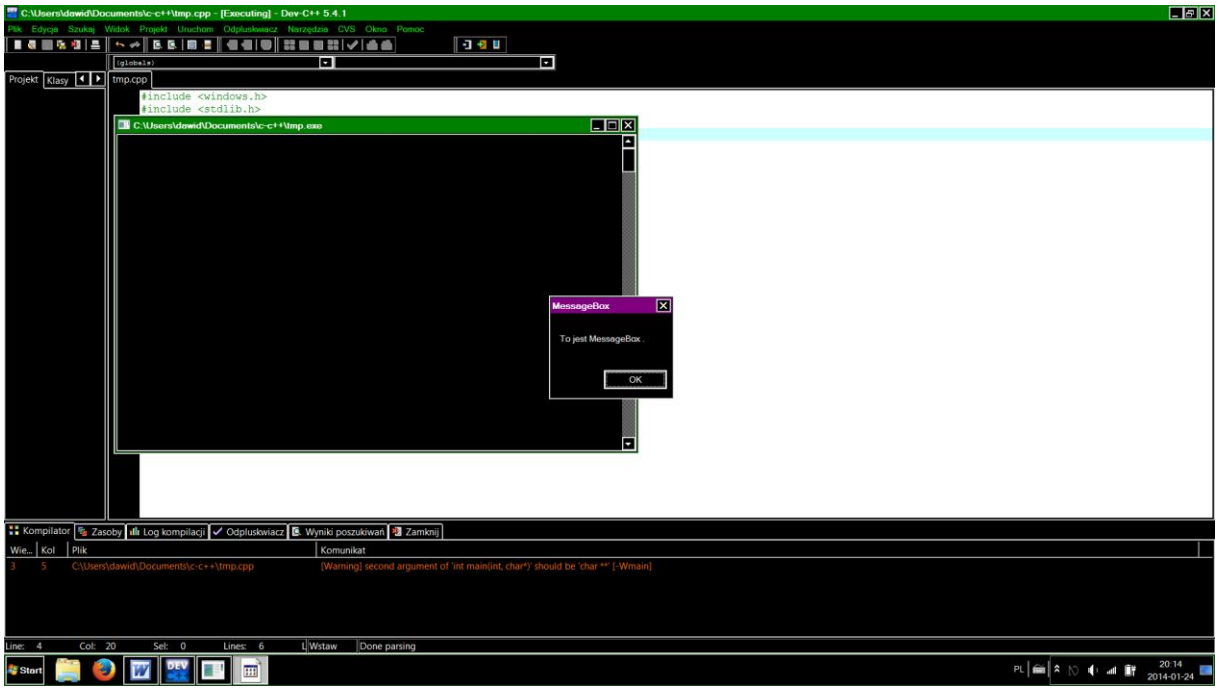
Najpierw zajmiemy się pierwszą, ważniejszą linijką kodu.

Argument	Znaczenie
Lokalizacja_biblioteki	Ścieżka dostępu do biblioteki bez rozszerzenia *.DLL
Adres_funkcji	Nazwa przywoływanej funkcji
Rodzaje_parametrów	O tym za chwilę
Parametry	Parametry przywoływanego polecenia

Zamiast „uchwyt” wpisz nazwę zmiennej z twoim poleceniem. Stanie się ona klasą/modułem z podfunkcją .call. Podfunkcja ta służy do wywołania biblioteki. Dobrze, o co chodzi z tymi rodzajami parametrów?
Każde polecenie przy bibliotece ma swój rodzaj. Na przykład: liczbę, uchwyt… Wywołując bibliotekę w C++ nie musimy podawać rodzajów parametrów, podobnie przy bibliotekach statycznych. Jednak w tym przypadku trzeba. Oto dostępne argumenty:

Parametr	Odpowiedniki C++	Opis
I	INT, DWORT, HWND, CHAR	Głównie liczby i wyrażenia TRUE/FALSE
L	LONGINT	Długie liczby (zwykle ponad 16384)
P	LPSTR	Wskaźnik na tekst lub inną zmienną (zwykle zmienną typu bufor)
V	void	Brak parametrów / pusty parametr

Korzystając z powyższej wiedzy możemy teraz wyświetlić jakiś tekst – napisać własny print. Użyjemy w tym celu funkcji MessageBox systemowej biblioteki user32.
Przykładowy MessageBox może wyglądać tak (C++):



Składnia polecenia biblioteki:

```
Int MessageBox(HWND uchwyt_okna_rodzica, LPSTR tekst, LPSTR tytuł,
UINT flagi)
```

Więc to przeanalizujmy. Najpierw musimy zainicjować nasz MessageBox. Zaglądamy wyżej i widzimy składnię Win32API. Najpierw wpisujemy ścieżkę do biblioteki. MessageBox to funkcja biblioteki user32.dll. Biblioteka ta jest biblioteką systemową, więc nie trzeba jej pobierać, wystarczy wpisać jej nazwę. Nazwa funkcji to oczywiście MessageBox. Teraz parametry!
Jak widać są cztery: okno rodzica, tekst, tytuł i flagi. Parametry wpisujemy w jednym ciągu znaków. Pierwszy parametr jest typu HWND, więc wpisujemy I. Drugi i trzeci to LPSTR, czyli P. Ostatni to UINT, czyli typ int – I. Ostatni parametr to wartość zwracana. Funkcja zaczyna się słowem int, więc parametr zwracany to 'I' – pamiętaj o apostrofach!
Więc ostatecznie deklaracja wygląda tak:

```
Mb = Win32API.new("user32", "MessageBox",'IPPI', 'i')
```

Mam nadzieję, że z inicjacją bibliotek nie ma problemu. Jedynym, co może być problemem jest napis mb. Skąd on się wziął? mb to zmienna inicjacji biblioteki. Jako że to zmienna definicyjna, jeśli chcesz jej użyć poza definicją, użyj zmiennej klasy, instancji lub globalnej.
Co jednak z tego, że funkcje zainicjowaliśmy, skoro nie wiemy jak jej użyć? Już bierzemy się do roboty. Użyjemy tutaj wywołania uchwytu mb.call. Mb to nazwa zmiennej inicjacji, a call pochodzi od angielskiego słowa oznaczającego przywołanie, wywołanie… Na początku zainicjowaliśmy cztery parametry, dlatego funkcja mb.call też przyjmie cztery. Pierwszy argument to uchwyt okna rodzica. Jest to okno, które zostanie przykryte nowym oknem. Nie posiadamy jednak uchwytu okna gry, gdyż go nie pisaliśmy sami. Wpisujemy 0. Drugi argument to tekst, a trzeci tytuł. Wpisujemy je jako normalne ciągi znaków. Nie polecam używania polskich liter, gdyż nie pokażą się

one. Później napiszę, jak je dodać. Czwarty argument to flagi. Na razie żadnych flag nie chcemy, więc piszemy 0. Cała funkcja wygląda na przykład tak:

```
Mb.call(0, "Testujemy MessageBox", "TEST", 0)
```

Niestety, nasz MessageBox ma pewną wadę. Gdy go szybko nie zamkniemy, pokaże się błąd: "Script is changing". By tego uniknąć, spróbuj dodać pętlę graficznej aktualizacji (Graphics.update).
Dobra, MessageBox działa. Teraz opracujemy flagi. Pozwalają one na drobną modyfikację MessageBox'a.
Pamiętaj, że MessageBox to funkcja biblioteki user32. Dlatego w przypadku dystrybucji gier pod system linux, wrzuć ją do folderu z grą. Znajdziesz ją w katalogu system32\.

Funkcja MessageBox – flagi i zwracana wartość

W poprzednim rozdziale omówiliśmy funkcję MessageBox jako przykład funkcji do wdrożenia. Funkcja ta sama z siebie – w podanej wyżej postaci –do niczego się nie przyda. Komu się chce inicjować biblioteki i przetwarzać jeszcze kody znaków i obsługę liter polskiego alfabetu, jeżeli istnieje wygodna i łatwa do użycia funkcja print? O wiele rozsądniej jest pójść na łatwiznę. Nie oznacza to jednak, że MessageBox się nie przyda. Nie musi on bowiem mieć tylko przycisku OK, może mieć ich więcej. W celu ich dodania, modyfikujemy parametr "flagi". Oto lista możliwych do dołączenia przycisków podstawowych:

Flaga	Przyciski
0	OK
1	OK, anuluj
2	Ponów próbę, ignoruj, przerwij
3	Tak, nie, anuluj
4	Tak, nie
5	Ponów próbę, anuluj
6	Anuluj, próbuj, kontynuuj

Więc jak widać możliwości jest kilka. Ponadto istnieje możliwość dodania przycisku Pomoc, ale nie jest on łatwy do odczytania przez sam program, więc się nim na razie nie zajmujemy.
Do funkcji MessageBox możemy jeszcze dodać ikonki. Poza znakiem graficznym, podczas wyświetlania usłyszymy odpowiedni dźwięk.

Flaga	Ikona
64	Informacja
0x30	Ostrzeżenie
16	Błąd
32	Pytanie
128	Użytkownika (potem pokażę jak ją dodać)
0	Brak

Mamy więc również ikonkę. Istnieje jeszcze coś takiego, jak waga. Określa ona, czy komunikat ma przykrywać jakieś okna. Może łatwiej wytłumaczę to informując od razu, co jak działa.

Flaga	Działanie
0	Zwykłe okno
0x20000	Na wierzchu pulpitu
0x2000	Najważniejsze okienko otwartej aplikacji
4096	Okienko wagi systemowej

Na razie tyle flag wystarczy. Może kiedyś wypiszę wszystkie, to wystarczy przeciętnemu użytkownikowi. Maski i przyciski podstawowe nie są ci jeszcze potrzebne.
Jak jednak to połączyć? Przecinkami nie można, bo program to potraktuje jako osobne parametry!
Każdą flagę od innej oddzielamy znakiem kreski pionowej | . Oczywiście nie możemy połączyć kilku flag z jednej grupy, na przykład wstawić do MessageBox'a zarówno opcji tak i nie, jak OK i anuluj, chyba logiczne.
Dobrze, mamy nasz wymarzony MessageBox, ale co nam po nim, skoro nie wiemy, co zostało kliknięte?
Dlatego funkcji mb.call przypisujemy zmienną. Będzie ona przetrzymywać to, co użytkownik kliknął. Robimy to tak:
Klikniecie = mb.call(…)
Zmienna kliknięcie przetrzymuje teraz to, co zostało wybrane przez użytkownika.
Niestety funkcja ta nie zapisuje wybranej opcji jako napisu lecz jako liczbę i trzeba wiedzieć, jaki numer oznacza dany przycisk.

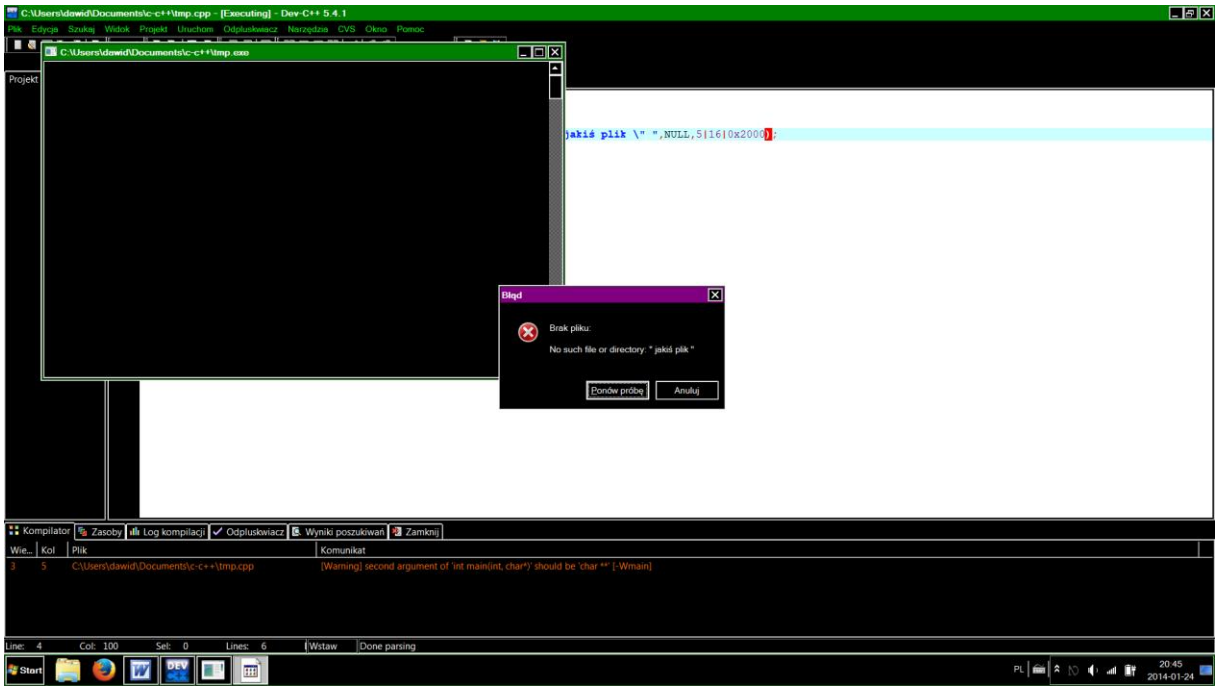
Numer	Przycisk
0	Wystąpił błąd przy przetwarzaniu polecenia
1	OK
2	Anuluj
3	Przerwij
4	Spróbuj
5	Ignoruj
6	Tak
7	Nie
8	Kliknięcie w krzyżyk zamykający okienko
9	Pomoc
10	Próbuj ponownie
11	Kontynuuj

Dobrze! Skoro jesteśmy tak napełnieni wiedzą, spróbujmy zrobić coś praktycznego. Po co jednak pisać skomplikowane skrypty po nic? Wyedytujemy przetwarzanie błędu braku pliku tak, by przy błędzie użytkownik był pytany, co zrobić. Skrypt wklejamy poza jakąkolwiek klasą.

```
Rescue Errno::ENOENT
filename = $!.message.sub("No such file or directory - ", "")
mb = Win32API.new("user32", "MessageBox", 'ippi', 'I')
retry if odp = mb.call(0, "Brak pliku: #{filename}", nil, 5|16|0x2000) == 4
end
```

Mhm, co tu tłumaczyć? Większość kodu jest wyjaśnionego w tym kursie lub kursie podstawowym. Jedyne, co może zastanawiać, to konstrukcja #{filename}. Zwraca ona ścieżkę do brakującego pliku w przypadku tego błędu.
Jak użytkownik kliknie "ponów próbę", program ponowi próbę odczytu dzięki omawianemu już poleceniu retry.
W przeciwnym wypadku okienko zostanie zamknięte.

Przykład powyższego okienka wygląda tak (C++):



Przypominam tylko, by nie stosować polskich znaków w Win32API dopóki nie dowiesz się, jak.

System szesnastkowy (heksadecymalny)

Zapewne zauważyłeś, że we flagach funkcji MessageBox występują takie, które zaczynają się od 0x…. Są to liczby systemu szesnastkowego. Jako że będziemy ich coraz częściej używać, postanowiłem zapoznać Cię z tymże systemem liczbowym.
System szesnastkowy – jak sama nazwa wskazuje – składa się z szesnastu cyfr. Jako że tylu cyfr normalnie się nie używa, liczby od jedenastej zapisane są kolejnymi literami alfabetu.
Liczby w tym systemie wyglądają tak:

Zapis dziesiętny (decymalny)	Zapis szesnastkowy (heksadecymalny)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F
32	20
33	21
34	22
35	23
36	24
37	25
38	26
39	27
40	28
41	29
42	2A
43	2B
44	2C
45	2D
46	2E
47	2F
48	30

Jak już się zapewne domyśliłeś, zapis liczby szesnastkowej poprzedzamy znakiem 0x. Przykładowo by zapisać liczbę 32, piszemy 0x20.

Dlaczego jednak podkreśliłem liczbę 32? Ponieważ jest to najmniejsza liczba szesnastkowa, używana w programowaniu. Wynika to stąd, że liczba ta jest łatwa do zapamiętania w systemie szesnastkowym (0x20), a jednocześnie jest wielokrotnością liczby 2, co jest niezbędne z powodu użycia systemu dwójkowego w procesorach komputerów.

Liczba	Wielokrotność potęgowa dwójki	Zapis szesnastkowy
2	1	0x2
4	2	0x4
8	3	0x8
16	4	0x10
32	5	0x20
64	6	0x40
128	7	0x80
256	8	0x100
512	9	0x200
1024	10	0x400
2048	11	0x800
4096	12	0x1000
8192	13	0x2000
16384	14	0x4000
32768	15	0x8000
65536	16	0x10000
131072	17	0x20000
262144	18	0x40000
524288	19	0x80000
1076576	20	0x100000

Dlaczego wypisałem tę tabelę? Przyda się ona jeszcze na przykład przy binarności plików. Osobiście zalecam nauczenie się tych wartości do potęgi dwudziestej, tak jak zapisałem, oczywiście nie za "jednym zamachem".

Poza tymi systemami istnieje jeszcze zapis ósemkowy, gdzie liczbę poprzedzamy zerem. Jest rzadko używany, wstawiam tabelę dla poglądu i na wszelki wypadek - gdyby miała się ewentualnie kiedyś przydać.

System ósemkowy	System dziesiętny
01	1
02	2
03	3
04	4
05	5
06	6
07	7

010	8
011	9
012	10
013	11

Pobieranie danych z Internetu

W wielu grach jest możliwość wysyłania swoich wyników do sieci, inne zgłaszają dostępność aktualizacji, a jeszcze inne posiadają Internetowy tryb multiplayer. Jak coś takiego osiągnąć?
Gry zgłaszające dostępność aktualizacji oraz te, które posiadają funkcję najlepszych wyników, wykorzystują protokoły http i bazy danych, czyli zachowują się jak normalne strony. Gry multiplayer wykorzystują tak zwane gniazda (WINSOCK), którymi teraz nie będę się zajmował – zacznijmy od czegoś łatwiejszego.

Nie zajmę się również bazami danych, wspomnę tylko, że są dwa sposoby ich obsługi:
1. Łączenie się z bazą danych w samym skrypcie RGSS – sposób niebezpieczny, gdyż jeśli ktoś zdobędzie kod źródłowy, będzie już znał nasze hasła.
2. Przekierowanie do skryptu PHP i obsługa bazy z jego poziomu.

Jeśli znasz dostatecznie PHP, po przeczytaniu tej lekcji i tak będziesz w stanie użyć drugiego podanego sposobu, tak więc do rzeczy!
Do łączenia się z Internetem możemy użyć bardzo wielu sposobów, między innymi dołączać się do gniazda http, obsłużyć moduł OPENURI… Ja pokaże sposób najłatwiejszy i najkrótszy – użyję biblioteki URLMON, a dokładniej polecenia URLDownloadToFile. Oto składnia tegoż polecenia:

```
HRESULT URLDownloadToFile(DPUNKNOWN Active_X, LPCSTR źródło, LPCSTR
cel, DWORT 0, LPBINDSTATUSCALLBACK pasek-stanu)
```

Jak zapewne zauważyłeś, czwarty argument w RGSS zawsze musi być równy 0.
Tak więc pobierzmy sobie przygotowany plik z Twierdzy RPG MAKER. Zapiszemy je pod nazwą twierdza_newsy.html.
Najpierw zainicjujmy naszą bibliotekę.

```
$download = Win32API.new("urlmon", "URLDownloadToFile", 'pppip',
'I')
```

Poznałeś kilka nowych rodzajów zmiennych. Jak widać HRESULT w RGSS zastępujemy znakiem 'I'.
Dobra, weźmy się za trudniejszą część i pobierzmy w końcu tę stronę.

Pierwszy argument polecenia wskazuje na obiekt ACTIVE_X. Jako, że obiektu takiego nie chcemy – może kiedyś je omówię – piszemy nil. Teraz źródło. Pobierzmy newsy z jakiejś strony internetowe, więc drugi argument to http://rpgmaker.pl/inne-pliki/news-test.txt. W trzecim argumencie wpiszemy "twierdza-newsy.html". Czwarty argument to oczywiście 0. Jako że nie omawialiśmy jeszcze pasków stanu, piąty argument to nil.
Wynik wygląda tak:

```
$download.call(nil,http://rpgmaker.pl/inne-pliki/news-
test.txt,"twierdza-newsy.html",0,nil)
```

Jak widać, obsługa pobierania plików nie jest taka trudna. Należy jednak pamiętać, by nie pobierać plików w trakcie gry w ten sposób, gdyż w trakcie pobierania praca programu jest wstrzymywana. Gdy nie ma dostępu do Internetu lub plik nie istnieje, nie zostanie utworzony plik – cel. Możemy to sprawdzić tak:

```
$download = Win32API.new("urlmon", "URLDownloadToFile", 'pppip', 'I')
$download.call(nil, http://rpgmaker.pl/inne-pliki/news-test.txt, "twierdza-
newsy.html", 0, nil)
If FileTest.exist?("twierdza-newsy.html")
Print("Plik testowy z wiadomościami z twierdzy rpg maker został pobrany.
Zostanie on otwarty po kliknięciu przycisku 'OK'")
System('start twierdza-newsy.html')
System('exit')
Else
Print("Nie udało się pobrać pliku. Sprawdź czy masz dostęp do Internetu i
spróbuj ponownie.")
End
```

Jak widać, wystarczy sprawdzić czy plik istnieje. Myślę, że nie muszę szczegółowo omawiać tego kodu, wszystko tu użyte już wyjaśniłem.

Pamiętaj, że z powodu nie wykonywania tych operacji w tle, przy pobieraniu zbyt dużych danych program zwróci komunikat Script is changing i zostanie zamknięty!

Dodać należy, że URLMON to kolejna biblioteka systemowa. Występuje ona w każdym systemie od systemu Windows 2000, a nie od początków systemu – jak było w przypadku biblioteki user32. Należy o tym pamiętać i napisać o tym w wymaganiach gry lub skopiować odpowiednie biblioteki do folderu z grą: URL.DLL oraz URLMON.DLL. Wszystkie znajdują się w %systemroot%\system32
Po zakończeniu pracy programu – jeśli chcemy zaoszczędzić miejsce na dysku – możemy usunąć wszystkie pliki, które pobraliśmy, a często jest ich naprawdę dużo. Inną metodą jest natychmiastowy odczyt każdego pobieranego pliku i zapisywanie jego treści w zmiennej, a następnie usuwanie go. Osobiście polecam pobieranie wszystkich plików do folderu z grą \ temp, a przy obsłudze wychodzenia z aplikacji dodać skrypt usuwający tenże folder.

Możemy też skopiować wszystkie nasze pliki do folderu temp systemu Windows. Lokalizacja do folderu temp znajduje się w stałej %temp% .

Uwaga! Nie pobieraj plików z nazwami zawierającymi polskie znaki, bo jeszcze nie wiesz, jak to robić, a przy standardowych ustawieniach i tak ci się nie uda.

Przy podawaniu ścieżek w DLL często używa się dwóch backslashów zamiast jednego, w celu uniknięcia problemów z błędnym odczytem stringu. Dla pewności możesz zawsze użyć znaku / .

Kopiowanie, przenoszenie i usuwanie plików oraz folderów

Już pokazywałem, jak wykonywać te operacje, używając systemowego wiersza poleceń. Miało to jednak kilka wad. Przede wszystkim pojawiało się okienko konsoli. Ponadto w celach kopiowania – z nieznanych mi przyczyn – trzeba utworzyć na niektórych systemach nowy plik wsadowy. Znakomicie! Chcąc stworzyć bardziej zaawansowane menu zapisu gry musimy stworzyć dziesiątki plików *.BAT lub *.CMD. Nawet jeśli nam to nie przeszkadza, rozmiar gry się zwiększy, a przy lekkiej edycji – nawet przypadkowej – tych plików, cały system gry straci rację bytu. Ponadto, gdy plików będzie około setki więcej, instalacja gry potrwa znacznie dłużej. Dlatego bezpieczniej jest użyć bibliotek systemowych i w spokoju skopiować pliki. Ponadto zawsze jest możliwość dodania jakichś pasków postępu i bycia przygotowanym na obsługę wyjątków, co przy plikach *.BAT jest znacznie trudniejsze.
W celu kopiowania plików użyjemy kolejnej systemowej biblioteki: kernel32. Minimalny system, na którym możemy kopiować pliki, to Windows xp – tylko aplikacje tzw. Desktopu.
Polecenie używane w celu kopiowania plików to – jak nietrudno się domyślić – CopyFile:

```
BOOL CopyFile(LPSTR źródło, LPSTR cel, BOOL co robić w razie
istnienia pliku)
```

Pierwsze dwa argumenty są chyba jasne. Dodam jeszcze tylko przypomnienie, że backslashe trzeba pisać podwójnie lub zastąpić slashami. Trzeci argument mówi poleceniu, czy ma zastąpić istniejący plik. Jeśli jest równy 0, plik zostanie zastąpiony, w przeciwnym wypadku kopiowanie nie powiedzie się, a polecenie zwróci wartość 0. Wartość inna od zera oznacza powodzenie. Przykładowo kopię pliku Uruchamiającego grę do innego folderu wykonuje się tak:

```
Cf = Win32API.new("kernel32", "CopyFile", 'ppi', 'I')
Cf.call("game.exe","c:\\game.exe",1)
```

Program skopiuje plik wykonywalny gier RPG MAKERA XP na dysk C:, myślę, że nie trzeba tu więcej wyjaśniać.
Poza kopiowaniem plików, często chcemy je przenosić. Oczywiście możemy najpierw plik skopiować, a potem usunąć oryginał, ale po co dodawać nowe linijki kodu, skoro istnieje polecenie MoveFile. Ma ono jednak jedną wadę. Jeśli plik docelowy istnieje, nie podmieni go.

```
BOOL MoveFile(LPSTR źródło, LPSTR cel)
```

Jak widać, polecenie to przyjmuje tylko dwa parametry. Chyba i tu nie trzeba wiele omawiać, oto mały przykładzik:

```
Mf = Win32API.new("kernel32", "MoveFile", 'pp', 'I')
Mf.call("game.rxproj","c:\\game.rxproj")
```

#Polecenie to może być również używane do zmieniania nazw plików

```
Mf.call("Game.rxproj","projekt_gry.rxproj")
```

Jak widać, dzięki temu poleceniu nie trzeba się męczyć z nowymi poleceniami do zmiany nazwy pliku.
Czas na ostatnie polecenie, które chcę omówić w tej części kursu, a mianowicie mowa o DeleteFile - jakież te nazwy skomplikowane!
Polecenie to ma tylko jeden parametr.

```
BOOL DeleteFile(LPSTR ścieżka)
```

Jak widać, w przypadku tego polecenia jeszcze jest łatwiej. Oto i przykład:

```
Df = Win32API.new("kernel32","DeleteFile",'p','i')
Df.call("Game.rxproj")
```

Chyba nie mam już nic do napisania o tych plikach, teraz zajmiemy się tworzeniem i usuwaniem katalogów. Tym razem okażemy się destrukcyjni i zaczniemy od usuwania, gdyż jest łatwiejsze. Nazwa jest niezrozumiała i nielogiczna: DeleteDirectory.

```
BOOL DeleteDirectory(LPSTR katalog)
```

Jak widać, parametr jest tylko jeden.

Pamiętaj jednak, że możesz usuwać tylko te katalogi, które są puste, to znaczy nie ma w nich żadnych danych.

Trudniej jest z funkcją CreateDirectory.

```
BOOL CreateDirectory(LPSTR nazwa, LPSTR atrybuty bezpieczeństwa)
```

Pierwszy parametr to nazwa tworzonego katalogu. Jeśli kursor przeglądu nie był zmieniany, nowy katalog zostanie utworzony w folderze z grą. Możesz też podać ścieżkę do nowo tworzonego katalogu. Drugi parametr to atrybuty bezpieczeństwa, ograniczające dostęp do folderu. Na razie wpisz tam nil. Już wkrótce się nimi również zajmiemy, gdyż jest to bardzo ciekawy temat.

To chyba tyle na ten temat, ale nie ma tak lekko. Kontynuujemy omawianie plików i folderów.

Tworzenie plików i ich otwieranie

Skoro już zajmujemy się plikami, to teraz stworzymy i odczytamy jakiś plik w WINAPI. Niewątpliwą zaletą tego sposobu jest możliwość edycji ich w notatniku, nie są one zapisywane binarnie, jak pliki metody Marshall, a stają się normalnymi plikami tekstowymi. Jak nietrudno się domyślić, użyjemy funkcji – jaka niespodzianka – CreateFile. Minimalny system użycia tej funkcji to Windows XP. Funkcja znajduje się w bibliotece, która zwykle plikami i procesami się zajmuje: kernel32. Oto też jej składnia.

```
HANDLE CreateFile(LPSTR nazwa, DWORT dostęp, DWORT współdzielenie,
LPSTR atrybuty bezpieczeństwa, DWORT parametry tworzenia, DWORT
flagi i atrybuty, HANDLE uchwyt pliku wzoru atrybutów)
```

Jak widać, parametrów jest całkiem sporo, teraz spróbujmy je przenieść i zapisać w formie odczytu Win32API. LPSTR to oczywiście P, a DWORT to I. Gorzej z parametrami HANDLE. Parametr HANDLE to uchwyt do danego pliku, zapiszmy go jako I. Dodam jeszcze, że radzę flagi plików zapisać jako P, gdyż jest ich sporo.
Wynik prac – inicjacja:

```
createfile = WIN32API.new("kernel32","CreateFile",'piipili','l')
```

Dobrze, teraz zajmiemy się inicjacją. Pierwszy argument to nazwa (lub ścieżka) do tworzonego pliku. Pamiętaj o podwójnej pisowni backslashów.
Teraz ustalamy dostęp. Kiedy napiszemy 0, możemy tylko czytać, 1 oznacza tylko zapis, natomiast 2 odczyt i zapis, proste.
Teraz zajmiemy się trybem współdzielenia pliku. Określa on, czy inne programy mają mieć możliwość zmian w naszym pliczku, dotyczy tylko chwil, gdy mamy uruchomiony program. Gdy go zamkniemy, straci to ważność.

Wartość	Znaczenie
0	Brak dostępu
1	Tylko odczyt
2	Tylko zapis
4	Pozwala na usuwanie pliku innym aplikacjom
1|2|4	Pełny dostęp

Naturalnie jeśli nie ma innej konieczności, radzę napisać 0, by inne programy nie grzebały w naszym pliku. Atrybuty bezpieczeństwa oczywiście ustawiamy na nil.
Teraz czas na sposób tworzenia. Mówi on interpreterowi, czy ma usunąć istniejący plik, czy zrobić coś innego.

Liczba	Co się stanie

1	Tworzy nowy plik, jeśli ten istnieje, przerywa działanie i zwraca błąd
2	Zawsze tworzy nowy plik, jeśli ten istnieje, nadpisuje go
3	Otwiera plik, jeśli ten istnieje. W przeciwnym wypadku generuje błąd
4	Otwiera istniejący plik, jeśli ten nie istnieje, to go tworzy
5	Zeruje otwierany plik, ale tylko jeśli on istnieje. W przeciwnym wypadku, przerywa działanie.

Teraz czas na flagi. Przypominam, że jeśli interesuje nas wiele flag, oddzielamy je znakiem kreski pionowej | . Oto lista flag.

Flaga	Działanie
268435456	Dostęp swobodny wskaźnika, więcej wkrótce
67108864	Plik zostanie usunięty po zamknięciu go lub aplikacji
33554432	W trakcie używania pliku, wykonuje jego kopię bezpieczeństwa
2097152	Otwiera plik z punktu odzyskiwania kopii bezpieczeństwa
0x20000000	Nie tworzy Bufora z pliku, nie kopiuje go do pamięci RAM
134217728	Dostęp sekwencyjny wskaźnika pliku – więcej poniżej
0x40000000	Plik nie jest do odczytu/zapisu. Służy do synchronizacji wejścia/wyjścia

Tych flag jest wiele więcej, ale wiele z nich się nie przydaje, a inne nie wiem do czego służą. Jakby ktoś był ciekawy, w MSDN, czyli dokumentacji WINAPI, znajduje się lista wszystkich flag. Poza tym w dziale z flagami możemy ustalać atrybuty pliku. Oto i one:

Numer	Atrybut
32	Archiwalny
16384	Plik jest spakowany
2	Ukryty
128	Brak atrybutów
4096	Plik nie do użycia przez Internet, OFFLINE
1	Tylko do odczytu
4	Plik systemowy
256	Plik tymczasowy

Ostatni argument to uchwyt do otwartego już pliku z atrybutami. Jeśli wpiszemy tam uchwyt pliku, atrybuty z niego zostaną skopiowane. Jeśli chcemy tego uniknąć, piszemy nil.
Jeśli funkcja się powiedzie, zwróci uchwyt do otwartego / utworzonego pliku. Używając tego uchwytu, będziemy w stanie edytować treść pliku. Oto przykładzik.

```
createfile = Win32API.new("kernel32","CreateFile",'piipili','l')
@pliczek =
createfile.call("temporaryfile.tmp",1,2,1|2|4,nil,0,256|67108864,ni
l)
```

Jak widać, polecenia bibliotek systemowych – jak zwykle – dają nam większe możliwości niż wbudowane polecenia RGSS. Ponadto w razie dalszej edycji pliku, jeśli tak chcemy, nie będzie on binarny, co pozwoli nam na jego używanie w innych programach. Dzięki możliwości tworzenia plików tymczasowych, możemy tworzyć różne tablice wyników, bestiariusze itd., gotowe do podglądu lub wydruku.
Co jeszcze można napisać o otwieraniu plików? Już wiem!
Co prawda nie chcesz teraz używać plików ani do odczytu, ani do zapisu. Dlatego warto albo nie uzyskiwać uchwytu, co spowoduje tylko jego stworzenie, a nie otwarcie. Jednak wkrótce zajmiemy się obróbką plików. Dlatego warto będzie przyzwyczaić się do uzyskiwania uchwytów i zamykania pliku, by nie spowalniać gry dzięki śmieciom w pamięci RAM. Plik zamykamy w ten sposób.

```
CloseHandle(HANDLE uchwyt)
```

Jak widać, polecenie to ma tylko jeden argument, którego na dodatek nie muszę omawiać. Jedyną wadą tego polecenia jest fakt, że trzeba je osobno inicjować (HANDLE to I). W wartości zwracanej użyj 'V'.
Ponadto, co już pokazałem, z otwartym plikiem możesz zrobić kilka przydatnych rzeczy, a nie tylko od razu go zamykać. Na przykład sprawdzić jakie ma atrybuty lub czas i datę modyfikacji.
To pierwsze robimy funkcją GetFileAttributes:

```
GetFileAttributes(LPSTR ścieżka)
```

Jak większość funkcji z zarządzania plikami, znajduje się ona w bibliotece kernel32.dll.
Naturalnie możemy zmienić atrybuty pliku bez otwierania go:

```
SetFileAttributes(LPSTR ścieżka, DWORT atrybut)
```

Pierwsza funkcja zwraca liczbę oznaczającą atrybut, druga go zastosowuje. Liczby oznaczające atrybuty są takie same, jak w powyższej tabelce.
Tak wygląda skrypt, który sprawdza atrybuty pliku Game.ini i zastosowuje je na pliku Game.rxproj:

```
Sprawdzatrybut = Win32API.new("kernel32","GetFileAttributes",'I','l')
Ustawatrybut = Win32API.new("kernel32","SetFileAttributes",'il','I')
Ustawatrybut.call("Game.rxproj",Sprawdzatrybut.call("Game.ini"))
```

Jak widać, nie zawsze trzeba używać szablonów plików i specjalnie użyć funkcji CreateFile.
Pokażę jeszcze, w jaki sposób sprawdzić datę zmiany pliku.
Jest to dość trudne. By to sprawdzić, musimy mieć otwarty plik w trybie do odczytu. Składnia:

```
GetFileTime(HANDLE uchwyt, LPSTR struktura czasu stworzenia, LPSTR
struktura czasu otwarcia, LPSTR struktura czasu modyfikacji)
```

A co to takiego struktura?
Struktura to zbiór podzmiennych przechowujących jakieś dane. Dobra wiadomość! Ta funkcja utworzy struktury automatycznie od dat. Oto przykład, jak to sprawdzić. Przyjmuję, że plik został załadowany do zmiennej plik.

```
Data = Win32API.new("kernel32","GetFileTime",'ippp','I')
Stworzenie = date.at(0)
Odczyt = date.at(0)
Edycja = date.at(0)
Data.call(plik, stworzenie, odczyt, edycja)
```

Jak widać, większych problemów z tymi funkcjami nikt nie powinien mieć, jeśli dotąd wszystko rozumiał. Myślę, że temat prostego zarządzania plikami możemy uznać za zakończony. Na razie opuścimy tematykę plików, by zająć się buforem.

Bufory

Bufory – co to takiego? Bufor to fragment pamięci ram, zarezerwowany specjalnie dla wykonania danej operacji. Ma określony rozmiar. W przeciwieństwie do zwykłej zmiennej, dokładnie zakładana jest jego pojemność, a on bez przerwy znajduje się w bazie danych, utrzymując łączność z każdym urządzeniem. Bufory przykładowo są niezbędne, jeśli chcesz napisać nagrywarkę plików na płyty, gdzie niezbędna jest zmienna połączona z dyskiem, z którego pliki pochodzą i bez przerwy nagrywająca je na krążku CD/DVD/BLURAY. Jednak zanim zabierzesz się za tak zaawansowane zagadnienia, bufory przydadzą ci się do wielu łatwiejszych operacji. Na przykład do odczytu danych z plików, gdzie taki bufor musi mieć ciągłą łączność z programem, odczytywanym plikiem, dyskiem oraz strukturą, do której treść jest przekazywana. Jak widać bufory są bardzo przydatne. Mimo to używaj ich rzadko i przemyśl to, gdyż bufory są jednym z najłatwiejszych sposobów na przepełnienie pamięci ram i uszkodzenie systemu, kończące się restartem komputera. Z tej przyczyny złośliwe oprogramowanie często operuje, dodając setki olbrzymich buforów. Dlatego jak już użyjesz buforu, możesz go skasować. Inną propozycją jest utworzenie jednego lub dwóch buforów, których będziesz używał w aplikacji – zawsze tych samych. W ten sposób zmniejszysz ryzyko niestabilnej pracy w systemie.

Dobrze, dość już teorii, jak wprowadzić ten bufor? Niespodzianka! RGSS coś potrafi bez Win32API i nie używamy do tego zewnętrznych bibliotek! Możemy to zrobić używając tylko RGSS. Dzięki temu nie trzeba inicjować kolejnych poleceń i marnować wielu linijek kodu. Tworzenie bufora jest bardzo proste, chodź trzeba to przemyśleć. Oto skrypt tworzący prosty bufor o pojemności 1024 bajtów.

```
B = "\0" * 1024
```

Bufor ma pojemność 1024 bajtów, oznacza to, że zajmuje 1kB pamięci RAM – niedużo, lecz pamiętaj, że wiele aplikacji tworzy setki takich buforów, a to już jest dużo. Utworzony bufor może przechowywać liczby. Od -1023 do 1023. Może też przechować napis o długości 1023 znaków. Pozostały jeden znak w liczbach oznacza 0, a w napisach zerową długość bufora. Warto jest tworzyć

minimalne bufory, takie, które akurat wystarczą na nasze potrzeby. W ten sposób ograniczamy ryzyko problemów wynikłych z braku pamięci RAM.
Przed użyciem bufora powinniśmy jeszcze usunąć wszystkie znaki zerowe, by i one nie zostały odczytane.

```
Zmienna_bufora.delete!("\0")
```

Jak już pisałem, po użyciu bufora możemy go usunąć. Zaskoczeniem może okazać się informacja, że usuwanie buforów jest trudniejsze niż ich tworzenie. Najpierw radzę – nie jest to konieczne – ustalenie dla bufora wartości NIL. W ten sposób unikamy ryzyka utraty kontroli nad danymi. Gotowe, bufor nie będzie już nam przeszkadzał, został usunięty.
Teraz zajmiemy się praktycznym zastosowaniem buforów, żeby nie było, że twórca tego kursu uczy niepotrzebnych i w dodatku ryzykownych rzeczy.

Odczyt plików *.INI

Zanim zajmiemy się odczytem jakichkolwiek plików, zaczniemy od łatwiejszych plik *.INI.
Co to jest plik *.INI? Plik *.INI jest również nazywany plikiem konfiguracyjnym, gdyż zwykle służy do zapisywania ustawień. Plik *.INI dzieli się na tak zwane grupy ustawień. Każda grupa zawiera już dane ustawienia. Każde ustawienie ma jakąś wartość. Może pokażę przykład? Jest to przykładowy plik Game.ini, czyli plik konfiguracyjny gier RPG MAKERA XP.

```
[Game]
Library=RGSS102E.dll
Scripts=Data\Scripts.rxdata
Title=temporary
RTP1=
RTP2=
RTP3=
```

Mhm, widać, że plik ten ma tylko jedną grupę ustawień: Game. Poszczególne grupy ustawień znajdują się w nawiasach kwadratowych. Każda linijka oznacza jedno kolejne ustawienie, którego wartość podawana jest po znaku równości = . Tekstu nie należy wpisywać w cudzysłowach, zawsze będzie traktowany jako napis. Nie ma też potrzeby unikania spacji, możemy ich używać bez ryzyka, że program je odczyta jako odstęp między liniami. Innym specyficznym plikiem *.INI jest autorun.inf. Co prawda nie jest to rozszerzenie konfiguracyjne, ale działa tak samo. Plik ten definiuje, co ma się stać po włożeniu płyty CD, na przykład uruchamia instalator gry. Oto budowa pliku autorun.inf.

```
[autorun]
Start=plik_do_uruchomienia.exe
Icon=ikonka_dysku.ico
Action=napis_przy_ikonce_uruchamiania
Label=wyświetlana_nazwa_dysku
```

Dla zainteresowanych napiszę, że plik ten należy wrzucić do głównego folderu nagrywanego dysku. Dlaczego piszę jednak o plikach autorun.inf? Tworząc instalatory możemy użyć takich plików razem z dyskiem. W pliku autorun.inf zdefiniujemy wersję aplikacji itp. Przydaje się to przy instalatorach, pobierających dane z Internetu. Możemy zdefiniować, co mają pobrać, używając plików autorun.inf. Dzięki temu przy wydawaniu nowych wersji gry nie trzeba zmieniać wielu skryptów. Nie trzeba też tworzyć dodatkowych plików, świetne rozwiązanie. Oto przykład takiego pliku autorun.inf.

```
[autorun]
Start=autorun.exe
Icon=autorun.ico
Action=Instalacja pakietu gry (nazwa gry)
Label=(nazwa gry)
[gamedata]
Version=0.0.0
Download-url=(adres z plikami)
Auto-update=true
Register-key=46A84BDE77F53CC7301847AA
Lang=polish
```

Jak widać, tworzenie takich plików nie jest zbyt trudne. No dobrze, już możemy zapychać swobodnie nasz dysk plikami konfiguracji. Nic nam jednak po nich, jeśli nie wiemy, jak je odczytać. Już idę z pomocą, a raczej idą programiści Microsoft, którzy dodali taką przydatną funkcję o nazwie GetPrivateProfileString do biblioteki – jakież zaskoczenie – kernel32.dll. Z przykrością stwierdzam, że jak zwykle wszystko idzie po górkę, gdyż nie dość, że funkcja wymaga bufora, to jeszcze jest strasznie skomplikowana. Ponadto oczekuje od programisty, że będzie on wiedział, jak długi napis chce pobrać – to akurat łatwo ominąć, pokażę jak. Oto i składnia tej cudownej funkcji!

```
DWORT GetPrivateProfileString(LPSTR nazwa-aplikacji, LPSTR nazwa-
klucza, LPSTR domyślność, LPTSTR wskaźnik-bufora, LPCSTR długość-
tekstu, LPSTR plik)
```

Ostrzegałem, funkcja jest dość skomplikowana. Zacznijmy od inicjacji. Pojawia się tu nowy rodzaj wartości: LPCSTR. Jest to – w tym przypadku – wskaźnik na bufor. Możemy go potraktować tak, jak LPSTR. Zainicjujmy więc tą biblioteke w Win32API.new!

```
Ini =
Win32API.new("kernel32","GetPrivateProfileString",'pppppp','I')
```

Jak widać, prawie wszystkie parametry są wskaźnikami. Teraz spróbujmy wypełnić tą funkcję.
Pierwszy argument wskazuje na nazwę czytającej aplikacji – w teorii. Programiści Microsoftu doszli do wniosku, że plik *.INI będzie grupowany według nazw aplikacji. Założenie to okazywało się słuszne, do póki nie powstały zaawansowane aplikacje i gry. Teraz jeden plik *.INI może być używany przez jedną aplikację, a mieć wiele grup, na przykład: Graphics, Audio, Data, Options, Player, Classes, 3D, Language, Package, Registration itp. Jak widać więc, założenie na dłuższą metę nie okazało się słuszne. Krótko mówiąc, polecenie to wskazuje na grupę ustawień, z której chcemy czytać. Wspaniale! Przeszliśmy przez pierwszy argument! Zostało tylko pięć! Możemy iść dalej!
Drugi argument to ustawienie, które chcemy odczytać z danej grupy. Myślę, że to jest logiczne. Teraz będzie gorzej.

Trzeci argument to wskazanie błędu. Jest to jedno z nielicznych ustawień, w którym wybieramy, co ma zwrócić w razie błędu. Krótko mówiąc, to co tu wpiszemy, zwróci polecenie, jeśli się nie powiedzie. Proponuję wpisać tu '', czyli po prostu nic nie zwróci.
Kolejny argument to bufor, który musimy wcześniej utworzyć.

Jeszcze następny argument to długość tekstu do odczytania. Aby się nie męczyć z liczeniem, wystarczy wpisać maksymalny rozmiar bufora, a funkcja i tak nie przeczyta więcej, niż znajdzie. Pamiętaj tylko, że rozmiar bufora o pojemności 256B pomieści 255 znaków, a nie 256. Wartość tego polecenia nie może być większa niż wartość bufora, jakież to logiczne!
Ostatni argument teoretycznie powinien być pierwszy. Okazuje się jednak, że pierwsi rzeczywiście będą ostatnimi. Jest to plik *.INI, który chcemy odczytać.
W tej chwili nasze polecenie jest już gotowe. Oto przykład, przy pomocy którego możesz odczytać nazwę gry.

```
Ini = Win32API.new("kernel32", "GetPrivateProfileString", 'ppppp', 'I')
Nazwa = "\0" * 64
Ini.call("Game","Title",'',nazwa,63,'Game.ini')
$nazwa_gry = nazwa
Nazwa.delete!("\0")
Nazwa = nil
```

Myślę, że z odczytem *.INI nie będziesz miał więcej problemów. Pozostaje jednak jeszcze zapis. Po co komu plik *.INI, skoro nie potrafi w nim pisać? Oczywiście można wczytać cały plik *.INI, zmodyfikować tylko zmienną z odpowiednim ustawieniem, a następnie wszystko po kolei zapisać. Jest to jakiś sposób, ale przy jego użyciu, sama idea plików *.INI traci sens, możemy równie dobrze użyć funkcji Marshall. Dlatego powinieneś nauczyć się również je edytować. Używamy do tego funkcji WritePrivateProfileString – dosyć rzadko spotykane, zwykle funkcje pobierające zaczynają się od słowa Get, a zapisujące Set. Tutaj jest troszkę inaczej, może programistom Microsoftu się nudziło…?
Oto i składnia tej funkcji.

```
BOOL WritePrivateProfileString(LPCSTR nazwa-aplikacji, LPCSTR
nazwa-klucza, LPCSTR wartość, LPCSTR plik)
```

Myślę, że wszystko jest jasne. Nazwa aplikacji to grupa ustawień, a wartość to zmieniana zawartość ustawienia. Myślę, że to jasne, ale na wszelki wypadek napiszę: tym poleceniem nie można dodawać grup ustawień.
To chyba wszystko na ten temat, zajmijmy się czymś innym.

Odczyt danych z plików, zapis danych do plików

Powoli zaczynają się nudzić te pliki. Dlatego jeszcze zajmę się dwiema kwestiami i kończymy ten nudny temat.
Teraz zajmiemy się odczytem danych z pliku. Oczywiście czytać możemy tylko z plików otwartych. Funkcja czytająca pliki nazywa się niezwykle tajemniczo: ReadFile.
Oto pierwszy wstęp do tortur – składnia funkcji.

```
BOOL ReadFile(HANDLE uchwyt-pliku, LPVOID bufor, DWORT długość-tekstu, LPWORD bufor-przeczytanych, LPOVERLAPPED struktura OVERLAPPED)
```

Radosna informacja: potrzebujemy tylko dwóch buforów!
Pierwszy argument to uchwyt do otwartego już pliku. Drugi parametr to bufor dla odczytywanych znaków. Trzeci oznacza liczbę znaków, które chcemy przeczytać. Czwarty z kolei argument to bufor zawierający liczbę przeczytanych znaków. Piąty argument to struktura OVERLAPPED. Takiej struktury nie chcemy, ustawiamy ".
Odczytajmy sobie teraz plik Game.ini jako jedną, gigantyczną zmienną.

```
createfile = Win32API.new("kernel32","CreateFile",'piipili','l')
gameini = createfile.call("Game.ini",1,2,1|2|4,nil,4,0,nil)
readfile = Win32API.new("kernel32","ReadFile",'ipiip','I')
b = "\0" * 16384
bp = "\0" * 16384
gameinitext = readfile.call(gameini,b,16383,bp,'')
b.delete!("\0")
bp.delete!("\0")
b = nil
bp = nil
print(gameinitext)
```

Myślę, że z powyższego skryptu wszystko wynika. Tekst odczytanego pliku jest zwracany przez funkcję ReadFile. Teraz zajmiemy się jeszcze zapisem danych do pliku. Jak zwykle w WINAPI, łatwiej jest pisać niż czytać.
Dodam jeszcze, że w RGSS istnieje taka fajna funkcja:

```
IO.readlines(ścieżka do pliku)
```

Zwraca ona nam siatkę, zaczynającą się od 0, która to siatka zawiera wszystkie linijki danego pliku. Czasem więc winapi staje się niepotrzebne.
Do zapisywania plików używana jest funkcja WriteFile. Składnia funkcji jest identyczna, a że autor tego kursu lubi sobie upraszczać życie jak tylko można, nie przepisze jej. Wspomni on tylko, że przy zapisie dane do zapisania znajdują się w pierwszym buforze – drugim poleceniu. Od razu pokażę przykład użycia tejże funkcji. Zapiszemy do pliku plik.txt słowo kurs.

```
createfile = Win32API.new("kernel32","CreateFile",'piipili','l')
plik = createfile.call("plik.txt",2,2,1|2|4,nil,2,0,nil)
writefile = Win32API.new("kernel32","WriteFile",'ipiip','I')
b = "\0" * 8
b= "kurs"
bp = "\0" * 1024
writefile.call(plik,b,8,bp,'')
b = nil
bp = nil
```

Myślę, że wyjaśniłem na ten temat wszystko. Jeszcze tylko jeden temat o wskaźnikach plików i możemy ten temat uznać za skończony!

Wskaźniki plików

Często chcemy odczytać jakąś treść z pliku, która znajduje się w innym miejscu, niż na początku. Czytając treść pliku, przesuwamy jego wskaźnik. Dzięki temu używając ReadFile możemy ów wskaźnik przesunąć, ale po co tworzyć kolejne bufory?
Istnieją dwa sposoby zapisywania wskaźnika pliku: swobodny i sekwencyjny.
Ten drugi pozwala na przesuwanie wskaźnika prostym poleceniem. Dzięki temu możemy przelecieć pół pliku bez zbędnego bufora wielkości np. 100000 bajtów.
Żeby móc używać dostępu sekwencyjnego, należy dodać odpowiednią flagę z tabelki kilka rozdziałów wyżej.
Oto i składnia funkcji przesuwającej wskaźnik.

```
DWORD SetFilePointer(HANDLE wskaźnik uchwytu pliku, LONG liczba
bajtów, o które chcesz przesunąć wskaźnik, LONG dystans górny,
DWORD sposób przesunięcia)
```

Pierwsze dwie funkcje są chyba logiczne. Zaraz wyjaśnię pozostałe, zainicjujmy tylko to polecenie.
Ach tak, zapomniałem napisać wcześniej. Funkcja działa od systemu – niech zgadnę – Windows XP?.

```
setfilepointer =
Win32API.new("kernel32","SetFilePointer",'ILLI','I')
```

Trzeci parametr możemy pominąć, wpisz tam 0.
Parametr ten wskazuje na resztę bajtów pliku. Jest wymagany przy plikach większych, niż 4GB, ja nie zakładam, że takie będziesz tworzył.
Ostatni parametr wskazuje, względem czego przesuwamy. Jeśli wpiszesz 0, przesuwasz względem początku pliku, jeśli wpiszesz 1, przesuwasz względem aktualnej pozycji, a jeśli wpiszesz 2, przesuwasz względem końca pliku.
Teraz przesuniemy wskaźnik przykładowego pliku o 16kB w przód – 16384 znaki.

```
setfilepointer.call(plik, 16384, 0, 1)
```

Myślę, że ten straszny temat plików mogę uznać za zakończony.
Przyznam się, że pliki mnie tak zmęczyły, że pomiędzy napisaniem tego, a poprzedniego tematu, minął jakiś miesiąc.
Dlatego pożegnam te przykre rzeczy i zajmę się czymś innym – dźwiękami.

Powiadomienia dźwiękowe

Dawno, dawno temu, w pierwszej części kursu podstawowego, pisałem, że napiszę w jaki sposób dodać własną kategorię dźwięków do gry.
Następne wspomnienia o dźwiękach pojawiły się trochę później, przy okazji funkcji Audio.typ_freeze.
Przy funkcji MessageBox odtwarzaliśmy dźwięki zależne od ikonki.
Co jednak, jeśli potrzebujemy danego dźwięku, a nie chcemy MessageBox'a?
Kilka najbliższych lekcji poświęcę tematowi dźwięków.
W tej krótkiej lekcji mam zamiar wyjaśnić, w jaki sposób dodać do gry dźwięki zdarzeń systemowych, takich jak na przykład błędy.
Naturalnie, moglibyśmy dodać dźwięki błędów i innych komunikatów do gry, ale przy różnych systemach brzmią one inaczej. Ponadto użytkownik mógł u siebie zmienić jakiś dźwięk.
Z pomocą przychodzi nam tu funkcja MessageBeep.
Jest to pierwsza od dawna funkcja, która nie jest z biblioteki kernel32.
To stara, dobra user32.

```
Int MessageBeep(UINT dźwięk)
```

Dźwięk to numer flagi z ikonką zwykłego MBoxa, NP..: ikonki błędu.
W ten sposób ów dźwięk odtworzymy.

```
Mbeep = Win32API.new("user32","MessageBeep",'I','I')
Mbeep.call(16)
```

Temat dość krótki, a i funkcja dość prosta. Teraz zajmiemy się czymś bardziej przydatnym i częściej używanym.

Funkcja dźwiękowa: Beep

W komputerach stacjonarnych znajduje się pewne urządzenie zdolne do wydawania dźwięków o różnych częstotliwościach. Często się odzywa w trakcie włączania komputera jako "piiip".
W laptopach dźwięki te są wydawane przez głośniki.
Urządzenie to bez trudu możemy wykorzystać do własnych potrzeb. W tym celu wystarczy odpowiednia funkcja biblioteki – jakież to nudne – kernel32.
Składnia funkcji jest dość prosta.

```
BOOL Beep(DWORD częstotliwość, DWORD czas)
```

czyli:

```
Beep = Win32API.new("kernel32","Beep",'ii','I')
```

Zwykła informacja z MSDN:
Minimum supported client
Windows XP [desktop apps only]
Minimum supported server
Windows Server 2003 [desktop apps only]

Teraz czas na jakiś przykładzik. W ten sposób spowodujemy zwykłe piknięcie.

```
Beep.call(440,1000)
```

Zapomniałem wspomnieć, że czas podajemy w milisekundach – 1s = 1000ms.
Częstotliwość podajemy w hercach.
Przy częstotliwościach prawie-milionowych dźwięku już nie słychać. O jego obecności możemy się przekonać tylko dzięki strasznemu bólowi uszu.
Pokażę teraz pewien przykład. Zastosowałem go w obecnym projekcie. Polega on na wznoszeniu się dźwięku i jednoczesnym zwiększaniu przeźroczystości obrazka. Kopiuję tutaj ów skrypt.

```
class Scene_Load
  def main
$game_system = Game_System.new
      #pominięto kod obsługi profilów I zapisu ustawień graczy
      if $shown == nil and $pomin != 1
$showm = Win32API.new 'user32', 'keybd_event', %w(l l l l), ''
$showm.call(18,0,0,0)
$showm.call(13,0,0,0)
$showm.call(13,0,2,0)
$showm.call(18,0,2,0)
sleep(3)
Graphics.update
sleep(2)
end
if $pomin != 1
    sprite = Sprite.new
    sprite.bitmap = RPG::Cache.picture("intro")
    sprite.opacity = 820
    opa = 820
    Graphics.transition
    Graphics.freeze
    beep = Win32API.new("kernel32","Beep",'ii','i')
  czest = 200
    for i in 0..79
```

```
      opa = opa - 8
      czest = czest + 10
      beep.call(czest, 5)
      sleep(0.01)
      Graphics.update
      sprite.update
      sprite.opacity = opa
                     end
       sprite.dispose
      end
    if $pomin != 1
      $scene = Scene_Title.new
     else
      sleep(1)
      $pomin = 0
      end
  end
end
```

Jak widać, skrypt może znaleźć pewne zastosowanie.
Jeśli ktoś czuje się na siłach, by grać w ten sposób melodie – tortura po wzięciu pod uwagę pauz i długości nut – dodaję tutaj tabelkę z częstotliwościami pojedynczych dźwięków w tonacji razkreślnej.

Nuta	Częstotliwość
C	261.6
CIS / DES	277.2
D	293.7
DIS / ES	311.2
E / FES	329.6
F	349.6
FIS / GES	370
G	391.9
GIS / AS	415.3
A	440
AIS / B / HES	466.2
H / CES	493.9

Osobom, które chcą tak grać, radzę napisać odpowiedni moduł, klasę lub definicję w celu ułatwienia sobie układania muzyki.
Teraz zajmiemy się prawdziwym odtwarzaniem dźwięków, a nie tylko prostymi komunikatami.
Ponadto użyjemy innej biblioteki niż kernel32!

Odtwarzanie plików *.WAV

Możemy tutaj użyć wielu poleceń, opiszę tu PlaySound.
PlaySound – sukces – jest poleceniem biblioteki winmm – Windows Multimedia. Jest to biblioteka obsługująca różne dźwięki, filmy, obrazki itp.
Funkcja działa od systemu Windows 2000 PROFESSIONAL.
Czas na to, co zwykle – składnię tej funkcji.

```
BOOL PlaySound(LPCSTR dźwięk, HMODULE uchwyt pliku zasobów, DWORD flagi)
```

Pierwszy parametr jest nazwą dźwięku, kluczem rejestru lub zasobem. Czym ma być, ustawia się we flagach. Domyślnie system szuka takiego klucza, a jeśli nie znajdzie, odtwarza plik o podanej nazwie. Jeśli wykorzystujemy zasoby, drugi parametr jest uchwytem do pliku z nimi – o zasobach będzie więcej.
Trzeci argument to flagi.

Flaga	Działanie
0x10000	Dźwięk jest aliasem do dźwięku systemowego
0x110000	Dźwięk jest kluczem dźwięku systemowego
0x20000	Dźwięk jest nazwą pliku lub do niego ścieżką
0x40004	Dźwięk jest zasobem pliku
0x80	Dźwięk jest aliasem aplikacji
0	Kierowanie aplikacją zostanie zwrócone dopiero po odtworzeniu dźwięku (domyślny) – nie polecam z racji script is changing. Lepiej już po dźwięku dodać pętelkę for.
1	Kierowanie aplikacją będzie dostępne w trakcie odtwarzania dźwięku
2	Domyślnie, jeśli dźwięk nie zostanie odnaleziony, odgrywany jest domyślny dźwięk systemu. Jeśli zastosujesz tą flagę, dźwięk ów nie zostanie odtworzony.
4	Dźwięk jest wskaźnikiem do pamięci MEMORY
8	Dźwięk będzie odtwarzany bez końca lub do chwili, gdy wywołasz tą funkcję bez parametru nazwy pliku.
16	W trakcie odtwarzania dźwięku, inne dźwięki nie zostaną zatrzymane

Teraz, jak już wszystko jest omówione, odtworzymy dźwięk o nazwie test.wav.

```
Psound = Win32API.new("winmm","PlaySound",'ppi','i')
Psound.call("test.wav",nil,0x20000|1|16)
```

Trzeba pamiętać, że przy zatrzymaniu tą funkcją dźwięków, zatrzymane zostaną tylko te, które zostały odtworzone przez tą funkcję.
Ułatwia to nam stworzenie nowego modułu.
Zastanówmy się, czy chcemy stworzyć moduł tła – BGM / BGS (dźwięk w pętli) czy dźwięk pojedynczy…
Stworzymy sobie Audio.TE – talk effect. Możemy tego używać do wstawiania dialogów.
Nie obsłużymy na razie _freeze, ale _play i _stop.

```
module Audio
psound = Win32API.new("winmm","PlaySound",'ppi','I')
def te_play(nazwa)
e = psound.call(nazwa,nil,0x20000|1)
return(e)
end
def te_stop
e = psound.call('',nil,4)
return(e)
end
```

```
end
```

Nie obsługujemy wysokości ani głośności, ale zawsze jest to coś. Później i wysokość, i głośność pewnie omówimy.
Poza tym przy dialogach wysokość nie będzie potrzebna.
Sposób wywołania:
Odtworzymy jakiś dźwięk:
Audio.te_play("Audio/TE/test.wav")
Zatrzymamy dźwięk:
Audio.te_stop
Istnieje również funkcja sndPlaySound w tej samej bibliotece. Ma ona tylko pierwszy i ostatni argument. Zawsze wskazuje na plik, więc pierwsze cztery flagi tabelki oraz 0x20 jej nie dotyczą.
W przypadku bardziej zaawansowanych operacji, należy stworzyć strukturę dźwiękową. Jako że struktury nie były na razie omawiane, prócz najprostszych, na razie o tym nie będę wiele pisał.
Tymczasem zakończę dział podstawowy i zajmiemy się okienkiem naszej gry.

Uwagi:
Funkcją zarówno PlaySound, jak i sndPlaySound, możemy odtwarzać tylko pliki *.wav. Jeśli na danym komputerze znajdują się odpowiednie kodeki – wystarczy Windows Media Player – możemy próbować odtworzyć również inne formaty, na przykład: *.MP3. Nie gwarantuję jednak rezultatu, wszystko zależy od systemu, funkcji i kodeków. Bezpieczniej jest, w ramach możliwości, konwertować wszystko do formatu wav, co zdecydowanie ułatwi pracę.

Przy odtwarzaniu dłuższych dźwięków, zalecam odtwarzanie asynchroniczne, gdyż inaczej po odtworzeniu ujrzymy komunikat "Script is changing". Proponuję zastosowanie pętli for z funkcją Graphics.update. Można też użyć sleep przedzielanego również Graphics.update. W takich chwilach żałuję, że w RGSS nie ma prostej i pełnej funkcji obsługi wątków, bo by się przydała.

Polskie znaki w Win32API

Długo miałem problem z polskimi znakami. Przez długi czas nie potrzebowałem funkcji tekstowych, używałem głównie bibliotek URLMON czy KERNEL32, a te zwykle nie przyjmują tekstu. Jak używałem MessageBoxów, układałem wiadomości tak, by polskie znaki omijać – lub od razu zmieniałem je, usuwając końcówki ą-a ć-c…
Pewnego dnia jednak postanowiłem wykorzystać Microsoft Speech API do udźwiękowienia gier. Wtedy zaczęły się problemy.
Szukałem wielu rozwiązań, doszło nawet do tego, że utworzyłem nową bibliotekę *.DLL, w której znaki były konwertowane. Efekt: error.
Szukałem rozwiązania miesiącami, a im dużej szukałem, tym mniej znajdowałem.
Silnik RGSS był częściowo pisany w DELPHI, postanowiłem więc spróbować tam szczęścia. I oto znalazłem taki skrypt DELPHI: UTF8ToSys("sprawdzamy działanie polskich znaków"). Efekt: ponownie error. Próbowałem więc wykorzystywać tak zwany unikom. Efekt: nie dość, że polskie znaki znikały, to jeszcze i łacińskie litery nie chciały się pojawić. W C++, a przynajmniej

w niektórych edycjach, przed ciągami tekstu dodaje się znacznik L w celu dodania obsługi ANSI. Przykład: L"działaj". Efekt: error.
Doprowadzony do ostateczności zapytałem o to, czy komuś się to udało na Ultimaforum. Ayene odpowiedziała prawie od razu. Ironia polega na tym, że rozwiązaniem jest ta funkcja, którą ja stosowałem. Nie można jej jednak wrzucać do biblioteki, trzeba ją wrzucić do samego RGSS.
Zanim przejdę do funkcji, jeszcze jedno, teoretycznie funkcja przetwarza do 255 znaków, w praktyce przetworzyłem ponad 2000 i działa. Domyślam się, że funkcja przetwarza do 65535 znaków, ale to tylko moja skromna teoria.

Funkcja wymaga dwóch buforów: bufora przetworzenia – miejsca w pamięci, w którym będzie przetwarzać tekst – i bufora wyjściowego – efektu. Chcę jeszcze zaznaczyć, że funkcje są dwie. Najpierw przetworzymy ciąg do ANSI, a potem do unikodu.
Podaję składnie obu funkcji.

```
MultiByteToWideChar( UINT strona kodowa, DWORD flagi, LPCSTR tekst
w ANSI, int rozmiar, LPWSTR bufor wyjścia, int rozmiar unicode)
WideCharToMultiByte( UINT strona kodowa, DWORD flagi, LPCWSTR ciąg
znakowy unikodu, rozmiar (znaki), LPSTR bufor wyjścia, int rozmiar
ANSI, LPCSTR znak, który zostanie wstawiony w przypadku braku odpowiednika
ANSI, LPBOOL flaga [dotyczy poprzedniego parametru])
```

Mhm, nie wygląda to dobrze. Te składnie są straszne. Podejmijmy szaleńczą próbę rozpisania tego do Win32API.

```
to_char = Win32API.new("kernel32", "MultiByteToWideChar", 'ilpipi',
'i')
to_byte = Win32API.new("kernel32", "WideCharToMultiByte",
'ilpipipp', 'i')
```

Kiedy to pierwszy raz zobaczyłem, a zwłaszcza liczbę buforów i kolejnych ściągów do wypełnienia, zastanawiałem się, czy to jest dobry pomysł tak się męczyć. Nie wyobrażam sobie kilku linijek kodu przy każdym wysłaniu jednej literki z polskiego alfabetu.
Na szczęście istnieje wspaniałe rozwiązanie.
Czas na drobną powtórkę.

Jak już kiedyś wspominałem, najważniejszą klasą RGSS jest klasa object. Wszystkie porzucone definicje właśnie do niej trafiają. Co najważniejsze, każda klasa dziedziczy po object!
Dlatego możemy stworzyć prostą definicję w object, kopiuję kod źródłowy z mojego projektu, dopisuję tylko komentarze.

Część opcji jest zbędna, jako taki otrzymałem skrypt na Ultimaforum i nie chciałem go już edytować.

```
def utf8(text)
#inicjacja bibliotek
  to_char = Win32API.new("kernel32", "MultiByteToWideChar", 'ilpipi', 'i')
to_byte = Win32API.new("kernel32", "WideCharToMultiByte", 'ilpipipp', 'i')
#.
utf8 = 65001 #tworzymy bufor
```

```
w = to_char.call(utf8, 0, text, text.size, nil, 0)  #każda zmienna ma
podzmienną *.size, która zawiera jej długość.
b = "\0" * (w*2) #tworzymy bufor o określonej długości, by uniknąć zbędnych
znaków zerowych
w = to_char.call(utf8, 0, text, text.size, b, b.size/2) #pierwsza konwersja
w = to_byte.call(0, 0, b, b.size/2, nil, 0, nil, nil)  #druga konwersja
b2 = "\0" * w  #bufor wyjścia
w = to_byte.call(0, 0, b, b.size/2, b2, b2.size, nil, nil) #trzecia
konwersja
return(b2) #zwracamy uzyskany ciąg jako efekt definicji
  end
```

teraz możemy wstawić nasz tekst do MessageBox'a.

```
mb = Win32API.new("user32","MessageBox",'ippi','I')
mb.call(0,utf8("Polski znaki działają, Jak nie wierzysz, sam
zobacz."),utf8("Wszystko działa!"),32)
```

Pamiętać musimy, że biblioteki zwracające tekst, na przykład funkcje z polami tekstowymi, również podają tekst jako swój format. Funkcja działa w obie strony, należy tylko ją zastosować na tekście zwróconym przez bibliotekę i wszystko sprawnie działa.
Teraz zajmiemy się czymś zupełnie innym. Zakończyliśmy dział podstawowy, możemy pobawić się okienkiem naszej gry. Najpierw dodamy mu ramkę, potem zmienimy wymiary i kilka innych podstawowych rzeczy. Najpierw jednak napiszemy sobie podsumowanie rozdziału w celu utrwalenia zdobytych wiadomości.

Podsumowanie

Zadanie
Napisz prosty aktualizator, sprawdzający dostępność nowej wersji. Załóż, że nowa wersja jest zapisana w pliku *.INI. Aktualizator ma możliwość wyświetlenia listy zmian z pliku *.TXT. Informacje o nowej wersji znajdują się na stronie http://localhost/wersja.ini, zaś lista zmian w pliku http://localhost/zmiany.txt. Aktualna wersja to 1.0.0.
Wersja składa się z trzech elementów wersja_a wersja_b i wersja_c. Na przykład dla wersji 1.0.2 wersja_a=1 wersja_b=0 wersja_c=3.
Po zaakceptowaniu aktualizacji, pobrany zostanie plik http://localhost/upgrade.exe , a następnie uruchomiony. Po pobraniu pliku, odtworzony zostanie plik Audio/sound/suc.wav przez winmm, a w przypadku porażki użyta zostanie funkcja MessageBeep z informacją o błędzie, a następnie funkcja Beep z długim sygnałem. O aktualizacji zostaniemy poinformowani poprzez restart gry i plik upgrade_ef.tmp z wartościom 1 – sukces, 0 – błąd.
Po zakończeniu wszystkiego, do pliku updates.log zostanie zapisana data oraz informacja o aktualizacji gry. Zanotowana zostanie również stara i nowa wersja. Wszystkie pobrane pliki zostaną usunięte, to znaczy: wersja.ini, upgrade.exe i zmiany.txt.

Rozwiązanie

```
def utf8(text)
  to_char = Win32API.new("kernel32", "MultiByteToWideChar", 'ilpipi', 'i')
to_byte = Win32API.new("kernel32", "WideCharToMultiByte", 'ilpipipp', 'i')
utf8 = 65001
w = to_char.call(utf8, 0, text, text.size, nil, 0)
b = "\0" * (w*2)
w = to_char.call(utf8, 0, text, text.size, b, b.size/2)
w = to_byte.call(0, 0, b, b.size/2, nil, 0, nil, nil)
b2 = "\0" * w
w = to_byte.call(0, 0, b, b.size/2, b2, b2.size, nil, nil)
return(b2)
  end
```

```
class Scene_Title
alias titinit initialize
def initialize
titinit
@download = Win32API.new("urlmon","urlDownloadToFile",'pppip','i')
@mb = Win32API.new("user32","MessageBox",'ippi','i')
@inir = Win32API.new("kernel32","GetPrivateProfileString",'pppppp','I')
@wersja = []
@wersja[1] = 1
@wersja[2] = 0
@wersja[3] = 0
end
alias titupd update
def update
$scene = Scene_Upgrade.suc(@nowawersja) if FileTest.exist("upgrade_ef.tmp")
@download.call(nil,"http://localhost/wersja.ini","wersja.ini",0,nil)
nowawersja = []
nw = "\0" * 32
@inir.call("wersja","wersja_a",'',nw,31,'wersja.ini')
nw.delete!("\0")
nowawersja[1] = nw
nw = "\0" * 32
@inir.call("wersja","wersja_b",'',nw,31,'wersja.ini')
nw.delete!("\0")
nowawersja[2] = nw
nw = "\0" * 32
@inir.call("wersja","wersja_c",'',nw,31,'wersja.ini')
nw.delete!("\0")
nowawersja[3] = nw
nw = nil
if nowawersja.to_i >> wersja.to_i
if @mb.call(0,utf8("Dostępna jest nowa wersja gry:" + nowawersja[1].to_s +
"." + nowawersja[2].to_s + "." + nowawersja[3].to_s + " . Czy chcesz ją
pobrać i zainstalować?"), "Nowa wersja gry", 4|32) == 6
```

```
$scene = Scene_Update.new(@wersja,@nowawersja)
end
end
titupd
end
end
```

```
class Scene_Update
def initialize
@download = Win32API.new("urlmon","urlDownloadToFile",'pppip','i')
@mb = Win32API.new("user32","MessageBox",'ippi','i')
@cf = Win32API.new("kernel32", "CopyFile", 'ppi', 'I')
@rf = Win32API.new("kernel32","ReadFile",'ipiip','I')
@wf = Win32API.new("kernel32","WriteFile",'ipiip','I')
@sfp = Win32API.new("kernel32","SetFilePointer",'ILLI','I')
@ch = Win32API.new("kernel32","CloseHandle",'i','i')
@beep = Win32API.new("kernel32","Beep",'ii','i')
@mbeep = Win32API.new("user32","MessageBeep",'i','i')
@df = Win32API.new("kernel32","DeleteFile",'p','i')
end
def main(@wersja,@nowawersja)
@download.call(nil,"http://localhost/upgrade.exe","upgrade.exe",0,nil)
Graphics.update
if FileTest.exist?("upgrade.exe")
print("Zakończono pobieranie aktualizacji. Kliknij przycisk OK, aby ją
zainstalować.")
system("start upgrade.exe")
system("exit")
$scene = nil
else
@mbeep(16)
print("Wystąpił błąd podczas aktualizacji!")
$scene = Scene_Title.new
end
end
def suc(@nowawersja)
fp = File.open("wersja.ini","r")
czaskoniec = fp.mtime
fp.close
log = @cf.call("updates.log",2,1|2,nil,4,0,nil))
@sfp.call(log,0,0,2)
bp = "\0" * 1024
@wf.call(log,b = czaskoniec.to_s + " - Aktualizacja gry. Stara wersja:
1.0.0. Nowa wersja: " + @nowawersja[1].to_s + "." + @nowawersja[2].to_s +
"." + @nowawersja[3].to_s + ".                          \n",b.size + 1,bp,'')
bp.delete!("\0")
bp = nil
@ch.call(log)
@beep.call(500,1000)
Graphics.update
print("Aktualizacja została zakończona sukcesem. Kliknij przycisk OK, aby
przeczytać listę zmian.")
zmiany = @cf.call("zmiany.txt",2,1|2,nil,4,0,nil))
b = "\0" * 1048576
bp = "\0" * 1048576
```

```
listazmian = @rf.call(zmiany,b,1047575,bp,'')
b.delete!("\0")
bp.delete!("\0")
b = nil
bp = nil
@ch.call(zmiany)
print(utf8("listazmian"))
Graphics.update
print("Proces aktualizacji zakończony, logi zostały zapisane w pliku
updates.log. Kliknij OK, aby usunąć pliki tymczasowe i rozpocząć grę")
@df.call("upgrade.exe")
@df.call("zmiany.txt")
@df.call("wersja.ini")
@df.call("upgrade_ef.tmp")
Input.update
Graphics.update
@wersja = @nowawersja
$scene = Scene_Title.new
end
end
```

W powyższym skrypcie nie odliczyłem czasu aktualizacji.
Jak widać aktualizator jest gotowy. Jeśli ktoś ma ochotę, może ten skrypcik wykorzystać, trzeba tylko stworzyć upgrade.exe i funkcje do niego. Trzeba też opóźnić start tegoż pliku o kilka sekund, by gra zdążyła się wyłączyć. W przeciwnym wypadku nic z aktualizacji nie wyjdzie.
Nie wykluczam możliwości, że w powyższym skrypcie jest błąd. Napisałem go na szybko i nie mam ochoty jeszcze tego sprawdzać, mam ochotę na coś ciekawszego: poczytać książkę ☺ albo napisać następną część tego kursu.

Jak już pisałem, teraz zajmiemy się okienkiem naszej gry i je troszkę zmodyfikujemy.
Niektórzy się uskarżają, że rpg maker'owe 640x480 jest za małe. Może pokażę w jaki sposób to zmienić na 1024x768? A może ustawienia rozdzielczości ekranu można by wpleść do gry?
Przed przeczytaniem następnej części, upewnij się, że dotychczas wszystko zrozumiałeś.

ROZDZIAŁ 2
Domena Windowsa
OKIENKA

Pierwszą rzeczą, która nam się kojarzy ze słowem Windows, są właśnie okna.

Jesteśmy przyzwyczajeni do takiego właśnie interfejsu użytkownika - składającego się z okien i kontrolek. Interfejs taki nazywamy GUI (graphic user interface).

W tym rozdziale omówię tworzenie, wypełnianie, edycję, modyfikowanie i rysowanie różnego rodzaju okien.

Uchwyt do okna naszej gry

Dawno, dawno temu, gdy świat był jeszcze młody – na początku tego kursu – pisałem, że RGSS nie przechowuje w żadnej znanej zmiennej uchwytu do okna naszej gry.
Zatem pierwszą rzeczą, którą powinniśmy zrobić przed jakąkolwiek zmianą naszego pięknego okna, jest uzyskanie do niego uchwytu – nie tego, za który się je otwiera i zamyka. W jaki sposób się za to zabrać? Przeanalizujmy dane, którymi dysponujemy:

Klasa okna: nieznana
Uchwyt okna: nieznany
Wymiary okna: znane (zapewne takie same, jak w setkach innych aplikacji)
Ustawienia dziedziczenia okna: nieznane
Tytuł okna: znany

Bingo! Tego potrzebowaliśmy. Jeśli znamy tytuł okna, wystarczy sprawdzić tytuły wszystkich Windowsowych okien w celu znalezienia tego jednego.
Fanom takich metod życzę powodzenia, kiedy napiszą pętlę, która rozszyfruje tytuły wszystkich okien, wszystkich wątków, wszystkich procesów. Po co jednak się tak męczyć? Microsoft przygotował taką pętelkę za nas i umieścił ją w bibliotece user32.
Mamy do dyspozycji dwie miłe funkcje: FindWindow oraz FindWindowEx. Różnią się tym, że druga przyjmuje więcej filtrów przy poszukiwaniu tego jednego, pięknego okna. Niestety tworząc tą funkcję, Microsoft przy okazji ułatwił tworzenie złośliwego oprogramowania i jednocześnie dał programistom w ręce dużą władzę - to tak nawiasem mówiąc, a raczej pisząc.
Oto składnie i możliwe filtry obu funkcji. Z nieznanej mi przyczyny wśród skrypterów większą popularnością cieszy się funkcja +Ex. Trochę dziwne, jeśli się weźmie pod uwagę fakt, że funkcja pierwsza w zupełności wystarczy, a filtrów +Ex i tak nie możemy w RGSS wypełnić.

```
HWND FindWindow(LPSTR klasa okna, LPSTR tytuł okna)
HWND FindWindowEx(HWND okno – rodzic, HWND okno – dziecko, LPSTR
klasa okna, LPSTR tytuł okna)
```

Jako że będziemy wkrótce to wykorzystywać, powinienem wytłumaczyć czym jest okno rodzica i okno dziecka.

Zapewne zauważyłeś, że w bardziej rozbudowanych programach, okna dzielą się na mniejsze okna. Te mniejsze okna to okna dziecka. Ponadto wszystkie kontrolki, takie jak przyciski, listy, menu czy pola tekstowe są potomstwem okna, w którym się znajdują. W ten sposób, dysponując uchwytem do kontrolki, jesteśmy w stanie kontrolować jej rodzica. W ten sam sposób możemy spróbować dojść do najwyższego okna systemu operacyjnego Microsoft Corporation ™ Windows OS. Okno to zowie się desktopem lub po prostu pulpitem. Co ciekawe, w rzeczywistości to, co my nazywamy pulpitem, czyli okno ze wszystkimi ikonami, tak naprawdę należy do okienka desktopu.
Postaram się tutaj mniej więcej wytłumaczyć zasady dziedziczności, gdyż to może się przydać.
Do okna desktopu należy pulpit, zegar, zasobnik systemowy, pasek szybkiego uruchamiania i każdy program uruchamiany bez dziedziczenia. W przypadku niektórych programów, na przykład Worda, część elementów również należy do desktopu.
Chciałbym jeszcze zaznaczyć, że HWND jest parametrem określającym tylko i wyłącznie okna, a zastępowanym jako integer 'I'.

Jeśli piszemy skrypty, które mają zmieniać rozmiary okna itp., musimy odczytać tytuł gry z pliku Game.ini.

Ja na czas tego kursu przyjmę, że nasza gra zwie się niezwykle elokwentnie: test.
$wnd = "win32API.new("user32","FindWindow",'pp','I').call(nil, "test")

Dlaczego tak mała rozdzielczość?

Wiele osób uskarża się na małe rozmiary okna RPG MAKERA XP. Nawiasem mówiąc, w przypadku VX wygląda to podobnie i równie łatwo jest to zmienić.
Z pomocą przychodzi nam user32.dll z funkcją SetWindowPos.
Słyszałem też o funkcji MoveWindow, ale nie szukałem jej w MSDN i nie wiem nawet, do jakiej dll'ki należy.
Ostrzeżenie: przed zapoznaniem się ze składnią funkcji weź głęboki wdech i nastaw się psychicznie na nadmiar informacji płynących z tego kursu.
Przygotowałeś się? Czas na składnię.

```
BOOL SetWindowPos(HWND uchwyt okna,, HWND układ okna względem dziedziczności, int pozycja x (w pikselach), int pozycja y (w pikselach), int szerokość okna (w pikselach), int wysokość okna (w pikselach), UINT flagi)
```

W pierwszym parametrze ustalamy, jakie okno edytujemy.
Ciekawostka: jeśli nie odpowiadają ci rozmiary dowolnej aplikacji, po prostu je zmień. Możesz przechwycić okno RPG MAKERA i zmienić jego rozdzielczość. Wiąże się to z pewnymi ograniczeniami, o których więcej za chwilę.
Drugi parametr to po prostu masakra. Określa on położenie okna względem potomstwa i rodziców. Oto co możemy ustawić. Parametr ten jest typu HWND, gdyż wstawiając tu chwyt okna, po prostu kopiujemy z niego ustawienia. Może być równy parametrowi pierwszemu, wtedy nic nie zmienimy.

Wartość	Ustawienie
1	Pod spodem okien nadrzędnych i równorzędnych
-2	Najwyższe okno, ponad rodzicami, sięga gwiazd ☺
0	Okno tam, gdzie powinno być – tak, jak dziedziczność wskazuje
-1	Okno ponad oknami równorzędnymi i podrzędnymi, jednak nie ponad nadrzędnymi

Parametry numer 3 i 4 powinny być jasne, pozycja okna względem ekranu. Domyślnie: 10, 10.
Parametry 5 i 6 określają rozmiar okna – domyślnie 640 i 480 pikseli.
Możemy to zmienić na minimalne ustawienia wielu kart graficznych: 800 i 600. Może to też się stać czymś normalnym: 1024x768, 1280x960 czy 1400x1050.
Przypominam tylko, że ekran musi daną rozdzielczość obsługiwać.
Przepisuję rozdzielczości ekranu obsługiwane przez system Windows 7 przy jakimś w miarę porządnym monitorze: 1024x768, 1052x862, 1280x600, 1280x768, 1280x800, 1280x900, 1360x768, 1366x768, 1400x1050, 1440x900, 1600x900, 1680x1050, 1920/1080.

Starsze modele monitorów działały w proporcjach 3:4, co oznacza, że ich wysokość była równa ¾ szerokości. Nowe monitory działają w proporcjach 9:16 i takie rozdzielczości najlepiej wybierać. 640x480 = 3x4, a na przykład 1366x768 ≈ 9x16.
Ostatni parametr to flagi, które w tym poleceniu są dość ważne.

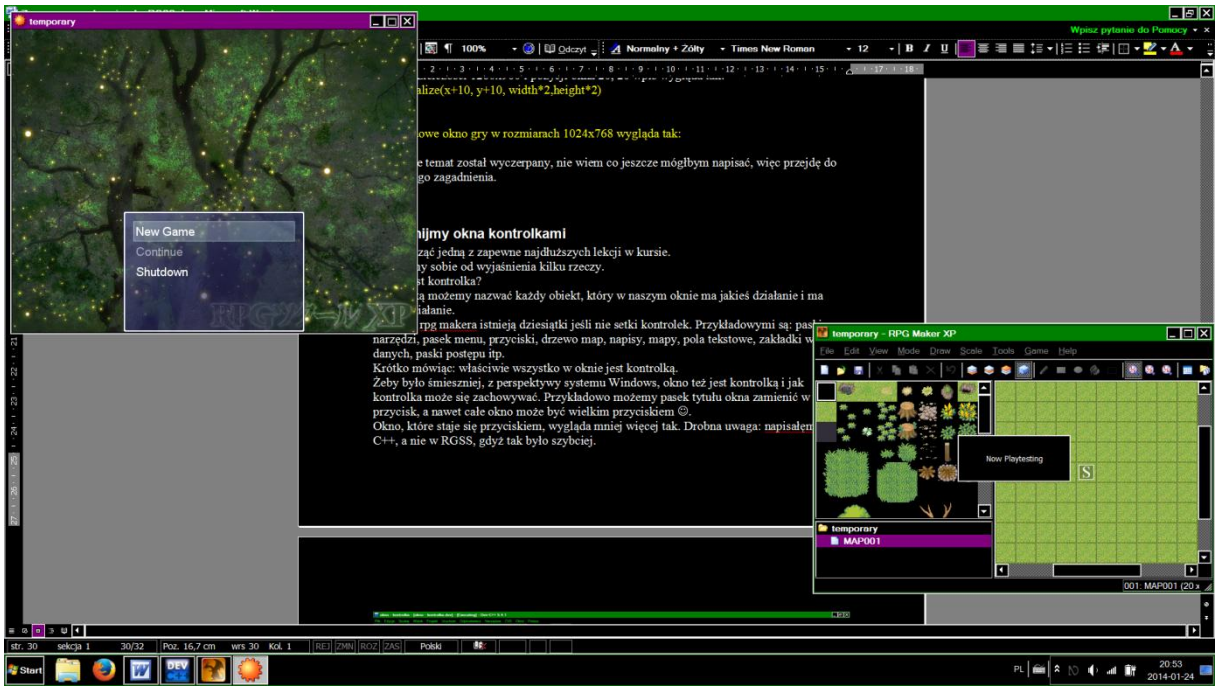
Wartość	Działanie
0x4000	Działanie asynchroniczne pozycji okna (nie muszą być zachowane proporcje na przykład 3x4) (zalecam)
0x20	Rysuje ramkę dookoła okna
0x80	Ukrywa okno
0x10	Nie zaznacza okna, nie przechodzi na nie fokusem (nie zalecam)
0x100	Nie zapamiętuje zawartości okna, czyści je
0x2	Nie przesuwa okna, ignoruje argumenty przesunięcia
0x4	Ignoruje ustalenie pozycji okna względem dziedziczności
0x1	Nie zmienia rozmiaru okna
0x8	Zapisuje zmiany, ale ich nie wciela, stare ustawienia pozostają. By wykonać, użyj funkcji UpdateWindow(HWND okno)
0x400	Nie informuje okna o zmianach (nie zalecam)
0x40	Wyświetla okno, jeśli to wyświetlonym nie było

Pokażę przykładowe zastosowanie. Przyjmuję, że gra dalej zowie się test.

```
Swp = Win32API.new("user32","SetWindowPos",'iiiiiii','i')
Fw = Win32API.new("user32","FindWindow",'pp','I')
Swp.call(fp.call(nil,"test"),0,0,0,1024,768,0x2|0x4)
```

Okno zmieni rozmiary, nic więcej.
Pisałem już o pewnych ograniczeniach. Polegają one na błędach w RGSS, które nie przewidują zmiany rozmiaru. Okno się powiększa, ale nie jest całe wypełnione. Tylko 640x480 jest zajęte, reszta jest pusta. Tą rzecz jest łatwo naprawić.
Wystarczy zajrzeć do klasy Window_Base i wyedytować definicję initialize. Jeśli powiększamy okno dwa razy, mnożymy width i high razy 2. Pozycje przesuwamy odpowiednio, dodając ustawione przesunięcie.
Dla rozdzielczości 1260x960 i pozycji okna 20, 20 wpis wygląda tak:

```
Def initialize(x+10, y+10, width*2,height*2)
#
End
Przykładowe okno gry w rozmiarach 1024x768 wygląda tak:
```



Myślę, że temat został wyczerpany, nie wiem co jeszcze mógłbym napisać, więc przejdę do następnego zagadnienia.

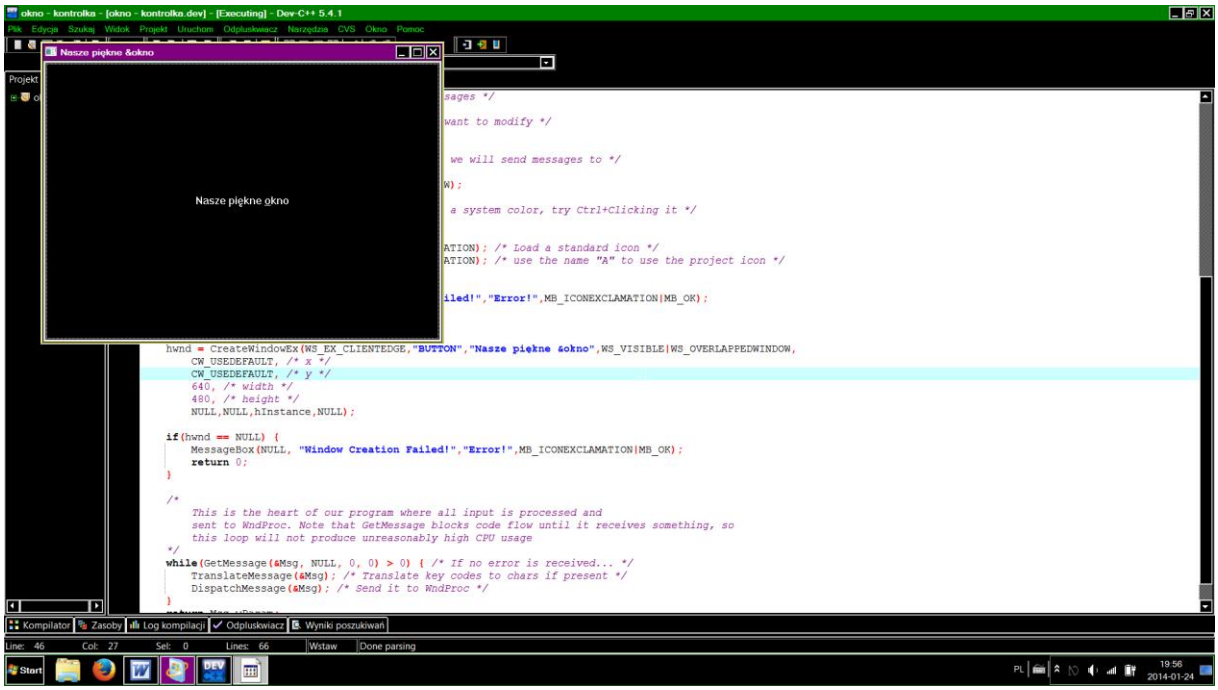
Wypełnijmy okna kontrolkami

Czas zacząć jedną z zapewne najdłuższych lekcji w kursie.
Zacznijmy sobie od wyjaśnienia kilku rzeczy.

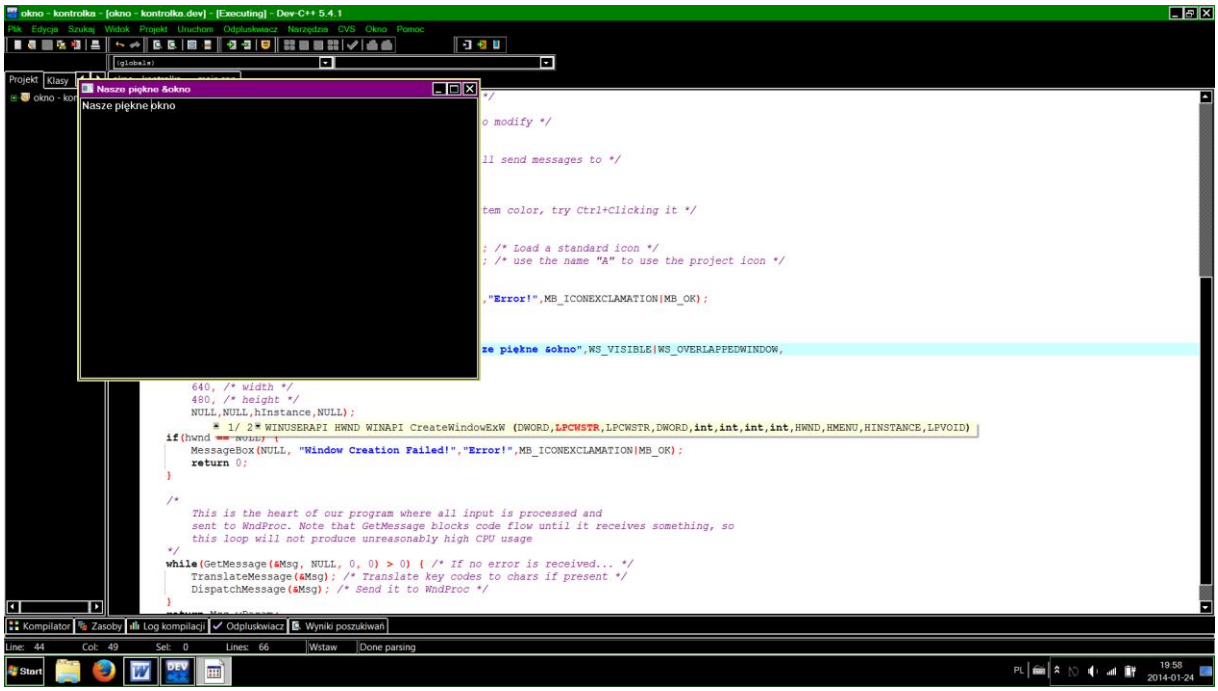
Czym jest kontrolka?
Kontrolką możemy nazwać każdy obiekt, który w naszym oknie ma jakieś działanie.
W oknie rpg makera istnieją dziesiątki - jeśli nie setki - kontrolek. Przykładowymi są: paski narzędzi, pasek menu, przyciski, drzewo map, napisy, mapy, pola tekstowe, zakładki w bazie danych, paski postępu itp.

Krótko mówiąc: właściwie wszystko w oknie jest kontrolką.
Żeby było śmieszniej, z perspektywy systemu Windows, okno też jest kontrolką i jak kontrolka może się zachowywać. Przykładowo możemy pasek tytułu okna zamienić w przycisk, a nawet całe okno może być wielkim przyciskiem ☺.

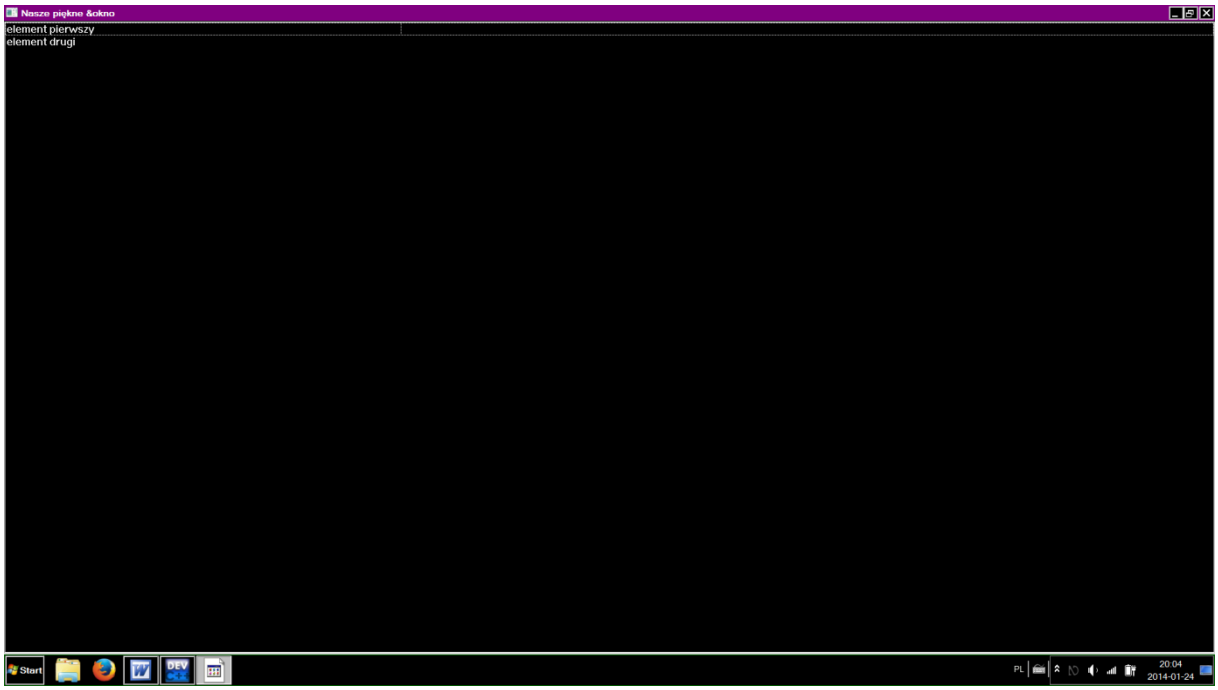
Okno, które staje się przyciskiem, wygląda mniej więcej tak. Drobna uwaga: napisałem je w C++, a nie w RGSS, gdyż tak było szybciej.



Równie dobrze okno może się stać na przykład polem tekstowym (C++):



Albo listą z elementami (C++):



Po co jednak to pokazuję? Informacja, że okno jest kontrolką, ułatwia zrozumienie prawdziwej natury kontrolek. Nie są one bowiem niczym innym, jak oknami. Różnica polega na tym, że mają one pewne właściwości: przyciski można wciskać (wielkie odkrycie), a w polach tekstowych pisać.

Co będzie nam potrzebne?
Uchwyt do naszego okna, Windows 2000 lub nowszy, biblioteka user32 i interpreter RGSS.
Dobrze, jakby tu zacząć tworzenie kontrolki?
Może od składni funkcji?

```
HWND CreateWindowEx(DWORD styl okna, LPCSTR klasa okna, LPCSTR tytuł okna, DWORD flagi okna, int rozmiar (x) okna, int rozmiar (y) okna, int szerokość okna, int wysokość okna, HWND okno rodzica, HMENU uchwyty menu i identyfikatorów, HINSTANCE instancja aplikacji, LPVOID struktura MDI)
```

Jak już się zapewne domyśliłeś, powyższa funkcja służy do tworzenia okien, ale jako że kontrolki są oknami, także do ich tworzenia.
I co teraz?
Zainicjujmy tę funkcję.

```
Createwindowex = Win32API.new("user32","CreateWindowEx",'ippiiiiiiiii','I')
```

Pierwszy argument to tak zwane style okna lub rozszerzone style okna. Jest ich "od groma i ciut, ciut". Podaję listę najpopularniejszych.

Flaga	Działanie
0x00040000	Okno "królem na pasku zadań" – priorytet okna na pasku zadań
0x00000200	Okno z ramką
0x00000001	Okno ma ramkę charakterystyczną dla okien dialogowych

	modalnych
0x00000004	Okno powstanie, ale okno – rodzic nie będzie o tym miało "zielonego pojęcia"
0x00020000	Okno ma ramkę taką jak przy kontrolkach, które nic nie robią – nie mają efektu po kliknięciu itp.
0x00000080L	Okno będzie traktowane jak pasek narzędzi. Będzie miało wygląd paska narzędzi, nie będzie się pojawiało po wciśnięciu alt+tab, a także nie pojawi się na pasku zadań.
0x00000008	Okno pojawi się ponad wszystkimi oknami bez tego stylu

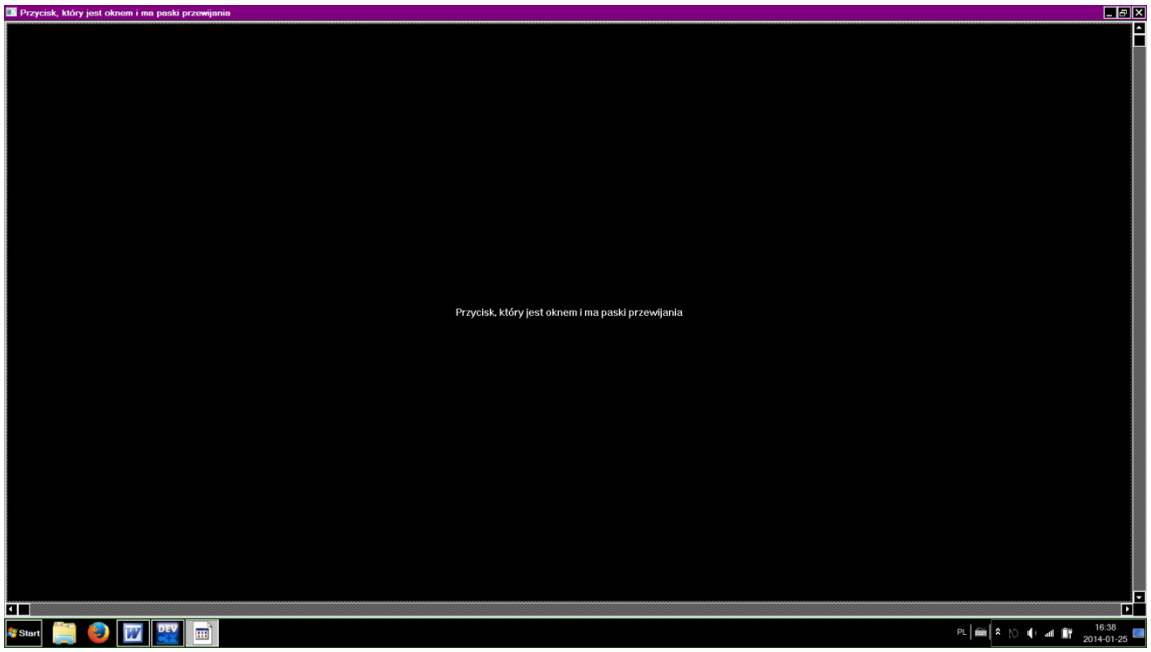
Niektóre style przydają się tylko przy oknach, inne przy kontrolkach. Ramka dla okna jest równa ramce dla przycisku itp.
Skoro mamy załatwiony ten problem, to czas na klasę okna. Klasa okna określa czym ono jest: przyciskiem, polem tekstowym czy oknem właściwym. Do klasy okna wrócimy, chciałbym jeszcze zająć się resztą.

Flagi okna są różne, przepiszę te, które są najważniejsze i wspólne dla wszystkich kontrolek i na dzisiaj skończę, bo już jest dość późno, a mam jeszcze kilka zajęć.

Flaga	**Działanie**
0x00800000	Ramka wokół okna
0x00C00000	Okno ma pasek z tytułem
0x40000000	Okno jest potomne (niezbędne w kontrolkach należących do innych okien)
0x08000000	Okno jest nieaktywne, nie działa, klikanie na przyciski nic nie daje, wpisywanie tekstu też. Aby je aktywować użyj biblioteki user32.dll, a dokładniej jej polecenia  `BOOL EnableWindow(HWND uchwyt okna, BOOL aktywowane(1) lub nie(0))`
0x00400000	Okno ma ramkę charakterystyczną dla okien dialogowych
0x00020000	Okno jest pierwszym obiektem grupy kontrolek
0x00100000	Okno ma poziomy pasek przewijania
0x00010000	Po kontrolce można się przesuwać klawiszem tab. Wymagana jest odpowiednia funkcja IsDialogMessage, o której może później
0x00000000	Okno z paskiem tytułowym i ramką
0x10000000	Okno jest widoczne (styl wymagany)
0x00200000	Okno ma pionowy pasek przewijania

Jak widać i tutaj część stylów się nie przyda niektórym kontrolkom. Po co na przykład przyciskom pasek przewijania?

Naturalnie, da się taki stworzyć (C++).



Dobrze. Oprócz stylów, parametry: x, y, CX i CY powinny być jasne.
Ani menu, ani identyfikatorów na razie nie używamy, wpisujemy tam nil.
Nie tworzymy również struktur MDI – nil.
Oknem rodzicem jest nasze okno gry, więc trzeba użyć funkcji FindWindow.
Zostało tylko jedno miejsce: instancja aplikacji.
Wesoła nowina: RGSS nam jej oczywiście nie dostarcza.
Na szczęście (lub nieszczęście) istnieje funkcja, która nam tę instancję zwróci.

```
HINSTANCE GetModuleHandle(LPCSTR moduł aplikacji)
```

W dokumentacji MSDN jest pewien piękny wpis. Mówi on, że jeśli argument tej funkcji jest pusty, zwróci ona instancję aktualnego modułu: naszej gry.

```
$hinstance =
Win32API.new("kernel32","GetModuleHandle",'p','i').call(nil)
```

Instancję aplikacji już mamy, zajmijmy się poszczególnymi kontrolkami.

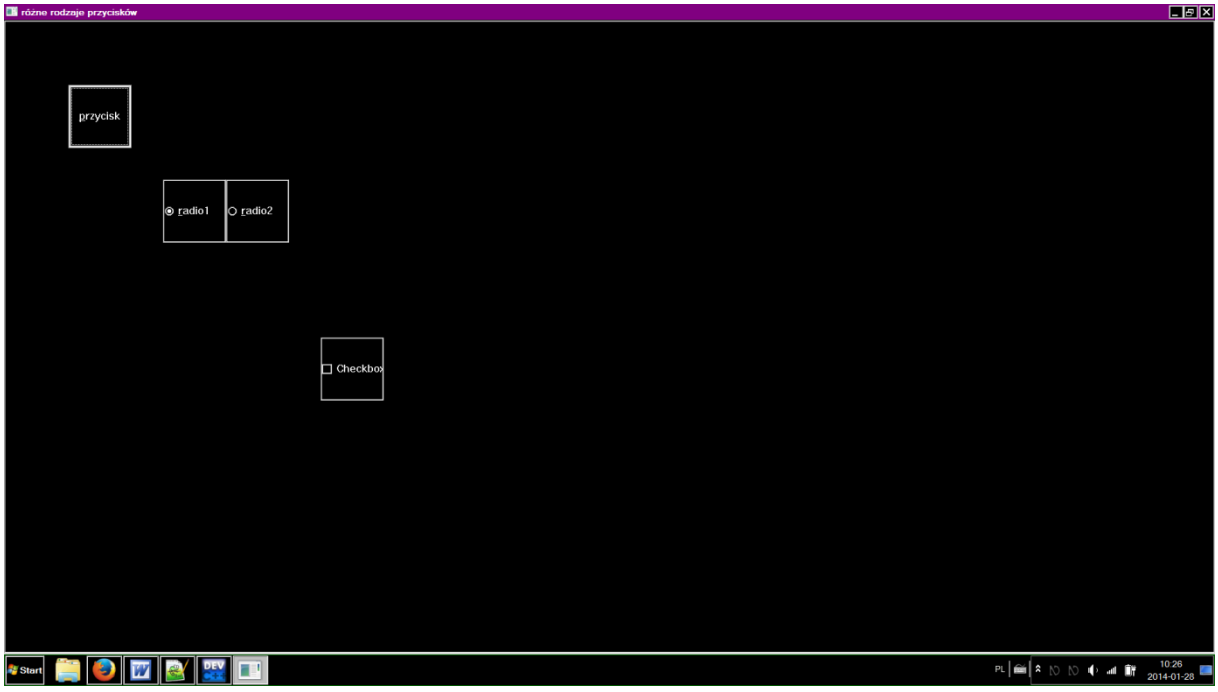
Przyciski

Chyba każdy wie, jak wygląda przycisk. Niektórych może jedynie zdziwić fakt, iż z perspektywy WINAPI, radioboxy i checkboxy są również przyciskami.
Aby stworzyć przycisk, w miejscu klasy okna wpisz: "BUTTON".
Jakie są natomiast flagi przycisków?

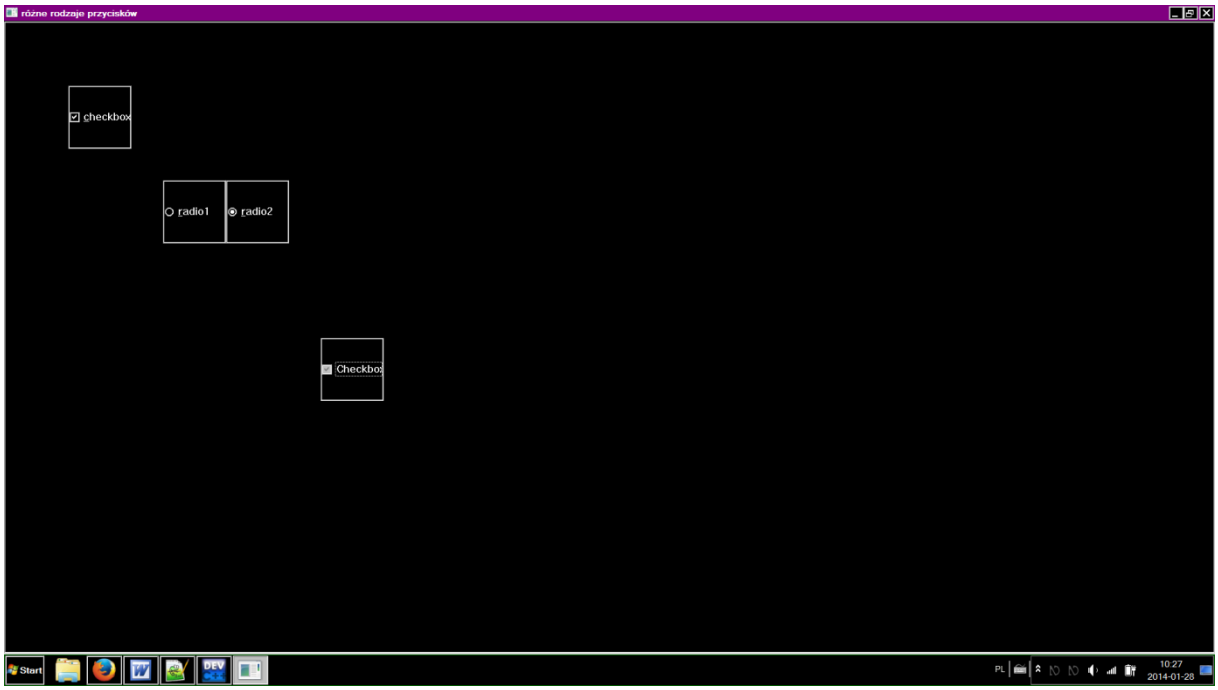
Poniższe flagi wpisz w czwartym parametrze.

Flaga	Działanie
5	Przycisk staje się checkboxem, który może być: zaznaczony, niezaznaczony lub nieokreślony.
6	To samo, co powyżej. Różnica polega na tym, że w powyższym przykładzie należy napisać kod, który obsłuży zaznaczanie i odznaczanie checkboxa. W tym przypadku, po kliknięciu checkbox zmieni swój stan automatycznie.
3	Automatyczny checkbox, który może być tylko zaznaczony lub nie (dwa stany). Zaznaczanie checkboxa zostanie obsłużone automatycznie.
2	Checkbox (dwa stany), który nie jest obsługiwany automatycznie.
9	Automatyczny przycisk radiowy. Użyj stylu grupy na pierwszym przycisku zgrupy radiowej i na pierwszym do niej nie należącym.
8	Przycisk radiowy nie obsługiwany automatycznie.
32	Tekst z lewej strony kontrolki.
128	Przycisk jest bitmapą (zamiast tekstu bitmapa).
64	Przycisk jest ikonką (zamiast tekstu ikonka).
0x800	Tekst nad grupą lub przyciskiem.
0x300	Wyśrodkowuje tekst w poziomie.
0x2000	Sprawia, że tekst na przycisku może mieć więcej, niż jedną linię.
4096	Tworzy przycisk, który wygląda jak zwykły przycisk, nawet jeśli jest checkboxem. Często używany w programach.
0x400	Tekst nad przyciskiem lub grupą.
0xC00	Wyrównuje tekst w pionie.
7	Tworzy grupę przycisku.

Sukces, skończone. Flag jest więcej, ale niektóre się nie przydają, inne działają tylko pod vistą i 6.00, a jeszcze inne nie działają poprawnie w RGSS.
Różne rodzaje przycisków wyglądają tak (C++):



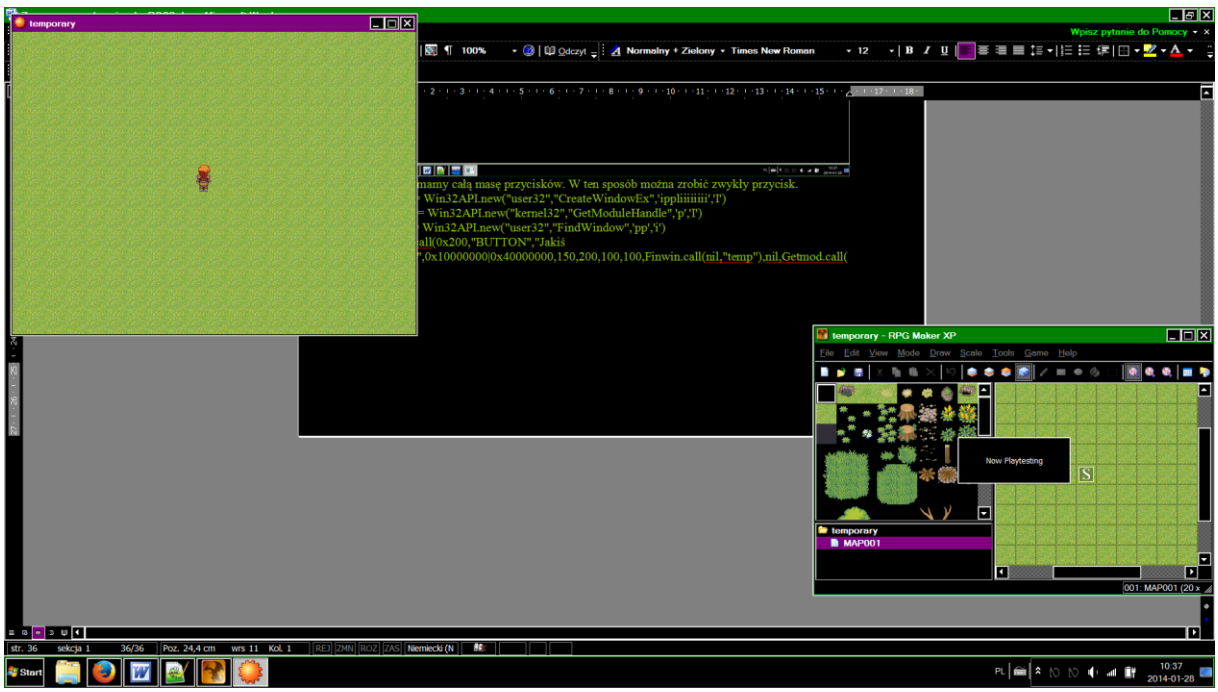
Inne stany przycisków (C++):



Dobrze, mamy całą masę przycisków. W ten sposób można zrobić zwykły przycisk.

```
Crewin = Win32API.new("user32","CreateWindowEx",'ippliiiiiiiip','I')
Getmod = Win32API.new("kernel32","GetModuleHandle",'p','I')
Finwin = Win32API.new("user32","FindWindow",'pp','i')
Crewin.call(0x200,"BUTTON","Jakiś przycisk",0x10000000|0x40000000,150,200,100,100,Finwin.call(nil,"temp"),0,Getmod.call(nil),nil)
```

Oto i efekt: mapa w RPG makerze wraz z przyciskiem.



Dobrze, mamy nasze piękne przyciski, ale co z tego, jeśli nie wiemy, jak ich użyć. Użytkownik może je wciskać, przełączać checkboxy, wciskać radioboxy i robić z nimi wszystko, a okno będzie wyglądało tak, jak wyglądało. Bez żadnych zmian.
Na szczęście da się te przyciski obsłużyć.

Nie znam żadnego sposobu na sprawdzenie wciśnięć zwykłych przycisków bez odczytu komunikatów okna. Kiedyś je może odczytamy i wtedy pokażę, jak sprawdzić stan wszystkich przycisków. Tym razem zadowolimy się czymś innym. Na szczęście jesteśmy w stanie obsłużyć checkboxy i radioboxy bez wspomnianej już procedury okna. W ten sposób, tworząc checkbox wyglądający jak przycisk, możemy obsłużyć przyciski w RGSS.

Do sprawdzenia stanu checkboxów służy funkcja IsDlgButtonChecked.
Funkcja ta przyjmuje dwa argumenty: uchwyt okna i identyfikator kontrolki.
Jak już pisałem, zamiast menu możemy ustawić identyfikator kontrolki – dotyczy tylko kontrolek. W tym celu wybieramy sobie jakąś liczbę z zakresu 1 | 1073741824-1.

Wybrałeś liczbę? Znakomicie, wpisz ją w parametr uchwytu menu w funkcji CreateWindowEx. Jeśli chcesz, możesz przechować ten uchwyt w jakieś zmiennej, żeby się nie pomylić w przepisywaniu. Najlepiej wybierać łatwe do zapamiętania parametry, na przykład: 111, 222, 333, 444, 555, 1111, 1234, 4321 itd.
Teraz możemy użyć funkcji IsDlgButtonChecked.
Przyjmijmy, że nasz identyfikator to 111, a uchwyt okna znajduje się w zmiennej $hwnd.

```
IsDlgButTonChecked =
Win32API.new("user32","IsDlgButtonChecked",'ii','I')
IsDlgButtonChecked.call($hwnd,111)
```

Funkcja zwraca jedną z poniższych wartości.

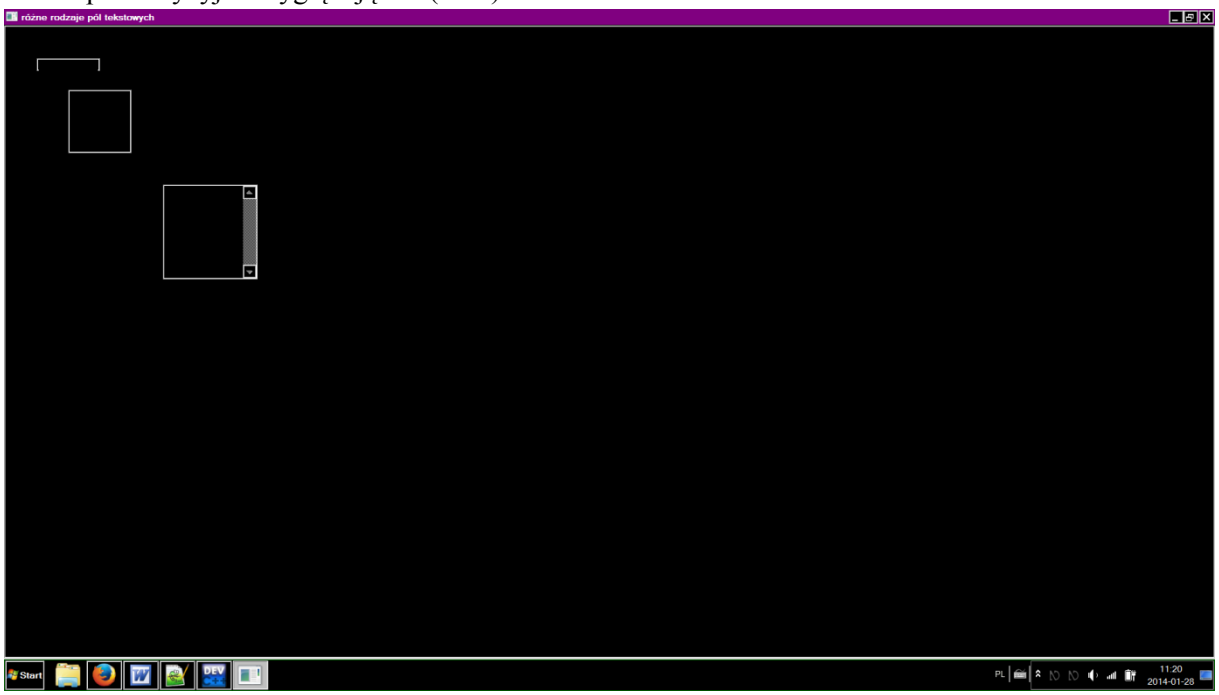
Wartość	Stan checkboxa
1	Zaznaczony
2	Częściowo oznaczony – niezidentyfikowany (tylko w trzy-stanowych checkboxach)
0	Niezaznaczony
8	Myszka na nim się znajduje
4	Wybrany

Nas interesują tylko trzy pierwsze wartości.
Możemy użyć również pewnej funkcji o nazwie CheckDlgButton, która nadaje zaznacza lub odznacza przyciski.
Funkcja znajduje się w bibliotece user32.dll. Jako pierwszy argument przyjmuje uchwyt okna, jako drugi identyfikator kontrolki, a jako trzeci jeden z trzech pierwszych argumentów z powyższej tabelki.

Pola edycyjne

Takowe pole wykorzystuję, teraz pisząc. Pola edycyjne to takie, w których możemy wprowadzać tekst.

Różne pola edycyjne wyglądają tak (C++).



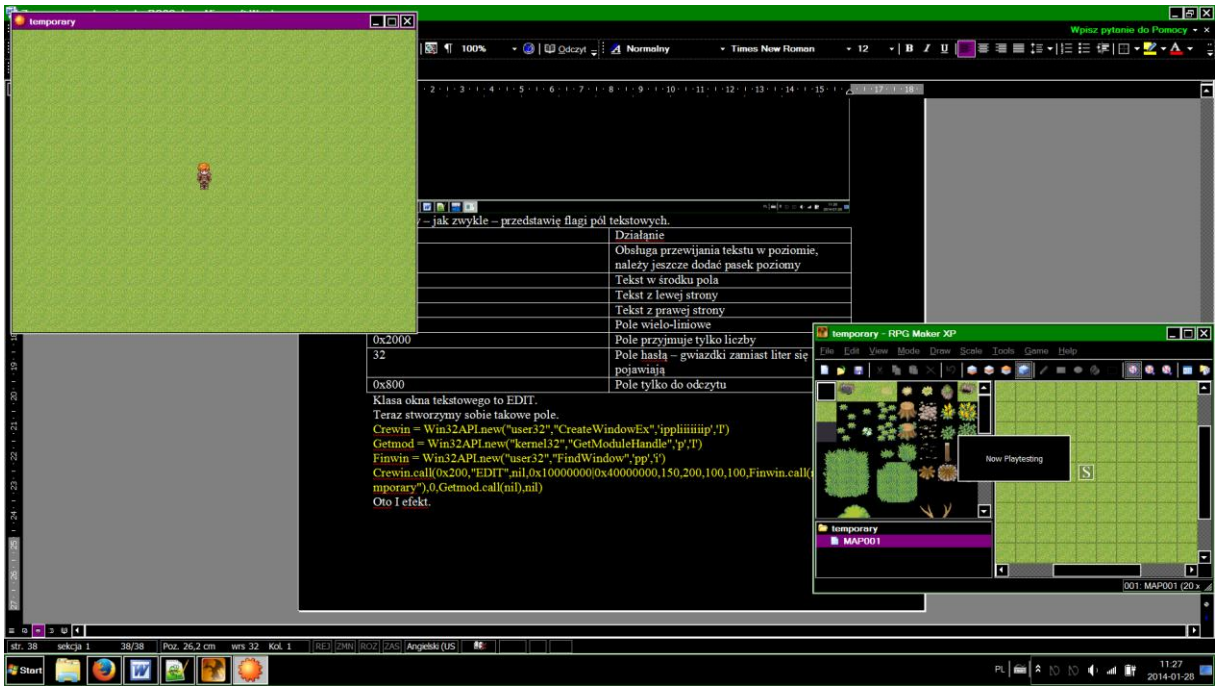
Najpierw – jak zwykle – przedstawię flagi pól tekstowych.

Flaga	Działanie
128	Obsługa przewijania tekstu w poziomie, należy jeszcze dodać pasek poziomy
1	Tekst w środku pola
0	Tekst z lewej strony
2	Tekst z prawej strony
4	Pole wieloliniowe
0x2000	Pole przyjmuje tylko liczby
32	Pole hasła – zamiast liter pojawiają się gwiazdki
0x800	Pole tylko do odczytu

Klasa okna tekstowego to EDIT.
Teraz stworzymy sobie takowe pole.

```
Crewin = Win32API.new("user32","CreateWindowEx",'ippliiiiiiip','I')
Getmod = Win32API.new("kernel32","GetModuleHandle",'p','I')
Finwin = Win32API.new("user32","FindWindow",'pp','i')
Crewin.call(0x200,"EDIT",nil,0x10000000|0x40000000,150,200,100,100,
Finwin.call(nil,"temporary"),0,Getmod.call(nil),nil)
```

Oto i efekt.



Jeszcze drobne wyjaśnienie. W przypadku pól tekstowych nie podajemy tytułu okna, gdyż zostałby on wpisany wewnątrz nich. W ten sposób ustawiamy domyślną zawartość pól tekstowych.
Z polami tekstowymi łączą się dwie ważne funkcje: GetWindowText oraz SetWindowText.

```
Int GetWindowText(HWND kontrolka, LPSTR bufor, DWORD długość)
```

Funkcja ta odczytuje tekst z pola tekstowego. Oto i przykład. Przyjmę, że uchwyt okna znajduje się w zmiennej $hwnd, a uchwyt pola tekstowego w zmiennej text.

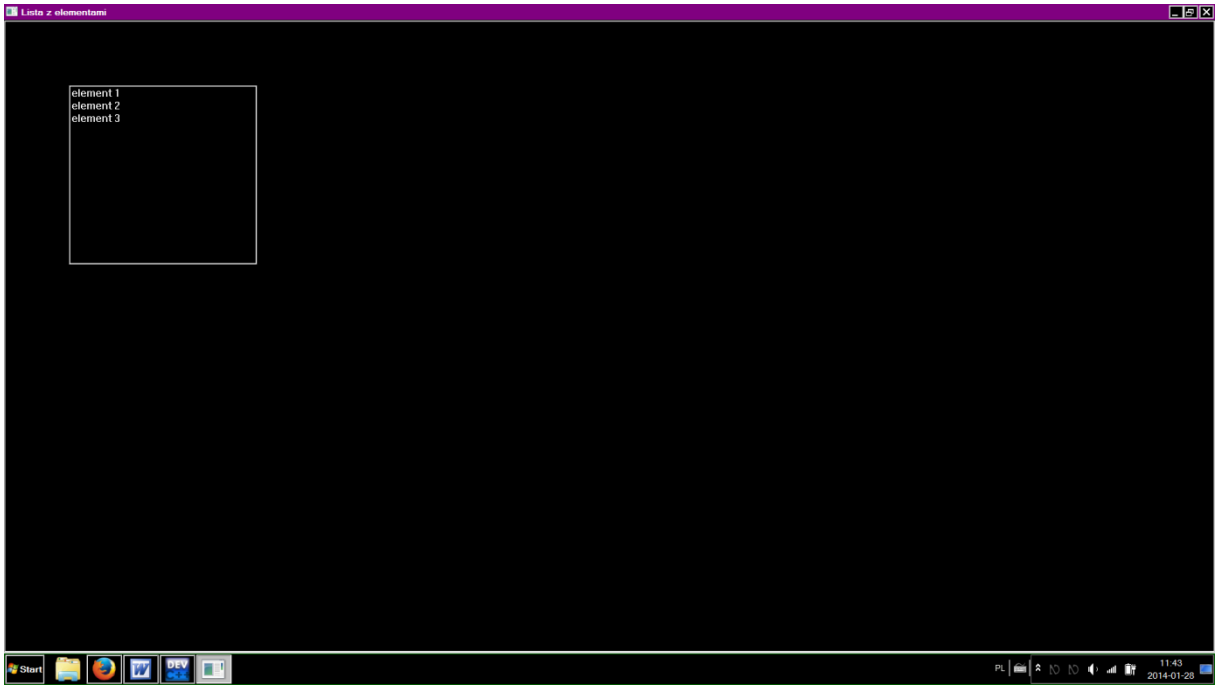
```
Bufor = "\0" * 512
Win32API.new("user32","GetWindowText",'ipi','I').call(text,Bufor,511)
Bufor.delete!("\0")
```

Druga funkcja to SetWindowText. Jej zadaniem jest wypełnianie pól tekstowych tekstem. Znajduje się również w bibliotece user32.dll.

```
BOOL SetWindowText(HWND kontrolka, LPSTR tekst)
```

Listy wyboru

Lista wyboru to coś takiego (C++).



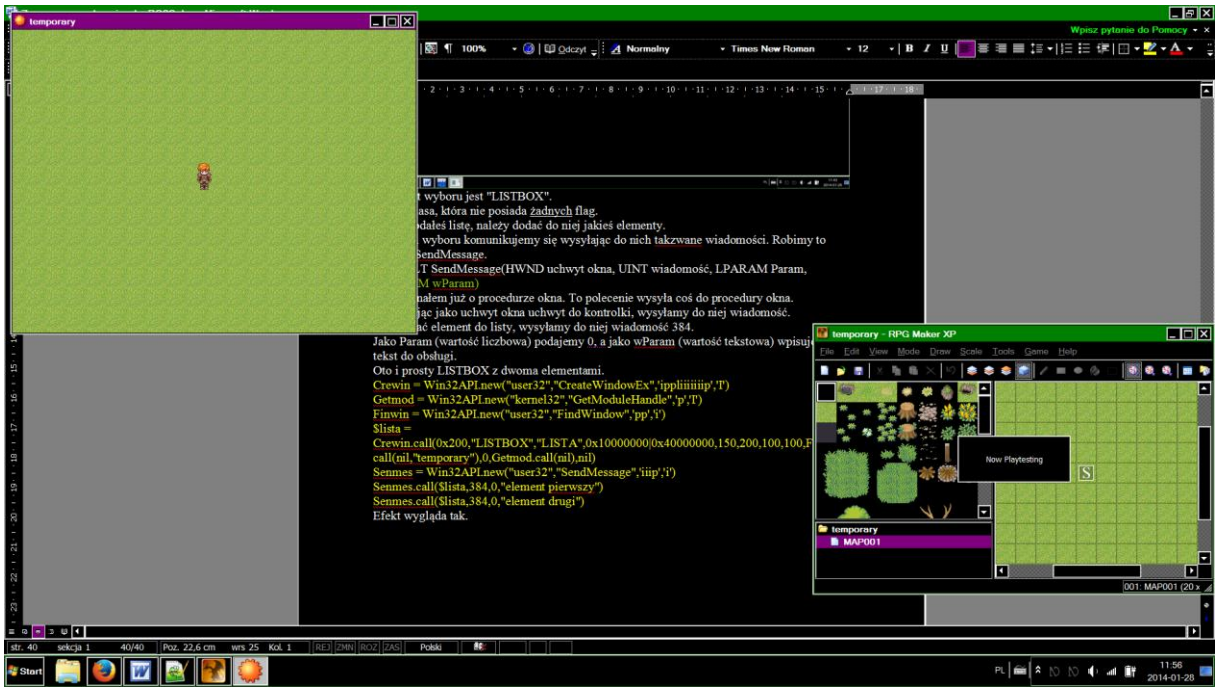
Klasą list wyboru jest "LISTBOX".
Jest to klasa, która nie posiada żadnych flag.
Kiedy dodałeś listę, należy dodać do niej jakieś elementy.
Z listami wyboru komunikujemy się, wysyłając do nich tak zwane wiadomości. Robimy to funkcją SendMessage.

```
LRESULT SendMessage(HWND uchwyt okna, UINT wiadomość, LPARAM Param,
WPARAM wParam)
```

Wspominałem już o procedurze okna. To polecenie wysyła coś do procedury okna. Wstawiając jako uchwyt okna uchwyt do kontrolki, wysyłamy do niej wiadomość.
Aby dodać element do listy, wysyłamy do niej wiadomość 384.
Jako Param (wartość liczbowa) podajemy 0, a jako wParam (wartość tekstowa) wpisujemy tekst do obsługi.
Oto i prosty LISTBOX z dwoma elementami.

```
Crewin = Win32API.new("user32","CreateWindowEx",'ippliiiiiiip','I')
Getmod = Win32API.new("kernel32","GetModuleHandle",'p','I')
Finwin = Win32API.new("user32","FindWindow",'pp','i')
$lista =
Crewin.call(0x200,"LISTBOX","LISTA",0x10000000|0x40000000,150,200,100,100,F
inwin.call(nil,"temporary"),0,Getmod.call(nil),nil)
Senmes = Win32API.new("user32","SendMessage",'iiip','i')
Senmes.call($lista,384,0,"element pierwszy")
Senmes.call($lista,384,0,"element drugi")
```

Efekt wygląda tak.



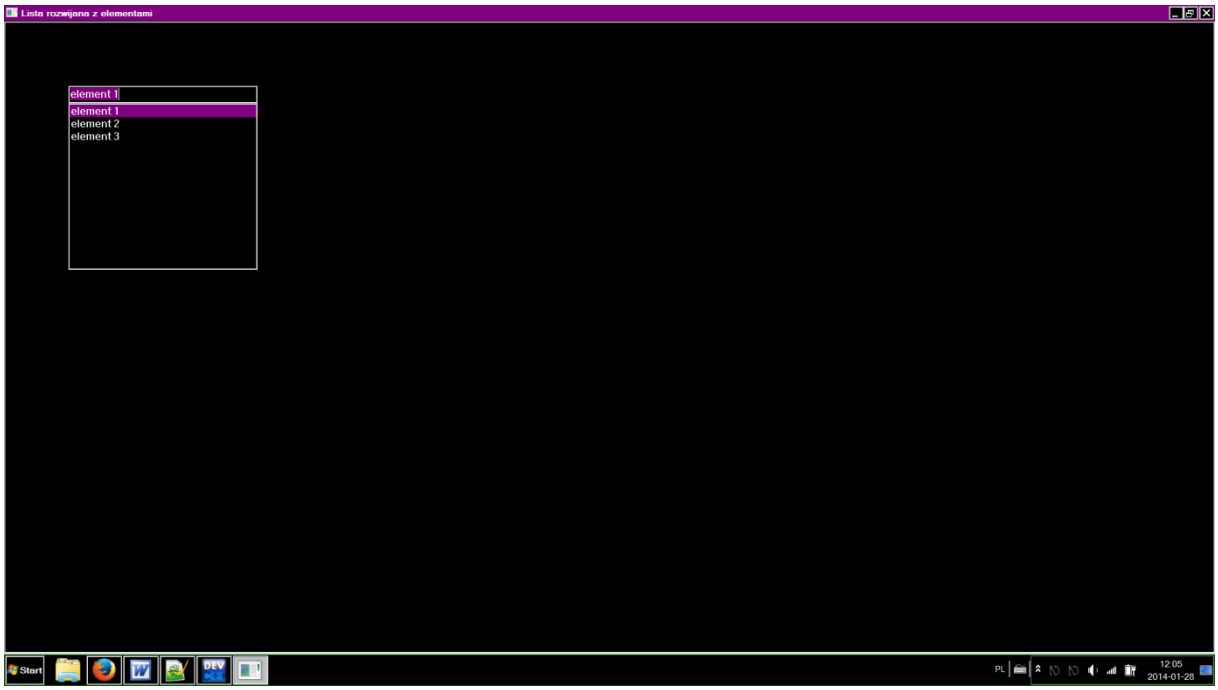
Oto i spis najważniejszych wiadomości do ListBoxów.

Wiadomość	LParam	wParam	opis
384	0	Tekst do wpisania	Dodaje element do listy
386	0	Tekst do usunięcia	Usuwa element
392	0	0	Zwraca kursor zaznaczonego elementu (0 = element pierwszy, 1 = element drugi itd.)
390	0	tekst do wybrania	Zaznacza wybrany tekst

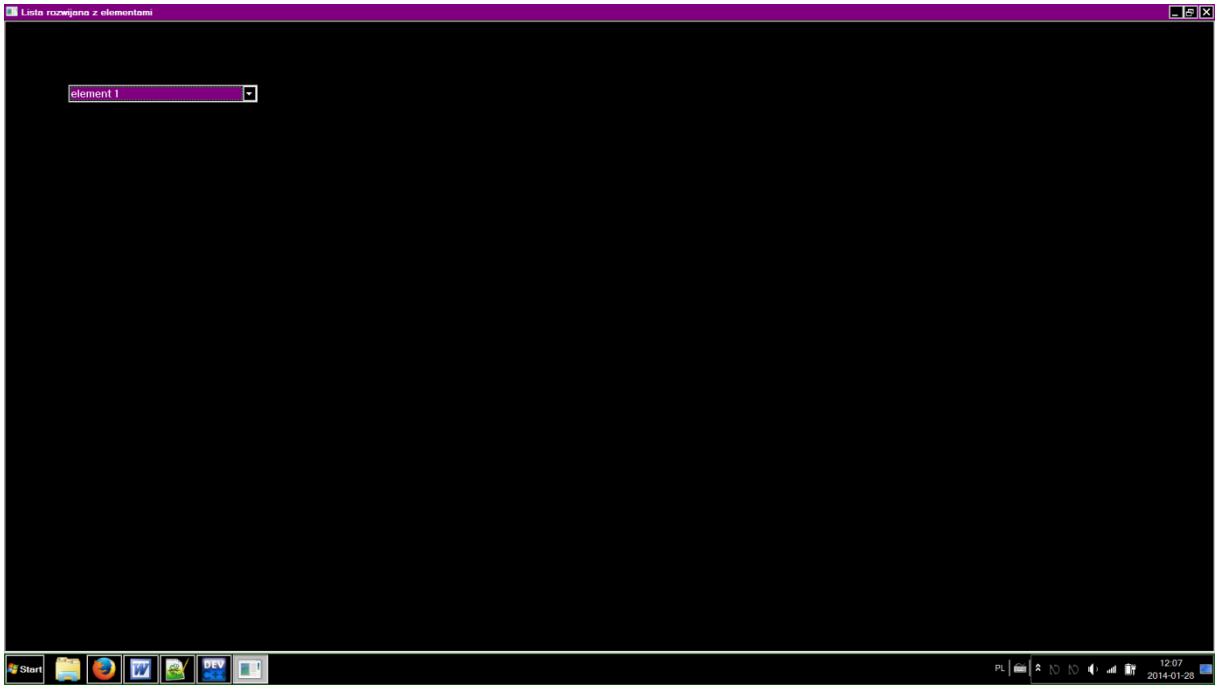
Myślę, że o listach to wszystko.

Listy rozwijane

Różnią się od zwykłych list tym, że można do nich wprowadzać tekst (można to wyłączyć) i są rozwijane – nie widać całej listy.
Oto i przykład (C++).



To pole tekstowe można uczynić nieedytowalnym z klawiatury, można wtedy wybrać tylko elementy z listy.
Wygląda to tak (C++).



Klasą list rozwijanych jest "COMBOBOX".
Oto i flagi COMBOBOX'ów.

Flaga	Działanie
64	Obsługa pasków poziomych
2	Przewijanie w dół

3	Przewijanie w dół – lista bez podawania tekstu w polu tekstowym
1	Najprostsza możliwa lista rozwijana

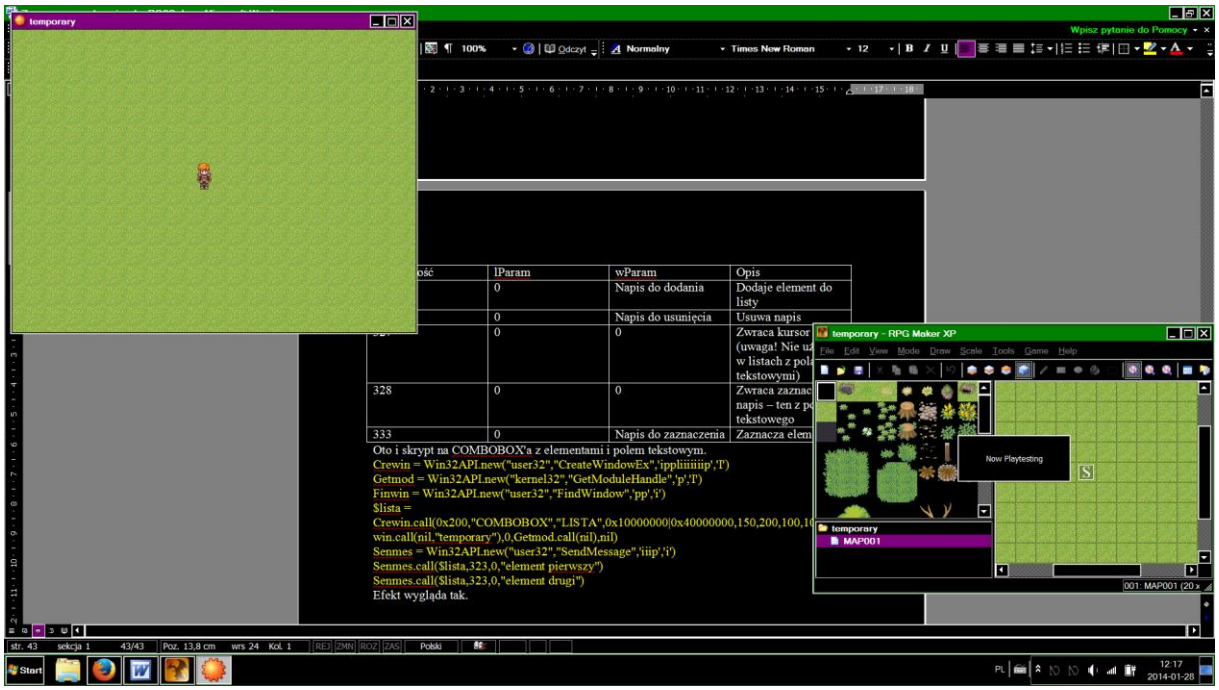
W Comboboxach – tak jak w LISTBOX'ach – używa się wiadomości do komunikacji. Oto i najważniejsze wiadomości.

Wiadomość	lParam	wParam	Opis
323	0	Napis do dodania	Dodaje element do listy
324	0	Napis do usunięcia	Usuwa napis
327	0	0	Zwraca kursor listy (uwaga! Nie używaj w listach z polami tekstowymi)
328	0	0	Zwraca zaznaczony napis – ten z pola tekstowego
333	0	Napis do zaznaczenia	Zaznacza element

Oto i skrypt na COMBOBOX'a z elementami i polem tekstowym.

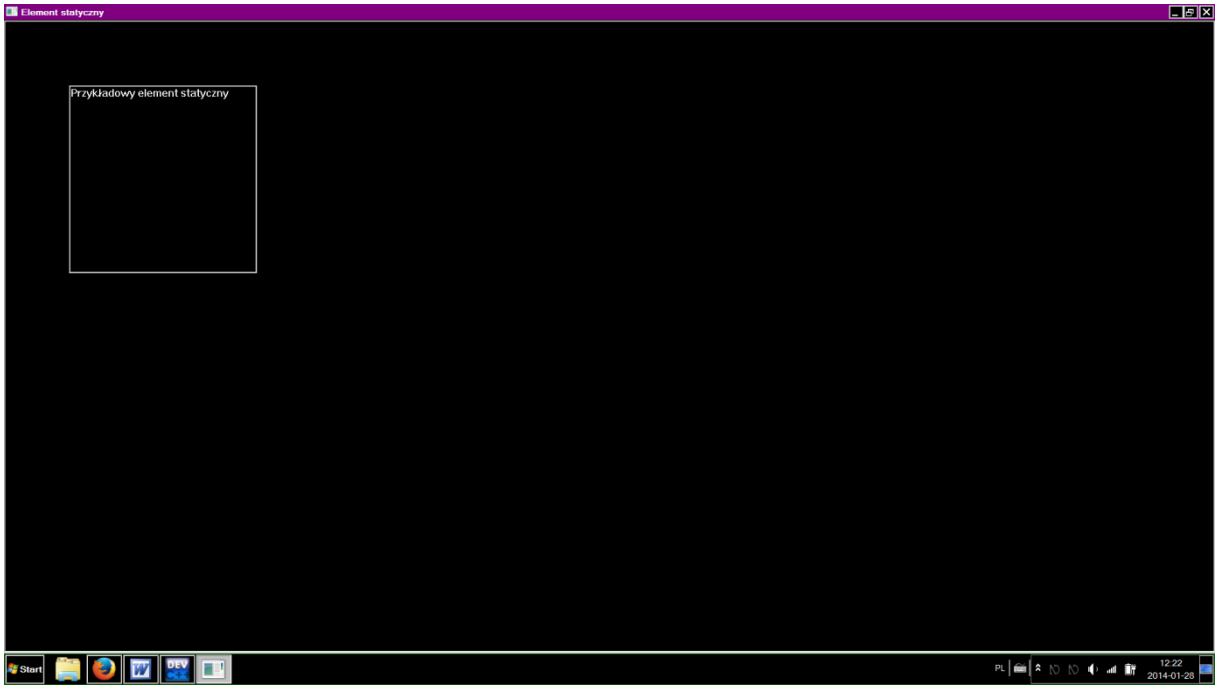
```
Crewin = Win32API.new("user32","CreateWindowEx",'ippliiiiiiip','I')
Getmod = Win32API.new("kernel32","GetModuleHandle",'p','I')
Finwin = Win32API.new("user32","FindWindow",'pp','i')
$lista =
Crewin.call(0x200,"COMBOBOX","LISTA",0x10000000|0x40000000,150,200,100,100,
Finwin.call(nil,"temporary"),0,Getmod.call(nil),nil)
Senmes = Win32API.new("user32","SendMessage",'iiip','i')
Senmes.call($lista,323,0,"element pierwszy")
Senmes.call($lista,323,0,"element drugi")
```

Efekt wygląda tak.



Elementy statyczne

Elementy statyczne to takie, które nie działają w żaden sposób, wyświetlają tylko jakiś tekst.



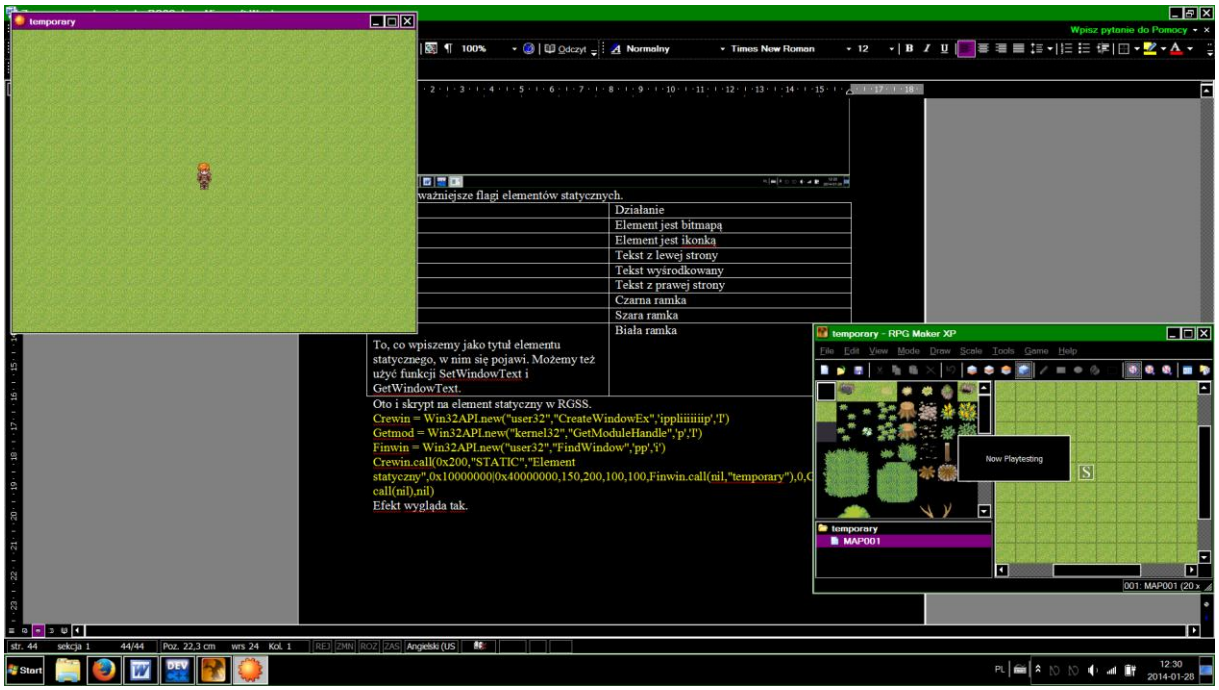
Oto i najważniejsze flagi elementów statycznych.

Flaga	Działanie
14	Element jest bitmapą
3	Element jest ikonką
0	Tekst z lewej strony
1	Tekst wyśrodkowany
2	Tekst z prawej strony
7	Czarna ramka
8	Szara ramka
9 To, co wpiszemy jako tytuł elementu statycznego, w nim się pojawi. Możemy też użyć funkcji SetWindowText i GetWindowText.	Biała ramka

Oto i skrypt na element statyczny w RGSS.

```
Crewin = Win32API.new("user32","CreateWindowEx",'ippliiiiiiip','I')
Getmod = Win32API.new("kernel32","GetModuleHandle",'p','I')
Finwin = Win32API.new("user32","FindWindow",'pp','i')
Crewin.call(0x200,"STATIC","Element
statyczny",0x10000000|0x40000000,150,200,100,100,Finwin.call(nil,"temporary
"),0,Getmod.call(nil),nil)
```

Efekt wygląda tak.



Jeszcze kilka informacji

Omówiliśmy już wszystkie najważniejsze kontrolki. Dodam jeszcze kilka ciekawostek.

TWORZENIE SKRÓTÓW KLAWISZOWYCH

Zapewne zauważyłeś, że w wielu programach przy kontrolkach jedna z liter jest podświetlona. Jeśli wciśniemy klawisz alt oraz tą literę, ustawiony zostanie fokus. Dodawanie takich skrótów jest bardzo łatwe.

Wystarczy dodać znak & w tytule kontrolki, a skrót klawiszowy zostanie dodany automatycznie.

PRZESUWANIE I ZMIANA ROZMIARU KONTROLEK

Robi się to tak, jak z okienkami. Nie chcę pisać skryptu. Napiszę tylko, że robi się to funkcją SetWindowPos.

ZAZNACZANIE KONTROLEK

Odpowiednia funkcja znajduje się w bibliotece user32.dll.

```
HWND SetFocus(HWND kontrolka lub okno)
```

Wielki finisz

UFF, przeszliśmy przez to. Jak pisałem, lekcja rzeczywiście jest długa, ale przydatna. Teraz możesz stworzyć np. system wpisywania imienia bohatera, używając do tego pól tekstowych i przycisków.

Ukrywanie i usuwanie kontrolek

Mhm, możemy wypełnić nasze okno elementami po brzegi.

Co nam jednak z tego przyjdzie? Możemy sobie przykładowo stworzyć okienko z wpisywaniem tekstu. Kiedy jednak tekst wpiszemy, nie zniknie ono.

Oczywiście można przesunąć kontrolkę poza ekran, ale nie jest to dobry sposób, w końcu i tam miejsce się skończy, a skróty klawiszowe nie znikną.

Kontrolek można się pozbyć w dwa sposoby: ukryć je lub usunąć.
Ukryte kontrolki tak naprawdę istnieją, tylko znikają. Ich skróty klawiszowe stają się nieaktywne. Jeśli chcesz by dana kontrolka pojawiła się, po prostu ją ukryj. Zaletą tej metody w przypadku pól tekstowych jest to, że po ich przywróceniu, pozostawia ich treść niezmienioną. Takie kontrolki jednak są przechowywane w pamięci ram. Dlatego jeśli tylko to nam nie utrudni zadania, usuń je.

Usuwanie kontrolek
Jest łatwiejsze niż ich ukrywanie. Używając tego samego polecenia, można usunąć okienko gry, nie wyłączając jej ☺.
Najpierw musimy uzyskać uchwyt do naszej kontrolki. Jest on zwracany przez funkcję CreateWindowEx. Wystarczy tylko przechować go w jakieś zmiennej.

```
BOOL DestroyWindow(HWND okno)
```

Chyba nie ma co się nad tym bardziej rozwodzić. Oto i przykład. Kontrolka zostanie w nim utworzona i zniknie po pięciu sekundach.

```
Cw = Win32API.new("user32","CreateWindowEx",'ippliiiiiiip','I')
Fw = Win32API.new("user32","FindWindow",'pp','I')
Dw = Win32API.new("user32","DestroyWindow",'I','I')
$hwnd = cw.call(nil,"temp")
$instance = Win32API.new("kernel32","GetModleHandle",'i','i').call(nil)
Przycisk =
Cw.call(0,"BUTTON","PRZYCISK",0x10000000|0x40000000,100,100,100,100,$hwnd,0
,$instance,nil)
Sleep(5)
Dw.call(Przycisk)
```

Ukrywanie okna

```
BOOL FindWindow(HWND okno, int rodzaj)
```

Oto I spis tego, co możemy wpisać jako rodzaj.

Flaga	Działanie
0	**Ukrycie**
1	**Normalne pokazanie**
2	**Pokazanie zminimalizowanego okna**
3	**Pokazanie maksymalizowanego okna**
4	**Pokazanie nieaktywowanego okna**
5	**Pokazanie tak, jak było**
6	**Minimalizuje, ale nie pokazuje, jeśli pokazanym nie było**
7	**Minimalizuje i dezaktywuje**
9	**Przywraca z zasobnika**
10	**Pokazuje tak, jak jest domyślnie**
11	**Maksymalny rozmiar**

Pominąłem ósemkę, gdyż nie mam pojęcia, jak działa.

Myślę, że temat jest dość krótki i dostatecznie omówiony.
Na koniec zalecam jeszcze użycie:
Graphics.update.

Okno dla kontrolek

Dowiedziałem się, że o ile programy odczytu tekstu widzą kontrolki w oknie RGSS, to jest ono na tyle wypełnione pustymi kontrolkami, że często po prostu nie widać tych zdefiniowanych przez nas. Postanowiłem więc znaleźć rozwiązanie i zamieścić je tutaj.
Jak już chyba wspominałem, w RGSS są problemy ze strukturami. O ile istnieją sposoby no ich tworzenie i odczyt, są bardzo mało precyzyjne i czasem po prostu nie działają.
Nie chciałem ryzykować tworzenia własnej klasy okna. Przecież to więcej niż struktura.
Mówiąc „klasa okna" nie mam na myśli klas ze zbiorami funkcji, a zarejestrowaną strukturę z danymi służącymi do rejestracji okna.

Każda klasa w RGSS – mówię o klasach funkcjonalnych – ma swoje miejsce w pamięci np.: 0xDE62A7C0FB. Klasy okna są zarejestrowane jako coś w stylu zmiennych, tyle że nie mamy do nich bezpośrednio dostępu – do ich zawartości. Dysponujemy tylko ich nazwą, a nie wartością.
Każda zmienna, którą tworzymy, tak naprawdę nie ma nazwy. Nazwy są tylko etykietami, które zapamiętywane są przez interpreter tylko na czas interpretacji. W językach kompilowanych zmienne przestają mieć wagę tuż po kompilacji i stają się jedynie adresami w pamięci, zawierającymi dane.
Jak więc widać, coś takiego jak nazwa zmiennych właściwie nie istnieje. Klasy okna są wyjątkami, nie dysponujemy ich wartością, a jedynie nazwą.

Dość jednak o prawdziwej naturze wszystkiego, bo jak tak dalej będę pisał, przejdziemy do Asemblera.

Zacząłem się zastanawiać nad oknem. Dzięki temu, że klasy mają stałą nazwę, a nie adresy w pamięci, ich dana się nie zmienia.

Gdybyśmy sprawdzili uchwyt do okna, wyłączyli i włączyli program i sprawdzili uchwyt ponownie, jego wartość by się zmieniła. Gdybyśmy sprawdzili nazwę klasy, wyłączyli i włączyli program ponownie, nazwa zostałaby bez zmian.

Dlatego postanowiłem sprawdzić nazwę klasy okna w RGSS i tutaj umieścić.
Teraz weź głęboki wdech i przygotuj się na tą wielką chwilę, gdy poznasz nazwę klasy okna w Ruby Game Scripting System. Ta nazwa to: "RGSS Player", myślę, że łatwo zapamiętać.
Dysponując tymi dwoma słowami i ich ukrytą potęgą, możemy stworzyć nowe okno dla gry. Ponadto możemy stworzyć okno identyczne, jak okno gry, a nawet okno gry skopiować. Wszystko to dzięki klasie "RGSS Player".

(Jeśli chcesz, możesz użyć też klasy "RGSS Player" zamiast tytułu okna w funkcji FindWindow.)
Teraz skopiuję przykładowy skrypt mojego autorstwa, który zamienia system wpisywania imienia na pole tekstowe w zewnętrznym oknie jako przykład stworzenia okna. W tworzonym oknie można wykorzystać wszystkie style i flagi wypisane kilkanaście stron wyżej.

```
$gametitle = "\0" * 512
fw = Win32API.new("user32","FindWindow",'pp','i')
inir = Win32API.new("kernel32","GetPrivateProfileString",'ppppip','i')
inir.call("Game","Title",'',$gametitle,511,".\\Game.ini")
$gametitle.delete!("\0")
$hwnd = fw.call(nil,$gametitle)
gmh = Win32API.new("kernel32","GetModuleHandle",'p','i')
$instance = gmh.call(nil)
$hwnd_class = "\0" * 256
Win32API.new("user32","GetClassName",'ipi','i').call($hwnd,$hwnd_class,255)
$hwnd_class.delete!("\0")
class Scene_Name
  def main
    @actor = $game_actors[$game_temp.name_actor_id]
    Win32API.new("user32","ShowWindow",'ii','i').call($hwnd,0)
    @crewin = Win32API.new("user32","CreateWindowEx",'ippiiiiiiiip','i')
    @nameparent =
@crewin.call(0x200,$hwnd_class,$gametitle,0xC00000|0x400000|0x10000000,10,1
0,640,480,0,0,$instance,nil)
    @name =
@crewin.call(512,"EDIT",@actor.name,0x10000000|0x40000000|1,30,30,600,450,@
nameparent,111,$instance,nil)
    Win32API.new("user32","SetFocus",'i','i').call(@name)
    Graphics.transition
    loop do
      Graphics.update
      Input.update
      update
      if $scene != self or @break == 1
        break
      end
    end
    Graphics.freeze
    Win32API.new("user32","ShowWindow",'ii','i').call($hwnd,0)
Win32API.new("user32","SetFocus",'i','i').call($hwnd)
Win32API.new("user32","DestroyWindow",'i','i').call(@nameparent)
Win32API.new("user32","UpdateWindow",'i','i').call($hwnd)
$scene = Scene_Map.new
      end
  def update
    if Win32API.new("user32","GetAsyncKeyState",'i','i').call(0x1B) != 0
      $game_system.se_play($data_system.cancel_se)
      @break = 1
    end
if Win32API.new("user32","GetAsyncKeyState",'i','i').call(0x0D) != 0
  $game_system.se_play($data_system.decision_se)
  @nazwa = "\0" * ($game_temp.name_max_char + 1)
Win32API.new("user32","GetWindowText",'ipi','i').call(@name,@nazwa,$game_te
mp.name_max_char)
@nazwa.delete!("\0")
@actor.name = @nazwa
@break = 1
  end
  end
  end
```

W skrypcie może być kilka niejasności. Funkcja GetAsyncKeyState pozwala na zbieranie danych z klawiatury. Wyjaśniłem to w ostatniej części kursu podstawowego, ale zrobiłem to szybko i wielu skryptów nie skopiowałem do końca. O funkcji tej jeszcze napiszę w dalszej części tego kursu.
Funkcja ta tutaj sprawdza, czy wciśnięty został klawisz enter. Nie mogłem wykorzystać Input.triggger, gdyż traktowałaby ona jako enter również spację i - co najgorsze - literę "C".
To ustawianie zmiennych zgodnie z jakimiś zmiennymi z $game_temp dotyczy tylko sytuacji wpisywania imienia bohatera, więc nie będę się nad nimi rozwodził.

$game_system.se_play to jedna z funkcji Game_System. Od Audio.se_play różni się tym, że mogę do niej wpisać zdefiniowane dźwięki w bazie danych, takie jak: $data_system.decision_se, $data_system.cancel_se, $data_system.cursor_se, $data_system.buzzer_se, $data_system.save_se, $data_system.load_se, $data_system.shop_se itd.

Jak widać po utworzeniu okna, przypisujemy do niego wszystkie tworzone kontrolki. Tymczasem ukrywamy prawdziwe okno gry, pozostaje po nim jedynie tytuł, ustawiany dla nowego okna. Po skończeniu wpisywania imienia, dane z pola tekstowego (tylko tyle liter, ile ustaliliśmy) wędrują do bufora i imię bohatera jest odpowiednio zmieniane. Okno z polem tekstowym jest niszczone (łącznie ze wszystkimi obiektami potomnymi) i przywracane jest okno gry. Uruchamiana jest klasa Scene_Map i kontynuowane jest przetwarzanie zdarzenia.
Myślę, że skrypt jest prosty do zrozumienia i nie pozostawia wiele wątpliwości po sobie. Nie wiem, co miałbym jeszcze tłumaczyć, jeśli chodzi o tworzenie nowych okienek.

Podsumowanie

Rozdział liczył z pewnością mniej lekcji niż poprzedni, był jednak mniej więcej tak samo długi i zapewne tak samo ważny.
Zadanie
MHM, jakie zadanie by tu znaleźć? Ach tak, już wczoraj wymyśliłem.
Temat o oknach liczył niewiele lekcji, ale był tak długi, jak część pierwsza, a może nawet dłuższy.
Znalazłem jednak zadanie przydatne w przyszłości i dobrze podsumowujące cały dział.
Spraw, by gra po włączeniu ukrywała swoje okienko. Następnie gra ta powinna uzyskać uchwyt do okienka RPG Makera XP, dalej działając w tle. Wymiary okienka RPG Makera winny być zmienione na 1200x900 pikseli. W prawym, dolnym rogu okienka należy zdefiniować prosty checkbox. Po jego kliknięciu winien on zniknąć, a okienko gry powinno się zmaksymalizować.

Rozwiązanie

```
def utf8(text)
  text = "\0" if text == nil
  to_char = Win32API.new("kernel32", "MultiByteToWideChar", 'ilpipi', 'i')
to_byte = Win32API.new("kernel32", "WideCharToMultiByte", 'ilpipipp', 'i')
utf8 = 65001
w = to_char.call(utf8, 0, text, text.size, nil, 0)
b = "\0" * (w*2)
w = to_char.call(utf8, 0, text, text.size, b, b.size/2)
w = to_byte.call(0, 0, b, b.size/2, nil, 0, nil, nil)
b2 = "\0" * w
```

```
w = to_byte.call(0, 0, b, b.size/2, b2, b2.size, nil, nil)
return(b2)
  end



begin
$gametitle = "\0" * 512
Win32API.new("kernel32","GetPrivateProfileString",'ppppip','i').call("Game"
,"Title","",$gametitle,511,".\\Game.ini")
$gametitle.delete!("\0")
$rpghwnd = Win32API.new("user32","FindWindow",'pp','i').call(nil,
$gametitle + " - RPG Maker XP")
if $rpghwnd <= 32 and $rpghwnd >= -32
print("Błąd! Nie udało się odnaleźć okna RPG Makera XP")
exit!
end
$hwnd = Win32API.new("user32","FindWindow",'pp','i').call("RGSS
Player",$gametitle)
if $hwnd <= 32 and $hwnd >= -32
print("Błąd! Nie udało się uzyskać uchwytu do okna gry")
exit!
end
Win32API.new("user32","ShowWindow",'ii','i').call($hwnd,0)
Win32API.new("user32","ShowWindow",'ii','i').call($rpghwnd,1)
Win32API.new("user32","SetFocus",'i','i').call($rpghwnd)
Graphics.update
Win32API.new("user32","SetWindowPos",'iiiiiii','i').call($rpghwnd,$rpghwnd,
0,0,1200,900,0x2|0x4|0x40)
$instance = Win32API.new("kernel32","GetModuleHandle",'p','i').call(nil)
$przyciskaktywacji =
Win32API.new("user32","CreateWindowEx",'ippliiiiiiiip','i').call(0x200,"BUTT
ON",utf8("Pokaż grę: " +
$gametitle),0x10000000|0x40000000|3,1000,700,300,300,$rpghwnd,123,$instance
,nil)
loop do
Graphics.update
if Win32API.new("user32","IsDlgButtonChecked",'ii','i').call($rpghwnd,123)
== 1
Win32API.new("user32","ShowWindow",'ii','i').call($hwnd,1)
Win32API.new("user32","SetFocus",'i','i').call($hwnd)
Win32API.new("user32","ShowWindow",'ii','i').call($przyciskaktywacji,0)
break
end
end
end
```

ROZDZIAŁ 3
Grafika

W poprzednim rozdziale omówiliśmy tworzenie okien, wypełnianie ich kontrolkami i zarządzanie nimi.

W tym rozdziale kontynuować będziemy naukę wszystkiego, co z oknami związane, tym razem skupimy się jednak na tworzeniu grafiki w WINAPI: począwszy od ręcznego wypełniania pikseli, poprzez tworzenie figur geometrycznych, po ładowanie obrazków z gotowych plików.

Przygotujmy wszystko, czego nam trzeba

W tym rozdziale chciałbym szczegółowo omówić grafikę, jej tworzenie i obsługę bitmap, animacji itp.
Zanim jednak coś przygotujemy, trzeba naszykować kilka rzeczy.
Po pierwsze, potrzebujemy okna, na którym będziemy malować, a co za tym idzie, uchwytu do niego.
Masz uchwyt do okna? Najłatwiejsze z głowy.
Mhm, oczywiście na samym oknie malować się nie da.
Musimy uzyskać tak zwany kontekst urządzenia, czyli wskaźnik karty graficznej na nasz obszar roboczy, w tym przypadku okno.
Kontekst urządzenia – o dziwo – można uzyskać bardzo łatwo.

```
HDC GetDC(HWND okno)
```

Funkcja ta znajduje się w bibliotece user32.dll.
Swoją drogą funkcji graficznych istnieje tyle, że aż dziwne, że nie powstałą dla nich nowa biblioteka ☺.
Dobrze, mamy nasz kontekst i co teraz?
Teraz cała masa teorii.
Zacznę od wspomnienia, że tworzyć będziemy grafikę 24bit. Tworzenie grafiki bardziej zaawansowanej jest trudniejsze, tak samo – jak o dziwo – tworzenie grafiki dla mniejszej liczby bitów.
Wykorzystamy tak zwany standard RGB (red, green, blue), o którym więcej będzie później.
Musimy też pamiętać, że część tworzonych buforów znajdzie się bezpośrednio przy karcie graficznej, która z reguły ma mniej pamięci niż RAM. Powinniśmy usuwać wszystkie zbędne dane, bo przy słabszych komputerach pojemność kart graficznych możemy liczyć nawet w kilkunastu MB.
Kiedy skończymy pracować nad grafiką, powinniśmy więc zwolnić nasz stworzony kontekst urządzenia. Zrób to poleceniem:

```
Int ReleaseDC(HWND okno, HDC kontekst)
```

Dobrze, mamy nasz piękny kontekst. Powinieneś wiedzieć jeszcze co WINAPI daje nam razem z kontekstem w pakiecie:

- Wszystkie niezbędne uchwyty
- Bufor dla bitmapy
- Czarne pióro
- Czarny pędzel
- Miejsce robocze w karcie graficznej
- Ustawienia głębi kolorów 24bit
- Standard RGB

Co jeszcze winienem tu napisać? Chyba nic, ewentualnie dopiszę później.

Pierwszy piksel

Czas umieścić coś na naszym obrazku – znaczy oknie.
Pierwszym, co zrobimy, będzie zmiana koloru jednego, jedynego piksela.
Domyślnie każdy piksel ma kolor tła okna, na przykład czarny. Wyjątkami są kontrolki.
Oto i składnia funkcji służącej do zmiany koloru piksela.

```
COLORREW SetPixel(HDC kontekst urządzenia, int pozycja x, int
pozycja y, COLORREW kolor)
```

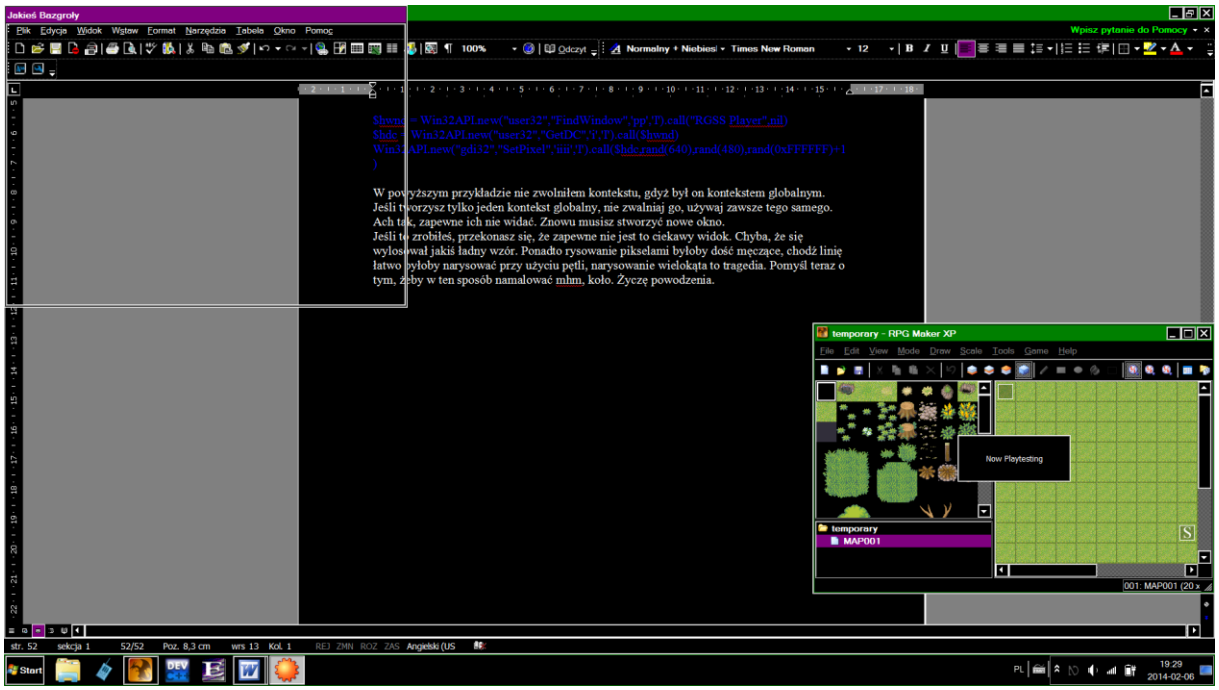
Jednak pomyliłem się i o czymś zapomniałem, ta funkcja nie znajduje się w bibliotece user32, a w bibliotece gdi32, która jest odpowiedzialna za karty graficzne.
Zarówno typ COLORREW, jak i HDC, w Win32API są typu 'i'.
Dobrze, ale skąd wytrzasnąć ten COLORREW.
I tu widać od razu dobrodziejstwo 24bit.
Wspomnę tylko, że w obrazach 24bit, obsługiwane jest 0xFFFFFF kolorów, czyli 16777215.
Teraz zobaczysz dobrodziejstwo metody RGB, a raczej RRGGBB.

Uwaga! Przedstawiona teraz prawda dotyczy tylko liczb heksadecymalnych!

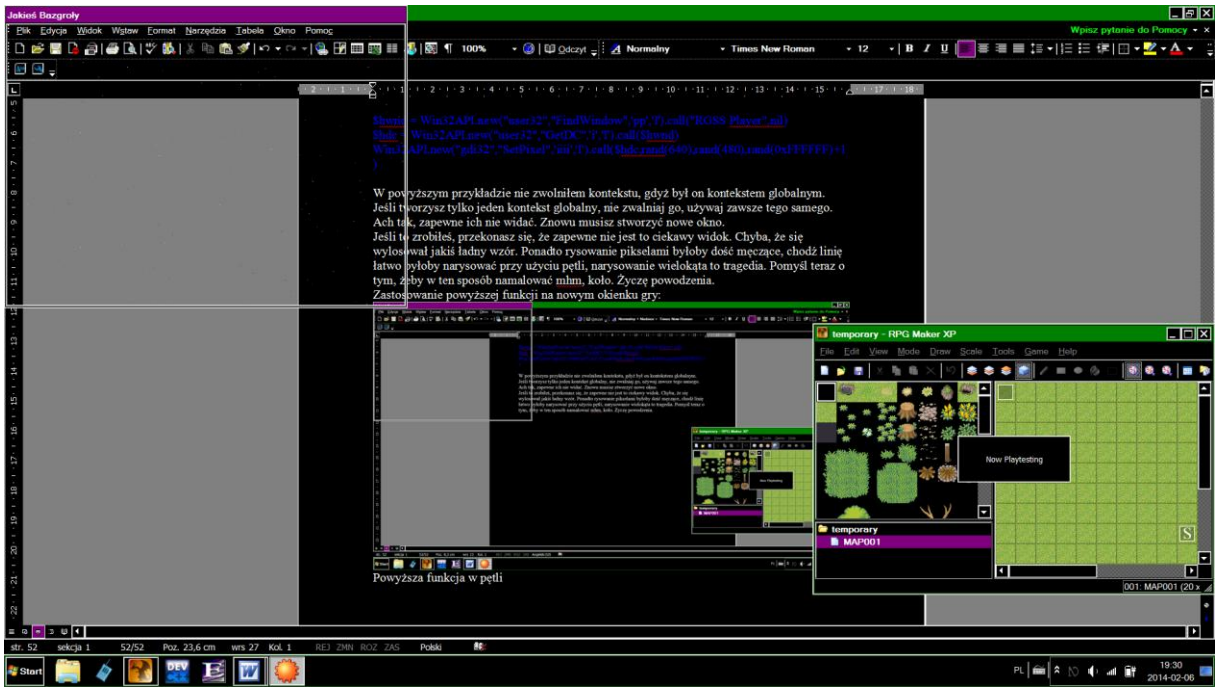
Jako, że 24bit kolorów obsługuje 2**24 (do potęgi), wynikową liczbą jest 0xFFFFFF. W tym układzie: dwie pierwsze cyfry oznaczają ilość niebieskiego składnika, dwie drugie ilość składnika zielonego, a dwie ostatnie ilość składnika czerwonego.
W sumie daje to ponad 33000000 kolorów, jeśli zaliczyć kryteria od -255 do 255.
Niestety ta funkcja pozwala nam tylko na uzyskanie niecałych 17000000 kolorów, gdyż nie można napisać 0x-F4F2-F5.
Oznacza to, że każdego koloru można dodać od 0 do 255.
Dość jednak o kolorach, wróćmy do naszego przykładu.
Oto i funkcja, która (przyjmując, że rozmiar okna to 640x480) narysuje w losowym miejscu okna piksel losowego koloru.

```
$hwnd = Win32API.new("user32","FindWindow",'pp','I').call("RGSS
Player",nil)
$hdc = Win32API.new("user32","GetDC",'i','I').call($hwnd)
Win32API.new("gdi32","SetPixel",'iiii','I').call($hdc,rand(640),rand(480),r
and(0xFFFFFF)+1)
```

W powyższym przykładzie nie zwolniłem kontekstu, gdyż był on kontekstem globalnym. Jeśli tworzysz tylko jeden kontekst globalny, nie zwalniaj go, używaj zawsze tego samego.
Ach tak, zapewne ich nie widać. Znowu musisz stworzyć nowe okno.
Jeśli to zrobiłeś, przekonasz się, że zapewne nie jest to ciekawy widok. Chyba, że się wylosował jakiś ładny wzór. Ponadto rysowanie pikselami byłoby dość męczące, bo choć linię łatwo byłoby narysować przy użyciu pętli, to narysowanie wielokąta to tragedia. Pomyśl teraz o tym, żeby w ten sposób namalować, mhm, koło. Życzę powodzenia.
Zastosowanie powyższej funkcji na nowym okienku gry:



Powyższa funkcja w pętli



Do uzyskania powyższego efektu użyłem takiego skryptu:

```
  to_char = Win32API.new("kernel32", "MultiByteToWideChar", 'ilpipi', 'i')
to_byte = Win32API.new("kernel32", "WideCharToMultiByte", 'ilpipipp', 'i')
utf8 = 65001
text = 'Jakieś Bazgroły'
w = to_char.call(utf8, 0, text, text.size, nil, 0)
b = "\0" * (w*2)
```

```
w = to_char.call(utf8, 0, text, text.size, b, b.size / 2)
w = to_byte.call(0, 0, b, b.size/2, nil, 0, nil, nil)
b2 = "\0" * w
w = to_byte.call(0, 0, b, b.size/2, b2, b2.size, nil, nil)
Win32API.new("user32","ShowWindow",'ii','i').call(Win32API.new("user32","Fi
ndWindow",'pp','i').call("RGSS Player",nil),0)
$okienko =
Win32API.new("user32","CreateWindowEx",'lppliiiiiiip','i').call(0x200,"RGSS
Player",b2,0x10000000,0,0,640,480,0,0,Win32API.new("kernel32","GetModuleHan
dle",'p','i').call(nil),nil)
$hdc = Win32API.new("user32","GetDC",'i','I').call($okienko)
for i in 0..199
Win32API.new("gdi32","SetPixel",'iiii','I').call($hdc,rand(640),rand(480),r
and(0xFFFFFF)+1)
end
for i in 0..199
  Graphics.update
  end
Win32API.new("user32","ShowWindow",'ii','i').call(Win32API.new("user32","Fi
ndWindow",'pp','i').call("RGSS Player",nil),1)
Win32API.new("user32","ShowWindow",'ii','i').call($okienko,0)
```

Okienko z obrazkiem pojawi się na jakieś 20 sekund, a następnie zniknie, zastąpione przez zwykłe okno gry.
Myślę, że zrobiliśmy nie jeden piksel, a wiele. Teraz namalujemy może linię?

Rysujemy linię

W winapi istnieje pewna funkcja, która rysuję linię od danego punktu. Funkcja jest chyba wygodniejsza, niż długa pętla.

```
BOOL LineTo(HDC kontekst, int x, int y)
```

W funkcji znajduje się ponoć pewien błąd.
Ostatni punkt linii nie jest malowany. Sprawia to, że długość linii musi być o punkt dłuższa, niż efekt. Nie znaczy to, że do x oraz y dodajemy 1. Nie jest malowany ostatni punkt powstającej linii, a nie ostatni punkt współrzędnych.
Funkcja jest dość prosta, od razu namalujemy linię.
Przyjmuję, że kontekst znajduje się w zmiennej hdc.

```
LineTo = Win32API.new("gdi32","LineTo",'iii','I')
LineTo.call(hdc,50,50)
```

Jak zapewne zauważyłeś, linia jest czarna, a my nie mamy na razie możliwości zmiany koloru.
Przyczyną są tak zwane pióra, o nich więcej już wkrótce.

Kilka figur geometrycznych

W winapi znajdują się specjalne funkcje, które rysują figury geometryczne.
Prostokąty
Prostokąty możemy oczywiście narysować zarówno używając LineTo, jak i SetPixel. Byłoby to jednak trudne, biorąc pod uwagę fakt, że trzeba je czymś wypełnić.
Funkcja rysująca prostokąty jest bardzo łatwa.

```
BOOL Rectangle(HDC kontekst, int x lewego-górnego rogu, int y górnego-lewego rogu, int x dolnego-prawego rogu, int y dolnego-prawego rogu.
```

```
Rectangle = Win32API.new("gdi32","iiiii",'I')
```

Myślę, że inicjacja jest bardzo prosta.
Przyjmując, że kontekst znajduje się w zmiennej hdc, tak narysujemy kwadrat.

```
Rectangle.call(hdc,10,10,20,20)
```

Elipsy
O ile kwadrat czy prostokąt jest narysować łatwo, elipsa, a nawet koło to czysta udręka.
Zacznijmy od prostego pytania: jakiemu programiście chce się obliczać przy pomocy PI wszystkie współrzędne koła?
Jeśli natomiast tworzymy jakiś edytor graficzny, nie będziemy chyba podawać komputerowi wzoru do obliczania, zwłaszcza wiernego.
Nie może być przecież tak, że za każdym razem komputer będzie obliczał wysokość koła przez skomplikowane definicje.
Tu z pomocą przychodzi nam funkcja Ellipse.
Nie musimy obliczać pola tego koła, ale nieco wiedzy geometrycznej niestety się przyda.
Funkcja ma jednak parę wad, o których łatwo się zapomina. Jeden piksel jest kwadratem, gdyż zwykle właśnie kwadraty tworzymy. Z tej przyczyny niemożliwym jest stworzenie idealnie idealnego koła. Możemy jednak stworzyć koło na tyle idealne, by na idealne wyglądało ☺.
Najlepiej, by pole takiego koła wynosiło 256px^2, wtedy koło wygląda dość wiarygodnie.
Składnia:

```
BOOL Elipse(HDC kontekst, int x lewo-góra prostokąta, Int y lewo-góra prostokąta, int x prawo-dół prostokąta, int y prawo-dół prostokąta)
```

O jakim znowu prostokącie autor tu plecie? My chcemy rysować koło!
Koło też narysujemy. Z powodu okrągłości koła, nie ma ono przecież wierzchołków, które możemy na nim umieścić.

Jako współrzędne podajemy wiec wierzchołki prostokąta, który zostanie opisany na tej elipsie. Oczywiście jeśli podane współrzędne będą tworzyć kwadrat, powstanie koło.
Przykładowe koło wygląda tak. Oczywiście przyjmujemy, że kontekst znajduje się w zmiennej hdc.

```
Win32API.new("gdi32","Ellipse",'iiiii','i').call(hdc,10,10,40,40)
```

Zajrzyjmy do piórnika

Ech, nudzi mnie ta grafika, ale już niewiele tematów i możemy przejść dalej. Osobiście za grafiką bowiem nie przepadam.
Autor się dziś zgłasza po ponad tygodniu nieobecności. ☺
Dzisiaj pomyślmy nad piórami.
Zapewne zauważyłeś, że wszystkie tworzone linie, okręgi, obwody itp. były koloru czarnego.
Jest to dość praktyczne, lepsze niż na przykład biel lub przezroczystość.
Często nie odpowiada nam jednak kolor czarny i chcemy sięgnąć głębiej.
Znów skorzystamy z RGB.
Pióro jest to obiekt, którym nanoszone są kontury i linie na obrazie (HDC). Domyślne pióro jest czarne, my stworzyć możemy jednak jakieś inne. Do dyspozycji mamy 256×256×256 barw.
Oto i składnia funkcji CreatePen (gdi32).

```
HPEN CreatePen(int rodzaj, int grubość, COLORREW kolor)
```

MHM, w MSDN nie widzę jakoś wartości zmiennych dla rodzaju, a jedynie nazwy stałych w C++.
Dlaczego zawsze trzeba zaglądać do kodu źródłowego Winapi? ☺

Styl	Efekt
0	Zwykłe, solidne pióro tworzące prostą linię
1	Pióro tworzące kreskowaną linię przerywaną
2	Pióro tworzące kropkowaną linię przerywaną
3	Pióro tworzące linię kropkowano-przerywaną
4	Pióro tworzące linię składającą się z jednej kreski i dwóch kropek
5	Tworzy najpiękniejszy obraz świata – pióro pozostawia niewidzialne linie
6	Trudno zrozumieć, o co chodzi. Wygląda na to, że pióro jest solidne, ale tylko wtedy, gdy używa go funkcja z gdi32.

Znakomicie, w C++ zdefiniowano więcej stylów, niż opisano w WINAPI. Jako że nie wiem, jak one działają, nie opisuję ich.
Pierwszy argument mamy więc z głowy.
Drugi argument określa szerokość pióra w pikselach. Zwykle jest to 1.
Trzeci argument to kolor w standardzie RGB.

Uwaga! Funkcja ta nie ustawia stworzonego pióra, jedynie je rejestruje.

Teraz możemy utworzyć nasz podstawowy piórnik.

```
CreatePen = Win32API.new("gdi32",'iii','I','I')
$bialepioro = CreatePen.call(0xFFFFFF)
$czerwonepioro = CreatePen.call(0,1,0x0000FF)
$zielonepioro = CreatePen.call(0x00FF00)
$niebieskiepioro = CreatePen.call(0,1,0xFF0000)
$czarnepioro = CreatePen.call(0x000000)
```

Skoro odwiedziliśmy już sklep i zakupiliśmy sześć piór, zajrzyjmy do piórnika i coś którymś namalujmy.
Do przypisania pióra używa się funkcji SelectObject (gdi32). Funkcja ta pozwala na przypisywanie różnych rzeczy do kontekstów, nie tylko piór.

```
HGDIOBJ SelectObject(HDC kontekst, HGDIOBJ obiekt)
```

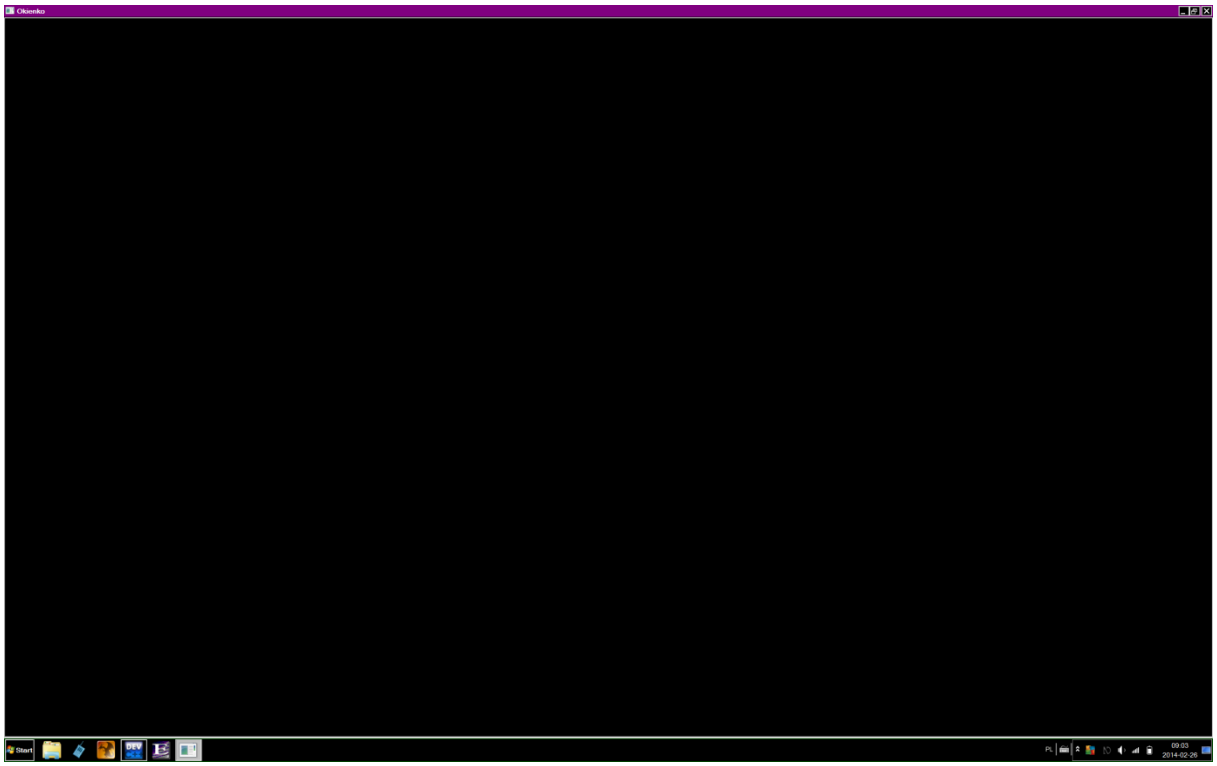
MHM co zwraca ta funkcja? Otóż funkcja zwraca uchwyt do poprzedniego obiektu danego typu, na przykład pióra.
Kiedyś pióra te się przechowywało i usuwało, teraz pamięć RAM jest większa i rzadziej się to robi. Pod koniec tej lekcji pokażę jednak to dla ciekawskich.
Przypuśćmy, że kontekst znajduje się w zmiennej hdc.
Wybierzmy sobie to ładne, niebieskie pióro.

```
Win32API.new("gdi32","SelectObject",'ii','I').call(hdc,$niebieskiepioro)
```

Teraz możemy namalować piękne kółko, którego okrąg będzie niebieski.

```
Win32API.new("gdi32","Ellipse",'iiiii','i').call(hdc,10,10,40,40)
```

Tak wygląda okienko z kółeczkiem, przy którego rysowaniu użyto czerwonego pióra (C++).



Teraz pokażę jeszcze jak usunąć niepotrzebne pióro.
Używana jest tu funkcja DeleteObject (gdi32).
Uwaga! Nie można usuwać obiektów przypisanych do kontekstu!

```
BOOL DeleteObject(HGDIOBJ obiekt)
```

Kupmy sobie pędzel

Zapewne zauważyłeś, że w kółeczku zmienił się tylko kolor obwodu, a nie środek.
Wnętrza figur nie są bowiem rysowane piórem, a pędzlem.
Stwórzmy więc sobie taki pędzel.
Użyjmy funkcji CreateSolidBrush (gdi32).
Funkcja ta tworzy solidny pędzel, który o dziwo nigdy się nie popsuje ☺.
Pędzel ten będzie tworzył proste piksele.

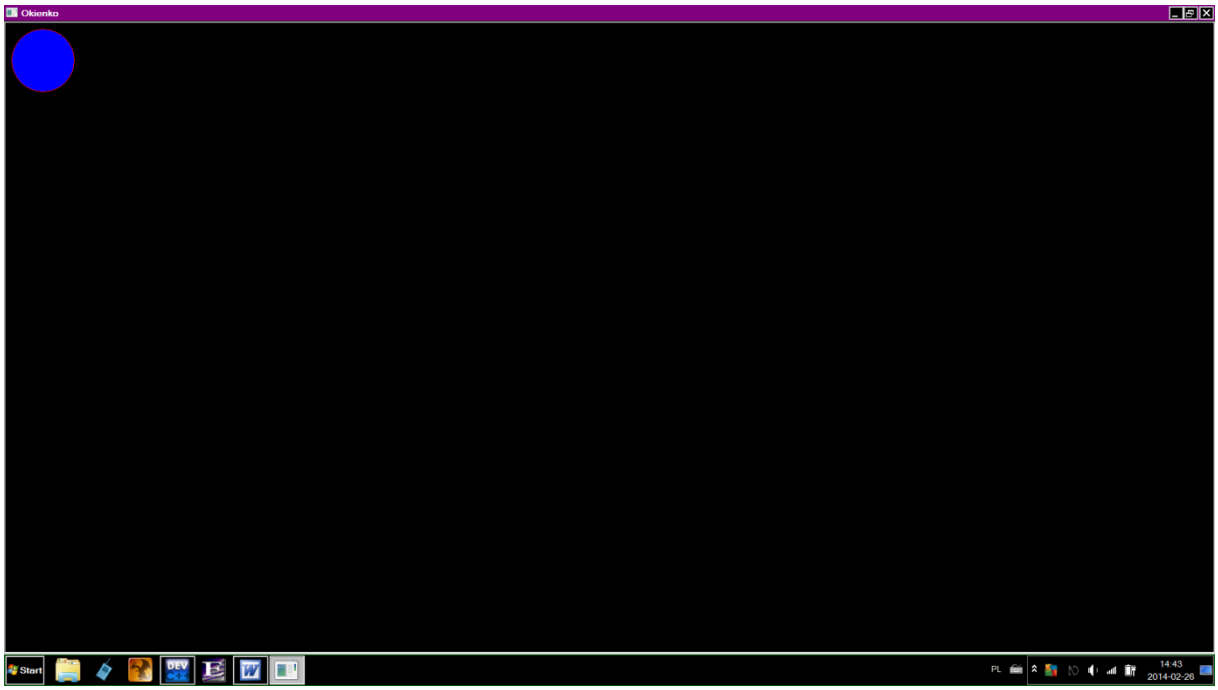
```
HBRUSH CreateSolidBrush(COLORREW kolor)
```

Jak widać, funkcja ma tylko jeden argument: kolor.
Stworzony pędzel przypisujemy funkcją SelectObject.
Oprócz funkcji CreateSolidBrush istnieją funkcje do tworzenia innych pędzli, ale są dość skomplikowane.

Kiedy już mamy pędzel, możemy go przypisać funkcją SelectObject. Usuwamy go natomiast DeleteObject.
Oto przykład zmiany pędzla, przyjmuję, że kontekst jest w zmiennej hdc.

```
$niebieskipedzel =
Win32API.new("gdi32","CreateSolidBrush",'i','i').call(0xFF0000)
Win32API.new("gdi32","SelectObject",'ii','I').call(hdc.$niebieskipedzel)
```

Jeszcze jakiś przykładzik w C++?
Okno z kółkiem, czerwone pióro i niebieski pędzel (C++).



Bitmapy

Oj, to będzie ciężki orzech do zgryzienia.
Co może być gorsze od bitmap w winapi?
Chyba tylko moduł graph Pascala lub rysowanie w asemblerze ☺.
Po pierwsze, choć to może się wydawać niemożliwe, formaty png i JPG trudniej ładować niż bitmapy. Dlatego zajmiemy się właśnie bitmapami.
Dalej korzystamy ze standardu RGB, więc potrzebujemy bitmapy najlepiej 24bitowej lub mniejszej, na przykład 256 kolorów.
Aby zdobyć bitmapę, potrzeba nam będzie, mhm, kontekstu tej bitmapy. Po zdobyciu kontekstu obrazka, skopiujemy go na kontekst okna i mamy bitmapę.
W praktyce jest to jednak trudniejsze niż w teorii.
Najpierw trzeba stworzyć dla bitmapy kontekst, który będzie rozmiarów naszego kontekstu. W efekcie niezbędny jest kontekst okna.

Funkcja CreateCompatibleDC (gdi32)

```
HDC CreateCompatibleDC(hdc kontekst źródłowy)
```

Po wykorzystaniu tej funkcji, stworzymy, nie przejmiemy, kontekst. Stworzony kontekst usuwa się inaczej, ale o tym później.
Teraz załadujmy jakąś bitmapę do zmiennej.

```
Handle LoadImage(HINSTANCE instancja, LPCTSTR nazwa, UINT typ, int
szerokość, int wysokość, UINT flagi)
```

Jak widać, składnia jest dość skomplikowana.
Funkcją tą można ładować bitmapy, ikony i kursory.
Pierwszy parametr jest niezbędny tylko wtedy, gdy ładujemy pliki z zasobów. W przeciwnym wypadku może być równy 0.
Drugi parametr to oczywiście ścieżka do pliku lub zasobu, podana jako string.
Trzeci parametr to rodzaj ładowanego pliku: bitmapa, kursor, ikonka lub meta. My ładujemy bitmapę, wpisz 0 (1 to ikona, 2 to kursor, 3 to metadata).
Parametry szerokości i wysokości nie są wymagane przy bitmapach, tylko przy kursorach i ikonach.
Flag jest całkiem sporo.

Flaga	Działanie
0x2000	Ładuje dipsekcję bitmapy, piksele a nie kolory, powoduje utratę informacji o kolorach
0x0	Zachowuje kolory z bitmapy
0x40	W przypadku kursorów i ikon, ustawia domyślne rozmiary systemowe
0x10	Drugi parametr wskazuje na nazwę pliku
0x1000	Włącza kolory 3D, jeśli takie obsługuje bitmapa
0x20	Wykorzystuje na zdjęciu kolory okna
0x1	I zastępuje inne kolory tymi barwami – podgląd monochromatyczny
0x8000	Pozwala na współdzielenie obrazu. Powoduje to, że wszystkie uchwyty do obrazu są wspólne. Pozwala to na współdzielenie obrazu między aplikacjami tak, by zmiany wykonane w jednej aplikacji, pojawiły się w drugiej.
0x80	Używa kolorów VGA

Sukces, mamy załadowaną bitmapę! Jak ją jednak wyświetlić? Na razie się nie pokazuje ☹.
Po miesiącu wracam, by to napisać.
Najpierw musimy stworzyć kontekst bitmapy (nazywać go będę oknem bitmapy, pamiętaj jednak, że nie jest to okno systemowe (HWND) i nie może być pokazane).
Najpierw stworzymy kontekst kompatybilny z kontekstem naszego okna. (gdi32)

```
HDC CreateCompatibleDC(HDC kontekst naszego okna)
```

Teraz, używając wcześniej poznanej funkcji SelectObject, do nowo utworzonego kontekstu przypisz naszą bitmapę.
Mhm, teraz czas na funkcję BitBlt.

Znajduje się ona w bibliotece gdi32.dll.
Oto i składnia.

```
BOOL BitBlt(HDC kontekst okna docelowego, int pozycja x w oknie docelowym, int pozycja y w oknie docelowym, int szerokość, int wysokość, HDC kontekst okna źródłowego, int pozycja x źródła, int pozycja y źródła, DWORD flagi)
```

Myślę, że nie ma co tłumaczyć, prócz kilku uwag. Źródłem jest nasz stworzony kontekst, do którego przypisaliśmy naszą bitmapę. Celem jest nasze okno gry, program czy czego tam używamy.
Pozycja x i y źródła winna być zawsze równa 0, by skopiować wszystko.
Flag nie będę pokazywał, gdyż są dość skomplikowane i opierają się głównie na sposobach kopiowania bitmapy.
Zamiast tego może pokażę skrypt, ładujący plik bitmap.bmp i go wyświetlający w nowym oknie.

Uwaga! Używając funkcji LoadImage i BitBlt, możemy operować tylko na plikach *.bmp.

```
$hwnd = Win32API.new("user32","FindWindow",'pp','i').call("RGSS Player",nil)
$hinstance = Win32API.new("kernel32","GetModuleHandle",'p','i').call(nil)
$hwndbitmap = Win32API.new("user32","CreateWindowEx",'lppliiiiiiiip','i').call(0x200,"RGSS Player","Bitmapa",0x10000000,0,0,640,480,0,0,$hinstance,nil)
Win32API.new("user32","ShowWindow",'ii','i').call($hwnd,0)
$hdchwndhbitmap = Win32API.new("user32","GetDC",'i','i').call($hwndbitmap)
$hbitmap = Win32API.new("user32","LoadImage",'ipiiii','i').call($hinstance,"bitmap.bmp",0,640,480,0x10|0x8000)
$hdchbitmap = Win32API.new("gdi32","CreateCompatibleDC",'i','i').call($hdchwndhbitmap)
Win32API.new("gdi32","SelectObject",'ii','i').call($hdchbitmap,$hbitmap)
Win32API.new("gdi32","BitBlt",'iiiiiiiii','i').call($hdchbitmap,0,0,640,480,$hdchwndhbitmap,0,0,0)
loop do
Graphics.update
End
```

Należałoby jeszcze usunąć utworzony kontekst.
Jako że kontekst był stworzony (create), a nie pobrany (get), usuwamy go, a nie zwalniamy.
Używamy zatem funkcji DeleteDC (gdi32).

```
BOOL DeleteDC(HDC kontekst)
```

Jak widać, ładowanie bitmap nie jest proste.
Musimy jednak tak się bawić, jeśli chcemy edytować bitmapy.

Jeśli nie robimy nic zbyt zaawansowanego, polecam RGSS-owy Sprite, którego się używa z pewnością szybciej, chyba że napiszemy odpowiednią funkcję.
Ponadto Sprite oferuje więcej opcji, choć są opcje oferowane przez LoadImage, których Sprite nie obsługuje.

Uwaga! Zapewne domyślasz się, że jeśli korzystamy z tak niskopoziomowych funkcji, wszystko nie może być tak łatwe. Otóż kiedy okno zostanie zminimalizowane lub fragment okna z bitmapą przez coś zamazany (dotyczy również rysunków), fragment ów zostanie usunięty.
Proponuje więc użycie wielowątkowości (czytaj niżej).

Podsumowanie

Zadanie
Narysuj okno z czerwonym kwadratem o niebieskiej obwódce. Potem dodaj do tego bitmapę bitmap.bmp. Po jakimś czasie bitmapa (w osobnym oknie) winna zniknąć, a poprzednie okno winno wrócić do łask.

Rozwiązanie

```
$hwnd_old = Win32API.new("user32","FindWindow",'pp','i').call("RGSS
Player",nil)
$hinstance = Win32API.new("kernel32","GetModuleHandle",'i','i').call(0)
Win32API.new("user32","ShowWindow",'ii','i').call($hwnd_old,0)
$hwnd =
Win32API.new("user32","CreateWindowEx",'lppliiiiiiiip','i').call(0x200,"RGSS
Player","bitmap.bmp",0x10000000,0,0,640,480,0,0,$hinstance,nil)
$hdc = Win32API.new("user32","GetDC",'i','i').call($hwnd)
pioro = Win32API.new("gdi32","CreatePen",'iii','i').call(0,1,0xFF0000)
pedzel = Win32API.new("gdi32","CreateSolidBrush",'i','i').call(0x0000FF)
Win32API.new("gdi32","SelectObject",'ii','i').call($hdc,pioro)
Win32API.new("gdi32","SelectObject",'i','i').call($hdc,pedzel)
Win32API.new("gdi32","Rectangle",'iiiii','i').call($hdc,0,0,20,20)
$hbitmap =
Win32API.new("user32","LoadImage",'ipiiii','i').call($hinstance,"bitmap.bmp
",0,640,480,0x10|0x8000)
$hdc_hbitmap =
Win32API.new("gdi32","CreateCompatibleDC",'i','i').call($hdchwndhbitmap)
Win32API.new("gdi32","SelectObject",'ii','i').call($hdch_bitmap,$hbitmap)
Win32API.new("gdi32","BitBlt",'iiiiiiiii','i').call($hdc_hbitmap,0,0,640,48
0,$hdc,0,0,0)
for i in 0..1048576
Win32API.new("user32","UpdateWindow",'i','i').call($hwnd)
Graphics.update
end
Win32API.new("user32","DeleteDC",'i','i').call($hdc_bitmap)
Win32API.new("user32","ReleaseDC",'i','i').call($hdc)
Win32API.new("user32","DestroyWindow",'i','i').call($hwnd)
Win32API.new("user32","ShowWindow",'ii','i').call($hwnd_old,1)
$hwnd = $hwnd_old
$hdc = Win32API.new("user32","GetDC",'i','i').call($hwnd)
```

ROZDZIAŁ 4
Zasoby

W tym dziale zajmiemy się zasobami, czyli materiałami dołączanymi bezpośrednio do plików wykonywalnych: dodawaniem muzyki i grafiki bezpośrednio do pliku *.exe, zmianą uprawnień pliku i tworzeniem menu aplikacji.

Czym są zasoby?

Nazwa rozdziału może brzmieć tajemniczo. Dlatego najpierw wyjaśnię, czym są zasoby.
Zasoby są to dane dołączone do pliku *.exe. Zasobami możemy nazwać wszystkie dołączone pliki – obrazki, menu, dialogi, pliki – o ile znajdują się one w samym pliku binarnym: *.exe, *.dll…
W skrócie mówiąc zasobami są materiały połączone z plikiem *.exe.
Zasobem nie można natomiast nazwać naszego kodu źródłowego, gdyż on sam tworzy plik *.exe, a nie jest do niego dołączony.
Edytory takich języków, jak C++, mają już domyślnie funkcję tworzenia zasobów. Nie trzeba wtedy ich tworzyć ręcznie.
Niestety, edytor skryptów nie daje nam takiej możliwości.
Zanim jednak napiszemy jakieś zasoby, przejrzymy domyślne zasoby naszej gry (Game.exe).
Otworzyłem ten pliczek w edytorze heksagonalnym. Co widzę?
Plik zawiera cztery tzw. Grupy zasobów: Icon, Icon Group, Version Info oraz 24. Pierwsze dwa zasoby odpowiedzialne są za ikonki. Do Icon należą zasoby numerowane jako 1—8. Do Icon Group należy jedynie zasób numerowany 1. Te zasoby jednak na razie nas nie interesują.
Do grupy Version należy tekst w specjalnym języku. Tekst ten zawiera informacje o wersji programu.
Wklejam go tutaj do wglądu.
Pełna ścieżka wygląda tak: Version\1\0.

```
1 VERSIONINFO
FILEVERSION 1,0,0,1
PRODUCTVERSION 1,0,0,1
FILEOS 0x40004
FILETYPE 0x1
{
BLOCK "StringFileInfo"
{
        BLOCK "000004b0"
        {
                VALUE "FileDescription", "RGSS Player"
                VALUE "FileVersion", "1, 0, 0, 1"
                VALUE "LegalCopyright", "Copyright (C) 2004 Enterbrain, Inc. / Yoji Ojima"
                VALUE "ProductVersion", "1, 0, 0, 1"
        }
}

BLOCK "VarFileInfo"
{
        VALUE "Translation", 0x0000 0x04B0
}
}
```

Na razie nas może zainteresować pierwsza linijka. Jej skład jest prosty:
NUMER_IDENTYFIKACYJNY TYP
Tutaj numerem jest 1, a typem VERSIONINFO, czyli informacje o wersji.

Zasób 24\1\1024 wygląda ciut inaczej:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
   version="1.0.0.0"
   processorArchitecture="X86"
   name="Enterbrain.RGSS.Game"
   type="win32"
/>
<description>RGSS Player</description>
<dependency>
   <dependentAssembly>
      <assemblyIdentity
         type="win32"
         name="Microsoft.Windows.Common-Controls"
         version="6.0.0.0"
         processorArchitecture="X86"
         publicKeyToken="6595b64144ccf1df"
         language="*"
      />
   </dependentAssembly>
</dependency>
</assembly>
```

Jak widać, jest to zwykły XML z pewnym dodatkiem.
Są to dwa najczęściej spotykane sposoby zapisu tekstowych zasobów.
Należy pamiętać, że informacje takie, jak VERSION INFO czy ICON nie są definiowane przez użytkownika.
Program, którego używam, sam rozpoznał, że zasób jest ikonką na podstawie liczbowego zapisu typu zasobu. W zasobie nie możemy napisać po prostu "ICON" chyba, że mamy program, który to rozpozna.
W poniższej tabeli pokazuje jaki numer przypada jakiemu typu zasobów.

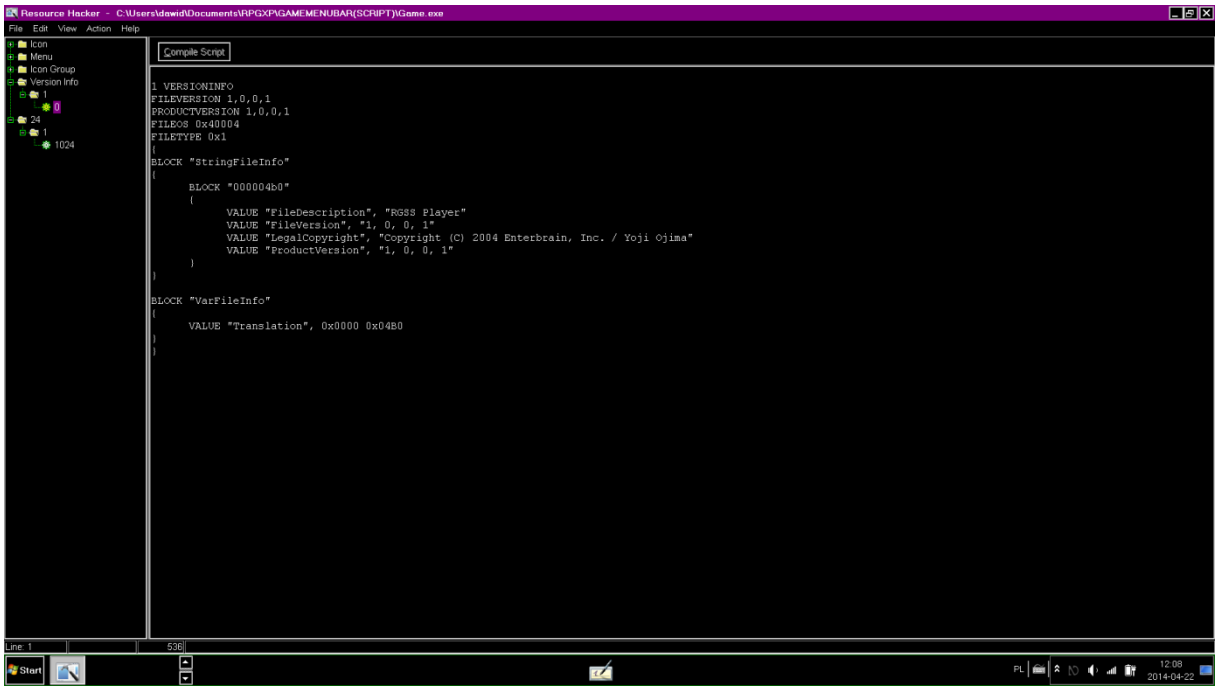
Numer	Typ
1	Kursor
2	Bitmapa
3	Ikona
4	Menu
5	Okno dialogowe
6	Łańcuch znakowy – jakiś napis
7	Katalog czcionek
9	Akceleratory
10	Inne zasoby
11	Grupa kursorów, grupa ikon, tabela wiadomości
16	Informacje o wersji
17	Dane dołączane do dialogów

19	Zasób PLUG AND PLAY
20	VXD
21	Kursor animowany
22	Animowana ikona
23	Kod HTML
24	Manifest

Jak widać, do zasobu jedenastego typu można przypisać wiele elementów. Przyjmuje on różne zbiory: ikon, kursorów czy komunikatów.
Tabele komunikatów są wykorzystywane w większych aplikacjach, aby ułatwić tłumaczenie tychże na inne języki.
Tabele komunikatów możemy znaleźć nawet w bibliotece DLL RGSS-a.
Chwilkę, zasoby znajdują się w plikach DLL?
Owszem. Zasoby można umieścić w absolutnie każdym pliku wykonywalnym - *.exe, *.dll czy *.elf.
Mhm. Wiemy już, czym są zasoby, ale jak je dodać, a tym bardziej obsłużyć?

Zapis i odczyt zasobów

Kiedy korzystamy z popularniejszych języków programowania lub, jak to nazwę, z tych przeznaczonych do tworzenia wielkich aplikacji, a nie głównie gier, dodania zasobów możemy dokonać w samym kodzie źródłowym aplikacji. Zwykle dodanie zasobów obejmuje stworzenie pliku zasobów *.rc i dodanie go do tak zwanego linkera, co większość programów robi automatycznie.
Oczywiście, w RGSS-sie nie ma ani zaawansowanego edytora, ani linkera.
Musimy więc sobie radzić sami.
Skorzystamy teraz z jakiegoś edytora heksadecymalnego. Edytorami heksadecymalnymi nazywamy aplikacje służące do modyfikacji plików binarnych. Mamy całkiem spory wybór naszego edytorka. Ja jednak polecam Resource Hackera, który nie ma olbrzymich funkcji, ale jest za to szybki i prosty w obsłudze, a co może okazać się ważne dla wielu użytkowników, ma polską wersję językową.
Zrzut jakiegoś zasobu z pliku Game.exe (Resorce Hacker, Angielska Wersja Językowa – tylko taką tutaj mam):



Dobrze. Zacznijmy od dodania do naszej gry jakieś bitmapy.
Jeśli chcemy, możemy dodać ją do jakieś dll'ki lub innego exe'ka. Po co jednak komplikować sobie życie? Użyjmy pliku Game.exe.
Kiedy go otworzysz w reshacku, wybierz z paska menu ACTION, a następnie ADD A NEW RESOURCE.
Teraz wybierz plik z bitmapą i wybierz dla niego jakieś ID oraz wewnętrzne ID.
ID to identyfikator, używając którego odnajdziemy nasz zasób. Wewnętrzne ID to tylko język zasobu, właściwie nie ma znaczenia i radzę tutaj wpisać 1024 (nieokreślony), 1033 (angielski) – tak się przyjęło. Nic nie stoi jednak na przeszkodzie wpisania tutaj 21 – język polski.
Aby ułatwić sobie życie, za ID zasobu uznajmy jakieś słowo, na przykład "MAINBITMAP".
W C++ istnieje makro MAKEINTRESOURCE, które bezproblemowo ładuje zasoby numerowane. Oczywiście, RGSS czegoś takiego nie posiada.
Dobrze. Jeśli dodałeś nowy zasób i zapisałeś plik, możemy teraz go odczytać.
By załadować bitmapę z zasobów, nie możemy użyć funkcji LoadImage. Użyjemy zamiennika zwanego LoadBitmap (user32).
Funkcja jest zdecydowanie prostsza od LoadImage. Niestety, można ją stosować tylko w przypadku zasobów.

```
HBITMAP LoadBitmap(HINSTANCE instancja, LPCSTR zasób)
```

Naszą bitmapę załadujemy więc tak.

```
Bitmapa =
Win32API.new("user32","LoadBitmap",'ip','I').call(Win32API.new("ker
nel32","GetModuleHandle",'I','i').call(0),"MAINBITMAP")
```

Nasza bitmapa jest załadowana.
Analogicznie, kursory ładujemy funkcją LoadCursor, a ikony LoadIcon. Składnia jest taka sama.

Wersja naszej aplikacji

Zapewne zauważyłeś, że jeśli wystąpi poważny błąd w naszej grze, system informuje, że program RGSS Player przestał działać. Ponadto, we właściwościach widnieje nazwa programu RGSS Player, mimo że plik zowie się Game.exe.
Przyczyną nie jest nazwa klasy okna, ale zdefiniowane informacje o wersji w zasobach.
Najpierw jednak pewna uwaga. Komentarze w zasobach są uprzedzone znakiem // .
Oto i skrypt wersji z komentarzami.

```
1 VERSIONINFO //numer zasobu I informacja o typie
FILEVERSION 1,0,0,1 //wersja pliku
PRODUCTVERSION 1,0,0,1 //wersja produktu
FILEOS 0x40004 //system operacyjny
FILETYPE 0x1 //typ pliku
{
BLOCK "StringFileInfo"
{
	BLOCK "000004b0"
	{
		VALUE "FileDescription", "Moja Gra" //nazwa produktu
		VALUE "FileVersion", "1, 0, 0, 1" //powtórzona wersja pliku
		VALUE "LegalCopyright", "Copyright (C) 2004 Enterbrain, Inc. / Yoji Ojima"
//copyright
		VALUE "ProductVersion", "1, 0, 0, 1" //powtórzona wersja produktu
	}
}

BLOCK "VarFileInfo"
{
	VALUE "Translation", 0x0000 0x04B0 //tłumaczenie
}
}
```

Wygląda to skomplikowanie, ale skomplikowane nie jest.
Zaraz, zaraz… Jak to dodać? Szczerze, wystarczy wyedytować już istniejące informacje o wersji i skompilować.

Tworzymy menu

Większość aplikacji, a nawet część gier, ma pasek Menu. My też sobie taki stwórzmy.
Najpierw stwórz plik w notatniku i zapisz go jako *.rc.
Teraz możemy napisać menu.
Skrypt zapisz w utworzonym pliku.

```
MAINMENU MENU
LANGUAGE LANG_POLISH, 0x0
```

```
{
POPUP "&Gra"
{
        MENUITEM "Pełny &Ekran...",  9001
        MENUITEM SEPARATOR
        MENUITEM "&Ustawienia RGSS...",  9002
        MENUITEM SEPARATOR
        MENUITEM "Menu &Tytułowe",  9003
        MENUITEM "Menu &Gry",  9004
        MENUITEM SEPARATOR
        MENUITEM "W&yjście",  9005
}
}
```

Od razu zaznaczę, że deklaracja języka nie jest konieczna, program sam ją dopisze.
MAINMENU to nazwa menu, której będziemy używać do identyfikacji.
Teraz trochę o samym Menu.
Aby stworzyć menu rozwijalne, używamy funkcji POPUP "NAZWA".
Takie menu można rozwinąć, by ukazać więcej elementów.
Nadrzędnym menu tego typu są na przykład: plik, edycja, widok…
Nic nie stoi jednak na przeszkodzie, by wewnątrz tych menu nie stworzyć kolejnych menu rozwijalnych.
Funkcja `MENUITEM "ETYKIETA", ID` służy do dodawania przycisków menu. Numer identyfikatora musi być większy od 1024 i jest wymagany, by zorientować się, który przycisk został wciśnięty.
Teraz mała niespodzianka. Zapewne zauważyłeś, że w menu występują również obiekty zaznaczone lub szare, których nie można wcisnąć.
Dodaje się je w ten sposób:

```
MENUITEM "Pełny &Ekran", 9071, CHECKED //zaznaczony obiekt
MENUITEM "W&yjście", 9075, GRAYED //zablokowany obiekt
```

OK, menu jest gotowe.
Teraz uruchom resorce hacker i otwórz w nim utworzony plik. Następnie z opcji ACTION, wybierz save as a res file. Zapisz gdzieś ten plik.
Teraz otwórz nasz plik Game.exe i wybierz opcje ADD New resorce i wybierz zapisany plik *.res.
Zasób został dodany.
Teraz wróćmy do Ruby'ego i dodajmy to menu.

```
$instance = Win32API.new("kernel32","GetModuleHandle",'i','I').call(0)
$wnd = Win32API.new("user32","FindWindow",'pp','I').call("RGSS Player",nil)
$menu =
Win32API.new("user32","LoadMenu",'ip','I').call($instance,"MAINMENU")
Win32API.new("user32","SetMenu",'ii','I').call($wnd,$menu)
```

Menu zostało dodane i możemy wciskać różne opcje. Nie widać jednak efektu, prócz tego, że menu się zamyka.

Efekt jednak dodamy dopiero wtedy, jak nauczysz się pisać pętlę komunikatów i procedurę obsługi wiadomości okna.
Oczywiście, menu możemy stworzyć bez zasobów, ale rejestrując strukturę dla każdego obiektu i tworząc wszystko od właściwie zera - przynajmniej dziesięć razy dłuższy kod.

Tablice łańcuchów znaków

Czasem może się zdarzyć sytuacja, że nasz program stanie się sławny i będzie tłumaczony na inne języki.
Nie chcemy zwykle oddawać tłumaczom kodu źródłowego, a sami nie znamy przecież każdego języka.
Ponadto, napisy w kodzie źródłowym można znaleźć niemal wszędzie.
Tu z pomocą przychodzą tablice ciągów znaków.
Czym one są?
Tablice ciągów znaków to obiekty w zasobach, który mają przypisane do identyfikatorów teksty.
W efekcie możemy stworzyć dwa łańcuchy o identyfikatorach 1001 i 1002. Pierwszy zawiera tekst "Dzień Dobry", drugi "Do Widzenia". Jeśli ktoś poczuje potrzebę przetłumaczenia naszej aplikacji, wystarczy, że w zasobach zmieni te napisy i nie musi już grzebać w kodzie źródłowym.
Dobrze, ale jak to wygląda w praktyce?

W ten sposób piszemy skrypt w zasobach. Nasz identyfikator zasobu to 1000, a napisów to kolejny: 1001, 1002, 1003, 1004 i 1005.

```
1000 11
{
1001, "Nowa gra"
1002, "Wczytaj grę"
1003, "Wyjście"
1004, "Zapisz grę"
1005, "Restart gry"
}
```

Zastanówmy się teraz jak odczytać ten napis.

```
Int LoadString(HINSTANCE instancja, UINT identyfikator, LPTSTR
bufor, int długość)
```

Mhm, ciężko byłoby to wszystko tworzyć za każdym razem używając stringów.
Lepiej napisać sobie taką funkcję.

```
Def getstring(id)
String = "\0" * 16384
Win32API.new("user32","LoadString",'iipi','I').call(Win32API.new("kernel32"
,"GetModuleHandle",'p','I').call(0),id,string,string.size)
```

```
String.delete!("\0")
Return string
End
```

Teraz wystarczy wpisać getstring(1001), aby uzyskać w wyniku "Nowa gra".

Żądamy dostępu administratora

Przyjmijmy, że w RGSS-sie albo w RUBY piszemy instalator lub inny program wymagający dostępu administratora.
W takim przypadku użytkownik musi wybrać z menu opcję "Uruchom jako administrator".
Może jednak o tym zapomnieć.
W konsekwencji program nie będzie poprawnie działał.

Może pamiętasz, że kiedy pisałem o rodzajach zasobów, pojawił się tam manifest?
W programowaniu manifest to zasób, który dostarcza informacje o pliku: wersja, rodzaj, język, sposób uruchamiania, środowisko, architektura procesora…
My wyedytujemy niezbędne uprawnienia.
Oto kod, który należy wpisać jako 24\1\1033 (lub inny język).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
 <security>
  <requestedPrivileges>
   <requestedExecutionLevel level="requireAdministrator" uiAccess="false"/>
  </requestedPrivileges>
 </security>
</trustInfo>
</assembly>
```

Jako uprawnienia możemy podać "requireAdministrator" – administrator lub "asInvoker" – użytkownik, bez wymaganych dodatkowych uprawnień.
Nawiasem mówiąc, chyba właśnie pokazałem jak dodawać skrypty manifest do pliku wykonalnego.

Dlaczego tylko plik naszej aplikacji?

Przecież możemy zechcieć załadować inny plik wykonywalny. Zacznijmy od pliku *.exe.
Aby pobrać zasoby z dowolnego pliku potrzebujemy jednej rzeczy – jego instancji.
Kiedy używamy naszego programu, mówimy o instancji aplikacji. Jest ona taka sama, jak instancja naszego pliku *.EXE. Z tej przyczyny nie musimy dodatkowo szukać naszego pliku. Może się jednak zdarzyć, że chcemy pobrać zasoby z innego pliku, gdyż np. nasza aplikacja składa się z kilku plików.
Zacznijmy od faktu, że dla bezpieczeństwa, stwórzmy w naszym pliku możliwość jego zamknięcia i sprawmy, by po uruchomieniu z jakimś parametrem, wszedł w pętlę, by nic nie robił – na wypadek uruchomienia go przez system.
Użyjemy kolejno dwóch funkcji.

```
DWORD LoadModule(LPCSTR nazwa, LPVOID parametry)
HMODULE GetModuleHandle(LPCTSTR nazwa)
```

Używając pierwszej funkcji, załadujemy moduł w systemie. Druga funkcja zwróci jego instancję.
Funkcji GetModuleHandle już używaliśmy, by uzyskać instancję naszego programu.
Jako parametr "nazwa", wpisz nazwę i (opcjonalnie) ścieżkę do pliku. Parametr "parametry" służy do przekazania parametrów do uruchamianego modułu.
Oto i przykład:

```
Win32API.new("kernel32","LoadModule",'pp','I').call("test.exe","/n"
)
$uchwyt =
Win32API.new("kernel32","GetModuleHandle",'p','I').call("test.exe")
```

Zasoby w bibliotekach *.dll

Zaczynamy już ostatni rozdział przed podsumowaniem. Nie znaczy to jednak, że o zasobach pisać nie będziemy, pewnie jeszcze coś się o nich pojawi później.
Na początku tego działu pisałem, że zasoby możemy umieszczać również w bibliotekach linkowanych dynamicznie (DLL). Teraz pokażę, jak wydobyć zasoby z takiej biblioteki.
Chodzi o to, co zwykle – o zdobycie instancji.
Można użyć tu funkcji LoadModule i GetModuleHandle, jest to jednak niezalecane – nie wiem, dlaczego.
Zamiast tego możemy użyć funkcji: LoadLibrary (kernel32) oraz FreeLibrary (kernel32).
Już używałeś tych funkcji, zapewne nie wiedząc o tym.
Każde wywołanie Win32API.new(*Arg) powoduje:
1. Załadowanie biblioteki (LoadLibrary)
2. Załadowanie procedury (GetProcAddress)
3. Wykonanie procedury
4. Zwolnienie biblioteki (FreeLibrary)
Nawiasem mówiąc, widać, że o ile trudniej jest obsługiwać biblioteki w C++.
Teraz składnie funkcji.

```
HMODULE LoadLibrary(LPCTSTR nazwa)
```

```
BOOL FreeLibrary(HMODULE moduł)
```

Obie funkcje znajdują się w bibliotece kernel32.
Dodam, że istnieje funkcja LoadLibraryEx. Ma ona więcej parametrów i jest całkiem ciekawa, ale ma swoje flagi, a w RGSS oznacza to, że trzeba albo zaglądać do poradnika po ich wartości, albo nauczyć się kilkunastu numerków wraz z działaniem.

Podsumowanie

Skończył się jeden z krótszych działów. Tym razem wyjątkowo nic nie zadaję, gdyż takie zadanie byłoby sztuczne i właściwie do niczego się nie przyda.
Przy jakimś innym temacie postaram się ułożyć takie zadanie, żeby i zasoby miały w nim swoje miejsce.

ROZDZIAŁ 5
Zarządzanie procesorem i pamięcią

Czyszczenie, zwalnianie, przesuwanie i kopiowanie pamięci

Zaczynamy jeden z najważniejszych działów tego kursu, gdyż wiedzy tutaj zawartej będziesz używać w prawie każdej bardziej zaawansowanej aplikacji.
Zacznijmy od zastanowienia się, czym jest pamięć RAM.
Pamięć RAM, nazywana również pamięcią operacyjną lub nietrwałą, zawiera aktualnie przetwarzane dane: uchwyty plików, zmienne, otwarte części plików, zarejestrowane moduły… Słowem niemal wszystko, co się dzieje. Pamięć ta jest czyszczona, gdy tylko dojdzie do wyłączenia komputera.
Nas w pamięci RAM chwilowo interesuje to, że są w niej przechowywane wartości zmiennych.
W tym rozdziale pokażę, w jaki sposób zarządzać pamięcią oraz procesorem.

Czyszczenie pamięci

Winapi było pisane dla różnych języków programowania. Stąd pojawiają się w nim makra. Czym są makra? Są to skróty wykonania kilku funkcji, zadeklarowane w samym kodzie źródłowym, na przykład w C++.
Niestety, nigdy nie powstała implementacja winapi dla RGSS, więc wykorzystujemy biblioteki DLL. To fajny sposób, kiedy makr nie potrzebujemy.
W C++ jest takie fajne makro: ZeroMemory, które czyści pamięć w wyznaczonym zakresie.
Niestety, w RGSS funkcji ZeroMemory nie ma.
Pozostaje nam jedynie posiadanie zmiennej, którą chcemy wyczyścić i ją wyzerować, na przykład tak.

```
Def zero(zmienna)
For i in 0..zmienna.size - 1
zmienna[zmienna.size] = 0
End
Return(zmienna)
End
Test = zero(test)
```

Właśnie wyzerowaliśmy zmienną zmienna.

Unieważnianie zmiennej

Kiedy posiadamy jakiś ciekawy bufor, na przykład rozmiaru 1048576B (1MB), na koniec warto go wyzerować, żeby nie zaśmiecać pamięci.
Ustawienie tej zmiennej jako nil nic nie da, gdyż pamięć jest dalej zarezerwowana.
Na szczęście możliwe jest unieważnienie zmiennej.
Wiąże się to jednak z pewnymi problemami.
Unieważniona zmienna dalej istnieje, ale nie należy już do naszej aplikacji. W rezultacie będzie ona przechowywać dane innych programów, więc może zawierać dość dziwne dane. Lepiej jej nie używać po kasacji lub przypisać innemu miejscu. Ponadto możemy już nie mieć dostępu do tego fragmentu pamięci.
GlobalFree (kernel32)

```
HGLOBAL GlobalFree(HGLOBAL pamięć)
```

Oto i przykład zastosowania tej funkcji.

```
Zmienna = "\0" * 1048576
Win32API.new("kernel32","GlobalFree",'p','I').call(zmienna)
```

Przesuwanie pamięci

Fajnie się tak usuwa i unieważnia fragmenty pamięci operacyjnej, ale może coś po prostu przesuniemy. Tak też można przecież coś popsuć ☺.
Przesuwanie pamięci pozwala nam na jednoczesne usunięcie danych z pierwszej zmiennej i przekazanie ich drugiej zmiennej.
Teraz niestety muszę napisać coś, co w MSDN określa się jako SECURITY REMARKS.
Otóż zmienna, do której przenosimy fragment naszej bezcennej pamięci musi być dość duża, by tę pamięć przyjąć. Inaczej nie będzie zbyt ciekawie, a bluescreen to tylko przykład tego, co teoretycznie stać się może.
Przyjmijmy jednak, że chwilowo bluescreenów nie lubimy – wiem, straszne – i spróbujmy zrobić to w miarę bezpiecznie.
Funkcja, którą się pobawimy, zowie się MoveMemory. W bibliotece DLL została jednak zadeklarowana jako RtlMoveMemory i tę nazwę podamy Win32API.

```
VOID MoveMemory(PVOID cel, CONST VOID źródło, SIZE_T rozmiar)
```

Funkcja ta nie przyjmuje zmiennych, a ich adresy.
Aby uzyskać adres zmiennej w adresie, należy dodać znak ":" przed jej nazwą np. :$scene .
":" można też dodawać przed nazwami funkcji czy klas.
Wtedy możemy przesuwać jedne funkcje do drugich.

```
zmienna1 = "To jest jakaś zmienna, której zawartość nie ma
większego sensu, ale liczy się to, że jest dość długa, by pomieścić
zmienną o numerze 2."
zmienna2 = "Tą zmienną będziemy przenosić."
Win32API.new("kernel32","RtlMoveMemory",'ppl','v').call(zmienna1,zmienna2,z
mienna2.size)
#inny sposób
Win32API.new("kernel32","RtlMoveMemory",'lll','v').call(:zmienna1,:zmienna2
,zmienna2.size)
```

Kopiowanie pamięci

Oczywiście jest to zwykle dodatkowa robota, bo taki sam efekt daje:
cel = zrodlo
Jednak kiedyś możesz zechcieć podczepiać się pod inne programy i wtedy ta funkcja się przyda.
Funkcja CopyMemory jest zadeklarowana jako RtlCopyMemory w bibliotece.
Składnia jest taka sama, jak w RtlMoveMemory, a przykład może być taki sam, więc nie będę przepisywał.
Powiem tylko, że funkcja ta zwykle jest marnowaniem czasu, a od zwykłego przypisania niczym się nie różni.

Jeśli jednak mamy odpowiednie uprawnienia, możemy coś zepsuć albo ulepszyć (raczej to pierwsze) coś w systemie.

Wielowątkowość

Zanim wyjaśnię, czym jest wielowątkowość, wytłumaczę jak działa procesor.
Do procesora przechodzą dane jakieś aplikacji – zwykle niekompletne, potem następnej i następnej, aż coś wyślą wszystkie programy. Potem wysyłane są następne dane, oczywiście z olbrzymią prędkością.
W rezultacie możemy właściwie wykonywać tylko jedną rzecz jednocześnie.
Wyobraź sobie teraz, że tworzysz grę sieciową. Będzie się ona składała z modułu potwierdzającego obecność, umożliwiającego poruszanie się, odświeżania ekranu…
Jeśli to wszystko będzie się działo po kolei, pobijemy zapewne rekord najbardziej zlagowanej gry na świecie.
Tutaj przychodzi nam z pomocą wielowątkowość.
Wielowątkowość – co to jest?
Jest to jakby podproces. Każdy proces dzieli się na wątki, które są wykonywane niemal jednocześnie. W efekcie możemy stworzyć wspomnianą grę sieciową.
Pamiętaj jednak, że każdy kolejny wątek jest większym obciążeniem komputera.
Dla testu podaję, że po zarejestrowaniu ok. 20000 wątków na moim komputerze (4GB RAM + 2 procesory * 4 rdzenie po 3,3ghz), komputer zaczął się na poważnie wieszać, więc próg jest duży, ale nie przesadzajmy.

Jak stworzyć nowy wątek?
W winapi powstało kilka fajnych funkcji do tworzenia nowych wątków. Mają tylko jedną wadę: nie działają z RGSS.
Próbowałem znaleźć sposób na wielowątkowość w aplikacjach RGSS przed długi czas. W Internecie pisze się, że to niemożliwe.
Cóż, możliwe musi być, gdyż sam RGSS rejestruje kilkanaście wątków.
Pytałem już nawet na różnych wątkach, jak stworzyć nowe wątki i oto znalazłem rozwiązanie przypadkiem.
Istnieje klasa Thread służąca do rejestracji wątków.
Oto i przykładowy wątek:

```
$wątek = Thread.new do
End
```

OK, stworzyliśmy wątek.
Z jakieś przyczyny nie chce on jednak działać długo, pojawia się na ułamek sekundy.
Przyczyna jest bardzo prosta. Po stworzeniu wątku, zaczyna on pracować. Kiedy jednak skończą się jego instrukcje, nastaje koniec.
Tego jednak nie chcemy.
Napiszmy więc tak:

```
$wątek = Thread.new do
Loop do
```

```
End
End
```

OK, mamy nasz wątek. Teraz możemy dodać do niego instrukcje.

Uwaga! Pamiętaj że dodanie do wątku funkcji Graphics.update spowoduje zawieszenie aplikacji na jedną ramkę. To właściwie nic, ale nie każ mu ciągle odświeżać ekranu.
Ponadto, lepiej w wątkach nie wyświetlać grafiki, gdyż może ona zostać źle zinterpretowana.

Ostatnia uwaga dotycząca wątków jest następująca: RGSS nie jest przystosowany do wielowątkowości samego skryptu. Jeśli wystąpi błąd w skrypcie wątku, gra się zamknie, ale nie pojawi się opis błędy – nic się nie pojawi.
Efekt będzie taki, jak byśmy wywołali komendę exit!

Struktury

Zmienne były całkiem przydatne. Jednak kiedy komputery zaczęły uzyskiwać coraz większe możliwości, okazało się, że potrzeba zwracać coraz więcej danych.
Wtedy pojawiły się struktury.
Struktury to odpowiedniki siatek tekstowych (tablic) w RGSS. Niestety, są one przeznaczone dla C++, więc nie możemy ich odczytać w samym RGSS'sie bez dodatkowej zabawy.
Struktury mogą być odbierane i wysyłane do funkcji. Ich największą zaletą jest to, że nie trzeba wypełniać ich wszystkich pól i nie trzeba znać kolejności tych pól.
W RGSS niestety zaleta staje się wadą, gdyż jedynym sposobem na zabawę ze strukturami jest poznanie kolejności pól i wypełnienie ich wszystkich, choćby NIL'EM.
Jak często struktury są używane w WINAPI? Prawie ciągle.
Pamiętasz może, że wspominałem, iż niemożliwe jest – jak na poziom twojej wiedzy, a właściwie kursu – napisanie na razie procedury okna? Przyczyną jest fakt, że pisząc procedurę okna, musimy używać struktur.
Podobnie jest z oknami dialogowymi, odczytem danych z pamięci, pętlą komunikatów i wieloma innymi ciekawymi rzeczami.
Niestety oznacza to, że musimy się pomęczyć i nauczyć obsługiwać struktury.
W strukturze każdy element ma swój typ: INT, LPSTR, LPCSTR, DWORD, LPCTSTR, CHAR, VOID, PVOID, LPVOID, itp. Musimy znać zatem kolejność elementów w strukturze i ich typy. Jeśli już jesteśmy przy dobrych wieściach, dopiszę, że w strukturach mogą się znajdować kolejne struktury itd. Prawie każda kolejna struktura jest własnego, innego typu.
W wyniku nasz kod staje się nieprzejrzysty, ale nic już chyba na to nie poradzimy.
Wyobraźmy sobie, że w zmiennej $str mamy strukturę z trzema polami: LPSTR, INT, LPSTR. Trzeba ją jakoś rozpakować. Służy do tego funkcja unpack.

```
pole1, pole2, pole3 = $str.unpack('ipi')
```

Jak widać, rozpakowanie struktury sprowadza się do wypisania kolejnych zmiennych, przetrzymujących zawartości pól struktury oraz prośby o jej rozpakowanie. Jako parametry podajemy typy obiektów. Są one takie same, jak podczas używania Win32API.new(*arg).

Umiemy już rozpakowywać struktury. Jak je jednak jeszcze spakować? Tak samo, a właściwie odwrotnie.

```
$str = [pole1,pole2,pole3].pack('ipi')
```

Pamiętaj o nawiasach kwadratowych.
Praktycznego przykładu zastosowania tutaj nie podam, ale pojawi się już w następnym rozdziale.

Informacje o pamięci

Czasem możemy pisać program odpowiadający za pamięć RAM albo po prostu chcemy sprawdzić, czy komputer ma dość dużo pamięci, by podźwignąć naszą grę.
Tutaj z pomocą przychodzi nam funkcja GlobalMemoryStatusEx.

```
BOOL GlobalMemoryStatusEx(LPMEMORYSTATUSEX bufor)
```

Jak zapewne się już domyśliłeś, buforem będzie struktura.
Składnia struktury wygląda następująco.

```
typedef struct _MEMORYSTATUSEX {
  DWORD     dwLength;
  DWORD     dwMemoryLoad;
  DWORDLONG ullTotalPhys;
  DWORDLONG ullAvailPhys;
  DWORDLONG ullTotalPageFile;
  DWORDLONG ullAvailPageFile;
  DWORDLONG ullTotalVirtual;
  DWORDLONG ullAvailVirtual;
  DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

Funkcja typedef w C++ służy do tworzenia typów zmiennych. Dodana jest tutaj informacja STRUCT. Oznacza to, że tworzymy strukturę. Potem następuje nazwa tej struktury. To jednak nie jest kurs C++, powiem więc tylko, że na końcu struktury występuje jeszcze ustalenie wszystkiego – zignorować. Nas interesuje tylko to, co jest pomiędzy klamrami {}, bo to jest lista pól struktury wraz z typami.

W ten sposób pobierzemy strukturę i ją rozpakujemy.

```
ram = "\0" * 16384
Win32API.new("kernel32","GlobalMemoryStatusEx",'p','I').call(ram)
ram.delete!("\0")
dwLength, dwMemoryLoad, ullTotalPhys, ullAvailPhyss,
ullTotalPageFile, ullAvailPageFile, ullTotalVirtual, ullAvailVirtual,
ullAvailExtendedVirtual = ram.unpack("IILLLLLLL")
```

Oczywiście, nazwy zmiennych nie muszą być takie same, jak nazwy pól. Tak łatwiej się po prostu zorientować, gdy się czyta MSDN przeznaczone dla C++ .

Teraz objaśnię wszystkie pola.

Typ	Pole	Znaczenie
DWORD	dwLength	Rozmiar całej struktury – w bajtach
DWORD	dwMemoryLoad	Zajęta pamięć – w procentach
DWORDLONG	ullTotalPhys	Rozmiar pamięci fizycznej
DWORDLONG	ullAvailPhys	Wolna pamięć RAM – w bajtach
DWORDLONG	ullTotalPageFile	Rozmiar pliku stronicowania pamięci
DWORDLONG	ullAvailPageFile	Wolny rozmiar pliku stronicowania pamięci
DWORDLONG	ulTotalVirtual	Rozmiar pamięci wirtualnej procesu
DWORD	ullAvailVirtual	Wolna pamięć wirtualna procesu
DWORDLONG	ullAvailExtendedVirtual	Parametr zastrzeżony dla Microsoftu dla najważniejszych programów. Zawsze jest równy 0 i szczerze radzę go nigdy nie zmieniać.

W ten sposób możemy napisać skrypt sprawdzający, czy na komputerze są dość duże zasoby, by działała nasza gra.

```
ram = "\0" * 16384
Win32API.new("kernel32","GlobalMemoryStatusEx",'p','I').call(ram)
ram.delete!("\0")
dwLength, dwMemoryLoad, ullTotalPhys, ullAvailPhyss, ullTotalPageFile,
ullAvailPageFile, ullTotalVirtual, ullAvailVirtual, ullAvailExtendedVirtual
= ram.unpack("IILLLLLLL")
if ullAvailPhys < 512*1024*1024 #512MiB
if ullTotalPhys < 1024*1024*1024 = 1GiB
print("Twój komputer ma zbyt niskie parametry, by obsłużyć ten program.")
exit!
else
print("Masz zbyt mało wolnej pamięci RAM, by obsłużyć ten program. Zamknij
część uruchomionych aplikacji i spróbuj ponownie.")
exit!
end
end
```

Podsumowanie

Zadanie

Napisz kontroler pamięci RAM działający w oddzielnym wątku.
O, wiem. Wykorzystamy zasoby.
Jeśli wolna pamięć będzie mniejsza niż 128MB, skrypt poinformuje o tym użytkownika.
Pamiętaj za każdym razem zwalniać bufor przez GlobalFree – myślę o strukturze.

Rozwiązanie

```
$hinstance = Win32API.new("kernel32","GetModuleHandle",'p','i').call(nil)
Thread.new do
loop do
ram = "\0" * 16384
Win32API.new("kernel32","GlobalMemoryStatusEx",'p','I').call(ram)
ram.delete!("\0")
dwLength, dwMemoryLoad, ullTotalPhys, ullAvailPhyss, ullTotalPageFile,
ullAvailPageFile, ullTotalVirtual, ullAvailVirtual, ullAvailExtendedVirtual
= ram.unpack("IILLLLLLL")
if ullAvailPhys <= 128*1024*1024
kom = "\0" * 16384
Win32API.new("user32","LoadString",'iipi','i').call($instance,9516,kom,kom.
size)
kom.delete!("\0")
kom.delete!("\0")
print(kom)
Win32API.new("kernel32","GlobalFree",'p','i').call(kom)
end
Win32API.new("kernel32","GlobalFree",'p','i').call(ram)
end
end
```

ROZDZIAŁ 5
O obsłudze różnych urządzeń i okien jeszcze kilka słów

W tym rozdziale omówię bardziej zaawansowane zagadnienia różnego typu.

Jest on swojego rodzaju dopełnieniem wszystkich innych rozdziałów i zamyka ten kurs. Omówię tutaj korzystanie z urządzeń wejścia i zarządzanie oknami na zaawansowanym poziomie.

Pętla komunikatów i procedura okna

Wielokrotnie tworzyliśmy różne kontrolki i inne ciekawe rzeczy, a w rezultacie nie mogliśmy ich używać z powodu braku procedury okna.
Przydałoby się więc sporządzenie takiej funkcji.
Zanim jednak się tym zajmiemy, wyjaśnię, czym jest procedura okna.
W Winapi każde okno ma swoją funkcję. Do tej funkcji są wysyłane wszystkie komunikaty, między innymi: o stworzeniu okna, o kliknięciu, o otwarciu menu, o wybraniu opcji menu, o aktywacji kontrolki, o zmianie czcionki… Niestety, o ile w C++ stworzenie procedury okna to kilka linijek, w RGSS jest trudniej.
Najpierw jednak trzeba wyjaśnić jeszcze jedno pojęcie – czym jest ta pętla komunikatów?
Pętla komunikatów jest pętlą – wielkie spostrzeżenie - która wysyła komunikaty do procedury okna. W C++ w pętli komunikatów chodzi o przetłumaczenie wiadomości tak, by program ją zrozumiał, a następnie wysłanie jej do procedury danego okna. W efekcie jedna pętla komunikatów może obsługiwać nawet tysiąc okien.
W RGSS pętla komunikatów będzie miała inne zastosowanie, zaraz wyjaśnię, jakie.
Najpierw zastanówmy się, jak napisać procedurę okna dla okna naszej gry.

Istnieją trzy metody.
1. Można użyć tak zwanego subclassingu. Używając go, możemy naszą własną procedurę okna przypisać jako procedurę najważniejszą. W rezultacie jednak obecna procedura okna stanie się nieważna, a RPG Maker straci sens. Oczywiście, można potem odnaleźć w pamięci adres poprzedniej procedury, a następnie wysyłać do niej dodatkowo wszystkie komunikaty. To jednak dość skomplikowane i niepotrzebne.
2. Można wykonać superclassing i stworzyć procedurę nadrzędną do procedury okna. Efekt jednak będzie taki sam.
3. Można napisać drugą procedurę okna o tej samej wadze, co procedura aktualnie istniejąca. W rezultacie każdy komunikat będzie wysyłany do dwóch procedur: domyślnej procedury RGSS PLAYERA i do naszej własnej procedury.
Takie właśnie rozwiązanie jest najprostsze i najszybsze.
Teraz możemy przystąpić do napisania pętli komunikatów.
Nasza pętla będzie przejmować wiadomości, a następnie kierować je do funkcji, którą nazwiemy WndProc. Funkcja ta będzie porównywała uzyskane efekty z tymi, których oczekujemy, czyli będzie robiła wszystko ☺ .
Na koniec wykonany zostanie efekt, a program wróci do odczytu pętli komunikatów.
Istnieje wiele funkcji, dzięki którym uzyskamy zawartość wiadomości wysłanej do naszego pięknego okna. Osobiście przyzwyczaiłem się do funkcji GetMessage, która skopiuje do bufora zawartość wiadomości.

```
BOOL GetMessage(LPMSG bufor, HWND okno, UINT min kod, UINT max kod)
```

Jak ja uwielbiam te skróty, po których nikt nie rozumie, o co chodzi ☺ .
Pierwszy wniosek po obejrzeniu parametrów jest zapewne taki, że będziemy mieli do czynienia ze strukturą.

Kolejne wnioski nie są już takie złe.
Po pierwsze, nasza wiadomość przechodzi przez pewien filtr. Wiadomości w winapi mają swoje kody. Jeden typ wiadomości ma jeden kod. Najlepiej jako filtr podać 1, 16384. Oczywiście nie ma wiadomości o kodzie 16384, więc dlatego przyjmiemy wszystkie komunikaty.
Podaję teraz skład struktury MSG.

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG, *PMSG, *LPMSG;
```

Funkcja, w przeciwieństwie do statusu pamięci, przekazuje zatem w miarę przyjazną strukturę.

Typ	Pole	Opis
HWND	hwnd	Uchwyt okna, z którego pochodzi wiadomość
UINT	Message	Kod wiadomości
WPARAM	wParam	Parametr wParam wiadomości (w Win32aPI P)
LPARAM	lParam	Parametr lParam (W Win32API "L")
DWORD	Time	Czas wysłania wiadomości
POINT	pt	Struktura zawierająca pozycję myszy (.unpack("ll"))

Jak widać, struktura ma tylko sześć pól, z czego chwilowo nas interesują trzy: Message, wParam i lParam.
Parametry lParam i wParam zawierają dodatkowe informacje o wiadomości. Na przykład, gdy wciśnięto klawisz, wParam zawiera jego kod, a gdy wciśnięto przycisk (kontrolka), lParam zawiera jej uchwyt, a wParam jej identyfikator – o ile istnieje.
Mimo, że wParam to string, zwykle zawiera liczby, więc będziemy go konwertować do liczb.
Dobrze, czy coś jeszcze winienem napisać? Owszem.
Jeśli nasza pętla komunikatów będzie działała przez cały czas, nastanie koniec sensu naszej gry, gdyż nasz wątek będzie przyznawał 100% wolnej mocy procesora na przetwarzanie kolejnych komunikatów.
Jeśli nasza gra będzie przeznaczona dla szybszych komputerów min. 4 rdzenie 2.4ghz lub coś koło tego, można zaryzykować 0.1 sekundy przerwy pomiędzy przejściami pętli. Inaczej optymalnie będzie wywołać przerwę 0.3s. Oczywiście, można nawet wywoływać pętle co 1s, ale po wciśnięciu przycisku, komunikat z dwie sekundy poczeka.
Napiszmy więc pętlę z opóźnieniem 0.5s, zbytnio to nic nie popsuje.

```
def WndProc(message, wparam, lparam)
#obsługa komunikatu
end
```

Naszą procedurę już mamy, robi ona dokładnie nic. Teraz możemy napisać pętlę komunikatów.
Przy okazji pokażę coś nowego. Do identyfikacji naszego okna używaliśmy funkcji FindWindow. Jakiś czas temu znalazłem jeszcze dwie funkcje. Mają one jedną wadę, o której za chwilę. Obie

znajdują się w bibliotece user32 i nie przyjmują parametrów, a są to: GetForegroundWindow – zwraca uchwyt do okna, które jest otwarte oraz GetActiveWindow – zwraca uchwyt do aktywnego okna lub kontrolki. W naszej grze kontrolek właściwie brak, więc drugiej funkcji też możemy użyć. Ja jednak użyję wszystkich trzech. W ten sposób pokażę, jak z pewnością odnaleźć to właściwe okno.

```
        $wnd = Win32API.new("user32","GetForegroundWindow",'v','I').call
        if $wnd != Win32API.new("user32","GetActiveWindow",'v','I').call
        $wnd = Win32API.new("user32","FindWindow",'pp','I').call("RGSS
        Player",nil)
end
klasa = "\0" * 32
Win32API.new("user32","GetClassName",'ipi','I').call($wnd,klasa,klasa.size)
klasa.delete!("\0")
if klasa != "RGSS Player"
$wnd = Win32API.new("user32","FindWindow",'pp','I').call("RGSS Player",nil)
end
```

Jest to, jak na wiedzę pokazaną w tym kursie, najdokładniejsza możliwa identyfikacja okna. Użyjemy funkcji FindWindow tylko wtedy, gdy otwarte jest inne okno, niż nasza gra.
Możemy jeszcze szukać okna po jego nazwie, ale użytkownik mógł przecież otworzyć naszą grę w kilku kopiach ☺ .
Wróćmy jednak do tematu i napiszmy tą nieszczęsną pętlę komunikatów.

```
Thread.new do
loop do
msg = "\0" * 16384
Win32API.new("user32","GetMessage",'piii','I').call(msg,$wnd,0,16384)
msg.delete!("\0")
hwnd, message, wparam, lparam, time, point = msg.unpack("llpllp")
WndProc(message,wparam,lparam)
sleep(0.5)
end
end
```

Skoro już się tak namęczyliśmy, by napisać tą pętlę, możemy się zastanowić, po co to na razie było, skoro żadnej wiadomości nie obsługujemy?
Może zatem obsłużymy nasze cenne menu?
Przyjmijmy, że w menu jest element o identyfikatorze 9001. Jak go odnaleźć?
Wciśnięcie elementu menu wywołuje komunikat o numerze 273. W przypadku tego komunikatu, parametr wParam jest równy numerowi obiektu w menu. Można więc zrobić coś takiego.

```
        def WndProc(message,wparam,lparam)
        if message == 273 and wparam.to_i == 9001
        p "Wcisnąłeś element menu o identyfikatorze 9001."
        end
end
```

Teraz możemy obsłużyć nasze menu.
Podobnie sprawa wygląda z kontrolkami. Nie będę tu pisał przykładu, powiem tylko, że wciśnięcie kontrolki lub wpisanie litery w polu tekstowym, zmiana obiektu w liście itd. wysyła komunikat 273. lParam jest uchwytem do kontrolki, a wParam jej identyfikatorowi.

Znowu powtarza się 273. Jest to najczęściej używany komunikat. Istnieją jednak również inne, nie mniej ważne, a właściwie może i mniej ☺ .
Wypiszę kilka z nich.

Komunikat	wParam	lParam	Opis
1	Nie używany	Struktura z informacjami o oknie	Stworzenie okna
2	Nie używany	Nie używany	Zniszczenie okna
273	Identyfikator kontrolki lub obiektu	Uchwyt kontrolki lub obiektu	Wywołanie jakieś komendy – wciśnięcie przycisku, edycja pola…
16	Nie używany	Nie używany	Zapowiedź zniszczenia okna, poprzedza komunikat 2
512	Informacje o wciśniętych przyciskach myszy i controlu	Pozycja myszy	Ruch myszą (o tym później)
24	Czy okno jest widoczne, czy ukryte (0/!0)	Dokładne informacje (patrz MSDN)	Informacja o zmianie rozmiaru okna lub zminimalizowaniu
0	Nie używany	Nie używany	Brak wiadomości w kolejce
256	Kod klawisza	Cała masa flag i kodów ☺	Określa, że wciśnięto klawisz
257	Kod klawisza	Cała masa dodatkowych informacji	Informuje, że klawisz został puszczony
274	typ	Pozycja kursora	Komunikat systemowy

Myślę, że ostatnie polecenie: 274 wymaga więcej dyskusji, a dokładniej możliwych typów.
Komunikat jest wysyłany, gdy wykonana zostanie operacja systemowa – na przykład wciśnięcie systemowej kontrolki typu "maksymalizuj".

	Opis
0x60	Zamknięcie okna
0xf180	Wybrano opcję pomocy z menu kontekstowego – w RGSS jej nie ma chyba, że ustawimy inaczej
0xf160	Użytkownik podwójnie kliknął na menu
0xf150	Aktywowano gorący klawisz (skrót) jakiegoś okna
0xf080	Użyto poziomego pasku przewijania
0x1	Wygaszacz ekranu jest bezpieczny – cokolwiek to ma znaczyć
0xf100	Otwarto pasek menu przy użyciu jakiegoś skrótu – na przykład alt + spacja
0xf030	Zmaksymalizowano okno
0xf020	Zminimalizowano okno
0xf170	. W przypadku tego wParam, lParam może zawierać: -1 – włączanie monitora 1 – wygaszanie monitora 2 – monitor jest wyłączony Ta funkcja dotyczy wygaszania ekranu
0xf010	Przesunięcie okna

0xf040	Przesunięcie fokusa do następnego okna
0xf050	Przesunięto fokus do poprzedniego okna
0xf120	Przywrócono standardowe: pozycję i rozmiar okna
0xf140	Uruchomiono wygaszacz ekranu
0xf000	Zmieniono rozmiar okna
0xf130	Aktywowano menu start
0xf070	Przesunięto pasek przewijania w pionie

Jak widać, polecenie przynosi ze sobą wiele informacji.
Ciekawostka jest taka, że istnieją polecenia często przetwarzane w C++, których w RGSS przetwarzać się nie opłaca i odwrotnie, więc to, co w dokumentacji winapi będzie podane jako nieporęczne, w RGSS może się przydać.

Może się zdarzyć, że stworzyliśmy jakąś kontrolkę i nie mamy kontroli nad tym, co w niej się dzieje – na przykład o wciśniętych klawiszach.
Chcielibyśmy zatem w naszej procedurze okna mieć również jej komunikaty.
Dla chcącego nic trudnego, musimy wykonać tak zwany subclassing.
Nie będę teraz tłumaczył wszystkiego z nim związanego.
Powiem tylko, że subclassing polega na przekierowaniu procedury okna w inne miejsce lub zmianie jego klasy.
Nie będę tutaj tłumaczył superclassingu, bo preferuję subclassing. Nie będę też szczegółowo tłumaczył, o co w każdej funkcji chodzi, gdyż to już jest bardzo zaawansowany temat.
Pokażę jednak w jaki sposób przekierować procedurę jakiejś kontrolki do naszej procedury okna.
W C++ jest oczywiście łatwiej.
Po pierwsze, musimy stworzyć jeszcze jedną funkcję, która musi mieć parametry identyczne procedurom okna w C++. Następnie, trzeba przekierować komunikaty do naszej procedury – zadziała tutaj automatycznie pętla komunikatów.
Na koniec, przetworzony komunikat oddajemy z powrotem do starej procedury, by na przykład pole tekstowe dalej mogło przyjmować tekst tak, jak dawniej. Inaczej stałoby się zwykłym oknem ze zdefiniowaną inną klasą i o innym wyglądzie.
Weźmy się zatem do roboty.
Istnieje wiele funkcji, dzięki którym dokonamy subclassingu. Ja się przyzwyczaiłem do SetWindowLong, która pozwala na podmianę wielu aspektów okna. Moje przyzwyczajenie pochodzi stąd, że autor kursu winapi, z którego się uczyłem, polecał właśnie tą funkcję. Dla zainteresowanych powiem, że z ciekawszych funkcji istnieje jeszcze SetWindowSubclass.
Możemy dokonać subklasy jednej kontrolki lub wszystkich kontrolek danego typu.
To drugie nie jest zalecane, bo:
1. Będziemy musieli sprawdzać, czy komunikaty pochodzą z naszej kontrolki.
2. Na końcu trzeba anulować każdą subklasę, bo inaczej:
3. Po zamknięciu naszej aplikacji komunikaty będą wysyłane do nieistniejącej procedury. Tutaj do gry wchodzą takie fajne rzeczy, jak błędy krytyczne, bluescreeny, restarty, otwarcie komputera na atak, utrata kontroli nad systemem itp.
Dlatego grzecznie posłuchamy się Microsoftu i będziemy unikać subklasy całej klasy, nawet rymuję ☺ .

```
LONG SetWindowLong(HWND okno, int index, LONG nowa wartość)
```

Index to parametr, który będziemy zmieniać, bo jak już wspomniałem, zmieniać możemy nie tylko procedury okna.
Krótko opiszę, co można zmieniać, ale w inne opcje nie będę się wgłębiał.

Numer	Opcja
-20	Rozszerzony styl okna (drugi parametr CreateWindowEx)
-6	Instancja aplikacji – właścicielki okna
-12	Identyfikator okna
-16	Styl okna
-21	Dane użytkownika
-4	Procedura okna – to nas interesuje

Istnieją jeszcze trzy indeksy, ale o nich nie będę pisał, bo służą do modyfikacji okien dialogowych, o czym jeszcze nie mówiłem w ogóle.
Przedstawię jeszcze skład funkcji, która nam wyśle na koniec komunikat tam, gdzie trzeba.

```
LRESULT CallWindowProc(WNDPROC procedura, HWND okno, UINT
wiadomość, WPARAM wParam, LPARAM lParam)
```

Teraz możemy przystąpić do napisania naszego skryptu. Uwaga! Nie zmieniaj kolejności ani nazw parametrów funkcji, gdyż takie niestety muszą być, by subklasa poprawnie działała – ach, to Winapi.

Dobrze, przy okazji powtórzymy sobie, w jaki sposób tworzymy okienka w Winapi. Powtórzę tutaj też część lekcji z działu drugiego – kiedy to było.

```
def EditProc(hwnd,Message,wParam,lParam)
Win32API.new("user32","CallWindowProc",'lilpi','I').call($oldeditproc,hwnd,
Message,wParam,lParam)
end
$hwnd = Win32API.new("user32","FindWindow",'pp','I').call("RGSS
Player",nil)
$hinstance = Win32API.new("kernel32","GetModuleHandle",'p','I').call(nil)
$hwnd_nowe =
Win32API.new("user32","CreateWindowEx",'lppliiiiiiiip','I').call(0x200,"RGSS
Player","Tekst",0x10000000,0,0,640,480,0,0,$hinstance,nil)
$edit =
Win32API.new("user32","CreateWindowEx",'lppliiiiiiiip','I').call(0x200,"EDIT
",nil,0x10000000|0x40000000,0,0,640,480,$hwnd_nowe,0,$hinstance,nil)
$oldeditproc =
Win32API.new("user32","SetWindowLong",'iil','l').call($edit,-4,:EditProc)
```

Jak widać, tworzenie subklasy nie jest zbyt skomplikowane.
W następnym rozdziale możemy napisać własną klasę dla naszego okna starego lub nowego.
Tymczasem, ten rozdział możemy uznać za skończony.

Piszemy klasę okna

Tworzenie nowej klasy okna sprowadza się do wypełnienia struktury i wykonania funkcji RegisterClass lub RegisterClassEx – zależy od rodzaju struktury.
My wykorzystamy drugą strukturę: WNDCLASSEX.

```
typedef struct tagWNDCLASSEX {
  UINT      cbSize;
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCTSTR   lpszMenuName;
  LPCTSTR   lpszClassName;
  HICON     hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

Typ	Pole	Znaczenie
UINT	cbSize	Rozmiar struktury w bajtach
UINT	style	Style klasy
WNDPROC	LPFNWndProc	Procedura okna
INT	cbWndExtra	Dodatkowe parametry
INT	cbWndExtra	Dodatkowe parametry
HINSTANCE	hInstance	Instancja aplikacji tworzącej klasę
HICON	hIcon	ikona
HCURSOR	hCursor	kursor
HBRUSH	hbrBackground	Pędzel, którym namalowane zostanie tło okna
LPCSTR	lpMenuName	Uchwyt menu, które automatycznie zostanie przypisane do wszystkich okien danej klasy
LPCTSTR	lpClassName	Nazwa klasy
HICON	hIconSm	mała ikona okna

Do załadowania ikony użyj LoadIcon (user32).

```
HICON LoadIcon(HINSTANCE instancja, LPCTSTR nazwa lub identyfikator
w zasobach)
```

Może się zdarzyć sytuacja, że nie chce ci się rysować ikonki. To nie problem, wystarczy wykorzystać jedną z ikon systemowych. W pierwszym parametrze wpisz 0, a w drugim identyfikator.

Identyfikator	Efekt
32512	Domyślna ikona aplikacji
32516	Gwiazdka

32513	Błąd
32515	Wykrzyknik
32513	Ikona dłoni
32514	Znak zapytania
32518	Ikona tarczy bezpieczeństwa

W ten sam sposób załaduj małą ikonkę.
Kursor załaduj funkcją LoadCursor – te same parametry, co LoadCursor.
Drugi argument ma jednak inne flagi.

Numer	Efekt
32650	Uruchamianie aplikacji (strzałka i klepsydra)
32512	Strzałka
32515	Krzyżyk
32649	Dłoń
32651	Strzałka i znak zapytania
32513	Belka
32648	Pocięte koło
32516	Pionowa strzałka
32514	klepsydra

Nie wypisałem oczywiście wszystkich kursorów, ale te najważniejsze.
Nie będziemy tworzyć nowej klasy, nie chce mi się ☺ . Pokażę natomiast, w jaki sposób zdobyć już istniejącą klasę, na przykład RGSS Player.

```
BOOL GetClassInfoEx(HINSTANCE instancja, LPCTSTR nazwa,
LPWNDCLASSEX struktura)
```

Celowo nie napiszę tego skryptu, byś przećwiczył odczyt i zapis struktur ☺ .

Obsługa klawiatury

Pokazywałem już funkcję WM_KEYDOWN (komunikat okna) oraz WM_KEYUP. Niestety, funkcje te są niepraktyczne w przypadku tworzenia gier, gdyż mamy do obsłużenia setki sytuacji, a nie chcemy biegać z klasy do klasy, przepisując kolejne warunki.
Dlatego polecam funkcję GetAsyncKeyState. Funkcja ta sprawdza stan danego klawisza (user32).

```
SHORT GetAsyncKeyState(int kod)
```

Jeśli klawisz nie jest wciśnięty, funkcja zwraca 0. Jeśli jest wciśnięty, zwraca więcej niż 0, a jeśli jest przytrzymany, mniej niż zero.
No to zabierzmy się do zabawy i przepiszmy najważniejsze kody klawiszy.

Kod	Klawisz
1	Lewy przycisk myszy

2	Prawy przycisk myszy
3	Klawisz anulacji
4	Środkowy przycisk myszy
5	Klawisz myszy x1
6	Klawisz myszy x2
7	niezdefiniowany
8	backslash
9	tabulator
10	Kod zastrzeżony
11	Kod zastrzeżony
12	Clear
13	Enter
14	Niezdefiniowany
15	Niezdefiniowany
16	Shift
17	Control
18	Windows
19	Pauza
20	Capslock
21	Kana/Hanguel/Hangul
22	Niezdefiniowany
23	Junja
24	Final
25	Hanja/Kanji
26	Niezdefiniowany
27	Escape
28	Convert
29	Nonconvert
30	Accept
31	Zmiana trybu
32	Spacja
33	Page Up
34	Page Down
35	End
36	Home
37	Strzałka w lewo
38	Strzałka w górę
39	Strzałka w prawo
40	Strzałka w dół
41	Wybierz
42	Drukuj
43	Wykonaj
44	Printscreen
45	Insert
46	Delete
47	Pomoc
48	0
49	1
50	2
51	3
52	4

53	5
54	6
55	7
56	8
57	9
58	Niezdefiniowany
59	Niezdefiniowany
60	Niezdefiniowany
61	Niezdefiniowany
62	Niezdefiniowany
63	Niezdefiniowany
64	Niezdefiniowany
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z

Skoro już się namęczyłem, by napisać tą tabelkę, może jakiś przykład?

```
if Win32API.new("user32","GetAsyncKeyState",'I','I').call(0xD) > 0
print("Wcisnąłeś enter.")
end
```

Prócz sprawdzania, czy przycisk został wciśnięty, możemy oszukać program, że coś naciśnięto, na przykład by otworzyć okno konfiguracji RGSS lub powiększyć ekran. Służy do tego funkcja keybd_event (user32).

```
VOID(BYTE kod, BYTE skan, DWORD flagi, ULONG dodatkowe informacje)
```

Drugim i czwartym parametrem nie będziemy się przejmować. Zostaje zatem pierwszy i trzeci.
Jako pierwszy parametr podaj kod klawisza, który jest rzekomo wciskany.
Parametr trzeci może mieć jedną z następujących trzech wartości:
1 – klawisz jest przytrzymany
0x2 – klawisz jest właśnie puszczony
0 – klawisz jest właśnie wciśnięty.
Tak więc możemy otworzyć okno konfiguracji RGSS Playera:

```
Win32API.new("user32","keybd_event",'llll','I').call(0x70, 0, 0, 0)
#0x70 to f1, 0x79 to f9, 0x7C to f12
Win32API.new("user32","keybd_event",'llll','I').call(0x70, 0, 2, 0)
```

Pełną listę kodów znajdziesz tutaj:
http://msdn.microsoft.com/en-us/library/windows/desktop/dd375731%28v=vs.85%29.aspx

Obsługa myszy i kursora

Przy każdej wysyłanej wiadomości, dodawana jest struktura POINT zawierająca informacje o pozycji myszy.

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

Podobnie jak w przypadku klawiszy, tutaj również ciągłe używanie procedury okna jest niepraktyczne, mimo że w C++ jest świetnym rozwiązaniem.
My jednak piszemy w RGSS, a nie C++ .
Dlatego skorzystamy z mniej popularnego sposobu, a mianowicie, z funkcji GetCursorPos (user32).

```
BOOL GetCursorPos(POINT bufor)
```

Ta funkcja również wypełnia nam strukturę LPPOINT.
Oto i przykładowy kod.

```
if Input.trigger?(Input::C)
buf = "\0" * 8
Win32API.new("user32","GetCursorPos",'p','i').call(buf)
```

```
x, y = buf.unpack("ll")
print "Pozycja myszy, ", x, "|", y
end
```

Prócz odnajdywania kursora, możemy go przesunąć. Służy do tego funkcja SetCursorPos (user32).

```
BOOL SetCursorPos(int x, int y)
```

Myślę, że nie ma więcej tutaj do wyjaśnienia.
Zastanówmy się, co jeszcze można robić z kursorem – prócz przesuwania?
Mhm, chyba można zmienić jego kształt!
Ta strzałka wygląda może i fajnie, ale czasem trzeba by wyćwiczyć u użytkownika cierpliwość i na przykład zmienić ją w klepsydrę.
Najpierw trzeba załadować kursor funkcja LoadCursor, przedstawioną wcześniej, więc nie będę przepisywał składni i wszystkich tabelek ☺ .
Następnie trzeba nasz nowy kursor ustawić.
SetCursor (user32)

```
HCURSOR SetCursor(HCURSOR kursor)
```

Funkcja zwraca uchwyt do poprzedniego kursora, na przykład jeśli zmieniamy strzałkę na klepsydrę, zwrócony zostaje uchwyt do strzałki.
Czasem jednak możemy potrzebować uchwytu do aktualnego kursora, bez jego zmiany. Nic prostszego, no dobrze, są rzeczy prostsze.
GetCursor (user32)

```
HCURSOR GetCursor(VOID)
```

Mhm, istnieje jeszcze jednak bardzo fajna funkcja, która wcale, a wcale się nam nie przyda ☺ . Nazywa się ona ShowCursor i służy do pokazania kursora. W RM'ie na pełnym ekranie staje się on bowiem ukrytym. My go znajdziemy ☺ .
ShowCursor (user32)

```
int ShowCursor(BOOL atrybut)
```

Jeśli jako argument podamy 0, kursor zniknie. Jeśli podamy więcej niż zero, kursor się zjawi – proste.

```
Win32API.new("user32","ShowCursor",'i','I').call(1)
```

Co jeszcze można powiedzieć o kursorach? Można tworzyć własne kursory, tworzyć dla nich konteksty itd. Wszystko wykonujemy tak, (prawie tak), jak w przypadku bitmap.

W przypadku kursorów również możemy używać LoadImage.

Różne kształty okien

Dawno, dawno temu, gdy pisałem jeszcze pierwszy czy drugi dział, natrafiłem na ciekawy artykuł.
Dotyczył on tworzenia – w C++ oczywiście – okien o różnych kształtach.
Właściwie uznałem, że eliptyczne czy wręcz okrągłe okno gry aż tak nie przeszkodzi, więc pokażę, jak coś takiego osiągnąć.
Okno wykorzystuje w Winapi tak zwane regiony. Wyznaczają one pewien kształt, na którym coś jest położone: okno, obraz itp.
Nas interesują okna.
Istnieją trzy przyjazne funkcje, których będziemy używać do tworzenia regionów.
Czwarta wiąże się z wypełnieniem struktury, a ten sam efekt można uzyskać bez niej.
Oto i owe funkcje.
Funkcje do tworzenia regionów znajdują się oczywiście w bibliotece gdi32.dll.
Funkcja do tworzenia zaokrąglonych kwadratów nazywa się CreateRoundRectRgn. Nie chce mi się przepisywać składni, więc ją opiszę. Pierwsze dwa argumenty to współrzędne x i y lewego, górnego rogu. Kolejne dwa argumenty to x i y prawego dolnego rogu okna. Ostatnie dwa argumenty to szerokość i wysokość elips tworzących zaokrąglone rogi. Im większa elipsa, tym bardziej zaokrąglone rogi okna.
Poza tym, istnieje również funkcja tworząca prostokątne regiony. Nazywa się ona CreateRectRgn. Pierwsze dwa parametry określają x i y górnego, lewego rogu, a pozostałe dwa określają x i y dolnego, prawego rogu.
Poza tym możemy tworzyć regiony eliptyczne – CreateEllipticRgn. Dwa pierwsze argumenty to x i y lewego, górnego rogu opisanego prostokąta, a pozostałe dwa argumenty to jego prawy, dolny róg.
Jak widać, tworzenie regionów nie jest trudne.

```
kwadratowy =
Win32API.new("gdi32","CreateRectRgn",'iiii','i').call(0,0,640,480
```

Tak mniej więcej wygląda domyślny region okna RGSS.

```
eliptyczny =
Win32API.new("gdi32","CreateEllipticRgn",'Iiii','i').call(0,0,640,4
80)
```

Tak natomiast możemy zmienić nasze okno w elipsę, nie tracąc żadnego z pikseli. Dzięki temu gra będzie działała na nim poprawnie.
Co jednak z tego, że stworzyliśmy regiony? Przecież okno się nie zmieniło!
Wiem, wiem. Nie przypisaliśmy regionu do okna, ale już to robimy.
SetWindowRgn (user32)

```
int SetWindowRgn(HWND okno, HRGN region, BOOL przerysować?)
```

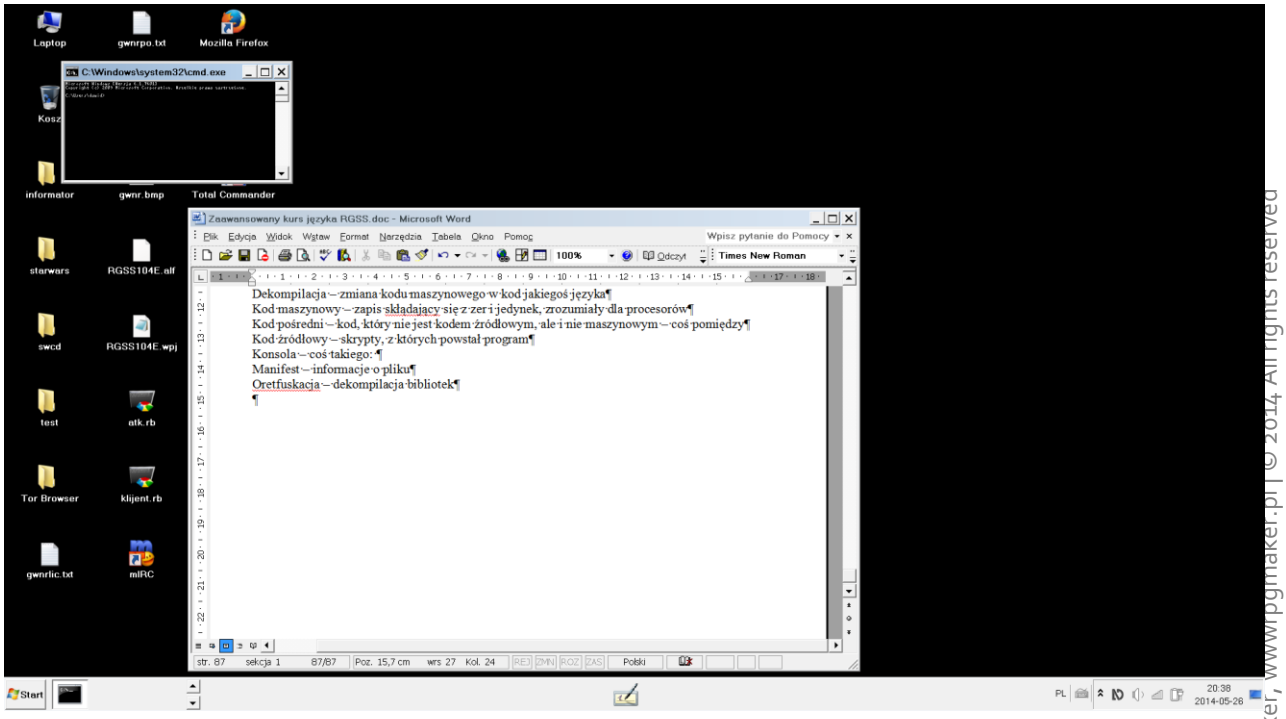
Jeśli nie zmienimy ostatniego parametru na liczbę większą, niż 0, region zostanie zmieniony, ale okno się nie przerysuje, więc nic się nie zmieni.
Ostatnim, co teraz pokażę, jest łączenie regionów.
Jeśli chcemy uzyskać maksymalnie pokręcony region, użyjmy tej funkcji.

```
int CombineRgn(HRGN cel, HRGN źródło, HRGN źródło, int opcje)
```

Pierwszy argument to pusty region, zarezerwowany bufor.
Dwa następne argumenty to po prostu regiony, które łączymy.
Nie mam na MSDN wartości flag, a nie chce mi się grzebać w kodzie źródłowym, tym bardziej, że najczęściej i tak byś pewnie używał flagi 0.
Uwaga! Nie podaję przykładu! Poćwicz sam! ☺

Słowniczek

Myślę, że używane przeze mnie słowa były jasne, ale zapobiegliwie przepiszę najważniejsze pojęcia, również te z "gwary programistycznej".
Algorytm – kolejność czynności niezbędna, by osiągnąć dany cel
Asemblacja – zmiana kodu asemblera (instrukcji procesora) na plik .exe
Asembler – kod instrukcji procesora, zapisany tako tekst, nie binarnie
Biblioteka – plik zawierający różne funkcje
Dezasemblacja – zmiana kodu maszynowego w skrypt asemblera
Dekompilacja – zmiana kodu maszynowego w kod jakiegoś języka
Kod maszynowy – zapis składający się z zer i jedynek, zrozumiały dla procesorów
Kod pośredni – kod, który nie jest kodem źródłowym, ale i nie maszynowym – coś pomiędzy
Kod źródłowy – skrypty, z których powstał program
Konsola – coś takiego:



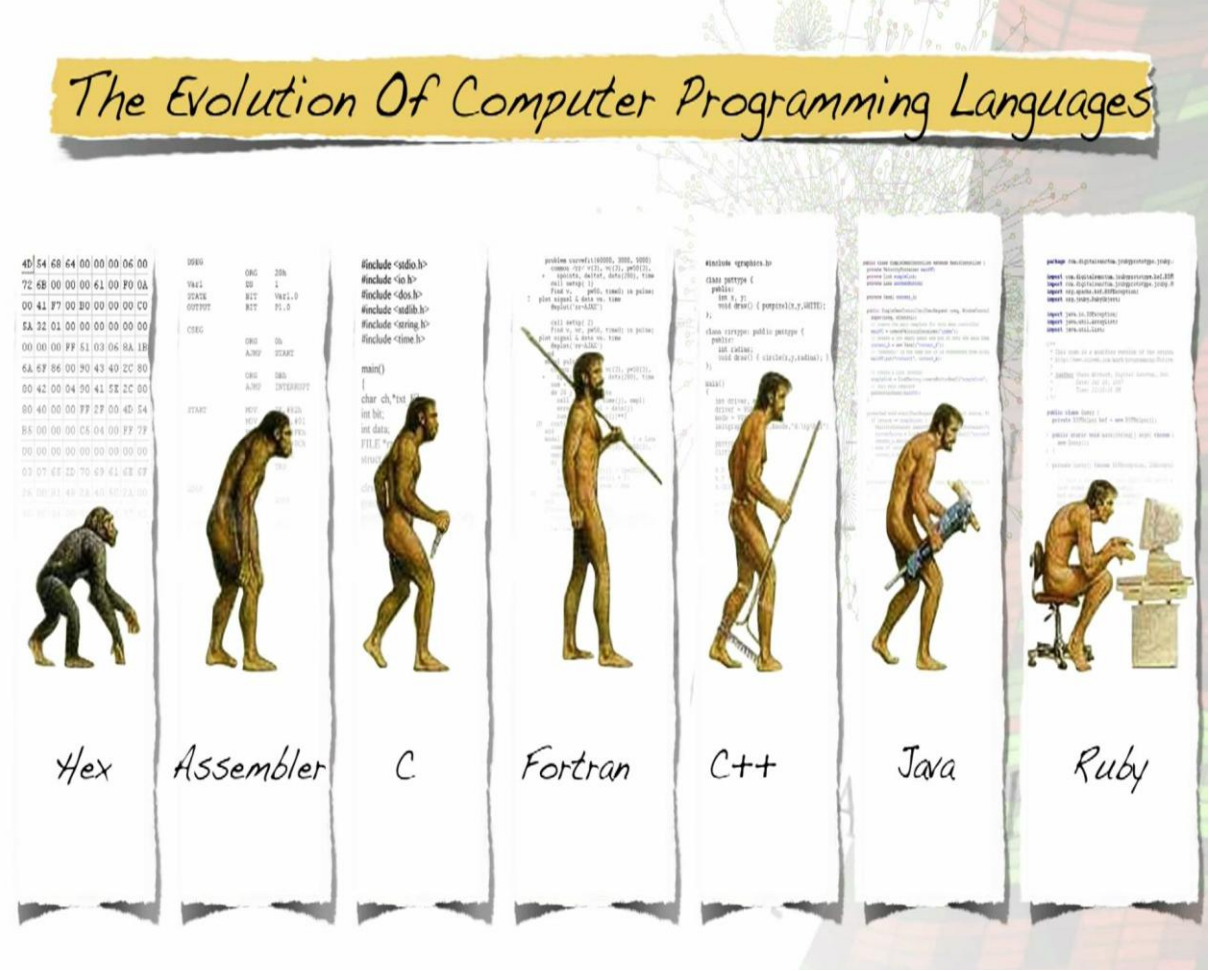
Manifest – informacje o pliku
Oretfuskacja – dekompilacja bibliotek
Plik binarny – plik niemożliwy do odczytania, nawet jako skrypt, jako tekst
Reszta pojęć została wyjaśniona już po tym, jak się pojawiły.

Podsumowanie

W kursie tym omówiono wiele tematów, zaczynając od podstaw biblioteki winapi, przez okna, obrazy, zasoby, pamięć operacyjną i procesor aż do zarządzania klawiaturą i myszą, wielowątkowości i podobnych tematów.
Tematów do omówienia jest jeszcze bardzo wiele. Przykładowo, ledwie zahaczyliśmy o temat związany z Internetem. Nie powiedzieliśmy w ogóle o bibliotece wininet, która posiada wiele funkcji. W ramach ciekawostki posiada funkcję DeleteUrlCacheEntry. Przyjmuje ona jeden parametr: napis, która to funkcja czyści zapamiętane dane ze strony, pozwalając na jej ponowne pobranie. Nie powiedzieliśmy też o winsocku czy obsłudze innych języków programowania, z assemblerem i C++ w roli głównej.
Po co jednak wymieniać te tematy, których nie było?
Cieszmy się tym, co jest.
Posiadłeś niemałą wiedzę na temat programowania w języku RGSS. Istnieje wiele tematów, o których nawet nie wspominałem, są to jednak naprawdę zaawansowane tematy, a zresztą co zabrania się doszkolić?
Pisząc ten kurs, pokazałem, w jaki sposób można wykorzystywać gotowe funkcje winapi czy też struktury. Dzięki temu możesz nauczyć się jakiejś funkcji w winapi i przetworzyć ją na kod w RGSS.
Nie powinieneś też mieć problemów z prostymi czy średnio-trudnymi strukturami.

RGSS jest jednym z najciekawszych języków programowania z powodu swojej wszechstronności. Jest świetny do tworzenia gier, ale nie tylko. Jeśli ktoś chce, może napisać odtwarzacz filmów, portal Internetowy czy edytor tekstu. Tak, wiem, C++ też jest wszechstronny, ale trzeba w nim pisać wszystkie funkcje. W RGSS mamy moduł Audio czy Graphics, które to w C++ trzeba pisać od podstaw.
Język ten oczywiście nie jest pozbawiony również wad, choćby moja największa zmora, błąd "script is changing". Nie zmienia to jednak faktu, że firma Enterbrain odwaliła "kawał dobrej roboty", tworząc ten język.
Na koniec chciałbym jeszcze przeprosić za to, że niektóre tabelki czy obrazki mogą się psuć. Niestety nie mam możliwości sprawdzenia – właściwie – czy wszystko działa.
Wszystkich zapraszam na stronę:
http://msdn.microsoft.com
gdzie znaleźć można spis funkcji Winapi. Warto też zajrzeć na stronę
http://cpp0x.pl
Jest to strona poświęcona językowi C++. W jej zasobach znajdują się kursy wielu bibliotek, choćby i winapi.
Ponadto radzę ci, skoro tak daleko zawędrowałeś, nauczyć się języka C++, bo nie ma co się oszukiwać, jest to najbardziej liczący się język programowania.

Pozdrawiam
Dawid Pieper



Poradnik RGSS (RPG Maker XP) część II. Zaawansowany kurs języka RGSS.
Autor: Dawid „Pajper” Pieper (dawidpieper@o2.pl)
Korekta, poprawki: Reptile, Kleo (reptile@o2.pl mail_kleo@o2.pl)

www.rpgmaker.pl