Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

Czwarty odcinek zaczniemy od sprzątania. A jest co robić - w jednym folderze są upchnięte wszystkie pliki, a że jest ich już całkiem sporo, lepiej będzie to wszystko poukładać. W folderze boxo tworzymy trzy katalogi: src, obj i exe. W pierwszym z nich będzie cały kod źródłowy, w drugim obiekty pośrednie utworzone podczas kompilacji, a w ostatnim pliki: wykonywalny, grafika do gry i plansze. Do katalogu src przenosimy pliku z rozszerzeniem .c i .h. W folderze obj umieszczamy te kończące się na .o. W exe znajdą się boxo, tile.pic i levels.bin. W głównym folderze boxo zostaną nam makefile.mak i pliki związane z Visual Studio. Nie wystarczy tylko poprzenosić pliki, trzeba także zmienić makefile - łatwo się o tym przekonać próbując skompilować projekt. Naiwnym rozwiązaniem byłoby powpisywać na sztywno ścieżki do plików w makefile.mak, lecz my ułatwimy sobie pracę i stworzymy dwie nowe zmienne: SRCDIR = src i OBJDIR = obj. Umieścimy je zaraz za definicją zmiennej CFLAGS. OBJ także trzeba zaktualizować. I tu jest dylemat, gdyż w jednej linii trzeba umieścić sporo rzeczy. Na szczęście jest możliwość łamania linii za pomocą znaku \. Należy pamiętam, aby ten znak kończył linie. W każdym razie zmienna ta teraz wygląda tak:

```
OBJ =   $(OBJDIR)\boxo.o \       $(OBJDIR)\window.o \       $(OBJDIR)\timer.o \       $(OBJDIR)\input.o \
 $(OBJDIR)\level.o \       $(OBJDIR)\tile.o \       $(OBJDIR)\game.o \       $(OBJDIR)\fileIO.o
```

Zauważmy, że ostatniego wpisu nie kończymy znakiem \, gdyż właśnie tej linii już nie będziemy łamać. Warto na to zwrócić uwagę, gdy będziemy dodawać nowy wpis, aby przypadkiem nie zapomnieć dodać tego znaku. Aktualizujemy zmienną LINKOBJ na tej samej zasadzie co OBJ. Modyfikujemy także BIN = exe\boxo. Ostatnią rzeczą, jaka musi ulec zmianie, to reguły tworzenia obiektów pośrednich (pliki zakończone .o). Dodajemy na początek nazwa.o $(OBJDIR)/, a wszędzie tam gdzie jest nazwa.c, dajemy na początku (SRCDIR)/. Aby wszystko było jasne zamieszczam trzy przykładowe reguły, które już są zmienione.

```
$(OBJDIR)/boxo.o: $(SRCDIR)/boxo.c    $(CC) $(CFLAGS) $(SRCDIR)/boxo.c -o $(OBJDIR)/boxo.o
$(OBJDIR)/window.o: $(SRCDIR)/window.c    $(CC) $(CFLAGS) $(SRCDIR)/window.c -o $(OBJDIR)/window.o
$(OBJDIR)/timer.o: $(SRCDIR)/timer.c    $(CC) $(CFLAGS) $(SRCDIR)/timer.c -o $(OBJDIR)/timer.o
```

W tym miejscu warto uruchomić nasz projekt i skompilować. W przypadku błędu należy go przeanalizować i ustalić o czym zapomnieliśmy. Dla tych, którzy już nie będą mieli sił, zamieszczam gotowe archiwum (odnośnik w ramce). Sprzątanie mamy za sobą, teraz zajmiemy się przeniesieniem plansz z kodu źródłowego do oddzielnego pliku. Można zapytać, co daje taki manewr poza zmniejszeniem pliku wykonywalnego? Niewątpliwie korzyścią jest to, że nie będziemy musieli po każdej zmianie planszy, kompilować naszego projektu. Przy tak małym projekcie to może mieć małe znaczenie, gdyż czas kompilacji jest krótki i okresy te są porównywalne. Przy dużych projektach nie jest tak różowo. Do wad przeniesienia plansz do pliku można zaliczyć nieoczywisty sposób aktualizacji planszy, nie wspominając o przypadku, gdy chcemy dodać nową planszę. O ile w przypadku drobnej zmiany możemy posiłkować się jakimś hex edytorem, to rozszerzanie bądź zwężanie ilości plansz wymaga większej wiedzy (znajomość formatu planszy). Te problemy rozwiązuje się przez napisanie edytora do gry bądź zmieniając format planszy na plik tekstowy. W każdym razie w obydwu przypadkach czeka nas praca. Wracając do importu plansz na zewnątrz, to przede wszystkim musimy zacząć od napisania procedur operujących na plikach. Spostrzegawczy czytelnik z pewnością zauważył, że podczas porządków z makefile, pojawił się plik pośredni o nazwie fileIO. Właśnie w nim umieścimy procedury, które będą nam pomocne przy rzeczach związanych z plikami. Dodajmy regułę na koniec makefile.mak.

```
$(OBJDIR)/fileIO.o: $(SRCDIR)/fileIO.c    $(CC) $(CFLAGS) $(SRCDIR)/fileIO.c -o $(OBJDIR)/fileIO.o
```

Należy nanieść też odpowiednie zmiany do zmiennych OBJ i LINKOBJ, które to już zrobiliśmy przy porządkach z wyżej wymienionym plikiem. Na sam początek potrzebujemy przynajmniej dwóch procedur: jednej do odczytu, a drugiej do zapisu. Zaczniemy od odczytu. Zastanówmy się przez chwilę, co musimy wiedzieć, aby załadować plik pod jakiś nam znany adres. Po cichu zakładam, że mamy dość miejsca na tenże plik. Na pewno musimy znać nazwę pliku i jego rozmiar. Razem z adresem, gdzie mają być umieszczone dane, wychodzą trzy parametry. Aby funkcja była elastyczna i możliwe było wczytanie nie tylko plansz, a także dowolnego pliku, to umieścimy te trzy rzeczy jako parametry naszej funkcji. Oto i ona:

```
int ReadFile(char* name, UBYTE* buffer, int size) {    FILE* f = fopen(name, "rb");    if (NULL == f)    {       return RT_FAILED_OPEN_FILE;    }    fread(buffer, 1, size, f);    fclose(f);    return RT_OK; }
```

Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

Pierwszym argumentem jest wskaźnik na nazwę, drugi to wskaźnik na miejsce w pamięci, a trzeci to rozmiar. W ciele funkcji użyłem procedur, które są w standardowej bibliotece io. Można w tym miejscu użyć odpowiednich funkcji bibliotecznych z dos.library, ale myślę, że tak jest o wiele łatwiej i przejrzyściej, co nie znaczy, że nie ma błędów. Jeśli przyjrzymy się bliżej, to zauważymy, że sporo rzeczy tutaj może powodować bardzo wiele zła - łącznie z zawieszeniem naszego programu. Warto zdawać sobie z tego sprawę. Nie sprawdzamy czy wskaźniki są poprawne, czyli czy są różne od NULL. Tak naprawdę to nie wiemy czy ktoś zaalokował wystarczającą ilość pamięci pod buffer oraz co się stanie, gdy ktoś poda niewłaściwy rozmiar pliku. Cały ten ciężar przeniosłem na zewnątrz, czyli do innych procedur bądź bloków kodu. To ich zadaniem będzie troska o poprawność argumentów tejże funkcji. W przypadku, gdy otwarcie pliku nie powiedzie się, dostajemy bardzo lakoniczną informację, że nie udało się wczytać pliku, co też może być uznane za wadę, a może też być furtką do dalszej analizy błędu. Jako ćwiczenie proponuję dodać obsługę przypadku, gdy funkcja fread nie przeczyta żądanej ilości informacji. Proszę pamiętać, że plik jest otwarty i wypada go zamknąć. Bardzo podobnie do ReadFile będzie wyglądać funkcja WriteFile, która to zapisuje plik i nie ma większego sensu rozpisywać się na ten temat. Oprócz odczytu i zapisu bardzo przydatną procedurą będzie pobranie rozmiaru pliku. To także można rozwiązać przy pomocy dos.library, ale aby być konsekwentnym używamy biblioteki io, którą to dołączamy za pomocą #include (to tak dla przypomnienia i na marginesie).

```
int GetFileSize(FILE* f) {    int size;    int current = ftell(f);     fseek(f, 0, SEEK_END);    size = ftell(f);    fseek(f, current, SEEK_SET);     return size; }
```

Pobieranie rozmiaru pliku sprowadza się do zręcznego manipulowania funkcjami fseek. Na początku zapamiętujemy obecną pozycję w pliku, potem przechodzimy na koniec pliku i pytamy się za pomocą ftell, jak daleko jest koniec, czyli jaki jest rozmiar i na koniec przywracamy pozycję w pliku taką, jaka była na początku. Dodałem także drugą wersję pobierania pliku, gdzie argument jest nazwą pliku - różni się ona od poprzedniej tym, że otwieramy i zamykamy plik. Zanim zabierzemy się za przenosiny plansz, zmodyfikujemy naszą planszę tak, aby nasz kosmiczny statek mógł się pojawiać w dowolnym miejscu naszej planszy. Bo jak przyjrzymy się funkcji NextLevel w pliku game.c, to nasz pojazd zawsze pojawia się prawie w górnym lewy rogu niezależnie od planszy. Aby to zmienić, trzeba naszego bohatera umieścić na planszy i zaktualizować jego początkową pozycję x i y. Wartością odpowiadającą na planszy naszemu statkowi jest TILE_SHIP i właśnie tę wartość umieszczamy w wybranym przez nas miejscu na planszy. Warto zaznaczyć, że ten element był tylko raz na danej planszy - w przeciwnym razie gracz może być delikatnie zdezorientowany. Umieszczamy statek we wszystkich planszach, by uniknąć kolejnej dziwnej sytuacji, gdy na ekranie nie ma naszego pojazdu (chyba, że kosmiczni krętacze maczali tam palce). Modyfikacji ulegnie funkcja NextLevel i tam umieszczamy w następnej linii, zaraz za procedurą LvlToWin, wywołanie nowej funkcji findShipPositionOnLevel. Sama funkcja wygląda tak:

```
static void findShipPositionOnLevel(void) {    int x;    int y;    UBYTE* p = g_pLevel;     for (y = 0; y
```

Przebiegamy całą planszę w poszukiwaniu elementu TILE_SHIP i gdy go już odnajdziemy, to ustawiamy pozycję początkową statku. W miejsce pojazdu wstawiamy pusty klocek, po czym bezzwłocznie opuszczamy funkcję. Łatwo zauważyć, że nie ma tu żadnego sprawdzania czy rzeczywiście jest statek na planszy, co jest na pewno minusem. Poza tym nie ma też obsługi przypadku, gdy ktoś przez pomyłkę wstawi dwa bądź więcej elementów typu TILE_SHIP.

Mamy już procedury umożliwiające nam podstawowe operacje na plikach, to teraz zabierzmy się na poważnie za przeniesienie plansz z kodu źródłowego do pliku levels.bin. Użyjemy sprytnej sztuczki, aby wygenerować plik z planszami. Dodajmy kod tuż przed instrukcją return RT_OK; do funkcji InitLevel w pliku level.c:

```
WriteFile("levels.bin", g_tabLevels, sizeof(g_tabLevels)/sizeof(g_tabLevels[0]));     return RT_OK
```

Pozostaje nam skompilować i uruchomić nasz projekt, wtedy to zostaną zapisane plansze do pliku levels.bin. Jeśli komuś się nie udało, to nie należy się przejmować - archiwum będzie już miało ten plik. Wykasujmy linię, gdzie zapisywaliśmy nasze plansze, bo swoje zadanie już spełniła. Teraz dodamy kod, który alokuję pamięć na plansze i załadujemy je tam. Wszystko to będzie odbywało się w InitLevel. Usunięcia wymaga tablica g_tabLevels a w jej miejsce deklarujemy zmienną g_pAllLevels, która jest wskaźnikiem na UBYTE. Funkcja InitLevel przedstawia się teraz następująco:

```
int InitLevels(void) {    g_nLvlMaxTWidth = 320/g_nTileWidth;    g_nLvlMaxTHeight = 192/g_nTileHeight;
```

Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

```
g_nLvlTWidth = g_nLvlMaxTWidth;  g_nLvlTHeight = g_nLvlMaxTHeight;  g_pLevel =
(UBYTE*)malloc(g_nLvlMaxTWidth*g_nLvlMaxTHeight);  if (NULL == g_pLevel)  {  return
RT_FAILED_MEM_LEVEL;  }  const int sizeAllLevels = GetFileSizeByName("levels.bin");  g_pAllLevels =
(UBYTE*)malloc(sizeAllLevels);  if (NULL == g_pAllLevels)  {  return RT_FAILED_MEM_ALL_LEVELS;  }
ReadFile("levels.bin", g_pAllLevels, sizeAllLevels);  return RT_OK; }
```

Nas najbardziej interesuje fragment zaczynający się od pobrania rozmiaru pliku levels.bin. Po otrzymaniu rozmiaru planszy, alokujemy pamięć funkcją malloc, sprawdzamy czy udało nam się pozyskać ją i po czym wczytujemy nasze plansze dokładnie w to zaalokowane miejsce. Oczywiście skoro przydzieliliśmy sobie pamięć, to musimy ją także oddać, co też czynimy w KillLevel, w standardowy sposób używając do tego celu funkcji free.

```
void KillLevels(void) {  if (g_pAllLevels)  {  free(g_pAllLevels);  }  if (g_pLevel)  {  free(g_pLevel);  } }
```

W ostatniej części tego odcinka zajmiemy się gameport.device, czyli dodamy obsługę joysticka do naszej gry. Pewne doświadczenie z device już mamy (timer.device), więc nie powinno być większych problemów. Podobnie jak w timer.device tworzymy MsgPort, który będzie potrzebny IO requestowi do komunikacji z devicem. Zobaczmy początkowy blok kodu z funkcji InitInput w pliku input.c.

```
int InitInput(void) {  m_pMsgPort = CreateMsgPort();  if (NULL == m_pMsgPort)  {  return
RT_FAILED_GAMEPORT_MSGPORT;  }  m_pIO = CreateIORequest(m_pMsgPort, sizeof(struct IOStdReq));  if
(NULL == m_pIO)  {  return RT_FAILED_GAMEPORT_IOREQ;  }  m_error = OpenDevice("gameport.device", 1,
(struct IORequest*)m_pIO, 0);  if (0 != m_error)  {  return RT_FAILED_GAMEPORT_DEVICE;  }
g_nGamePortSignal = 1L mp_SigBit;
```

W skrócie - tworzymy MsgPort, tworzymy request IOStdReq, otwieramy device i zapamiętujemy sygnał z MsgPortu. Przy otwieraniu device ustawiamy unit na 1, co oznacza, że będziemy korzystali z drugiego portu - przeważnie tam mamy podłączony joystick. Przy timer.device to było w zasadzie wszystko, jeśli chodzi o inicjalizację, a przy gameport musimy wykonać jeszcze parę rzeczy. Przede wszystkim sprawdzimy czy ktoś przypadkiem nie używa już urządzenia i jeśli tak jest, to wysyłamy informacje, że my będziemy go używać. Ponieważ pracujemy w multitaskingu, to na czas odpytywania i ustawiania zatrzymamy go tak, aby ktoś inny nie podebrał nam joysticka. Druga część kodu z InitInput:

```
Forbid();  const BYTE type = getControllerType();  if (GPCT_NOCONTROLLER == type)  {
setControllerType(GPCT_ABSJOYSTICK);  }  Permit();  setTriggerConditions();  clearBuffer();
sendGamePortRequest();  return RT_OK;
```

Najpierw zatrzymujemy multitasking za pomocą funkcji Forbid z exec.library. Pobieramy typ kontrolera. Jeśli nikt go nie używa, to ustawiamy joystick (GPCT_ABSJOYSTICK) i przywracamy multitasking funkcją Permit. W następnym kroku ustawiamy warunki otrzymywania informacji z urządzenia, dalej czyścimy urządzenie i na końcu wysyłamy request z prośbą o odczytanie stanu joysticka. Przyjrzyjmy się bliżej wywoływanym funkcjom.

Funkcja getControllerType wysyła prośbę (DoIO) za pomocą komendy GPD_ASKCTYPE o typie kontrolera. Dane o nim zostaną zapisane do bajtu result, stąd rozmiar jest ustawiony na 1 (jeden bajt) i io_Data zawiera adres zmiennej result. Sama funkcja DoIO, wykona się w tym przypadku szybko, bo w zasadzie nie korzystamy z urządzenia, a pytamy jak się przedstawia stan rzeczy.

```
static BYTE getControllerType(void) {  BYTE result;  m_pIO->io_Command = GPD_ASKCTYPE;
m_pIO->io_Flags = IOF_QUICK;  m_pIO->io_Data = (APTR)&result;  m_pIO->io_Length = 1;  DoIO((struct
IORequest*)m_pIO);  return result;  }
```

Bardzo podobna do niej jest funkcja setControllerType, która ustawia typ kontrolera na GPCT_ABSJOYSTICK. Oznacza to, że będziemy dostawać pojedynczą informację o stanie joysticka (wychylenie bądź puszczenie w danym kierunku) i przyciskach (naciśnięcie bądź puszczenie). Na przykład, gdy gracz wykona sekwencję ruchów: lewo a potem prawo i wtedy naciśnie przycisk, to urządzenie wyśle nam informacje: wychylony lewo, puszczony lewo, wychylony prawo, przycisk naciśnięty. Funkcja setTriggerConditions odpowiada za wybór informacji/warunków, jakie będziemy otrzymywać. Innymi słowy - w tej procedurze precyzujemy jakie i kiedy dostaniemy wiadomości o stanie joysticka i przycisków. Do tego celu używa się struktury GamePortTrigger. Pole gpt_Keys ustawiamy na GPTF_DOWNKEYS, co oznacza, że interesują nas wiadomości o wciśnięciu przycisku. gpt_XDelta i gpt_YDelta ustawiamy na 1, gdy chcemy

Asman

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

otrzymywać informację lewo/prawo i góra/dół. Można oczywiście ustawić gpt_YDelta na zero lub też wybrać wartość różną od 1, a wtedy nie dostaniemy wiadomości o stanie góra/dół. Do gpt_Timeout możemy wstawić odstęp czasowy. Jednostką jest 1/50 sekundy dla PAL i 1/60 dla NTSC. Jeśli pracujemy w trybie PAL, to ustawienie 25 spowoduje, że co pół sekundy będziemy otrzymywali wiadomości o joysticku i przykładowo, gdy gracz wychylił joystick w lewo przez 4 sekundy, to otrzymamy o tym 8 wiadomości (dla jasności - będzie to 8 wiadomości o tym samym stanie joysticka). My nie potrzebujemy takiej informacji i dlatego to pole jest ustawione na zero. Dalej w InitLevel wywołujemy funkcję clearBuffer, której zadaniem jest wyczyszczenie urządzenie, bo może się zdarzyć, że ktoś przed nami korzystał z joysticka i jakieś informacje pozostały w buforze. A na samym końcu, za pomocą sendGamePortRequest, wysyłamy naszą prośbę o odczytanie urządzenia, a wynik będzie zapisany w zmiennej m_event, która jest strukturą InputEvent.

```
static void setTriggerConditions(void) {  struct GamePortTrigger gpt;  gpt.gpt_Keys = GPTF_DOWNKEYS;
gpt.gpt_Timeout = 0;  gpt.gpt_XDelta = 1;  gpt.gpt_YDelta = 1;  m_pIO->io_Command = GPD_SETTRIGGER;
m_pIO->io_Flags = IOF_QUICK;  m_pIO->io_Data = &gpt;  m_pIO->io_Length = (LONG)sizeof(struct
GamePortTrigger);  DoIO((struct IORequest*)m_pIO); }
```

Inicjalizację gameport mamy za sobą - czas na funkcję przetwarzającą zdarzenia z joysticka. W standardowy sposób, za pomocą sygnału, który jest przechowywany w zmiennej g_nGamePortSignal, będziemy odbierać wiadomości z urządzenia. Oczywiście, aby było to możliwe, dodajemy nasz sygnał do maski sygnałów w funkcji Wait (tak na marginesie - na kilka sygnałów to już czekamy), która jest w metodzie loop w pliku boxo.c. Po odebraniu sygnału z gameport.device wywołujemy funkcję przetwarzającą zdarzenia z joysticka. I w podobny sposób do funkcji signalWindow z pliku window.c, odbieramy komunikaty za pomocą GetMsg. Wtedy to po odebraniu komunikatu, jesteśmy pewni, że m_event zawiera aktualne informacje o stanie joysticka. Pole ie_Code zawiera informację o przyciskach. IECODE_LBUTTON oznacza przycisk w joysticku, a IECODE_RBUTTON to drugi przycisk, który notabene jest traktowany w grach po macoszemu. My robimy z niego użytek i za pomocą niego możliwe jest natychmiastowe opuszczenie gry. Pola ie_X i ie_Y, które tak na marginesie są wygodnym skrótem do ie_position.ie_xy.ie_x i do ie_position.ie_xy.ie_y, zawierają informacje o kierunkach joysticka. W przypadku ie_X mówimy o wychyleniu w lewo bądź w prawo (wartość 1 oznacza prawo a -1 - lewo). Oczywiście 0 oznacza, że drążek nie był wychylony w tychże dwóch kierunkach. Dla ie_Y wartość 1 to wychylenie w dół a -1 to górne wychylenie. Po analizę zdarzenia i ustawieniu odpowiednich zmiennych wysyłamy naszą prośbę o ponowne odczytanie joysticka.

```
BOOL SignalsInput(void) {  BOOL bEnd = FALSE;  while (TRUE)  {  struct IntuiMessage* pMsg = (struct
IntuiMessage*)GetMsg(m_pMsgPort);  if (NULL == pMsg)  {  break;  }  const UWORD button =
m_event.ie_Code;  if (IECODE_LBUTTON == button)  {  g_bFire = TRUE;  }  else if
(IECODE_RBUTTON == button)  {  bEnd = TRUE;  }  const WORD x = m_event.ie_X;  const WORD y =
m_event.ie_Y;  if (1 == x)  {  g_bRight = TRUE;  }  else if (-1 == x)  {  g_bLeft = TRUE;  }  if
(1 == y)  {  g_bDown = TRUE;  }  else if (-1 == y)  {  g_bUp = TRUE;  }  }
sendGamePortRequest();  return bEnd; }
```

Została nam jeszcze jedna funkcja – KillInput. Jak łatwo się domyśleć jej zadaniem jest zakończenie pracy z gameport.device. Najpierw przerywamy poprzednio wystartowaną prośbę, informujemy gameport.device, że urządzenie jest wolne, ustawiając odpowiedni typ GPCT_NOCONTROLLER. A potem to już zamknięcie device i skasowanie naszego IORequesta i MsgPorta.

To wszystko w tym odcinku, zachęcam do eksperymentowania z projektem; zmieniania grafiki, plansz, kodu. W przypadku jakichkolwiek pytań, proszę zadawać je na forum PPA.

Paczka ze źródłami

Artykuł oryginalnie pojawił się w dziewiątym numerze Polskiego Pisma Amigowego.