Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

Zanim rozpoczniemy szóstą odsłonę naszego kursu tworzenia gry, poprawimy drobne błędy z poprzedniego odcinka - przynajmniej tam je znalazłem, więc niezwłocznie naprawiam. Przejdźmy do funkcji initTiles w pliku tile.c. Zawsze zwracamy tam wartość RT_OK, co jest błędem, bo gdy funkcja - mówiąc kolokwialnie - się wysypie, to i tak powróci wyżej z poprawną wartością, co jest niedopuszczalne. Szybciutko należy to zmienić na iResult. W pliku fileIO.c w funkcji WriteFile odnajdziemy błędnie zapisany kod powrotu. W przypadku gdy plik nie zostanie otwarty, funkcja powinna zwracać kod RT_FAILED_WRITE_FILE. O zgrozo - tejże wartości nie odnajdziemy w pliku types.h, gdzie zwyczajowo tam się jej spodziewamy i należy ją dopisać w enum ReturnCodes. Także tam dopisujemy RT_FAILED_TIMER_MSGPORT, RT_FAILED_TIMER_DEVICE, RT_FAILED_TIMER_IOREQ, które przydadzą się w nowej odsłonie modułu timer. Biję się w piersi, bo tam też były co najmniej dziwne wartości zwracane w przypadku niepowodzenia. Oprócz tego w pliku types.h dopiszemy makro, które znakomicie ułatwi nam życie z wielkością tablicy - przestaniemy się martwić jej rozmiarem. Dopisujemy linię.

```
#define ARRAY_SIZE(array)  sizeof (array) / sizeof (array[0])
```

Makro to znajduje zastosowanie wszędzie tam, gdzie potrzebujemy przebiec całą tablicę i znudziło nam się ręczne wklepywanie jej rozmiaru. W pojedynczym przypadku użycia tablicy wydawać by się mogło, że lepiej jest napisać magiczną liczbę, oznaczającą ilość elementów tablicy. To niestety jest bardzo złudne i prędko można się przekonać, że program działa niewłaściwie, gdy tylko przyjdzie nam zmienić ilość elementów tablicy i zapomnimy zaktualizować ową magiczną liczbę. Na podstawie własnego doświadczenia powiem, że z całą pewnością zapomnimy. Poniżej prosty przykładzik użycia makra.

```
int i; UBYTE tab[] = { 0,1,2,3,5,6,7,8,10 };  for (i = 0; i
```

Proste ćwiczenie dla czytelnika, to policzenie rozmiaru tej tablicy - najlepiej dwa razy, aby przekonać się, że lepiej używać makra.

W tym odcinku będziemy dalej ulepszali grę i dodamy krótką demonstrację, przy czym nie mam tu na myśli wypuszczenia gry w wersji demo. To o co chodzi? A to, że często w grach można zauważyć, że jak nie uruchamiamy gry przez jakiś czas, to ona "przechodzi się sama" pokazując nam, jak mamy grać, zazwyczaj pokazując odległe plansze. Wszystko po to, aby nas zachęcić do grania, gdyż po samym tytule czasami ciężko dojść o co w grze chodzi i jakiego typu jest to gra. Problem dodania demonstracji gry można rozwiązać brutalnie, żeby nie powiedzieć banalnie - wystarczy grać na emulatorze i zrobić filmik, po czym załadować go i odtworzyć. Pomijam tu kwestie formatu filmu i potrzebnej pamięci. Ta brutalna i naiwna solucja, w naszym przypadku, nie wchodzi w grę. Śmiesznie by wyglądała gra, która jest plikiem rzędu 20 kB, a do niej dołączony plik z animacją zajmującą lekko ponad 20 MB. W takim razie jakie są inne rozwiązania tego problemu? Mi do głowy przyszło jedno i myślę, że jest dobre. W grze poruszamy się statkiem kosmicznym za pomocą joysticka bądź klawiatury, a ruchy przechowujemy w zmiennych boolowskich. A gdybyśmy tak zasymulowali ruchy, jakie wykonuje gracz, zmieniając wartości zmiennych odpowiedzialnych za ruch naszego bohatera, to wtedy rozwiązalibyśmy nasz problem. Na pewno będziemy potrzebowali tablicę, która te ruchy (lewo, prawo, góra, dół) będzie zawierać. Ponadto każdy taki ruch musi następować w określonym czasie. A zatem będziemy przechowywać w tablicy zarówno ruch, jak i czas, po którym należy wykonać następną czynność. Z pomocą nam przyjdzie taka struktura:

```
typedef struct {     UBYTE dir;     UBYTE secs; } DemoDir;
```

Aby ułatwić sobie zadanie, skorzystaliśmy z dobrodziejstwa typedef i w ten oto sposób wystarczy pisać tylko Demodir, zamiast struct DemoDir. Sprytny czytelnik z pewnością zapyta o sens budowania takiej struktury, skoro oba typy są

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

takie same. Ja zrobiłem to ze względu na czytelność kodu, a jeśli ktoś chce, może użyć zwyczajnej tablicy jednowymiarowej. Pozostawiam to jako ćwiczenie dla chętnych.

Skoro już wiemy, co mamy przechowywać w tablicy, to pozostaje nam sama tablica.

```
#define Right 1 #define Left  2 #define Up    3 #define Down  4 #define Fire  5  static DemoDir DemoDirTable[] = {
{Left, 3},     {Left, 1},     {Down, 1},     {Right, 1},     {Right, 4},     {Down, 1},     {Fire, 1}, };
```

Użyłem #define, aby kod był bardziej czytelny, bo ciężko byłoby stwierdzić, co jest ruchem a co czasem. W tej tablicy "Fire" będzie miało specjalne znaczenie - spowoduje zakończenie demonstracji i powrót do ekranu tytułowego. Powyższą tablicę należy czytać w następujący sposób: ruszamy joystickiem w "Lewo", czekamy 3 sekundy, znowu ruch w "Lewo" i czekamy sekundę i tak dalej. Odliczanie czasu można zrealizować za pomocą timer.device (pliki timer.c i timer.h w naszym projekcie), tyle że będzie to wymagało większych zmian w timer.c i timer.h. Zabieramy się do pracy.

Na początku dwie pomocne konstrukcje - w sumie ułatwiające pisanie. Do tego dorzućmy nową strukturę, trzymającą potrzebne wskaźniki do sprawnego używania timer device.

```
typedef struct timerequest TimeReq; typedef struct IORequest ReqIO; typedef struct {     struct MsgPort* port;
TimeReq* request; } Tim;
```

Przedstawię teraz zmienione dwie główne funkcje, które są lokalne w module timer. Są to: init i kill, które, jak łatwo się domyślić, odpowiadają za inicjalizację i zwolnienie zasobów.

```
static int init(Tim* t) {     t->port = CreateMsgPort();     if (0 == t->port)     {         return RT_FAILED_TIMER_MSGPORT;
}     t->request = (TimeReq*)CreateIORequest(t->port, sizeof(TimeReq));     if (0 == t->request)     {         return
RT_FAILED_TIMER_IOREQ;     }     {     BYTE error = OpenDevice(TIMERNAME, UNIT_VBLANK, (ReqIO*)t->request,
0);     if (0 != error)     {         return RT_FAILED_TIMER_DEVICE;     }     }
t->request->tr_node.io_Message.mn_Node.ln_Type = NT_UNKNOWN;     t->request->tr_node.io_Command =
TR_ADDREQUEST;     return RT_OK; }
```

Najpierw tworzymy port, na którym będziemy nasłuchiwać wiadomości z device'a. Następnie alokujemy strukturę timerequesta, a dalej otwieramy device. Nie będę się rozpisywał, bo temat już jest znany z wcześniejszych odcinków. Nowością jest ustawienie typu noda w wiadomości na nieznany (NT_UNKNOWN). Dzięki takiemu zabiegowi będziemy mogli bez problemu dowiedzieć się w funkcji kill czy request został wysłany i przedsięwziąć odpowiednie kroki typu przerwanie requesta. Pokrótce teraz o funkcji zwalniającej użyte zasoby.

```
static void kill(Tim* t) {     if (t->request)     {         if (NT_UNKNOWN !=
t->request->tr_node.io_Message.mn_Node.ln_Type)         {             AbortIO((ReqIO*)t->request);
WaitIO((ReqIO*)t->request);         }         CloseDevice((ReqIO*)t->request);         DeleteIORequest(t->request);     }     if
(t->port)     {         DeleteMsgPort(t->port);     } }
```

Na początku robimy użytek z ustawienia typu noda na NT_UNKNOWN i jeśli request został wysłany, to typ noda został zmieniony i spokojnie możemy użyć AbortIO i WaitIO. Następnie zamykamy device, zwalniamy strukturę timerequesta i usuwamy port. W związku z tym że moduł timer (timer.c i timer.h) będzie odpowiadał zarówno za odmierzanie czasu w grze co 1/50 sekundy, jak i będzie liczył czas pomiędzy ruchami w demonstracji gry, to będziemy potrzebowali dwóch struktur Tim i czterech funkcji odpowiedzialnych za inicjalizację i zwalnianie. W przypadku pierwszego odmierzania wyglądać to będzie tak:

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

```
Tim mainTimer; int initMainTimer(void) {     int nResult = init(&mainTimer);     g_nMainTimerSig = 1L mp_SigBit;     if (RT_OK == nResult)     {         mainTimer.request->tr_time.tv_secs = 0;         mainTimer.request->tr_time.tv_micro = 20000;         SendIO((ReqIO*)mainTimer.request);     }     return nResult; } void killMainTimer(void) {     kill(&mainTimer); }
```

Mamy tu mainTimer, który będzie trzymał port i timerequest, aby móc odmierzać co 1/50 sekundy. Tutaj podczas inicjalizacji od razu wysyłamy requesta, dzięki czemu główna pętla będzie działać tak, jak poprzednio. Uwolnienie sprowadza się do wywołania wcześniej opisanej funkcji. Oprócz tego mamy funkcję signalsMainTimer, którą będziemy wywoływać w boxo.c.

```
void signalsMainTimer(void) {    while (TRUE)    {      if (0 == GetMsg(mainTimer.port))      {         break;      }   }
 mainTimer.request->tr_time.tv_secs = 0;    mainTimer.request->tr_time.tv_micro = 20000;
 SendIO((ReqIO*)mainTimer.request); }
```

W tym przypadku najpierw odbieramy wszystkie wiadomości z naszego portu i wysyłamy ponownie timerequest. Tak na marginesie wytłumaczę, dlaczego tutaj stosuję pętlę while(TRUE). Uważny czytelnik powie, że przecież mogłem napisać to o wiele prościej pisząc.

```
    while (GetMsg(mainTimer.port))         ; /* nic nie rób w pętli while*/
```

Muszę przyznać, że początkowo tak sam robiłem, ale po wygenerowaniu kodu źródłowego okazało się, że VBCC (z włączonymi optymalizacjami) generuje gorszy kod niż w przypadku, gdy używamy konstrukcję while(TRUE). Gorszy w tym sensie, że jest go więcej i jest powtórzony, co poniekąd wynika ze specyfikacji instrukcji while. Nie oznacza to jednak, aby wszystkie pętle zamieniać na while(TRUE) - trzeba to empirycznie zbadać dla każdego przypadku. Dlatego nie powinno się optymalizować za wcześnie, bo może się okazać, że po poprawkach kodu, poprzednia wersja była o wiele lepsza. To taka mała dygresja, a my wracamy do głównego wątku.

Teraz czas na funkcję init dla odliczania czasu w demonstracji gry. Funkcję kill pomijamy, bo wygląda bardzo podobnie, jak wyżej.

```
int initDemoTimer(void) {     int nResult = init(&demoTimer);     g_nDemoTimerSig = 1L mp_SigBit;     return nResult; }
```

Od razu rzuca się w oczy brak wysłania requesta, ale po chwili zastanowienia wszystko jest jasne, gdyż wysłanie requesta zostanie wywołane, gdy uruchomione zostanie demo gry. Funkcję signalsDemoTimer będziemy wywoływali w demo.c.

```
void signalsDemoTimer(ULONG secs) {    while (TRUE)    {      if (0 == GetMsg(demoTimer.port))      {         break;      }
 }     demoTimer.request->tr_time.tv_secs = secs;     demoTimer.request->tr_time.tv_micro = 0;
SendIO((ReqIO*)demoTimer.request); }
```

Oczywistą konsekwencją zmian w module timer będzie zmiana w pliku boxo.c odpowiadającym za inicjalizację naszej gry. W funkcji init dodajemy initMainTimer() i initDemoTimer() tak, jak jest to czynione dla innych funkcji inicjujących. Bardzo podobnie robimy w close(). Pętla główna, czyli loop też ulegnie zmianie, a kilka słów wyjaśnienia wymaga blok kodu:

```
        if (g_bDemoMode)        {            if (signals & g_nDemoTimerSig)            {                DemoRun();            }        }
```

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

Dopiero gdy zmienna g_bDemoMode jest ustawiona na wartość TRUE, następuje sprawdzenie czy przyszedł sygnał z demoTimera i następuje wywołanie funkcji sterującej demem gry. Sama funkcja DemoRun została zaszyta w nowym module demo (demo.c i demo.h). Zwyczajowo tworzymy dwa nowe pliki demo.c i demo.h, po czym modyfikujemy makefile.mak dodając odpowiednie formy dla zmiennych OBJ i LINKOBJ. Tam też wstawiamy informacje dla kompilatora o sposobie przetwarzania tychże plików.

```
OBJ =   $(OBJDIR)\boxo.o \    ...    $(OBJDIR)\sound.o \    $(OBJDIR)\demo.o  LINKOBJ =   $(OBJDIR)\boxo.o \    ...
  $(OBJDIR)\sound.o \    $(OBJDIR)\demo.o     ...  $(OBJDIR)/demo.o: $(SRCDIR)/demo.c    $(CC) $(CFLAGS)
$(SRCDIR)/demo.c -o $(OBJDIR)/demo.o
```

Plik nagłówkowy zawiera deklarację wyżej wymienionej struktury DemoDir i deklarację funkcji widocznej na zewnątrz - DemoRun(). W demo.c umieszczamy tablicę zawierająca ruch wraz z czasami i ciało funkcji DemoRun.

```
static UBYTE cnt = 0; void DemoRun(void) {     if (cnt >= ARRAY_SIZE(DemoDirTable))     {         cnt = 0;     }
UBYTE dir = DemoDirTable[cnt].dir;     if (dir == Right)     {         g_bRight = TRUE;     }     else if (dir == Left)     {
g_bLeft = TRUE;     }     else if (dir == Down)     {         g_bDown = TRUE;     }     else if (dir == Up)     {         g_bUp =
TRUE;     }     else if (dir == Fire)     {         g_pFnc = &Title;     }     signalsDemoTimer((ULONG)DemoDirTable[cnt].secs);
    cnt++; }
```

Sama funkcja jest mało skomplikowana. Pierwszy if służy do zapętlenia tablicy ruchów, a następne gniazdo "ifów" przekształca wartości z tablicy na ruchy w grze. Ostatnie przekształcenie przycisku spowoduje powrót do ekranu tytułowego.

Do omówienia pozostały nam zmiany w module game. Z nowych rzeczy znajdziemy tam licznik, który jest zerowany w funkcji Title, a uruchomienie dema gry następuje w funkcji titleLoop. Jak można się spodziewać demo odpalamy tylko wtedy, gdy licznik spełnia warunek i nie jesteśmy w trybie demo. pokazujemy na ekranie krótki komunikat, że będzie demo i wywołujemy funkcję odpowiedzialną za demo gry. Zmieniamy także wskaźnik tak, aby gra uruchomiła się w trybie demo.

```
     cnt++;     if (cnt == 125 && g_bDemoMode == FALSE)     {         PrintTxt("demo", 128,32);         cnt = 0;
g_bDemoMode = TRUE;         DemoRun();         g_pFnc = &NewDemoGame;     }
```

Kod odpowiedzialny za uruchomienie dema gry pozwala nam na modyfikacje planszy startowej. Oczywiście wtedy musimy zmienić odpowiednio tablicę ruchów. Można też zwiększyć atrakcyjność dema dodając losowy wybór planszy. Sama funkcja jest bardzo prosta i nie wymaga większego komentarza - najważniejszy jest numer planszy, który został ustawiony na pierwsze etapy.

```
static void NewDemoGame(void) {     CleanLevel();    g_nLives = 1;    g_nLvlNumber = 0;    g_pFnc = &NextLevel; }
```

Warto dodać informację, że gra uruchomiła się w trybie demo. Najlepiej posłużyć się tym, co już mamy, czyli użyć funkcji PrintTxt pokazującej tekst na ekranie. Najważniejsze to użyć jej w takim miejscu, aby napis nie był przysłaniany przez planszę bądź przez nasz statek kosmiczny. Innymi słowy - napis musi być na pierwszym planie. Zatem umieszczamy go na samym końcu funkcji GameLoop.

```
     if (g_bDemoMode)     {         PrintTxt("demo", 128,96);     }
```

Ciekawym urozmaiceniem może być dodanie migającego tekstu "demo". To wiąże się z dodatkową zmienną

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

odpowiadającą za miganie tekstu i trzeba zmienić bądź napisać drugą funkcję, która będzie zmazywać tekst. Minusem tego podejścia będzie to, że musimy umieścić tekst w takim miejscu, gdzie tło pod tekstem nie jest grafiką. Kompleksowym rozwiązaniem będzie zapamiętywanie tła pod tekstem i dlatego proponuję, aby czytelnik w ramach ćwiczenia zaimplementował podejście łatwiejsze. Jako wskazówkę dodam, że trzeba się przyjrzeć bliżej funkcji BltBitMapRastPort.

W tym odcinku to wszystko. Zachęcam do eksperymentów i zadawania pytań na forum PPA.

Artykuł oryginalnie pojawił się w jedenastym numerze Polskiego Pisma Amigowego.

www.ppa.pl