

Programowanie gier dla Symbian OS

Budujemy grę!

W poprzednim, pierwszym odcinku z serii „Programowanie gier dla Symbian OS” Czytelnicy mieli okazję przebrnąć przez gąszcz stosunkowo zawiłych szczegółów dotyczących programowania aplikacji pod Symbian OS. W rezultacie powstał prosty szkielet gry. W niniejszej części cyklu zajmiemy się tematem znacznie ciekawszym – odejdziemy (chwilowo) od niskopoziomowych zagadnień systemowych i zrobimy to co Tygrysy lubią najbardziej: zbudujemy grę!

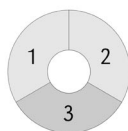
Dowiesz się:

- Jak zaprojektować architekturę prostej gry i jak zaimplementować ją pod system operacyjny Symbian;
- Jak zaprojektować i zaimplementować silnik gry oparty na przyrostach czasu;
- Jak oprogramować menu gry;

Powinieneś wiedzieć:

- Jak się programuje w języku C++;
- Co to jest pętla gry;
- Jak działa szkielet aplikacji opisany w poprzedniej odsłonie cyklu.

Poziom trudności



Jest to bardzo trafne pytanie, które powinna zadać sobie każda osoba planująca stworzyć grę komputerową. Zakładam, że wielu z czytelników SDJ, będących z przyczyn oczywistych programistami, odpowiedziałoby: *nie ma na co czekać, bierzemy się za kodowanie!*. Takie podejście jest właśnie przyczyną upadku wielu przedsięwzięć związanych z produkcją gier. Otóż to od czego zawsze powinno zaczynać się tworzenie gry, to opracowanie jej koncepcji. Ponieważ temat projektowania gier (ang. *game design*), chociaż bardzo ciekawy, nie pokrywa się z tematem niniejszego artykułu – pójdziemy na łatwiznę i zaczerpnijemy pomysł na naszą aplikację z klasyki polskiej myśli kreatywnej w dziedzinie komputerowej rozrywki. Opisywana w niniejszym artykule gra – LaserQuest – bazuje na koncepcji logicznej gry pt. Lasermania, wydanej pierwotnie na ośmiobitowe Atari przez Laboratorium Komputerowe Avalon. LaserQuest imple-

mentuje podstawowy podzbiór funkcjonalności oryginalnej Lasermanii – szczegółowy opis koncepcji gry, do którego lektury gorąco namawiam przed przejściem do kolejnej części artykułu, znajduje się w ramce zatytułowanej *LaserQuest: skrócony podręcznik użytkownika*.

Gotowi...? Zaczynamy!

Punktem wyjścia do naszych rozważań będzie szkielet aplikacji opracowany w poprzedniej odsłonie niniejszego cyklu. Dla porządku przypomnę, że szkielet ten oferuje nam:

- implementację pętli gry;
- obsługę zdarzeń klawiatury;
- podstawową obsługę przerw (utrata i uzyskanie focusa).

Pierwszym zadaniem – być może niespecjalnie twórczym, aczkolwiek niezbędnym – jest dostosowanie istniejącego szkieletu do wymogów naszego projektu. Wbrew pozorom – zmienia się wiele rzeczy: nazwa aplikacji, UI, identyfikator zasobów, autor itd. Oczywiście można pokusić się o ręczne przerabianie kodu, ale podejście takie jawnie gwałci zasadę DRY (ang. *Don't Repeat*

Yourself). Czytelników niezaznajomionych z tą zasadą zapraszam do lektury wspomnianej książki pt. *Pragmatyczny Programista* autorstwa Andrew Hunta i Davida Thomasa. Tym, którzy mimo wszystko zdecydowali się podjąć tę żmudną pracę, gwarantuję wiele niezapomnianych emocji związanych z próbami uruchomienia programu. Tych, którzy wolą bardziej pragmatyczne podejście namawiam na zautomatyzowanie tego procesu. Ja do wygenerowania szkieletu projektu LaserQuest wykorzystałem proste narzędzie o nazwie Template-2-Text (w skrócie t2t). T2t to prosty silnik renderowania tekstu na bazie zadanych szablonów. To co wyróżnia to narzędzie od całego zestawu innych, podobnych rozwiązań, to możliwość generowania całego drzewa katalogów na bazie drzewa szablonowego, możliwość wykorzystania zmiennych szablonowych w nazwach plików oraz automatyczny mechanizm wykrywania tych zmiennych w celu pobrania ich od użytkownika. Aby dostosować nasz szkielet aplikacji do wymagań t2t, skopio瓦łem go do oddzielnego katalogu i zmodyfikowałem nazwy i zawartości plików. Rozważamy plik *GameSkeletonApplication.cpp* (Listing 1).

W nowej odsłonie plik ten będzie miał nazwę `[%PROJECT-NAME%]Application.cpp`. Jak się można łatwo domyśleć, tag o strukturze `' [%NAZWA%]'` oznaczają zmienną, która przy generowaniu tekstu na bazie zadanego szablonu zostanie podmieniona na konkretną wartość. Zawartość szablonu `[%PROJECT-NAME%]Application.cpp` przedstawiona jest na Listingu 2.

Analizując plik szablonu można zauważyć takie zmienne jak: `[%PROJECT-NAME%]`

(nazwa projektu), [%AUTHOR%] (autor), [%EMAIL%] (kontaktowy adres email autora) oraz [%UID%] (unikalny identyfikator aplikacji). W podobny sposób przekształcone zostały pozostałe pliki projektu *GameSkeleton*. Narzędzie *t2t* oraz szablony szkieletu gry jest dostępne do pobrania z witryny SDJ. Po skonfigurowaniu narzędzia (patrz Ramka *Konfiguracja narzędzia t2t*) wystarczy wywołać komendę:

```
t2t -d GameSkeletonS60 LaserQuest
```

W dalszej kolejności narzędzie poprosi o wartości kolejnych zmiennych: UID, AUTHOR, RESOURCE-ID, VENDOR, EMAIL oraz PROJECT-NAME po czym wygeneruje w bieżącym katalogu podkatalog *LaserQuest* z odpowiednią zawartością. Bardziej dociekliwi Czytelnicy mogą w tej chwili zadać pytanie: skąd mam wziąć UID aplikacji? Dla nich właśnie przygotowałem specjalną Ramkę o tytule podobnym do przedstawionego wyżej pytania.

LaserQuest – pierwsze wyjście z mroku

Tytuł niniejszego podpunktu brzmi nieco tajemniczo, czy wręcz złowieszczo. Skąd taka nazwa? Sprawa jest prosta. W punkcie tym opiszemy sposób uruchomienia nowo wygenerowanej aplikacji na telefonie. Dlaczego już teraz? Otóż – na nieszczęście programistów – program działający idealnie pod emulatorem urządzenia, na realnym sprzęcie ma szansę działać zupełnie niepoprawnie. Wynika to z istnienia szeregu subtelnych różnic pomiędzy tymi dwoma środowiskami. Zasada jest prosta – im dłużej będziemy odwlekać uruchomienie aplikacji na telefonie, tym większe jest prawdopodobieństwo wystąpienia takich błędów i tym większe będą koszty ich znalezienia.

Aby uruchomić naszą grę na urządzeniu musimy wykonać następujące kroki: zbudować binarną wersję aplikacji przeznaczoną do uruchomienia na telefonie, wygenerować instalator i podpisać go oraz przetransportować powstałą paczkę instalacyjną na telefon.

Pierwsze dwa kroki można zautomatyzować przy pomocy środowiska Carbide. W tym celu należy przestawić aktualną konfigurację na *Phone Release (GCCE)* a następnie kliknąć prawym przyciskiem na nazwie projektu w panelu *Project Explorer* i wybrać zakładkę *Carbide.c++ -> Carbide Build Configurations -> SIS Builder* (Rysunek 1). Po wciśnięciu przycisku *Add* możemy skonfigurować sposób budowania aplikacji (Rysunek 2). Kluczową rolę odgrywa tutaj plik *LaserQuest.pkg* (można go znaleźć w podkatalogu *sis* projektu) opisujący dokładnie sposób generowania paczki instalacyjnej. Do zawartości tego pliku wrócimy w kolejnych podpunktach. Oprócz wpisania nazw generowanych plików

należy ustawić dodatkowo opcję *Self sign sis file*. Osoby zainteresowane szczegółami podpisywania aplikacji zapraszam do Ramki *Jak podpisać aplikację dla Symbian OS?*. Po takich zabiegach konfiguracyjnych wystarczy wybrać opcję *Build Project* i spokojnie czekać aż w podkatalogu *sis* projektu pojawi się podpisana paczka instalacyjna *LaserQuest.sisx*. Kwestia przetransportowania paczki z grą na telefon jest mocno zależna od posiadanego systemu. W większości przypadków wykorzystuje się w tym celu sieć Bluetooth – należy przy tym pamiętać o włączeniu odbiornika Bluetooth w telefonie i ustawianiu jego widoczności dla innych urządzeń (między innymi dla komputera z którego będzie-

est.sisx. Kwestia przetransportowania paczki z grą na telefon jest mocno zależna od posiadanego systemu. W większości przypadków wykorzystuje się w tym celu sieć Bluetooth – należy przy tym pamiętać o włączeniu odbiornika Bluetooth w telefonie i ustawianiu jego widoczności dla innych urządzeń (między innymi dla komputera z którego będzie-

Listing 1. Zawartość pliku *GameSkeletonApplication.cpp*

```
//
// FILE NAME:
//   GameSkeletonApplication.cpp
//
// AUTHOR(S):
//   Rafal Kocisz <rafal.kocisz@gmail.com>
//

#include "GameSkeletonApplication.h"
#include "GameSkeletonDocument.h"

const TUid KUidGameSkeletonApp = { 0xA000958F };

CApaDocument* CGameSkeletonApplication::CreateDocumentL()
{
    return ( static_cast< CApaDocument* >(
        CGameSkeletonDocument::NewL( *this ) ) );
}

TUid CGameSkeletonApplication::AppDllUid() const
{
    return KUidGameSkeletonApp;
}

// Eof
```

Listing 2. Zawartość szablonu [%PROJECT-NAME%]Application.cpp

```
//
// FILE NAME:
//   [%PROJECT-NAME%]Application.cpp
//
// AUTHOR(S):
//   [%AUTHOR%] <[%EMAIL%]>
//

#include "[%PROJECT-NAME%]Application.h"
#include "[%PROJECT-NAME%]Document.h"

const TUid KUid[%PROJECT-NAME%]App = { [%UID%] };

CApaDocument* C[%PROJECT-NAME%]Application::CreateDocumentL()
{
    return ( static_cast< CApaDocument* >(
        C[%PROJECT-NAME%]Document::NewL( *this ) ) );
}

TUid C[%PROJECT-NAME%]Application::AppDllUid() const
{
    return KUid[%PROJECT-NAME%]App;
}

// Eof
```

Listing 3. Definicja klasy CLaserQuestGame

```

class CLaserQuestGame : public CBase
{
public:
    enum TMode { EMenu, EPlay };

    static CLaserQuestGame* NewL(
        const TLaserQuestKeyState& aKeyState );
    CLaserQuestGame(
        const TLaserQuestKeyState& aKeyState );
    ~CLaserQuestGame();

    void Draw( CBitmapContext& aGc ) const;
    void Update( TInt64 aDt );
    TBool ExitFlag() const { return iExitFlag; }

private:
    void ConstructL();
    void UpdateMenuMode( TInt64 aDt );
    void UpdatePlayMode( TInt64 aDt );

    const TLaserQuestKeyState& iKeyState;
    TMode iMode;
    CLaserQuestGamePlayMode* iPlayMode;
    CLaserQuestGameMenuMode* iMenuMode;
    TBool iExitFlag;

}; // class CLaserQuestGame
    
```

Listing 4. Definicja metody Draw() w klasie CLaserQuestContainer

```

void CLaserQuestContainer::Draw(
    CBitmapContext& aGc ) const
{
    aGc.Reset();
    aGc.SetBrushColor( KRgbBlack );
    aGc.SetBrushStyle( CGraphicsContext::ESolidBrush );
    aGc.DrawRect( Rect() );
    iGame->Draw( aGc );
    TBuf< KFpsMessageMaxLength > fpsMessage;
    fpsMessage.Format( KFpsMessageFormatString,
        iFpsCounter->Fps() );
    aGc.UseFont( iEikonEnv->NormalFont() );
    aGc.SetPenColor( KRgbWhite );
    aGc.DrawText( fpsMessage,
        TPoint( KFpsMessagePosX,
            KFpsMessagePosY ) );
    aGc.DiscardFont();
}
    
```

Listing 5. Definicja metody Update() w klasie CLaserQuestContainer

```

void CLaserQuestContainer::Update( TInt64 aDt )
{
    iGame->Update( aDt );

    if ( iGame->ExitFlag() )
    {
        if ( iAppUi != NULL )
        {
            iAppUi->Exit();
        }
    }
}
    
```

my wysłać paczkę). Paczka instalacyjna pojawia się jako nowa wiadomość w skrzynce odbiorczej. Po jej otwarciu automatycznie rozpoczyna się proces instalacji. Po jego pomyślnym zakończeniu pozostaje jedynie uruchomić aplikację. W naszym przypadku otrzymujemy znajomy z poprzedniej części cyklu czarny ekran z mrużającym przyjaźnie licznikiem FPS. Kolejny etap naszych zmagania zakończył się sukcesem!

Architektura gry

Dobrze wiemy wreszcie do momentu kiedy można zacząć myśleć o tym, co nas najbardziej interesuje – mowa tu oczywiście o implementacji zawartości gry. Zanim przejdziemy do kodowania, wypada zastanowić się nad architekturą naszego rozwiązania.

Rozwiązanie, które zdecydowałem się zastosować w przypadku projektu LaserQuest, to drzewiasta hierarchia obiektów reprezentujących poszczególne składniki gry. W hierarchii tej wyróżnić będziemy kilka poziomów. Na szczycie hierarchii umieszczony zostanie obiekt reprezentujący grę jako całość – w naszym przypadku obiekt ten będzie reprezentowany przez klasę `CLaserQuestGame`. Obiekt ten będzie odpowiedzialny za integrację warstwy gry z warstwą szkieletu aplikacji oraz za zarządzanie trybami gry. Obiekty reprezentujące wspomniane tryby stanowiącą kolejną warstwę w naszej hierarchii. W naszym przypadku wyróżnimy dwa rodzaje trybów: tryb menu (klasa `CLaserQuestGameMenuMode`) oraz tryb rozgrywki (klasa `CLaserQuestGamePlayMode`). Tryb gry, to według prezentowanego tu zamysłu, maszyna stanów określająca, co się w danej chwili z grą dzieje. Dla przykładu – tryb rozgrywki (`CLaserQuestGamePlayMode`) przechowuje i zarządza takimi informacjami jak poziom etapu gry, warunki jej zakończenia itd. Tryb gry może z kolei przechowywać obiekt reprezentujący silnik określonej części rozgrywki. W przypadku LaserQuest wyróżnimy tylko jeden silnik, reprezentowany przez klasę `CLaserQuestGameBoardEngine`. Obiekt silnika – w naszym przypadku stanowiący liść hierarchii – reprezentować będzie planszę na której odbywa się faktyczna rozgrywka. W tym miejscu zaimplementowana będzie mechanika gry oraz reguły nią sterujące. Gwoli jasności należy w tym miejscu mocno podkreślić, że przedstawiona tu hierarchia bazuje na kompozycji (to znaczy: obiekty wchodzące w jej skład zawierają siebie nawzajem), a nie na dziedziczeniu.

To co łączy klasy opisujące obiekty na poszczególnych poziomach to wspólna koncepcja przetwarzania oparta na dwóch metodach: `Draw()` oraz `Update()`. Zadaniem pierwszej z tych metod jest renderowanie danego składnika, druga odpowiada za uaktualnianie oraz obsługę ewentualnych przejść po-

między stanami gry (czytaj: przekazywania kontroli pomiędzy poszczególnymi składnikami). W przypadku programowania bardziej skomplikowanych gier warto zastanowić się nad zbudowaniem szkieletu gry opartego na powyższej koncepcji – tak aby raz zaprogramowaną funkcjonalność można było wykorzystać w przyszłości. Można by dla przykładu stworzyć bazową klasę reprezentującą składnik gry (np. `CLaserQuestGameEntity`) i z niej dziedziczyć klasy bazowe reprezentujące poszczególne składniki hierarchii. W przypadku LaserQuest – aby nie zaciemniać tematu – zbudujemy opisaną wyżej hierarchię w sposób bezpośredni.

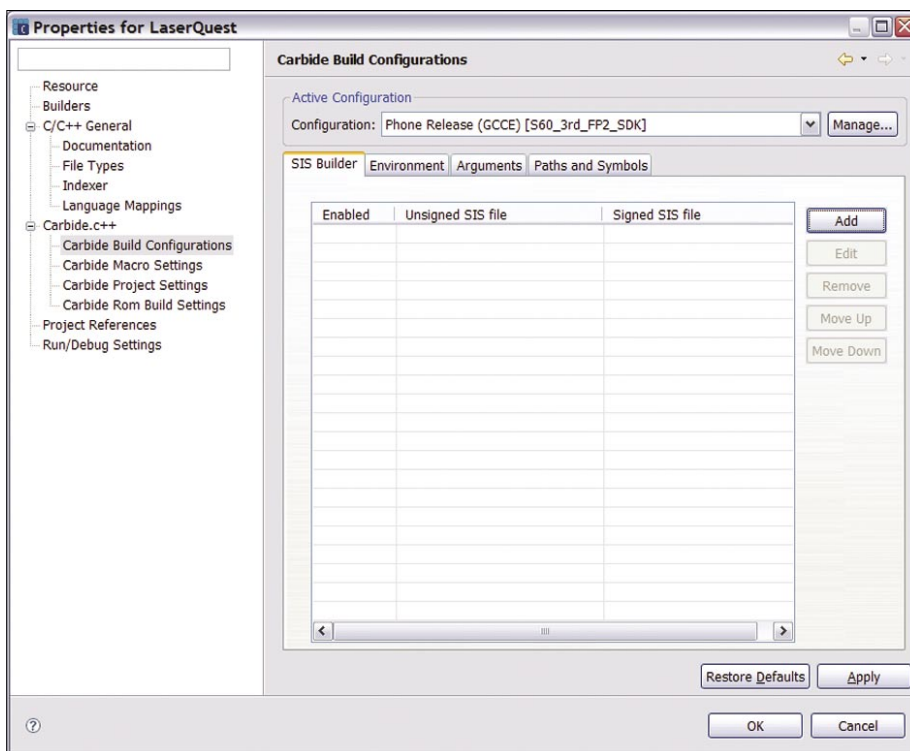
Warto w tym miejscu zatrzymać się na chwilę i zwrócić uwagę na zastosowaną konwencję nazewnictwa klas w naszym projekcie. Ponieważ przy programowaniu aplikacji C++ dla Symbian OS nie stosuje się zazwyczaj przestrzeni nazw, dlatego zdecydowałem się nadawać wszystkim klasom reprezentującym składniki gry nazwy rozpoczynające się prefiksem `CLaserQuestGame`. Podejście takie pozwala w łatwy sposób odróżnić klasy szkieletu aplikacji od klas reprezentujących samą grę.

Warto w tym punkcie poruszyć jeszcze jedną kwestię natury architektonicznej. Podejrzewam, że wielu purystów projektowania obiektowego obruszy się widząc w jednej klasie metody odpowiedzialne za przetwarzanie i renderowanie swojej zawartości. Słyszę niemalże jak padają groźnie pytania pokroju: a co z separacją warstwy logiki od warstwy prezentacji?! Odpowiedź jest prosta: w przypadku programowania gier rzadko zachodzi ku temu potrzeba. Co więcej – w grach komputerowych warstwa logiki często łączy się z warstwą prezentacji (np.: badanie kolizji pomiędzy obiektami występującymi w grze na zasadzie badania zawartości sprite'ów) i czasami wprowadzanie takiego podziału jest sztuczne i powoduje niepotrzebny narzut. W takich sytuacjach warto jest stosować zasadę Brzytwy Ockhama: nie mnożmy niepotrzebnych bytów (czytaj: klas) jeśli nie ma po temu rzeczywistej potrzeby. Co więcej – nasza koncepcja wcale nie kłóci się z ideą separacji wspomnianych warstw. Po prostu użyliśmy do tego innego narzędzia – zamiast umieszczać te funkcjonalności w różnych klasach, umieściliśmy je w różnych funkcjach. Dodatkowo, fakt iż metoda `Draw()` posiada modyfikator `const` jest dodatkowym gwarantem, iż wspomniane warstwy nie będą się przeplatać.

W kolejnych podpunktach opiszę szczegółowo kolejne klasy w przedstawionej tu hierarchii. Schemat hierarchii przedstawiony jest na Rysunku 3.

Gra zamknięta w obiekcie

W niniejszym podpunkcie opisuję interfejs oraz implementację klasy `CLaserQuestGame`,



Rysunek 1. Narzędzie do konfiguracji procesu budowania aplikacji

Listing 6. Definicja metody Draw() w klasie CLaserQuestGame

```
void CLaserQuestGame::Draw( CBitmapContext& aGc ) const
{
    switch ( iMode )
    {
        case EMenu: iMenuMode->Draw( aGc ); break;
        case EPlay: iPlayMode->Draw( aGc ); break;
        default: break;
    }
}
```

Listing 7. Definicja metody Update() w klasie CLaserQuestGame

```
void CLaserQuestGame::Update( TInt64 aDt )
{
    switch ( iMode )
    {
        case EMenu: UpdateMenuMode( aDt ); break;
        case EPlay: UpdatePlayMode( aDt ); break;
        default: break;
    }
}
```

Konfiguracja narzędzia t2t

Aby skonfigurować narzędzie t2t (pod systemami z rodziny Windows) należy podjąć następujące kroki:

- rozpakować archiwum z narzędziem do wybranego podkatalogu;
- ustawić zmienną środowiskową `%T2T_HOME%`, przypisać jej nazwę wybranego podkatalogu i dodać ją do zmiennej `%PATH%`.

Po takich zabiegach konfiguracyjnych (zakładając, że w systemie dostępny jest interpreter języka Perl) narzędzie t2t można uruchomić z dowolnego miejsca w systemie wpisując z linii poleceń komendę `t2t`. Po uruchomieniu tej instrukcji bez żadnych argumentów wyświetlona będzie lista poleceń narzędzia wraz prostą instrukcją obsługi opatrzoną przykładami.

Listing 8. Definicja metody *CLaserQuestGame::UpdateMenuMode()*

```
void CLaserQuestGame::UpdateMenuMode( TInt64 aDt )
{
    iMenuMode->Update( aDt );

    if ( iMenuMode->IsFireKeyPressed() )
    {
        switch ( iMenuMode->ActiveOption() )
        {
            case CLaserQuestGameMenuMode::EExit:
            {
                iExitFlag = ETrue;
                break;
            }
            case CLaserQuestGameMenuMode::EStart:
            {
                iMode = EPlay;
                iPlayMode->Reset();
                break;
            }
        }
    }
}
```

Listing 9. Definicja metody *CLaserQuestGame::UpdatePlayMode()*

```
void CLaserQuestGame::UpdatePlayMode( TInt64 aDt )
{
    iPlayMode->Update( aDt );

    if ( iPlayMode->MenuKeyPressed() )
    {
        iMode = EMenu;
        iMenuMode->Reset();
    }
}
```

LaserQuest – skrócony podręcznik użytkownika

Jak wspomniano we wstępie artykułu, LaserQuest to gra logiczna, bazująca na Lasermanii. Rozgrywka LaserQuest odbywa się na dwuwymiarowej planszy podzielonej na równomierne pola, wypełnionej przeróżnymi elementami. Elementy występujące w grze to:

- emiter lasera, czyli urządzenie generujący wiązkę skondensowanego światła;
- ściany absorbujące promień lasera;
- ściany odbijające promień lasera;
- ruchome lustra odbijające promień lasera;
- czujniki układu alarmowego;
- teleport do kolejnej planszy;
- pojazd sterowany przez gracza.

Zadaniem gracza jest zniszczenie wszystkich czujników układu alarmowego i wejście do teleportu prowadzącego do następnej planszy. Czujniki można niszczyć promieniem lasera. W praktyce oznacza to, że gracz musi w taki sposób manipulować ruchomymi częściami planszy (lustrami) aby nakierować odbijający się promień lasera na kolejne czujniki, a gdy zostaną one zniszczone – dotrzeć do teleportu. Wspomniane manipulacje wykonywane są za pomocą sterowanego joystickiem pojazdu, który można wykorzystać do przesuwania luster.

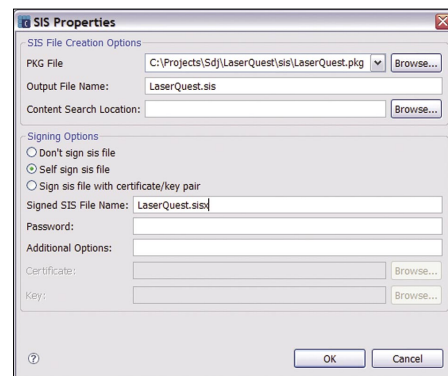
W niniejszym odcinku cyklu logika otaczająca główny silnik gry jest bardzo uboga, aczkolwiek kompletna. Po uruchomieniu gry użytkownik ma do dyspozycji proste menu z dwoma opcjami: Start i Exit. Opcja pierwsza powoduje przejście do testowej planszy i rozpoczęcie rozgrywki. Opcja druga powoduje natychmiastowe opuszczenie aplikacji. Po rozpoczęciu rozgrywki gracz ma następujące opcje. Może sterując pojazdem próbować rozwiązać planszę – jeśli mu się to uda, to rozgrywka zostanie zakończona i gra przejdzie ponownie do trybu menu. Gracz może wrócić do wspomnianego trybu w dowolnym momencie rozgrywki – wciskając prawy softkey. Jeśli w trakcie rozgrywki gracz uzna, że znalazł się w położeniu bez wyjścia, to może zrestartować grę przy pomocy lewego softkey'a.

która, jak wspomniałem wyżej, reprezentuje grę jako całość. Klasa ta jest – niejako z powołania – punktem styku silnika gry oraz opisanego w poprzednim odcinku szkieletu aplikacji. Na początek przyjrzyjmy się definicji klasy (plik *LaserQuestGame.h* w podkatalogu *inc* katalogu projektu; paczkę z kodem źródłowym do niniejszego artykułu można pobrać z witryny SDJ). Definicja klasy przedstawiona jest na Listingu 3.

Prześledźmy na początek publiczne składniki interfejsu tej klasy. Statyczna metoda *NewL()*, konstruktor oraz prywatna metoda *ConstructL()* stanowią część mechanizmu dwufazowej konstrukcji obiektów (ang. *two-phase construction*). Osoby planujące na poważnie zająć się programowaniem natywnych aplikacji dla Symbian OS powinny zapoznać się z tym mechanizmem – tak samo jak z mechanizmem stosu czyszczącego (ang. *cleanup stack*). Obydwa mechanizmy opisane są w Ramce zatytułowanej *Stos czyszczący oraz dwufazowa konstrukcja obiektów w Symbian OS*. W tym punkcie zakładamy, że metoda *NewL()* przejmuje rolę konstruktora klasy. Destruktor klasy uchwalił się na szczęście od Symbianowych dziwactw i nadal pozostaje zwykłym destruktorem. Metody *Draw()* i *Update()* mają szczególne znaczenie. Do *Draw()* przekazywany jest kontekst bitmapy reprezentującej tylny bufor aplikacji. Z kolei *Update()* otrzymuje przyrosty czasu pomiędzy poszczególnymi iteracjami pętli gry. Znaczenie publicznej metody *ExitFlag()* opiszę w dalszej części tego podpunktu. Obiekt klasy *CLaserQuestGame* jest przechowywany i tworzony w obiekcie kontenera (klasa *CLaserQuestContainer*, pliki *LaserQuestContainer.h/cpp*). Tworzenie obiektu gry zrealizowane jest w metodzie *CLaserQuestContainer::ConstructL()*:

```
iGame = CLaserQuestGame::NewL( iKeyState );
```

Warto zwrócić uwagę na to, w jaki sposób do obiektu gry przekazywany jest stan klawiatury. Prywatna składowa *iKeyState* przechowywana jest w klasie



Rysunek 2. Narzędzie do konfiguracji procesu budowania paczki instalacyjnej programu

CLaserQuestGame jako referencja do stałej. Idea tej decyzji projektowej polega na tym, iż mając referencję do zmiennej reprezentującej stan klawiatury wszelkie dotyczące jej zmiany (wykonywane na poziomie klasy CLaserQuestContainer) będą widoczne w klasie CLaserQuestGame. Referencję tę możemy oczywiście przekazać dalej – tak aby obiekty niżej w hierarchii również mogły śledzić stan klawiatury. Wykorzystanie referencji do stałej daje nam gwarancje, że nikt nie popsuje wartości składowej iKeyState (tylko klasa CLaserQuestContainer ma prawo ją uaktualniać, wszyscy inni mogą ją tylko czytać). W rezultacie – stosując takie podejście – możemy badać stan klawiatury na zasadzie bezpośredniego odpytywania (ang. *key polling*), a nie na zasadzie notyfikacji (ang. *key event notification*). Inny mi słowo – to obiekt zainteresowany stanem klawiatury pyta o jej stan, w odróżnieniu od sytuacji gdy obiekt śledzący stan klawiatury informuje o zmianach wszystkich zainteresowanych. Stosowanie pierwszego podejścia w przypadkach pisania prostych gier znacznie ułatwia oprogramowywanie warstwy logiki.

Kolejne miejsca w których *podpinamy* nasz silnik gry to metody Draw() i Update() w klasie CLaserQuestContainer (Listingi 4 i 5).

W przypadku metody CLaserQuestContainer::Draw() stosujemy proste podejście – wystarczy w odpowiednim miejscu wywołać funkcję Draw() na obiekcie iGame. Zawartość metody Update() jest trochę bardziej skomplikowana. Problem z tą metodą polega na tym, że oprócz komunikacji przebiegającej *w dół* hierarchii (w tym konkretnym przypadku mowa jest o wywołaniu iGame->Update(aDt)) musimy również odczytać wiadomość wędrującą *w górę* hierarchii: obiekt reprezentujący grę powiadamia swojego rodzica (czyli klasę CLaserQuestContainer) o tym, że gdzieś w jej wnętrzu zapadła decyzja o zakończeniu działania aplikacji. Aby przekonać się o tym, kontener sprawdza odpowiednią flagę w silniku gry.

Nasi znajomi puryści projektowania obiektowego mogliby tym razem zauważyć, że alternatywą dla tego rozwiązania mógłby być wzorzec projektowy znany pod nazwą Obserwatora, bazujący na wywoływaniu funkcji zwrotnych.

Owszem – jest to alternatywa warta rozważenia, aczkolwiek w przypadku niewielkich gier zastosowanie prostego odpytywania jest zwyczajnie prostsze. Żądanie zakończenia gry trzeba przekazać jeszcze o jeden poziom wyżej, do obiektu klasy CLaserQuestAppUi. W tym celu trzeba było zmodyfikować kod szkieletu i przekazać wskaźnik na wspomniany obiekt do kontenera (patrz: nowa składowa iAppUi w klasie

kontenera). Wywołanie iAppUi->Exit() w CLaserQuestContainer::Update finalnie ułatwia sprawę.

Omówienie mechanizmu interakcji na linii silnik gry – pętla gry mamy za sobą. Wróćmy ponownie do implementacji sil-

Listing 10. Definicja klasy CLaserQuestGameMenuMode

```
class CLaserQuestGameMenuMode : public CBase
{
public:
    enum TOption { EStart, EExit };
    static CLaserQuestGameMenuMode* NewL(
        const TLaserQuestKeyState& aKeyState );
    CLaserQuestGameMenuMode(
        const TLaserQuestKeyState& aKeyState );
    ~CLaserQuestGameMenuMode();
    void Reset();
    void Draw( CBitmapContext& aGc ) const;
    void Update( TInt64 aDt );
    TOption ActiveOption() const
    { return iActiveOption; }
    TBool IsFireKeyPressed() const
    { return iIsFireKeyPressed; }
private:
    void ConstructL();
    const TLaserQuestKeyState& iKeyState;
    TOption iActiveOption;
    CFbsBitmap* iTitle;
    CFbsBitmap* iStart;
    CFbsBitmap* iActiveStart;
    CFbsBitmap* iExit;
    CFbsBitmap* iActiveExit;
    TLaserQuestPeriod
        iMenuOptionsTransitionDelayPeriod;
    TBool iIsFireKeyPressed;
}; // class CLaserQuestGameMenuMode
```

Listing 11. Definicja zasobu LaserQuestMenu.mbm w pliku LaserQuest.mmp

```
START BITMAP LaserQuestMenu.mbm
SOURCEPATH ..\gfx
TARGETPATH resource\apps
HEADER
SOURCE c24 LaserQuestMenuExit.bmp
SOURCE c24 LaserQuestMenuExitActive.bmp
SOURCE c24 LaserQuestMenuStart.bmp
SOURCE c24 LaserQuestMenuStartActive.bmp
SOURCE c24 LaserQuestMenuTitle.bmp
END
```

Listing 12. Zawartość wygenerowanego automatycznie pliku nagłówkowego LaserQuestMenu.mbg

```
// LaserQuestMenu.mbg
// Generated by BitmapCompiler
// Copyright (c) 1998-2001 Symbian Ltd. All rights reserved.
//
enum TMbmLaserquestmenu
{
    EMbmLaserquestmenuLaserquestmenuexit,
    EMbmLaserquestmenuLaserquestmenuexitactive,
    EMbmLaserquestmenuLaserquestmenustart,
    EMbmLaserquestmenuLaserquestmenustartactive,
    EMbmLaserquestmenuLaserquestmenutitle
};
```

nika i przeanalizujemy wewnętrzne mechanizmy jego działania. Jak wspomina-

łem wcześniej, gra LaserQuest składa się z dwóch podstawowych trybów: menu oraz

rozgrywki. Podział ten uwzględniony jest w klasie `CLaserQuestGame`. W enumeracji `CLaserQuestGame::TMode` zdefiniowane są dwie wartości reprezentujące wspomniane tryby: `EMenu` oraz `EPlay`. Informacja o tym jaki jest aktualny tryb gry przechowywany jest w składowej `iMode`. Jeśli zajrzemy do implementacji metod `CLaserQuestGame::Draw()` oraz `CLaserQuestGame::Update()` (Listingi 6 i 7) to zauważymy, że są one zaimplementowane na bazie instrukcji `switch`.

Instrukcje te delegują zadania do obiektów reprezentujących poszczególne tryby gry w zależności od wartości zmiennej `iMode`. W przypadku metody `Draw()` sytuacja jest prosta – wystarczy wywołać metodę `Draw()` dla obiektu reprezentującego aktywny tryb gry. W przypadku metody `Update()` należy dodatkowo obsłużyć komunikację zwrotną – kod odpowiedzialny za to zadanie umieszczony jest w metodach `UpdateMenuMode()` oraz `UpdatePlayMode()` (Listingi 8 i 9).

W `UpdateMenuMode()` oprócz wywołania metody `Update()` na obiekcie `iMenuMode`, sprawdzamy czy użytkownik podjął jakąś akcję związaną z tym trybem. W tym celu wykorzystujemy metody: `IsFireKeyPressed()` oraz `ActiveOption()` stanowiące część publicznego interfejsu klasy `CLaserQuestGameMenuMode`. Pierwsza metoda zwraca wartość `ETrue` w przypadku gdy w trybie menu wykryto wciśnięcie przycisku `fire`. Po wykryciu takiej sytuacji, przy pomocy drugiej metody – `ActiveOption()` – pobierany jest identyfikator opcji, która jest na dany moment aktywna i na tej podstawie podejmowane są kolejne akcje: bądź to ustawienie flagi wyjścia z programu, bądź przejście do trybu rozgrywki. W przypadku `UpdateMenuMode()` dzieje się podobnie – tyle, że do obsłużenia jest tylko jedna możliwość – powrót z trybu rozgrywki do trybu menu.

W jaki sposób tryby gry kontrolują swój stan – o tym już za moment. Na początek sprawdzimy...

...co w menu piszczy

Po zapoznaniu się ze szczegółami implementacji obiektu reprezentującego grę jako całość, czas przyjrzeć się klasom reprezentującym obiekty z drugiej warstwy hierarchii – mowa tu o warstwie trybów gry. Na początek przyjrzymy się prostszej z dwóch klas występujących w `LaserQuest`: `CLaserQuestGameMenuMode`.

Przeanalizujemy Listing 10, na którym znajduje się definicja wspomnianej klasy.

Duża część interfejsu klasy wygląda znajomo – wiele metod z tej klasy było wykorzystywane w obiekcie klasy `CLaserQuestGame`. To co na pewno rzuci się w oczy to zestaw wskaźników do obiektów klasy `CFbsBitmap`. Tak, tak – wreszcie dotarliśmy do miejsca

Listing 13. Fragment definicji metody `CLaserQuestGameMenuMode::ConstructL()` – wczytywanie bitmapy

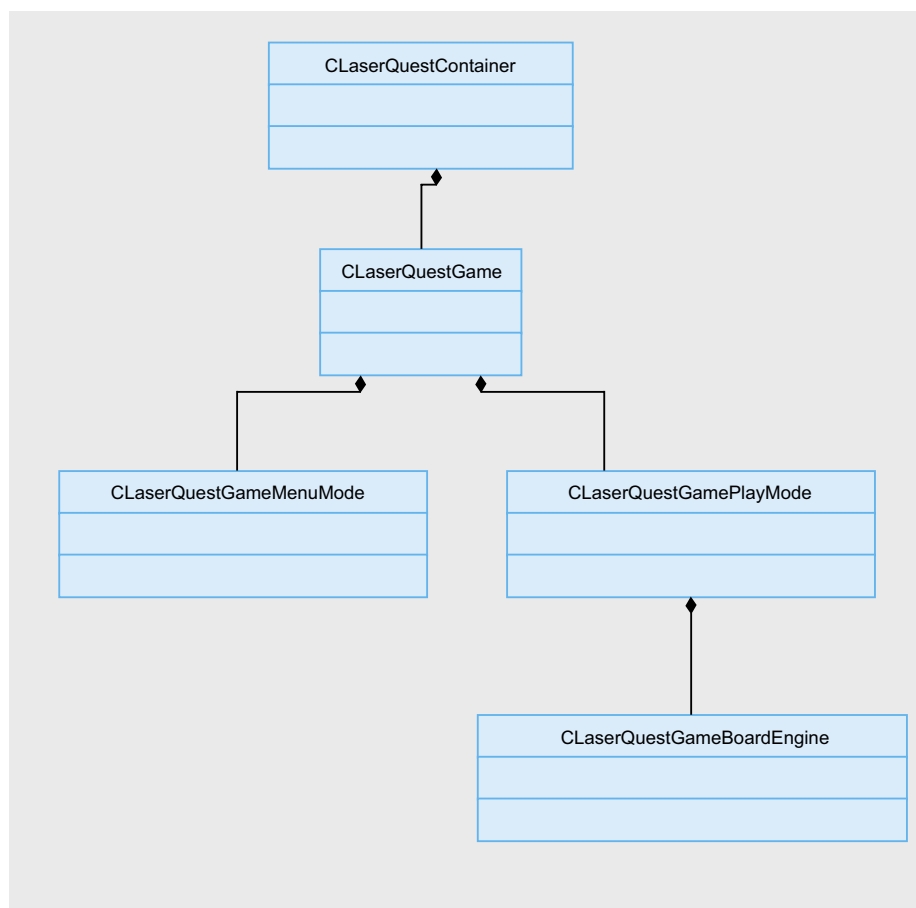
```
void CLaserQuestGameMenuMode::ConstructL()
{
    iTitle = new ( ELeave ) CFbsBitmap();
    User::LeaveIfError( iTitle->Load(
        KLaserQuestMenuMbmFilePath,
        EMbmLaserquestmenuLaserquestmenutitle ) );
    // ...
}
```

Listing 14. Definicja stałej określającej ścieżkę dostępu do pliku `mbm`

```
#ifdef __WINS__
_LIT( KLaserQuestMenuMbmFilePath,
    "z:\\resource\\apps\\LaserQuestMenu.mbm" );
#else
_LIT( KLaserQuestMenuMbmFilePath,
    "c:\\resource\\apps\\LaserQuestMenu.mbm" );
#endif // __WINS__
```

Listing 15. Fragment definicji metody `CLaserQuestGameMenuMode::Draw()`

```
void CLaserQuestGameMenuMode::Draw(
    CBitmapContext& aGc ) const
{
    aGc.BitBlt( TPoint( KTitlePosY, KTitlePosY ),
        iTitle,
        iTitle->SizeInPixels() );
    // ...
}
```



Rysunek 3. Schemat architektury gry LaserQuest

gdzie narysujemy coś więcej niż czarny prostokąt czy informacja o liczniku FPS. Menu w naszej grze jest na tyle proste, że nie warto rozbijać go na mniejsze klasy. Gdyby było inaczej, to zapewne architektura tej części gry byłaby o wiele bardziej złożona; oprogramowanie skomplikowanych interfejsów użytkownika w grach to jednak temat na oddzielny, wcale niekrótki artykuł. W przypadku LaserQuest klasa `CLaserQuestGameMenuMode` jest odpowiedzialna za obsługę menu oraz za jego renderowanie. Przyjrzyjmy się jak zostało to zrealizowane.

Klasa `CFbsBitmap` służy w Symbian OS do reprezentacji specjalnie spreparowanej bitmapy. Bitmapy takie są w procesie budowania aplikacji przetwarzane dedykowanym narzędziem (*bmconv*), a następnie pakowane w pliki z rozszerzeniem *mbm*. Tak spreparowane paczki bitmap są następnie dołączone do instalatora aplikacji i stanowią integralną ich część. Nagrodą, którą programista otrzymuje za uporanie się z całym tym galimatiasem jest możliwość synchronicznego wczytania bitmapy na podstawie automatycznie wygenerowanego identyfikatora (zapisanego jako enumeracja w nagłówkowym pliku z rozszerzeniem *mbg*). Brzmi skomplikowanie? Cóż – niestety – po części tak jest. Prześledźmy jak w praktyce wygląda proces przygotowania bitmap reprezentujących menu gry LaserQuest.

Jako że ponoć jeden obraz wart jest tysiąca słów, zapraszam Czytelników do zapoznania się z Rysunkiem 4, na którym pokazane jest menu gry LaserQuest. Obraz menu składa się z kilku bitmap wyświetlonych na czarnym tle. Bitmapy te są reprezentowane przez następujące składowe w klasie `CLaserQuestGameMenuMode`: `iTitle` (tytuły gry), `iStart` (napis *Start*), `iExit` (napis *Exit*), `iActiveStart` oraz `iActiveExit` (podświetlone napisy *Start* i *Exit*). Bitmapy zostały narysowane w zwykłym edytorze grafiki i zapisane w podkatalogu *gfx* projektu LaserQuest. W dalszej kolejności zmodyfikowano odpowiednio plik definicji projektu (*mmp*) dodając fragment pokazany na Listingu 11.

W definicji określone są źródłowa i docelowa ścieżka dla zasobów oraz lista bitmap, które składają się na paczkę *LaserQuestMenu.mbm*. To, na co warto zwrócić uwagę, to specyfikacja formatu piksela dołączona do każdego elementu we wspomnianej liście. W naszym przypadku używamy wartości `c24`, która oznacza, iż każdy piksel w docelowej bitmapie reprezentowany jest przez 24 bity. Na tym kończy praca programisty związana z przygotowaniem bitmap – pozostała część procesu przygotowania pliku *mbm* zajmuje się łańcuch budowania aplikacji. W tym momencie warto wspomnieć, że Carbide oferuje wygodne, wizualne narzędzie wspomagające dołączanie bitmap do projek-

tu i zwalniające programistę z ręcznego modyfikowania plików *mmp*. Aby dostać się do wspomnianego narzędzia należy otworzyć plik *mmp* w eksploratorze projektu, a nastę-

Listing 16. Definicja metody `CLaserQuestGameMenuMode::Update()`

```
void CLaserQuestGameMenuMode::Update( TInt64 aDt )
{
    iMenuOptionsTransitionDelayPeriod.Update( aDt );

    if ( iMenuOptionsTransitionDelayPeriod.Passed() )
    {
        if ( iKeyState & KKeyUp || iKeyState & KKeyDown )
        {
            switch ( iActiveOption )
            {
                case EStart: iActiveOption = EExit; break;
                case EExit: iActiveOption = EStart; break;
            }
            iMenuOptionsTransitionDelayPeriod.Reset(
                KOneSecondInMicroSeconds / 5 );
        }
    }

    if ( iKeyState & KKeyFire )
    {
        iIsFireKeyPressed = ETrue;
    }
}
```

Listing 17. Definicja klasy `TLaserQuestPeriod`

```
class TLaserQuestPeriod
{
public:
    TLaserQuestPeriod( TInt64 aPeriod = 0 )
        : iPeriod( aPeriod ) { }
    void Reset( TInt64 aPeriod )
        { iPeriod = aPeriod; }
    void Update( TInt64 aDt )
        { if ( iPeriod > 0 ) { iPeriod -= aDt; } }
    TBool Passed() const
        { return iPeriod <= 0; }

private:
    TInt64 iPeriod; // In microseconds.
};
```

Skąd mam wziąć UID dla aplikacji Symbian OS?

UID w kontekście Symbian OS, to unikalny w skali światowej identyfikator aplikacji. Utrzymaniem puli i przydzielaniem UID-ów zajmuje się firma Symbian – producent Symbian OS. Nie każda aplikacja musi mieć unikalny UID – np. do celów testowych można wykorzystywać jeden UID wielokrotnie, jednakże należy mieć świadomość, że gdy zainstalujemy na jednym telefonie dwie różne aplikacje z identycznym UID-em to efekty będą nieprzewidywalne (czytaj: coś się na pewno popsuje). Z tego względu – wszystkie aplikacje pretendujące do uzyskania certyfikacji Symbian Signed (jeden z podstawowych wymogów przy rozpowszechnianiu aplikacji komercyjnych) muszą posiadać swój własny UID. Generalnie, osoby planujące udostępnić swoją aplikację na publicznym forum (niekoniecznie w trybie komercyjnym – może to być również dobrze aplikacja oferowana na zasadach Open Source) powinna postarać się o taki numer. Aby uzyskać UID na własne potrzeby trzeba zarejestrować się na stronie Symbian Signed (<https://www.symbiansigned.com>). Po pomyślnej rejestracji należy przejść do zakładki My Symbian Signed a następnie wybrać z bocznej belki opcję UIDs i Request. Dalej należy postępować według przedstawionych instrukcji. Zamówione UID-y odbiera się za pomocą poczty elektronicznej.

nie wybrać zakładkę *Sources*. Przyjrzymy się teraz jak obsłużyć nasze bitmapy z poziomu

kodu źródłowego. W tym celu dobrze jest zajrzeć do pliku *LaserQuestGameMenuMo-*

de.cpp. Na początku należy dołączyć plik nagłówkowy zawierający definicję enumeracji z identyfikatorami bitmap:

Listing 18. Definicja metody *CLaserQuestGameBoardEngine::ComputeNextVehicleMovePosX()*

```
TInt CLaserQuestGameBoardEngine::NextVehicleMovePosX(
    TDirection aDir ) const
{
    static const TInt KDir2Dx[] = { -1, 1, 0, 0 };
    return iVehiclePosX + KDir2Dx[ aDir ];
}
```

```
#include <LaserQuestMenu.mbg>
```

Plik ten jest generowany automatycznie w trakcie budowania aplikacji i można go znaleźć w katalogu z systemowymi nagłówkami SDK. Plik nagłówkowy dla zasobu *LaserQuestMenu.mbm* w przypadku rozważanej aplikacji wygląda tak jak przedstawiono na Listingu 12.

Wczytywanie bitmap reprezentujących menu zrealizowane jest w metodzie *ConstructL()* klasy *CLaserQuestGameMenuMode*. Przyjrzymy się fragmentowi tej metody (Listing 13).

Na początek należy stworzyć przy pomocy operatora *new* obiekt reprezentujący bitmapę. W drugim kroku – przy pomocy metody *CFbsBitmap::Load()* wczytujemy zawartość bitmapy z pliku *mbm*. Oprócz identyfikatora bitmapy podajemy również ścieżkę dostępu do pliku *mbm*. Ścieżka ta zdefiniowana jest jako stała tekstowa na początku pliku *LaserQuestGameMenuMode.cpp* (Listing 14).

Z racji tego, że ścieżki dostępu do pliku *mbm* są różne w środowisku emulatora i na urządzeniu, dlatego do wyboru właściwej zastosowałem instrukcję warunkową preprocesora. Dla ścisłości należy dodać, iż zmienna *__WINS__* jest definiowana automatycznie w momencie budowania aplikacji w wersji działającej pod emulatorem, co – mam nadzieję – finalnie rozjaśnia sprawę. Niestety, podejście takie nie rozwiązuje do końca naszego problemu.

Chodzi konkretnie o to, że ścieżka będzie niepoprawna w sytuacji gdy zainstalujemy aplikację na karcie pamięci (o ile taka jest dostępna w telefonie). Sytuację tę można obsłużyć sprawdzając w czasie wyko-

Stos czyszczący oraz dwufazowa konstrukcja obiektów w Symbian OS

Pisząc aplikacja C++ dla Symbian OS nie da się uniknąć mechanizmu stosu czyszczącego i idiomu dwufazowej konstrukcji. W tym miejscu opiszę obydwa mechanizmy tak, aby Czytelnik był w stanie zrozumieć prezentowane kody źródłowe i w pewnym ograniczonym zakresie stosować je przy budowaniu własnych gier. Czytelników zainteresowanych szczegółowym zgłębieniem tej dziedziny wiedzy odsyłam do książki *Symbian OS Explained* autorstwa Jo Stichbury. Drugi, trzeci oraz czwarty rozdział tej książki stanowią w mojej opinii najlepsze dostępne wprowadzenie do tematyki opisywanej bardzo pobieżnie w niniejszej Ramce.

Potrzeba wprowadzenia stosu czyszczącego oraz dwufazowej konstrukcji obiektów wzięła się z tego, iż kompilatory używane do budowania aplikacji pod pierwotne wersje systemu Symbian nie obsługiwały sytuacji wyjątkowych. Na dzień dzisiejszy sytuacja ta może bawić, aczkolwiek na ówczesne czasy (początek lat 90) taka była rzeczywistość. Jako, że Symbian OS miał być systemem dedykowanym dla urządzeń mobilnych o mocno ograniczonych zasobach, to jego autorzy byli zmuszeni wprowadzić spójny i przekrojowy system zabezpieczeń przed gubieniem tych zasobów. W tej sytuacji opracowano alternatywny mechanizm obsługi wyjątków w postaci funkcji *User::Leave()* oraz makra *TRAP*. Rozważając temat w kategoriach języka C, mechanizmy te odpowiadają bibliotecznym funkcjom *setjmp()* and *longjmp()*. W kontekście sytuacji wyjątkowych języka C++, funkcja *User::Leave()* działa podobnie jak instrukcja *throw*, zaś *TRAP* pełni rolę bloku *catch*. Jednakże wbudowany mechanizm wyjątków C++ oferuje coś więcej: mowa tu oczywiście o automatycznym wywołaniu destruktorów po pojawieniu się sytuacji wyjątkowej co – przynajmniej w teorii – zapobiega powstawaniu luk w zasobach. Pomijając szereg subtelnych zagrożeń, mechanizm wyjątków C++ stanowi stosunkowo wygodne narzędzie do obsługi nieprzewidzianych sytuacji w programie. W tym kontekście, symbianowy stos czyszczenia stanowi brakujący ów składnik mechanizmu obsługi wyjątków. Niestety – ze względu na oczywiste ograniczenia (jest to w końcu tylko biblioteka próbująca imitować mechanizmy wbudowane w język) – programiści są zmuszeni ręcznie zarządzać obiektami znajdującymi się na takim stosie. To z kolei ciągnie za sobą fakt, iż tylko wybranie klasy mogą ze stosem czyszczenia współpracować – mowa tu o obiektach dziedziczących z klasy *CBase*. Analizując przedstawione w poszczególnych Listingach nagłówki klas Czytelnik może się łatwo przekonać, iż zasadniczo wszystkie klasy reprezentujące silnik *LaserQuest* dziedziczą z *CBase* (stąd też bierze się konwencja zakładająca dodawanie klasom prefiksów – z góry wtedy wiadomo, że klasy o nazwach rozpoczynających się na *C* reprezentują obiekty przechowujące zasoby). W tym miejscu dojdziemy do genezy problemu na które remedium stanowić ma idiom dwufazowej konstrukcji obiektów. Chodzi o to, że wbudowany konstruktor obiektów klasy *C* nie może rzucać wyjątków (tj. wywoływać operacji *leavującej*). Jeśli przyjrzymy się dowolnej klasie przechowującej zasoby (np. dynamicznie alokowane obiekty reprezentujące bitmapy) to zauważymy, że operacje związane z ich alokacją mają nazwy zakończone sufiksem *L*. To oznacza właśnie, iż operacje te mogą bezpośrednio lub pośrednio wywołać operację *User::Leave()*. Z drugiej strony, alokacja pamięci jest sama w sobie operacją potencjalnie *leavującą*. W tej sytuacji – zakładając iż konstruktor klasy może wywoływać operację *User::Leave()*, przy dynamicznej alokacji obiektu mielibyśmy do czynienia z dwoma takimi operacjami wywołanymi pod rząd. A jest to sytuacja niedopuszczalna, gdyż po wykonaniu pierwszej z operacji *leavujących* należy bezzwłocznie umieścić wskaźnik do zaalokowanego zasobu umieścić na stosie czyszczenia (przy pomocy wywołania *CleanupStack::PushL()*). Aby obejść ten problem stosuje się prosty chwyt – wszystkie potencjalnie *leavujące* operacje związane z konstrukcją obiektu umieszcza się w metodzie *ConstructL()*, którą należy wywołać ręcznie po uprzednim włożeniu wskaźnika do zaalokowanego obiektu na stos czyszczenia. Proces ten można ukryć przed użytkownikiem umieszczając opisane wyżej operacje w statycznej metodzie klasy, nazywanej według przyjętej konwencji *NewL()* lub *NewLC()*. Metody te zastępują wbudowany konstruktor, tak że obiekt można zkonstruować w następujący sposób:

```
iMenuMode = CLaserQuestGameMenuMode::NewL( iKeyState );
```

W ramach ćwiczenia zachęcam wszystkich Czytelników do przeanalizowania dołączonych źródeł projektu i zbadania implementacji opisanych powyżej mechanizmów w praktyce.



Rysunek 4. Menu gry *LaserQuest*

niania programu miejsce jej instalacji i dynamicznie budując ścieżkę dostępu. Wróć do tego tematu w kolejnych odsłonach niniejszego cyklu.

Wracając jeszcze do zawartości Listingu 13, warto zwrócić uwagę na użycie konstrukcji `User::LeaveIfError()`. Funkcja ta powoduje rzucenie symbianowego wyjątku w sytuacji kiedy operacja wczytywania zawartości bitmapy z jakichś przyczyn się nie powiedzie. Voila! Jedyne co nam pozostało to wyświetlenia bitmapy, co też czynimy w funkcji `CLaserQuestGameMenuMode::Draw()` (Listing 15).

Rysowanie bitmapy zrealizowane jest przy pomocy operacji szybkiego kopiowania fragmentu pamięci zaimplementowanego jako metoda `CBitmapContext::BitBlt()`. Warto też podkreślić, iż zgodnie z dobrym zwyczajem programistycznym unikamy tak zwanych *magicznych numerów* zapisanych bezpośrednio w kodzie źródłowym, a w zamian za to wykorzystujemy stałe (`KTitlePosX`, `KTitlePosY`). Patrząc z szerszej perspektywy, należy uwypuklić fakt, iż w tym miejscu również idziemy na pewne uproszczenie. Otóż przy profesjonalnej produkcji gier rozłożenie elementów ekranu realizuje się – w miarę możliwości – w sposób niezależny od wyświetlacza. W naszym przypadku, zamiast stałej powinniśmy wykorzystać raczej wartość wyliczaną, bazującą na takich parametrach jak szerokość i wysokość ekranu urządzenia (pobieranych w czasie działania programu). W prezentowanym przykładzie zakładam, że wyświetlacz ma stałe rozmiary (240x320 pikseli).

Przeanalizujemy teraz jak działa metoda `CLaserQuestGameMenuMode::Update()` (Listing 16).

Wspomniana metoda sprawdza stan klawiatury i ustawia odpowiednie flagi. To co ewidentnie rzuca się w oczy to wykorzystanie składowej `iMenuOptionsTransitionDelayPeriod`. Składowa ta jest zdefiniowana w klasie `CLaserQuestGameMenuMode` w następujący sposób:

```
TLaserQuestPeriod
    iMenuOptionsTransitionDelayPeriod;
```

Użyta powyżej klasa `TLaserQuestPeriod` reprezentuje okres czasu wyrażony w milisekundach, a także oferuje proste operacje do zliczania tego okresu na podstawie przyrostów czasu (Listing 17).

Obiekt tej klasy wykorzystywany jest w metodzie `Update()` w celu wprowadzenia opóźnienia przy zmianach opcji w menu. Gdyby ta funkcjonalność zaimplementowana była bez opisanego mechanizmu to, ze względu na fakt, iż wspomniana funkcja wywoływana jest około 30 razy na sekundę, opcje w menu zmieniałyby się tak szybko, że

użytkownik nie byłby w stanie kontrolować tego procesu.

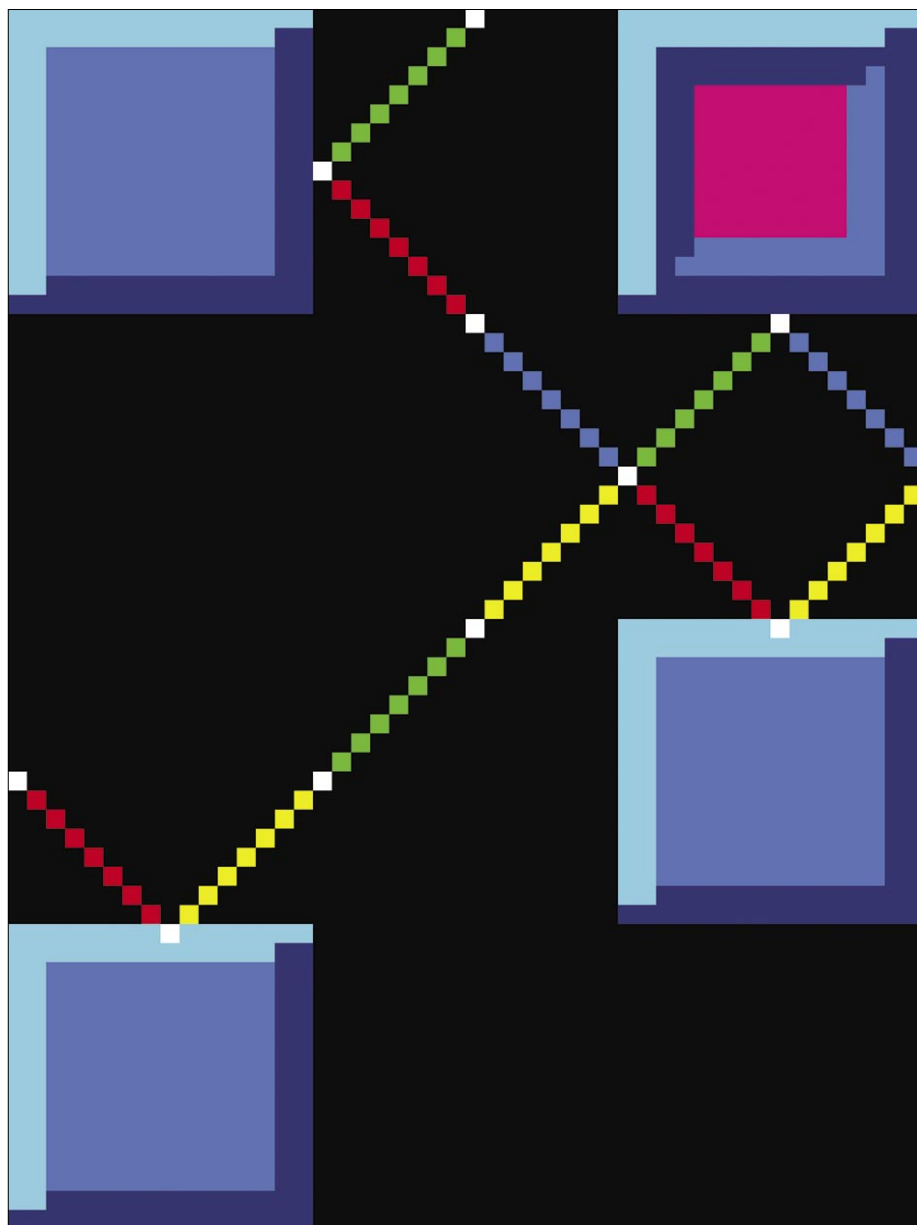
Kończąc niniejszy podpunkt warto zwrócić uwagę na metodę `CLaserQuestGameMenuMode::Reset()`. Metoda ta stanowi pewną architektoniczną konwencję – jest ona wykorzystywana do resetowania stanu obiektu, co jest bardzo przydatne w momencie kiedy przechodzimy pomiędzy trybami gry lub restartujemy tryb gry (będzie napisane więcej na ten temat w kolejnym podpunkcie). Wizualny efekt naszych wysiłków związanych z oprogramowaniem menu gry przedstawiony jest na Rysunku 4.

Rozgrywkę czas zacząć!

Kolejny tryb gry, zaimplementowany w ramach klasy `CLaserQuestGamePlayMode` reprezentuje rozgrywkę. Tryb ten – na ten moment – dostarczony jest nieco na wyrost. Aczkolwiek jego wprowadzenie daje

nam wysoki poziom elastyczności architektury w kontekście kolejnych etapów implementacji gry.

Kiedy w następnych odcinkach cyklu będziemy rozszerzać `LaserQuest` o obsługę wczytywania kolejnych plansz czy system zapisywania stanu gry – tryb ten będzie bardzo przydatny. W bieżącej fazie projektu nie ma się w tym miejscu specjalnie o czym rozpisywać. Praktycznie wszystkie operacje w tym trybie polegają na delegowaniu zadań do silnika planszy. To, na co warto zwrócić uwagę, to mechanizm restartu rozgrywki zaimplementowany w metodzie `Update()`, z wykorzystaniem wspomnianej wcześniej metody `Reset()`. Metoda `CLaserQuestGamePlayMode::Draw()` w przeszłości również będzie bardziej interesująca – umieścimy tam kod odpowiedzialny za wyświetlanie aktualnego numeru planszy oraz statystyk gry, a także napisów reprezentują-



Rysunek 5. Przechodzenie fragmentów wiązki lasera przez elementy planszy

cych opcje przypisane do softkey'ów (restart gry i powrót do menu).

Laserowa łamigłówka

I oto dotarliśmy do implementacji upragnionego silnika planszy. Monolityczna klasa `CLaserQuestGameBoardEngine` stanowi niewątpliwie serce naszej aplikacji: tutaj zaimplementowano mechanikę działania gry. Tym razem podarujemy sobie zamieszczenie Listingu zawierającego pełną definicję klasy – skupimy się na jej publicznym interfejsie oraz zapoznamy się z wybranymi szczegółami jej wewnętrznej implementacji. W tym podpunkcie Czytelnik znajdzie wiele odniesień do plików projektu, a w szczególności do dwóch z nich: `LaserQuestGameBoardEngine.h` oraz `LaserQuestGameBoardEngine.cpp`. Z tego względu, sugerowane jest zaopatrzenie się w paczkę z kodami źródłowymi projektu przed rozpoczęciem czytania niniejszego podpunktu. Przypominam, że wspomniana paczka dostępna jest do pobrania na witrynie SDJ.

Publiczny interfejs klasy reprezentującej planszę gry, czyli punkt styku pomiędzy silnikiem LaserQuest, a obiektem reprezentującym tryb rozgrywki, nie zadziwia niczym niezwykłym. Ponownie widzimy tu takie metody jak `Draw()`, `Update()` czy `Reset()`. Uwagę należy zwrócić za to na metodę `IsSolved()`. Metoda ta odpowiada na pytanie, czy nasza *laserowa łamigłówka* została pomyślnie rozwiązana. Przypominam, że według opisanych wcześniej założeń, warunkiem zakończenia gry jest zniszczenie wszystkich umieszczonych na planszy sensorów i dojście do pola oznaczonego jako wyjście.

Prawdziwa mechanika LaserQuest – zgodnie z zasadą ukrywania informacji – zamknięta jest w metodach stanowiących niepubliczny interfejs klasy. Zanim jednak przejdziemy do opisu tej mechaniki, zbadamy jaką strukturę danych reprezentuje planszę gry. Dla większości Czytelników nie będzie zapewne wielkim zaskoczeniem, iż do

modelowania planszy wykorzystałem zwykłą tablicę liczb całkowitych. Aczkolwiek, diabeł – jak zwykle – tkwi w szczegółach. Pierwsza rzecz – rozmiar planszy: 14 na 18 pól, z czego tak naprawdę wykorzystane jest tylko 12 na 16 pól. Zewnętrzny obszar to strefa tak zwanych wartowników (ang. *sentinels*). Wykorzystanie wartowników to dość prosty zabieg pozwalający uniknąć wstawiania kłopotliwych i podatnych na błędy warunków brzegowych. Przy programowaniu logiki planszy często zachodzi potrzeba sprawdzenia tego co znajduje się na polu o indeksie $i + 1$, lub co gorsza – wstawienia tam nowej wartości. Jeśli nie skorzystamy z wartowników, to w podobnych sytuacjach musimy posiłkować się dodatkowymi warunkami, które zazwyczaj mocno zaciemniają kod.

O błąd naruszenia pamięci w takich warunkach nietrudno... Korzystając z wartowników mamy gwarancję, że przy takich operacjach wyjście poza bufor tablicy nam nie grozi, gdyż wszystkie algorytmy działające na planszy operują na jej wewnętrznym otoczonym warstwą ochronną. Druga sprawa, to reprezentacja poszczególnych składników planszy. A jest tego trochę: puste pole, ściana, ściana-lustro, lustro ruchome, emiter promienia laserowego czy wreszcie sama wiązka lasera – to tylko niektóre elementy rozgrywki. Pomyśl na upakowanie tego wszystkiego w jednej tablicy jest następujący. Wartość zero oznacza wartownika. Wartości mniejsze od zera oznaczają kolejne elementy planszy. Wartość jeden oznacza pole puste. Wartości powyżej jeden to maski bitowe określające jaką część wiązki lasera przechodzi przez dane pole. Warto w tym miejscu zauważyć, że puste pole planszy może zawierać cztery fragmenty wiązki – w różnych konfiguracjach. Idea ta jest przedstawiona na Rysunku 5, gdzie każdy fragment wiązki przedstawiony jest oddzielnym kolorem (zielony – `EBeamUpperLeft`, niebieski – `EBeamUpperRight`, czerwony – `EBeamBottomLeft` oraz żółty – `EBeamBottomRight`).

Innymi słowy – jeśli dane pole ma wartość większą od jeden, to zakładamy, że jest to puste pole przez które przechodzą fragmenty wiązki lasera. Aby przeprowadzić przez dane pole wiązki, możemy skorzystać z operacji bitowych. Na przykład wyrażenie:

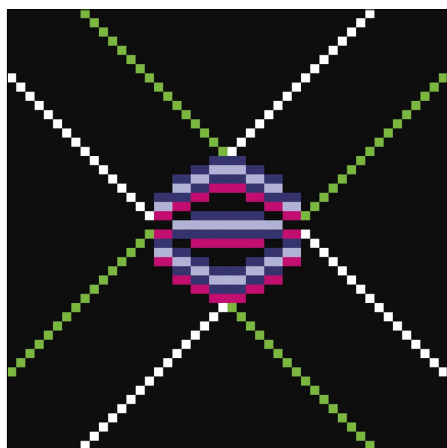
```
iCurrentGridData[ 6 ][ 4 ] |=
    EbeamBottomRight;
```

dodaje do pola o określonych indeksach ($x=6$ i $y=4$) fragment wiązki znajdujący się w jego dolnym prawym rogu. W tym miejscu warto zauważyć, iż plansza gry przechowywana jest w dwóch wersjach: jedna w postaci oryginalnej i druga – zawierająca modyfikacje wprowadzone w trakcie gry. Pojedyncze takie ułatwia implementację mechanizmu restartu, który w przypadku LaserQuest jest nieodzowny, jako że gracz poprzez niewłaściwe przesuwanie luster może doprowadzić do sytuacji, w której ukończenie planszy jest niemożliwe.

Kolejnym ciekawym aspektem związanym z implementacją mechaniki gry jest emisja lasera.

Na początek warto zauważyć, że emiter zaimplementowany w grze może działać w 8 konfiguracjach (jest w stanie wysyłać promień lasera na 8 sposobów). Wspomniana konfiguracja opisana jest przez pozycję emitera na planszy, aktywny bok (tj. bok z którego emitowany jest promień lasera) oraz orientację promienia (lewo- lub prawostronną). Poszczególne konfiguracje pracy emitera pokazane są na Rysunku 6, przy czym wiązki lasera o orientacji lewostronnej zaznaczone są kolorem zielonym, zaś prawostronnej – białym.

Kluczową rolę w implementacji silnika odgrywa funkcja `TrackBeam()`. Funkcja ta odpowiada za śledzenie wiązki lasera od momentu wyjścia z emitera aż do chwili, kiedy trafi ona w materiał, który nie jest w stanie jej odbić. Algorytm ten korzysta intensywnie z tablic przejść (ang. *lookup tables*) opisujących zmiany fragmentów wiązki dla różnych kon-



Rysunek 6. Konfiguracje pracy emitera

Errata

Nikt nie ma patentu na nieomyślność. W tym miejscu opisane są usterki, które wkrały się do poprzedniej części artykułu. Do implementacji metody `CGameSkeletonContainer::OnTick()` wkrał się drobny aczkolwiek znaczący błąd. Chodzi konkretnie o fragment kodu w którym obliczany jest przyrost czasu pomiędzy kolejnymi iteracjami pętli gry:

```
TInt64 dt = iTickStart.Int64() - iTickStop.Int64();
```

Przy tak skonstruowanym zapisie przyrosty przekazywane do metody `CGameSkeletonContainer::Update()` są ujemne...

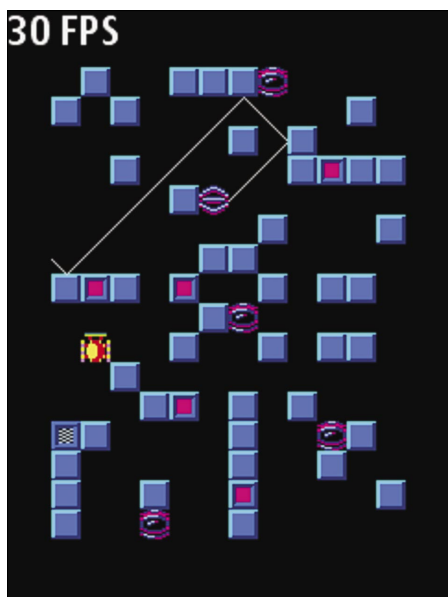
Drugi błąd polega na zastosowaniu zbyt ogólnego kontekstu graficznego w metodzie `CGameSkeletonContainer::Draw()`. Uważni czytelnicy zauważą, iż w projekcie LaserQuest jako kontekst graficzny przekazywany jest obiekt klasy `CBitmapContext`, w odróżnieniu do `CGraphicsContext` używanego w przypadku projektu GameSkeleton. Problem polega na tym, iż ten drugi nie oferuje kluczowej przy programowaniu gier operacji szybkiego kopiowania bitmap (`BitBlt`). Warto w tym miejscu podkreślić iż poprawki opisanych usterek zostały zaaplikowane w przykładowych kodach źródłowych są dostępne na witrynie SDJ.

figuracji wejściowych i zonglując przyrostami określa drogę wiązki, na której pozostawia ślad przy użyciu odpowiednich operacji bitowych. Osoby zainteresowane szczegółami implementacji tego algorytmu zapraszam do przeanalizowania kodów źródłowych. To na co chciałbym zwrócić jeszcze raz uwagę, to wykorzystanie tablic przejść. Ta użyteczna technika pozwala niejednokrotnie uczynić dany fragment kodu krótszym i czytelniejszym, toteż gorąco polecam stosowanie jej przy konstruowaniu swoich własnych aplikacji. Jako przykład stosowania tej techniki mogę podać implementację pomocniczej metody `CLaserQuestGameBoardEngine::ComputeNextVehicleMovePosX()`, przedstawioną na Listingu 18.

Wspomniana metoda oblicza wartość składową na osi X dla kolejnej pozycji pojazdu kierowanego przez gracza przy zadanym kierunku. W implementacji funkcji korzystamy z faktu, iż enumeracje opisujące kierunki ruchu przyjmują wartości z zakresu od 0 do 3. Biorąc to pod uwagę, wykorzystujemy wartość kierunku jako indeks w tablicy przejść zawierającej odpowiednie przyrosty na osi X. Ten sam efekt można by uzyskać stosując operację `switch` do obsługi kolejnych wartościach enumeracji, aczkolwiek przedstawione rozwiązanie wydaje się być bardziej proste, eleganckie i jednocześnie – kompaktowe.

Oprócz struktur danych opisujących rozgrywkę, silnik planszy przechowuje cały szereg bitmap wykorzystywanych do renderowania składników gry. W przypadku kafli (ang. *tiles*), czyli statycznych składników planszy sytuacja jest prosta – poszczególne bitmapy przechowywane są w tablicy:

```
CFbsBitmap* iTileBitmaps[ KLaserQuestTileBitmapCount ];
```



Rysunek 7. Plansza gry

Analizując kod źródłowy silnika warto zwrócić uwagę na makro `TILE_ENUM_VAL_2_TILE_BITMAP_INDEX`, które w sprytny sposób odwzorowuje wartości enumeracji identyfikujących poszczególne kafle na indeks w tablicy z odpowiadającymi im bitmapami. Nieco bardziej złożone struktury danych wykorzystywane są przy animacji. Jako przykład można podać tu dwuwymiarową tablicę służącą do przechowywania klatek animacji pojazdu którym steruje gracz:

```
CFbsBitmap* iVehicleFrames[ KLaserQuestVehicleFrameCount ]
[ KLaserQuestVehicleFrameBitmapCount ] ;
```

Pierwszy wymiar tablicy odpowiada numerom klatek animacji zaś wymiar drugi – poszczególnym kierunkom ruchu.

Zarówno rysowanie jak i uaktualnianie wartości planszy przebiega w kilku etapach – niektóre z nich aktywowane są jedynie w specyficznych przypadkach – tak na przykład dzieje się z animacją zanikającego sensora.

Aby zwiększyć czytelność kodu starałem się grupować metody i składowe w grupy *tematyczne*.

Niestety, w przypadkach takich jak silnik gry LaserQuest, gdzie mamy do oprogramowania zestaw mocno powiązanych i przenikających się reguł gry, ciężko jest uniknąć monolitycznych klas. Zresztą LaserQuest jest i tak projektem stosunkowo nieskomplikowanym – sama Lasermania stanowiąca jego pierwowzór zawiera o wiele szerszy zestaw reguł rozgrywki powiązanych z dodatkowymi typami elementów występujących na planszy gry. Czytelników zainteresowanych szczegółami implementacji silnika zapraszam do analizy kodu źródłowego. Tych mniej cierpliwych zachęcam do obejrzenia Rysunku 7, na którym przedstawione są efekty działania opisanych wyżej mechanizmów.

Gra w kieszeni

Nadszedł czas aby przetestować naszą grę w jej docelowym środowisku. Procedura budowania paczki instalacyjnej jest identyczna do tej opisanej na początku artykułu. Czytelników, którzy są szczęśliwymi posiadaczami komórek Nokia z serii S60 3rd Edition, zapraszam do przetestowania LaserQuest. Mam nadzieję, że rozwiązanie przykładowej planszy (ściągniętej zresztą z oryginalnej Lasermanii) nie sprawi nikomu dużego problemu. Już po kilku minutach grania widać jest, że grze bardzo dużo brakuje do finalnej postaci. Nadal brakuje dźwięku, obsługi wielu plansz, czy wykrywania orientacji ekranu. Do aspektów tych wrócimy w kolejnych odsłonach cyklu, a teraz czas na...

Podsumowanie

Tak oto kończymy drugi odcinek cyklu artykułów o programowaniu gier przeznaczonych dla Symbian OS. Niestety – mój ambitny plan opisany w podsumowaniu z poprzedniego odcinka – spalił na panewce: wiele z obiecanych wtedy zagadnień zmuszony byłem przenieść do kolejnego odcinka. Głównym tego powodem stało się ograniczenie wielkości artykułu. Z drugiej strony – staram się traktować prezentowane zagadnienie bardzo poważnie, opisując niełatwy proces tworzenia gier od podszewki i zahaczając o szczegóły pomijane często w publikacjach o podobnej tematyce.

Mam nadzieję, iż prezentacja cyklu tworzenia gry – od początku do końca – z uwzględnieniem form przejściowych, da pełniejszy pogląd na to zagadnienie. Na sam koniec – zgodnie z tradycją chciałbym zaproponować Czytelnikom zadanie domowe. Proponuje, korzystając z dołączonego do niniejszego artykułu szablonu szkieletu aplikacji oraz narzędzia `t2t`, stworzyć własny projekt i zaimplementować prostą grę o tematyce podobnej do tej, którą reprezentuje LaserQuest (dobrymi przykładami tego rodzaju gier są takie tytuły jak Sokoban czy Boulder Dash). Osoby, które zdecydują się na taki krok, zachęcam również do rozwijanie obranego projektu na bazie technik opisywanych w kolejnych odcinkach niniejszego cyklu – w myśl starego porzekadła, które w zawodzie programisty sprawdza się jak nigdzie indziej: *nie zrozumiesz dopóty, dopóki nie zrobisz tego sam*.

Na koniec chciałbym wymienić osoby, które dzięki swoim wnikliwym recenzjom przyczyniły się do znacznej poprawy jakości niniejszego artykułu. Wspomniane osoby to: Dawid de Rosier, Jacek „Noe” Cybularczyk oraz Piotr Buła.

Chciałbym również tradycyjnie podziękować mojej ukochanej żonie – Oli, za cierpliwość, którą mi nieustannie okazuje.

RAFAŁ KOCISZ

Rafał Kocisz pracuje na stanowisku Dyrektora Technicznego w firmie Gamelion, wchodzącej w skład Grupy BLStream. Rafał specjalizuje się w technologiach związanych z produkcją oprogramowania na platformy mobilne, ze szczególnym naciskiem na tworzenie gier. Grupa BLStream powstała, by efektywniej wykorzystywać potencjał dwóch szybko rozwijających się producentów oprogramowania – BLStream i Gamelion. Firmy wchodzące w skład grupy specjalizują się w wytwarzaniu oprogramowania dla klientów korporacyjnych, w rozwiązaniach mobilnych oraz produkcji i testowaniu gier.

Kontakt z autorem: rafal.kocisz@game-lion.com