Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

Siódmy odcinek zaczniemy zwyczajowo od naprawienia błędów z poprzedniego odcinka. Rozszerzając naszą grę o demonstrację gry w szale twórczym zapomniałem o małym detalu, jakim jest wyjście z demo za pomocą wciśnięcia przycisku w joysticku bądź też klawisza A na klawiaturze. Nie jest to skomplikowane i uczynimy to teraz. Jak dobrze pamiętamy, w trybie demo nie zbieramy żadnych informacji czy to z klawiatury, czy z joysticka. Powinniśmy jedynie sprawdzać czy g_bFire został ustawiony, czyli że przycisk w joysticku lub odpowiedni klawisz został wciśnięty. Mimo tego, że kod zostanie później odrobinę zmieniony, to pozwoliłem sobie zamieścić ten kawałątek kodu. Poniższy fragment wstawiamy do window.c w linii 114, czyli zaraz za instrukcją warunkową IF dotyczącą sprawdzania czy nie jesteśmy w trybie demo.

```
else {    if (msg_code == KEY_A)    {      g_bFire = TRUE;    } }
```

To jednak dopiero połowa pracy. Aby tryb demonstracji zakończyć, pozostaje dodać odpowiedni fragment kodu w demo.c, który sprawdza czy przycisk został wciśnięty i jeśli tak było, to powracamy do ekranu tytułowego przekazując wskaźnik na funkcję Title.

```
if (g_bFire) {    g_bFire = FALSE;    g_pFnc = &Title;    return; }
```

Drugi błąd też jest związany z trybem demonstracyjnym. Otóż w module odpowiedzialnym za zbieranie informacji o joysticku zabrakło sprawdzenia czy jesteśmy w trybie demo. Błąd ten powodował, że w trybie demo, użytkownik mógł ruszać joystickiem i mieć mylne wrażenie, że uczestniczy w grze, mimo napisu "demo" na środku ekranu. Pierwsze co, to musimy dodać brakującą dyrektywę include tak, aby kompilator poradził sobie z kompilacją zmiennej g_bDemoMode. Umieszczamy stosowną linię zaraz za pierwszym #include input.h - to proste ćwiczenie pozostawiam dla czytelnika. Musimy jeszcze dodać brakujący warunek w funkcji SignalsInput, który zbiera informacje o wychyleniu joysticka tylko wtedy, gdy gramy. Przedstawiam stosowny kawałek kodu, który trzeba podmienić w tej procedurze.

```
if (FALSE == g_bDemoMode) {    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;    } }
```

Następny błąd, jaki znalazłem to niewłaściwie napisana funkcji GetFileSizeByName, zwracająca kod błędu jako rozmiar. Moim zdaniem, poprawianie tej procedury nie ma większego sensu, bo zbytnio skomplikuje zarówno kod jak i używanie jej, co czyni kod mniej czytelnym. Zalecaną szczepionką na takie przypadki jest usunięcie kodu i przerobienie miejsc, gdzie wołaliśmy funkcję. W tym przypadku została ona użyta w module level.c do załadowania wszystkich plansz do pamięci. Zmianie ulegnie funkcja InitLevels. Należy, mówiąc brzydko, wywalić cały kod od linii, gdzie występuje GetFileSizeByName (wraz z tą linią) aż do klamry kończącej funkcję. Następnie dodajemy deklarację funkcji i zmiennej przetrzymującej nazwę pliku, zawierającego wszystkie plansze. A całość wygląda tak:

```
static int loadAllLevels(void); static const char* m_AllLvlName = "levels.bin";
//========================================================================== int InitLevels(void)
{    int iResult = RT_FAILED_MEM_LEVEL;    g_nLvlMaxTWidth = 320/g_nTileWidth;    g_nLvlMaxTHeight =
192/g_nTileHeight;    g_nLvlTWidth = g_nLvlMaxTWidth;    g_nLvlTHeight = g_nLvlMaxTHeight;    g_pLevel =
(UBYTE*)malloc(g_nLvlMaxTWidth*g_nLvlMaxTHeight);    if (NULL == g_pLevel)    {      return iResult;    }    iResult =
loadAllLevels();    return iResult;
```

Przejdźmy do loadAllLevels. Poza wczytaniem plansz, najważniejsze dla nas jest zaalokowanie potrzebnego obszaru pamięci i prawidłowa obsługa prawie wszystkich przypadków, gdy coś się nie powiedzie. Mam tu na myśli zwracanie

Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

przez funkcję odpowiednich kodów błędów dla odpowiedniej sytuacji. Celowo pominąłem tu sprawdzanie czy funkcja fread wczytała tyle danych ile powinna - to łatwe zadanie pozostawiam czytelnikowi. Z początku może dziwić trochę konstrukcja funkcji, gdzie zmienna iResult przyjmuje różne wartości błędów, aby w wewnętrznej pętli ustawić poprawny kod powrotu. W tym przypadku startujemy od najbardziej pesymistycznego wariantu, aby dojść do pożądanego RT_OK. Czyli początkowo zakładamy, że plik nie zostanie otwarty, potem, skoro jednak się powiodła sztuka otwarcia, to ustawiamy błąd związany z niewłaściwym rozmiarem pliku. Dalej alokujemy obszar pamięci, w którym to za chwilę umieścimy dane dla plansz. I w końcu czytamy dane plansz do pamięci. Oczywiście nie zapominamy dodać brakujących kodów błędów do pliku types.h.

```
 static int loadAllLevels(void) {    int iResult = RT_FAILED_OPEN_LEVELS;    FILE* f = fopen(m_AllLvlName, "rb");    if (f)    {       iResult = RT_FAILED_LEVELS_SIZE;       const int size = GetFileSize(f);       if (size > 0)       {          iResult = RT_FAILED_MEM_ALL_LEVELS;          g_pAllLevels = (UBYTE*)malloc(size);          if (g_pAllLevels)          {             iResult = RT_OK;             fread(g_pAllLevels, 1, size, f);          }       }       fclose(f);    }    return iResult; }
```

W paru innych miejscach także można znaleźć elementy do poprawy, które zmodyfikowałem. Warto wspomnieć o zmianie procedury WaitTicks w module game - dodałem tam możliwość szybszego wyjścia z pętli za pomocą przycisku joysticka lub odpowiedniego klawisza. Dzięki temu bardziej nerwowi użytkownicy mogą szybciej przejść z "Game Over" do obrazka tytułowego, co niewątpliwie jest zaletą.

W dzisiejszym odcinku skupimy się na dalszym usprawnianiu gry. Zaczniemy od zmian w pliku boxo.c, dodamy nowy plik boxo.h i przeniesiemy z modułu game dwie zmienne: g_pFnc i g_bDemoMode. Dodamy tam także nową deklarację zmiennej typu BOOL g_bExitFromGame. Ta nowa zmienna umożliwi nam, jak się łatwo domyśleć, lepszą kontrolę wyjścia z gry. Oczywiście definicję tychże zmiennych umieścimy w boxo.c. Ulepszymy statyczne (czyli widziane tylko w boxo) funkcje init i close. O ile ta druga wygląda jeszcze znośnie, to z pierwszą nie jest już tak różowo. W zasadzie można zauważyć powtarzającą się sekwencję kodu, wywołanie funkcji init dla danego modułu, przypisanie go do zmiennej iResult, testowanie czy wszystko odbyło się tak, jak chcieliśmy (czyli, że zmienna ma wartość RT_OK) i ewentualne zakończenie funkcji w przypadku błędu. Udało nam się wydzielić pewien powtarzalny ciąg instrukcji, a więc możemy z tego wykombinować pętlę. I tak też zrobimy. Zdefiniujmy najpierw wskaźnik na funkcję bez parametrów zwracającą INT. Będzie to podobnie brzmiało do umieszczonej w types.h definicji PVF. Tę definicje także tam umieścimy.

```
 typedef int(*PIF)(void);
```

Definiujemy teraz strukturę o wdzięcznej nazwie InitKill i tworzymy tablicę przechowującą odpowiednie wskaźniki. W ten sposób za jednym zamachem będziemy mieli z głowy zarówno funkcje inicjalizujące, jak i robiące porządek w danym module. Niewątpliwie dodatkową korzyścią, płynącą z takiej struktury będzie zmniejszona ilość błędów w kodzie oraz łatwiejsze i bardziej intuicyjne dodawanie nowych modułów. Wystarczy dodać odpowiednią linię w odpowiednim miejscu i nie musimy pamiętać czy dodaliśmy do funkcji close odpowiedni kod zamykający dany moduł.

```
 struct InitKill {    PIF init;    PVF kill; }; static struct InitKill initTable[] = {    {&initLibs, &closeLibs},    {&InitCfg, &KillCfg},    {&initWindow, &killWindow},    {&initMainTimer, &killMainTimer},    {&initDemoTimer, &killDemoTimer},    {&InitInput, &KillInput},    {&initTiles, &killTiles},    {&InitLevels, &KillLevels},    {&InitSound, &KillSound},    {&InitTitle, &KillTitle}, };
```

Dzięki temu w łatwy sposób zmodyfikujemy init. Całe ciało tej funkcji zastępujemy takim o to zgrabnym kodem.

```
   int i = 0;    while (i
```

Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

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

Jak widać zrobiliśmy użytek z makra ARRAY_SIZE. Najciekawsza rzecz została odrobinę zagmatwana, ale szybka analiza pomoże zrozumieć, "w czym tkwi diabeł". Wywołanie funkcji Init dla danego modułu uzyskujemy przez wyłuskanie odpowiedniego wskaźnika z tablicy zawierającej struktury typu InitKill. Samo initTable[i] wskazuje na i-ty wiersz w naszej tablicy. Aby dobrać się do wskaźnika na funkcję inicjującą dany moduł, stosujemy notację initTable[i].init. Następnym krokiem, zresztą już nam znanym, będzie wywołanie tej funkcji za pomocą (*wskaźnik)(). Bardzo podobnie będzie wyglądać zmieniona funkcja close, która nie wymaga komentarza.

```
static void close(void) {    int i = ARRAY_SIZE(initTable) - 1;    while (i > -1)    {      (*initTable[i].kill)();      i--;    } }
```

Nadszedł czas na dodanie pliku konfiguracyjnego naszej gry. Taki plik konfiguracyjny na samym początku będzie bardzo skromny - umieścimy w nim kody klawiszy odpowiedzialne za poruszanie naszym stateczkiem. Dotychczas w grze ustawione kody klawiszy były na sztywno i na przykład ruchowi w lewo odpowiadał kursor w lewo. Oczywiście wybór tego klawisza nie był przypadkowy, ale myślę, że możliwość niejako konfigurowania gry podnosi jej wartość. Z racji tego, że wcześniej nie myśleliśmy nad tym tematem, to musimy trochę zmodyfikować kod. Plik nazwiemy boxo.cfg, który zostanie zapisany w tym samym katalogu, gdzie znajduje się plik wykonywalny. Na wstępie awansem dodajemy 5 nowych plików: boxo.h, cfg.c, cfg.h, title.c, title.h i modyfikujemy plik makefile tak, aby było możliwe kompilowanie modułów cfg i title. Myślę, że temat dodawania nowych plików do makefile nie nastręczy problemów i dlatego pominę ten krok.

Przejdźmy do cfg.c i cfg.h. Zdefiniujemy trzy zmienne globalne. Pierwsza g_pCfg, będzie wskazywać na konfigurację naszej gry - w sumie powinniśmy zdefiniować strukturę odpowiedzialną za konfigurację. Miałaby ona przechowywać informacje w naszym przypadku o kodach klawiszy. Dla prostoty przyjąłem, że plik boxo.cfg będzie zawierał pięć bajtów i każdy bajt będzie odpowiadał jednemu klawiszowi. Druga zmienna g_nCfgSize przechowywać będzie rozmiar konfiguracji, a trzecia, g_bCfgChanged, to flaga mówiąca czy konfiguracja się zmieniła, czy też nie. Zrobimy z niej użytek przy zapisywaniu boxo.cfg. Inicjacja modułu nie powinna rodzić problemów, gdyż coś bardzo podobnego uczyniliśmy podczas wczytywania plansz w pliku level.c. Nowością jest jedynie kod odpowiedzialny za wywołanie domyślnej konfiguracji, a odbywa się to zawsze, gdy był jakikolwiek problem z odczytaniem pliku boxo.cfg. Wtedy to ustawiamy domyślne wartości dla kodów klawiszy. Jak łatwo się domyśleć, są to kody, które na twardo zapisane były w pliku window.c. Mam tu na myśli między innymi #DEFINE KEY_CURSOR_UP 0x4c. I jeśli ustawialiśmy domyślną konfigurację, to przestawiamy flagę g_bCfgChanged, aby był możliwy zapis, który odbywa się w funkcji KillCfg. Można spróbować zapisywać zaraz po zmianie, ale to podejście zwiększa ilość zapisywania konfiguracji - wystarczy sobie wyobrazić użytkownika, który co chwilę zmienia konfigurację. Przy naszym podejściu konfiguracja zostanie zapisana po wyjściu z gry. Dla jasności przedstawiam tylko funkcje KillCfg i defaultCfg.

```
void KillCfg(void) {    if (g_pCfg && g_bCfgChanged)    {      WriteFile(m_name, g_pCfg, g_nCfgSize);    } }
/*-------------------------------------------------------------------------*/ static void defaultCfg(void) {    m_defaultCfg[0] =
KEY_CURSOR_UP;    m_defaultCfg[1] = KEY_CURSOR_DOWN;    m_defaultCfg[2] = KEY_CURSOR_LEFT;
  m_defaultCfg[3] = KEY_CURSOR_RIGHT;    m_defaultCfg[4] = KEY_A;    g_nCfgSize = DEFAULT_CFG_SIZE;
  g_pCfg = m_defaultCfg;    g_bCfgChanged = TRUE; }
```

Oczywiście nie zapominamy o dodaniu odpowiedniego wiersza w naszej tablicy struktur tak, aby funkcje Init i Kill się uruchomiły oraz aby nasze zmiany przyniosły pożądany skutek i było możliwe poruszanie statkiem kosmicznym za pomocą zdefiniowanych klawiszy (na ten moment ustawiliśmy te same klawisze, co wcześniej). Musimy zrobić jeszcze jedną rzecz - mianowicie przepisać kody klawiszy z konfiguracji do zmiennych odpowiedzialnych za ruch bohatera. W input.c dodajemy funkcję setKeys:

```
static void setKeys(UBYTE* pCfg) {    g_nKeyUp = pCfg[0];    g_nKeyDown = pCfg[1];    g_nKeyLeft = pCfg[2];
```

```
  g_nKeyRight = pCfg[3];    g_nKeyFire = pCfg[4]; }
```

Nastepną rzeczą, jaką się zajmiemy, to animacja elementów w grze, a na deser zrobimy sobie proste menu w obrazku tytułowym. W pierwszym rzędzie zajmiemy się animowaniem kostki odpowiadającej przejściu do następnej planszy w grze. Na tę okazję upichciłem nową grafikę - nie jest ona najwyższych lotów, ale mam nadzieję, że widać na niej, o co chodzi. W naszej grze plansza składa się z kostek i jeśli będziemy podmieniać jeden blok 16x16 na inny, to w ten sposób uzyskamy animację. Zdefiniujmy nowe elementy w tile.h, które będą odpowiadać za poszczególne fazy animacji i, jak nie trudno się domyśleć, TILE_EXIT0, TILE_EXIT1 i TILE_EXIT2 to animowana kostka dla wyjścia. Aby ułatwić sobie deczko sprawę i nie modyfikować plansz, pozostawiłem TILE_SHIP bez zmian. Z nowych elementów dodałem TILE_WALL1, która będzie symulować uderzenie kierowanego przez użytkownika bohaterskiego statku. W level.c pojawiła się nowa funkcja calculateLevel, której zadaniem jest animowanie elementów planszy. W samej funkcji nie ma nic tajemniczego - przechodzimy całą planszę, sprawdzamy na jaki element trafiliśmy i zamieniamy go na inny. Nieocenione usługi tu oddaje instrukcja switch i dlatego też tylko ją przedstawię.

```
    switch (a)   {      case TILE_EMPTY: break;      case TILE_EXIT0:          *p = TILE_EXIT1;
      BltBitMapRastPort(g_pTileBMap, TILE_EXIT1*g_nTileWidth, 0, g_pRpMain, x,y, g_nTileWidth, g_nTileHeight,
0xc0);          break;      case TILE_EXIT1:          *p = TILE_EXIT2;          BltBitMapRastPort(g_pTileBMap,
TILE_EXIT2*g_nTileWidth, 0, g_pRpMain, x,y, g_nTileWidth, g_nTileHeight, 0xc0);          break;      case TILE_EXIT2:
      *p = TILE_EXIT0;          BltBitMapRastPort(g_pTileBMap, TILE_EXIT0*g_nTileWidth, 0, g_pRpMain, x,y,
g_nTileWidth, g_nTileHeight, 0xc0);          break;      case TILE_WALL1:          *p = TILE_WALL;
      BltBitMapRastPort(g_pTileBMap, TILE_WALL*g_nTileWidth, 0, g_pRpMain, x,y, g_nTileWidth, g_nTileHeight,
0xc0);          break;   }
```

Jako bardzo proste ćwiczenie pozostawiam czytelnikowi zamianę długiego BltBitMapRastPort na zgrabną funkcję przyjmującą trzy parametry. Wówczas powyższy kod, będzie wyglądał bardziej estetycznie i przejrzyście. Procedurę calculateLevel wywołujemy w pętli głównej rozgrywki. Aby animacja nie była za szybka, bo mamy mało klatek animacji, wprowadziłem licznik opóźniający. Oprócz tego mniejszym zmianom ulegną warunki sprawdzania czy statek uderzył w ścianę i czy osiągnął wyjście. Po prostu musimy wziąć poprawkę na to, że na planszy będzie więcej różnych klocków i musimy zadbać o prawidłową obsługę. W przeciwnym razie gra nie miałaby sensu, gdyby bohater zamiast zatrzymać się na ścianie przenikałby przez nią i tym sposobem tracił życie. A jeśli już jesteśmy przy temacie statku kosmicznego, to aby go zanimować, musimy zmienić funkcję PasteShip. Niestety nie możemy podpiąć logiki animacji do powyższego kodu, bo jak dobrze pamiętamy (bądź zaglądamy do findShipPositionOnLevel), to po znalezieniu kostki TILE_SHIP, stawiamy tam TILE_EMPTY. W każdym razie animacja jest bardzo podobna do powyższego kodu i nie będę się nad tym rozwodził.

Ponieważ jest nowa grafika, zaszalałem i napisałem kilka nowych procedur. Obrazek tytułowy zyskał pokaźne logo, zmieniłem czcionkę na bardziej czytelną i dodałem stylową strzałkę pokazującą, gdzie obecnie znajduje się wybór użytkownika. Aby to obsłużyć, w module tile pojawiły się procedury odpowiedzialne za: pokazanie logo – Print, postawienie pojedynczego znaku – PrintChar, pokazanie napisu Game Over – PrintGameOver, pokazanie liczby – PrintNumber. Ostatnia będzie wykorzystana do pokazania, na którym etapie jesteśmy. Nadszedł czas na deser, czyli proste menu, a wszystko to umieścimy w module title. Na wstępie przenosimy procedury Title i titleLoop z modułu game. W funkcji Title zmieniamy sposób pokazania logo, wywołując PrintLogo. Tutaj też będzie narysowane menu. Aby nie było zbyt małe, zrobimy trzy pozycje: Start, Options i Exit, a do tego dochodzi strzałka symbolizująca, gdzie aktualnie użytkownik się znajduje. Z racji tego, że są trzy klatki animacji to wprowadziłem licznik opóźniający. Na razie Options pozostanie nieaktywne i zajmę się nim w kolejnym odcinku. Cała logika związana z obsługą menu będzie się mieściła w titleLoop. Rozszerzeniu ulegnie warunek wciśnięcia przycisku – dodano tam sprawdzanie, w której linii znajduje się strzałka i na podstawie tej informacji podejmujemy stosowną akcję. Przy próbie

Asman

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

wychylenia joysticka w dół bądź też wciśnięcia klawisza odpowiedzialnego za ruch w dół, strzałka przemieszcza się w dół, a po dotarciu do granicy następuje powrót do pierwszej pozycji menu. Podobnie jest z ruchem w górze, wtedy to strzałka pojawi się na samym dole.

To wszystko w dzisiejszym odcinku, jak zwykle zachęcam do pytań na łamach forum PPA, eksperymentowania i zgłaszania uwag.

Artykuł oryginalnie pojawił się w dwunastym numerze Polskiego Pisma Amigowego.

www.ppa.pl