Nasza pierwsza gra - kurs programowania AmigaOS i C - część

Asman

W tym odcinku dla uatrakcyjnienia gry zajmiemy się efektami dźwiękowymi. Innymi słowy - będziemy odtwarzać sample, czyli próbki dźwiękowe, które to można znaleźć choćby na Aminecie. Aby ułatwić możliwość podmiany sampli na jakiejś inne, będziemy posiłkować się dźwiękami w formacie 8svx. Oznacza to, że musimy sobie poradzić z obsługą tego formatu. Tutaj z pomocą przyjdzie nam datatype. W jednym z poprzednich odcinków już używaliśmy datatypów do załadowania obrazka z kostkami. Pozwoliło nam to zachować większą elastyczność projektu, gdyż format obrazka jest zupełnie dowolny - istotne jest tylko to, aby datatype umiał sobie z nim poradzić. Nie inaczej jest w przypadku dźwięków - specjalny datatype o znajomo brzmiącej nazwie sound.datatype jest odpowiedzialny za dźwięki. Dzięki niemu bez problemów załadujemy sample do pamięci, a odtwarzać je będziemy za pomocą audio device.

Obsługę dźwięków zaszyjemy w osobnym module, który będzie odpowiadał za wszystko, co jest z tym związane. Na samym początku tworzymy dwa nowe pliki sound.h i sound.c. Dodajemy odpowiednie formuły dla OBJ i LINKOBJ do pliku makefile.mak, wstawiamy też informację dla kompilatora, w jaki sposób ma skompilować nowe pliki.

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

Na wstępie wypada określić liczbę sampli występujących w grze - dla naszych celów wystarczą trzy próbki. Stworzymy typ numeryczny, który będzie pełnił funkcję informacyjną o używanych numerach sampli. W ten sposób stanie się jaśniejsze ich użycie - na przykład PlaySfx(SND_EXIT) zamiast PlaySfx(0).

```
enum SoundNames {   SND_EXIT = 0,   SND_KILL = 1,   SND_ROCK = 2,    AMOUNT_OF_SND };
```

Pierwszy sampel będzie odtwarzany, gdy statek doleci do wyjścia, drugi to zniszczenie pojazdu kosmicznego, czyli wylecenie poza planszę, a trzeci dźwięk będzie odgrywany, gdy statek zderzy się ze skałą. AMOUNT_OF_SND - jak łatwo się domyślić - będzie miał w tym przypadku wartość 3 i posłuży nam jako ograniczenie w pętlach. Dzięki temu zyskujemy większą czytelność źródeł, gdyż pozbędziemy się magicznej liczby 3, która musiałaby się pojawić choćby w pętli odpowiedzialnej za załadowanie sampli do pamięci. Oprócz tego w miarę łatwo jest rozszerzyć ilość sampli w programie - wystarczy dodać SND_COS = 3 przed AMOUNT_OF_SND.

Zastanówmy się teraz, jakie informacje o próbkach będą nam potrzebne, aby je odtwarzać. Na pewno będzie to adres pamięci, gdzie załadowaliśmy dany sampel, a jak już jest w tej pamięci, to musi mieć znany nam rozmiar. Trzeba wiedzieć z jaką prędkością ma być grany przez audio.device i tu pojawia się okres (period) i głośność (volume) sampla. Wszystkie te informacje najlepiej będzie zgrupować w strukturę MyAudio.

```
typedef struct {    Object* dtp;    BYTE* address;    ULONG length;    UWORD period;    UWORD volume; } MyAudio;
MyAudio m_sfx[AMOUNT_OF_SND];  //tablica nazw naszych próbek dźwiękowych, kolejność //i ilość musi być taka
sama jak w enum SoundNames char* m_fileNames[AMOUNT_OF_SND] = {    "exit.8svx",    "kill.8svx",    "knock.8svx",
};
```

W MyAudio, oprócz wymienionych wyżej informacji, pojawił się wskaźnik na Object. Umożliwi nam to sprawniejsze zarządzanie życiem próbek dźwiękowych - mam na myśli zarówno załadowanie próbek, a co za tym idzie zaalokowanie zasobów, jak i późniejsza ich dealokacja. Z dodatkowych rzeczy pojawiła się deklaracja tablicy, która będzie przechowywała stosowną ilość struktur. Dalej widzimy tablicę nazw dla sampli.

Zabieramy się do dzieła. Zwyczajowo tworzymy dwie funkcje: InitSound i KillSound. Pierwsza odpowiada za

Asman

przygotowanie rzeczy związanej z dźwiękami, czyli tutaj zostaną między innymi załadowane próbki dźwiękowe do pamięci, a zadaniem drugiej jest posprzątanie po tej pierwszej, czyli zwolnienie zasobów, które to zajęły sample. Przejdźmy do pierwszej funkcji. Na samym początku zainicjujemy tablicę m_sfx.

```
  //wyczyszczenie pola dtp w tablicy sfx for (i = 0; i
```

Wydawałoby się, że trzeba wyczyścić wszystkie pola struktury, ale wyzerowanie pola dtp jest zupełnie wystarczające, gdyż tylko ta informacja jest potrzebna, aby bez komplikacji zwolnić zasoby. Pozostałe pola będą uzupełnione w dalszej części funkcji InitSound. Najbardziej interesujące jest wczytanie sampli i uzyskanie informacji, które to umieścimy w strukturze MyAudio.

```
  //załadowanie plików dźwiękowych for (i = 0; i
```

Jak widać, zrobiliśmy użytek z AMOUNT_OF_SND (zamiast pisać magiczną liczbę 3). Dalej mamy standardowe załadowanie próbki za pomocą datatypu, czyli podanie nazwy pliku, który musi być zsynchronizowany z naszym typem numerycznym - tak więc SND_EXIT musi odpowiadać pierwszej nazwie w tablicy m_fileNames. Można się pokusić o połączenie tych informacji w strukturę - dla prostoty zrezygnowałem z tego. Następnie podajemy informację, że dane będą pochodzić z pliku, a na końcu określamy typ obiektu, czyli dźwięk. Wygląda to bardzo podobnie jak przy wczytywaniu obrazka. Za pomocą funkcji GetDTAttrs pobieramy interesujące nas właściwości i umieszczamy je w odpowiednim miejscu struktury. Myślę, że jasna jest przyczyna, dla której nie zwalniamy tutaj obiektu dtp po pobraniu atrybutów, gdyż datatype zwolniłby zasoby. Ostatnią rzeczą, jaką zrobimy w InitSound, będzie zainicjowanie audio.device. Robimy to w osobnej funkcji InitSoundPaula.

```
  static int initSoundPaula(void) {    int i;    LONG error;      //tworzymy msg port    m_pMsgAudioPort = CreateMsgPort();
  if (0 == m_pMsgAudioPort)    {      return -1;    }    //umieszczamy go w tworzonym audio requeście    m_pAudioIO =
(struct IOAudio*)      CreateIORequest(m_pMsgAudioPort, sizeof(struct IOAudio));    if (0 == m_pAudioIO)    {      return
-1;    }    //otwieramy audio device i jednocześnie alokujemy kanał dźwiękowy
  m_pAudioIO->ioa_Request.io_Message.mn_ReplyPort = m_pMsgAudioPort;
  m_pAudioIO->ioa_Request.io_Message.mn_Node.ln_Pri = 127;    m_pAudioIO->ioa_Request.io_Command =
ADCMD_ALLOCATE;    m_pAudioIO->ioa_Request.io_Flags  = ADIOF_NOWAIT;    m_pAudioIO->ioa_AllocKey = 0;
  m_pAudioIO->ioa_Data = m_whiChannel;    m_pAudioIO->ioa_Length = sizeof(m_whiChannel);    error =
OpenDevice(AUDIONAME, 0L, (struct IORequest*)m_pAudioIO, 0L);    if (0 != error)    {      return -1;    }    return 0; }
```

Na początku widzimy standardową sekwencję kodu dla otwarcia device, czyli stworzenie msg portu i audio requesta z tymże portem do komunikacji. Omówienia wymaga otworzenie samego audio.device. W tym przypadku pieczemy dwie pieczenie na jednym ogniu - otwieramy i alokujemy kanał audio. Do tego celu wykorzystujemy ADCMD_ALLOCATE w połączeniu z flagą ADIOF_NOWAIT. Sama komenda ADCMD_ALLOCATE służy do rezerwacji kanału. Ustawienie flagi ADIOF_NOWAIT pozwoli nam od razu otrzymać błąd, w przypadku gdy alokacja kanału się nie powiodła. Pole ioa_AllocKey ustawiamy na zero, co oznacza wygenerowanie nowego klucza. Ten klucz jest używany za każdym razem, gdy wysyłamy naszego requesta i jest porównywany z obecnym kluczem kanału audio. Jeśli te wartości są różne, to oznacza, że inny program właśnie używa naszego kanału dźwiękowego. Ponieważ programujemy w zgodzie z systemem operacyjnym, to oczywiście może się zdarzyć, że kilka programów używa tego samego kanału dźwiękowego. Nasuwa się pytanie, kto pierwszy powinien być obsłużony - tu pojawia się priorytet. Ustawiamy go w węźle naszej wiadomości na 127 (m_pAudioIO->ioa_Request.io_Message.mn_Node.ln_Pri). Jest to najwyższa możliwa wartość, dzięki czemu wiemy, że nasz request będzie obsłużony pierwszy. W polu ioa_Data podajemy wskaźnik na tablicę, która zawiera kombinację kanałów do zaalokowania. Kanałom przyporządkowane są następujące numery: 1 - pierwszy lewy, 2 - pierwszy prawy, 4 - drugi lewy, 8 - drugi prawy. W naszym przypadku chcemy zaalokować jeden kanał, a ponieważ

są cztery kanały audio, to w tablicy umieszczamy ich wartości. W ten sposób otrzymamy jeden z czterech kanałów do naszej dyspozycji (oczywiście o ile jakiś jest wolny).

```
static UBYTE m_whiChannel[] = {1,2,4,8};
```

Przykładowo, aby uzyskać pierwszy lewy i pierwszy prawy kanał (czyli jeden kanał stereo), wystarczy ustawić m_whichChannel[] = { 1+2 }. W polu ioa_Length podajemy rozmiar tablicy alokacji. W grze wykorzystamy bardzo proste podejście do efektów dźwiękowych - będziemy odtwarzać sample na jednym i tym samym kanale, nie będzie używać priorytetów sampli, co oznacza, że w zależności o odegranie jakiego sampla poprosimy, to taki zostanie zagrany. Z pewnością budzi to wątpliwości, bo przecież dźwięk zabicia naszego pojazdu kosmicznego powinien być ważniejszy niż odtworzenie dźwięku zetknięcia się ze skałą. Dodanie priorytetu nie powinno nastręczać problemów - wystarczy dodać odpowiednie pole w strukturze i zdefiniować priorytet, który jest aktualnie odtwarzany. Proste porównanie w PlaySfx załatwia sprawę. Pozostawiam to jako ćwiczenie.

Najważniejszą funkcją jest PlaySfx, a jej jedynym argumentem jest numer sampla. Jak widać, nie ma tu sprawdzania czy numer sampla przekracza ilość sampli, bo przy wywołaniu podajemy zawsze właściwy numer. Jak ktoś chce, to pozostawiam to do zrobienia jako bardzo proste ćwiczenie. Na wstępie sprawdzamy czy AudioRequest nie odgrywa jakiejś próbki, a robimy to za pomocą funkcji CheckIO (exec.library). Jeśli jest odtwarzany sampel, to zatrzymujemy go za pomocą ADCMD_FINISH. Użyliśmy tutaj BeginIO, gdyż SendIO albo DoIO czyszczą flagi używane przez audio.device i powinno się zawsze używać BeginIO w połączeniu z audio.device. Aby było możliwe ponowne użycie tego samego audio requesta, musimy wywołać WaitIO i dopiero wtedy możemy wysłać naszego requesta. Co też niżej czynimy, wywołując komendę CMD_WRITE. Podajemy tam też stosowne informację o samplu, który chcemy odtwarzać, czyli adres, rozmiar, okres i głośność. Pole ioa_Cycles ustawiliśmy na 1, co oznacza, że chcemy odtworzyć dźwięk tylko raz.

```
void PlaySfx(int sfxNumber) {    //sprawdzamy czy czasem nie odgrywamy już czegoś    if (0 == CheckIO((struct IORequest*)m_pAudioIO))    {      //tak odgrywamy, więc, prosimy audio.device       //o zakończenie odtwarzania sampla     m_pAudioIO->ioa_Request.io_Command = ADCMD_FINISH;      m_pAudioIO->ioa_Request.io_Flags  = ADIOF_SYNCCYCLE;      BeginIO((struct IORequest*)m_pAudioIO);      WaitIO((struct IORequest*)m_pAudioIO);    }  //odtwarzamy sfx o zadanym numerze    m_pAudioIO->ioa_Request.io_Command = CMD_WRITE;  m_pAudioIO->ioa_Request.io_Flags = ADIOF_PERVOL;    m_pAudioIO->ioa_Data = m_sfx[sfxNumber].address;  m_pAudioIO->ioa_Length = m_sfx[sfxNumber].length;    m_pAudioIO->ioa_Volume = m_sfx[sfxNumber].volume;  m_pAudioIO->ioa_Period = m_sfx[sfxNumber].period;    m_pAudioIO->ioa_Cycles = 1;    BeginIO((struct IORequest*)m_pAudioIO); }
```

Ostatnią funkcja jest KillSound, która, jak wspomniałem wcześniej, czyści po naszej zabawie z dźwiękami. Sprzątanie zaczynamy od zamknięcia audio.device wywołując funkcję killSoundPaula i uwalniając zasoby zaalokowane za pomocą datatype.

```
    for (i = 0; i
```

Samo zamykanie device też nie jest dla nas nowe - oprócz niszczenia requesta i kasowania message port, pojawia się wywołanie CMD_STOP przed zamknięciem audio.device. Dzięki temu wszystkie odtwarzane dźwięki zostaną zatrzymane i spokojnie można zamykać device. Ponieważ przy jego otwieraniu robiliśmy także alokację kanałów dźwiękowych, to zostaną one automatycznie zwolnione.

Skoro już mamy omówiony sposób, w jaki będziemy odtwarzać dźwięki, to teraz pozostają nam dwie sprawy. Pierwsza

Asman

to dodać wywołania InitSound i KillSound w pliku boxo.c i w odpowiednich miejscach dodać wywołania funkcji PlaySfx. Dźwięk zniszczenia pojazdu kosmicznego dodajemy w funkcji GameLoop tuż zaraz za warunkiem sprawdzającym czy statek nie wyleciał poza planszę. Pozostałe sample trzeba dodać w moveShip(): PlaySfx(SND_ROCK) w miejscu gdzie jesteśmy pewni, że nasz bohater dotknął kostki TILE_WALL a PlaySfx(SND_EXIT), gdy dotknął TILE_EXIT. Dodatkowo, aby sampel oznajmujący znalezienie wyjścia z planszy był słyszalny, dodałem prostą pętlę czekającą chwilę tak, aby gracz usłyszał efekt dźwiękowy. Sama pętla wydawać by się mogła odrobinę skomplikowana, ale jeśli przypomnimy sobie, w jaki sposób działa nasza gra i w jaki sposób działają wskaźniki na funkcję, to wszystko staje się proste.

```
 static int m_nTicks = 0; static PVF m_pFncAfter = NULL;  static void waitTicks(void) {     if (m_nTicks > 0)     {
m_nTicks--;     }     else     {         if (m_pFncAfter)         {             g_pFnc = m_pFncAfter;         }     } }
```

Sercem tej funkcji (która tak na marginesie nic nie robi) jest odliczanie do zera i jeśli tak się stało, to zmieniamy wskaźnik na funkcję, która jest wywoływana co określony czas. A przykładowe użycie tej funkcji wygląda tak:

```
        m_nTicks = 50;         m_pFncAfter = &NextLevel;         g_pFnc = &waitTicks;
```

Na zakończenie dwa przykłady użycia sound.datatype. Pierwszy odgrywa sampla, a drugi go zatrzymuje. Aby odegrać próbkę, którą wcześniej załadowaliśmy za pomocą sound.datatype do obiektu sfxObject, ustawiamy adres sampla i wykonujemy metodę STM_PLAY za pomocą DTM_TRIGGER na tymże obiekcie. W zasadzie nie musimy ustawiać adresu sampla, ale przyda nam się to, gdy chcemy go odegrać więcej niż jeden raz. Zatrzymanie odgrywania sampla to nic innego jak ustawienie adresu sampla na zero i wywołanie STM_PLAY.

```
 //odegraj sampla za pomocą sound.datatype SetDTAttrs(sfxObject, NULL, NULL, SDTA_Sample, sfxBuffer,
TAG_END); DoDTMethod(sfxObject, NULL, NULL, DTM_TRIGGER, NULL, STM_PLAY, NULL);  //zatrzymaj sampla
SetDTAttrs(sfxObject, NULL, NULL, SDTA_Sample, 0, TAG_END); DoDTMethod(sfxObject, NULL, NULL,
DTM_TRIGGER, NULL, STM_PLAY, NULL);
```

W tym odcinku to wszystko. Jak zwykle namawiam do eksperymentowania, dodawania własnych sampli i zadawania pytań na forum PPA.

Paczka ze źródłami

Artykuł oryginalnie pojawił się w dziesiątym numerze Polskiego Pisma Amigowego.