Rozdział 13

Duszki i animacja

GDK (*Graphics Drawing Kit*) potrafi więcej, niż tylko rysować linie i prostokąty. Przy naszej niewielkiej pomocy może również zajmować się animacją, co można wykorzystać w grach. GDK nie jest szczególnie wydajną biblioteką do pisania gier, więc próba stworzenia przy jej użyciu kolejnego Dooma albo Quake'a ma niewielkie szanse powodzenia. Jednakże gry, które nie mają wielkich wymagań co do wydajności, mogą korzystać z GDK.

Animacja

Animacja w grach komputerowych polega na szybkim przesuwaniu rysunków po ekranie. Mogą to być statki kosmiczne, ludziki, albo inne obiekty, które mogą reprezentować gracza i komputer. Takie ruchome rysunki często bywają nazywane duszkami (*sprites*).

Animacja wygląda najlepiej, jeśli rysunki na ekranie nie migoczą. Najłatwiejszą metodą poradzenia sobie z tym problemem jest użycie podwójnego buforowania (przypomnijmy sobie przykładowy program zegara z rozdziału 10). Sekwencje animacji w naszych przykładach będą tworzone przy pomocy piksmap, ponieważ ich format jest już nam znany i łatwo jest je rysować z uwzględnieniem przezroczystych obszarów, przez które widać tło.

Animowane sekwencje tworzymy w trzech krokach. Najpierw należy wyczyścić tło i narysować drugoplanową scenerię. Następnie należy umieścić w tle rysunki, które będziemy animować. Trzeci krok polega na skopiowaniu narysowanej klatki animacji do okna w celu wyświetlenia jej na ekranie.

Używanie duszków

Aby przeprowadzić animację, musimy najpierw zdefiniować duszka jako rysunek xpm, z którego możemy uzyskać nadającą się do wyświetlenia piksmapę i maskę. Maska definiuje obszar, na którym będziemy rysować. Rysunki są prostokątne, ale nie chcemy rysować całego prostokąta-lepiej jest zamaskować obszar rysunku i rysować tylko w tych miejscach, gdzie w rysunku są jakieś dane.

Chodzącego człowieka można stworzyć przy pomocy trzech rysunków-nogi razem, lewa noga w przedzie i prawa noga w przedzie. Rysunek ze złączonymi nogami można wykorzystać dwukrotnie, jako fazę przejściową pomiędzy rysunkami „lewa noga w przedzie" – „prawa noga w przedzie" oraz „prawa noga w przedzie" – „lewa noga w przedzie". W jednej ręce człowiek niesie książkę, aby było widać ruch jego ramienia. Używane obrazki wyglądają następująco:

```
static char *xpm_stoi[] = {
"24 24 4 1",
"  c None",
"X c #FFFFFF",
". c #000000",
"b c #0000FF",
"        ......          ",
"       .XXXXXX.         ",
"      .XXXXX.XX.        ",
"     .XXXXXXXX.         ",
"      .XXXXXX.          ",
"        .XX.            ",
"       .XXXX.           ",
"       .X.XX.           ",
"       .X.XX.           ",
"       .X.XX.           ",
"       .X.XX.           ",
"       .X.XX.           ",
"       .X.XX.           ",
"        .b..            ",
"        .bb.            ",
"        .bb.            ",
"        .bb.            ",
"        .bb.            ",
"        .bb.            ",
"        .bb.            ",
"        .bb...          ",
"        .XXXXX.         ",
"        .XXXXX.         ",
"         .....          ",
};

static char *xpm_idzie1[] = {
"24 24 4 1",
```

```
"  c None",
"X c #FFFFFF",
". c #000000",
"b c #0000FF",
"        ......          ",
"       .XXXXXX.         ",
"      .XXXXX.XX.        ",
"      .XXXXXXXX.        ",
"       .XXXXXX.         ",
"         .XX.           ",
"        .XXXX.          ",
"        .XXXX.          ",
"       .X.XX.X.         ",
"       .X.XX.X..        ",
"      .X..XX..XX.       ",
"      .X..XX.XXXX.      ",
"       . .XX..XX.       ",
"         .bb. ..        ",
"         ..b.           ",
"        .b..b.          ",
"        .b..b.          ",
"       .b. .b.          ",
"       .b.  .b.         ",
"      .b.   .b.         ",
"      .b.   .b...       ",
"      .XXX. .XXX.       ",
"      .XXX. .XXX.       ",
"       ...   ...        ",
};

static char *xpm_idzie2[] = {
"24 24 4 1",
"  c None",
"X c #FFFFFF",
". c #000000",
"b c #0000FF",
"        ......          ",
"       .XXXXXX.         ",
"      .XXXXX.XX.        ",
"      .XXXXXXXX.        ",
```

```
"      .XXXXXX.        ",
"       .XX.          ",
"      .XXXX.         ",
"      .XXXX.         ",
"      .X.XX.X.       ",
"     ..X.XX.X.       ",
"    .XX. XX..X.      ",
"   .XXXX.XX..X.       ",
"    .XX..XX..         ",
"     .. .bb.          ",
"       .b..          ",
"      .b..b.         ",
"      .b..b.         ",
"      .b. .b.        ",
"      .b.  .b.       ",
"     .b.   .b.       ",
"     .b..  .b..      ",
"     .XXX. .XXX.     ",
"     .XXX. .XXX.     ",
"      ...   ...      ",
};
```

Rysunki będziemy przechowywać w strukturze danych duszka, zdefiniowanej następująco:

```
typedef struct {

   char **dane_xpm;
   GdkPixmap *piksmapa;
   GdkBitmap *maska;

} typDuszek;
```

Pole dane_xpm przechowuje dane xpm piksmapy, a pola piksmapa i maska przechowują rezultaty wywołania funkcji gdk_pixmap_create_from_xpm_d. Chodzący mężczyzna będzie opisany przez tablicę elementów typDuszek, w której będą przechowywane przedstawione wyżej rysunki. Pola piksmapa i maska struktury są początkowo ustawiane na NULL.

```
/*
 * Chodzący mężczyzna
 */
typDuszek duszek_pan[] =  {
```

```
    { xpm_stoi, NULL, NULL },
    { xpm_idzie1, NULL, NULL },
    { xpm_stoi, NULL, NULL },
    { xpm_idzie2, NULL, NULL },
    { NULL, NULL, NULL }
};
```

Tablica duszka definiuje rysunki oraz kolejność, w jakiej powinny być wyświetlane. Struktura typAktor definiuje położenie duszka na ekranie, numer kolejny obecnie wyświetlanego rysunku, oraz rozmiar rysunku (zakładamy, że wszystkie rysunki w sekwencji mają te same rozmiary).

```
typedef struct {

    int sekw;
    int x;
    int y;
    int wysok;
    int szerok;
    int nSekwencje;
    typDuszek *duszki;

} typAktor;
```

Ładowanie rysunków

Funkcja LadujPiksmapy dołącza do struktury typAktor elementy typDuszek, aby aktor wiedział, który duszek jest wyświetlany i w jakim punkcie ekranu. Wywołujemy funkcję wraz z oknem, aktorem i tablicą duszków, stanowiących poszczególne klatki ruchu aktora. Poniżej wiążemy z aktorem tablicę duszków duszek_pan i inicjujemy strukturę aktora:

```
LadujPiksmapy (okno, &pan, duszek_pan);
```

Kod funkcji LadujPiksmapy zamienia każdy rysunek xpm w piksmapę i maskę. Tablica duszków jest przechowywana w strukturze typAktor, wraz z liczbą duszków w sekwencji oraz rozmiarami duszków.

```
/*
 * LadujPiksmapy
 *
 * Ładuje piksmapy aktorów, pobiera informacje o duszkach
 * z danych xpm i inicjuje dane animacji aktorów.
 */
void LadujPiksmapy (GtkWidget *kontrolka, typAktor *aktor,
```

```
                typDuszek *duszki)
{

  int i = 0;

  /* --- Pobieramy każdą klatkę --- */
  while (duszki[i].dane_xpm) {

    /* --- Zamieniamy dane xpm na piksmapę i maskę --- */
    duszki[i].piksmapa = gdk_pixmap_create_from_xpm_d (
                kontrolka->window,
                &duszki[i].maska,
                NULL,
                duszki[i].dane_xpm);
    i++;
  }

  /* --- Pobieramy wysokość i szerokość duszka --- */
  PobierzSzerIWys (duszki[0].dane_xpm, &aktor->szerok,
            &aktor->wysok);

  /* --- Inicjacja danych duszka --- */
  aktor->sekw = 0;
  aktor->nSekwencje = i;
  aktor->duszki = duszki;
}
```

Wyświetlanie rysunków

Po powiązaniu wszystkich aktorów z duszkami możemy zacząć animację. Podobnie jak w innych przykładach z podwójnym buforowaniem, procedura przerysowująca operuje na drugoplanowej piksmapie. Kiedy rysowanie dobiegnie końca, piksmapa jest kopiowana do obszaru rysunkowego.

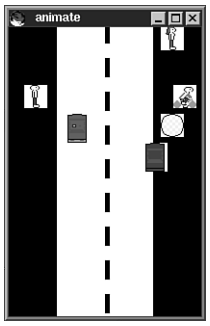
```
/*
 * Przerysuj
 *
 * dane - kontrolka do przerysowania
 */
gint Przerysuj (gpointer dane)
{
  GtkWidget*    obszar_rys = (GtkWidget *) dane;
  GdkRectangle  uakt_prostokat;
```

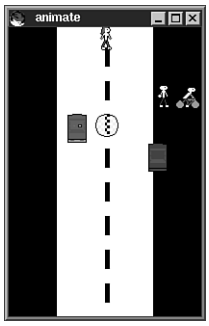
```
static przesuniecie = 0;
int nGora;

/* --- czyścimy piskmapę (rysunek drugoplanowy) --- */
gdk_draw_rectangle (piksmapa,
        obszar_rys->style->black_gc,
        TRUE,
        0, 0,
        obszar_rys->allocation.width,
        obszar_rys->allocation.height);

/* --- Rysujemy ulicę --- */
gdk_draw_rectangle (piksmapa,
        obszar_rys->style->white_gc,
        TRUE,
        50, 0,
        100,
        obszar_rys->allocation.height);

/*
 * Rysujemy linie na jezdni
 */

/* --- Ustalamy, gdzie powinna być pierwsza linia --- */
przesuniecie++;
if ((przesuniecie - DLUG_PRZERWY) >= 0) {
   przesuniecie -= (DLUG_LINII + DLUG_PRZERWY);
}

/* --- Rysujemy wszystkie linie na jezdni --- */
nGora = przesuniecie;
do {
   gdk_draw_rectangle (piksmapa,
        obszar_rys->style->black_gc,
        TRUE,
        100, nGora, 5, DLUG_LINII);
   nGora += DLUG_LINII + DLUG_PRZERWY;
} while (nGora < obszar_rys->allocation.height);

/* --- Rysujemy wszystkie duszki --- */
KlatkaAnimacji (obszar_rys, &rower, 3);
KlatkaAnimacji (obszar_rys, &pan, 2);
```

```
    KlatkaAnimacji (obszar_rys, &pani, 2);
    KlatkaAnimacji (obszar_rys, &policja, 0);
    KlatkaAnimacji (obszar_rys, &pilka1, 2);
    KlatkaAnimacji (obszar_rys, &auto1, 3);

    /* --- Trzeba uaktualnić cały ekran --- */
    uakt_prostokat.x = 0;
    uakt_prostokat.y = 0;
    uakt_prostokat.width = obszar_rys->allocation.width;
    uakt_prostokat.height = obszar_rys->allocation.height;

    /* --- Uaktualniamy go --- */
    gtk_widget_draw (obszar_rys, &uakt_prostokat);
}
```

Funkcja KlatkaAnimacji rysuje każdego animowanego duszka na drugoplanowej piksmapie przy pomocy funkcji gdk_draw_pixmap. Zwykle rysowanie duszka spowodowałoby narysowanie całego zawierającego go prostokąta (patrz rysunek 13.1).

Można na szczęście pozbyć się tego efektu, używając maski, zwróconej przez funkcję gdk_pixmap_create_from_xpm_d. Maska tworzona jest na podstawie kolorów, zdefiniowanych w danych xpm. Jedynym kolorem, który nie jest uwzględniany w masce, jest kolor None; staje się on przezroczystym kolorem maski. Funkcja gdk_gc_set_clip_mask ustawia maskę, która zostanie wykorzystana w funkcji gdk_draw_pixmap. Maska musi zostać umieszczona w punkcie, gdzie będzie rysowana piksmapa, przy pomocy funkcji gdk_gc_set_clip_origin. Dzięki tym dwóm funkcjom pozbywamy się prostokąta wokół umieszczanego na ekranie rysunku (patrz rysunek 13.2).





Rysunek 13.1. Duszki bez przezroczysto-

ści.

Rysunek 13.2. Duszki z przezroczystością.

Funkcja KlatkaAnimacji przyjmuje aktora, uaktualnia jego położenie, dodając odległość (nOdlegl) do jego współrzędnej x i rysuje aktora na ekranie. Rysowanie może odbywać się albo przy pomocy maski, albo bez niej-określa do znacznik bUzyjMaski. Rysowanie bez maski sprawia, że wokół duszka pojawia się duży prostokąt (patrz rysunek 13.2). Oczywiście, jeśli podczas rysowania duszka używaliśmy maski, powinniśmy po zakończeniu rysowania ustawić ją na NULL, ponieważ nie jest już potrzebna. Oto pełny kod funkcji, wyświetlającej całą sekwencję animacji:

```
/*
 * KlatkaAnimacji
 *
 * Przechodzi do następnego duszka w sekwencji
 * i rysuje go przy użyciu maski.
 */
KlatkaAnimacji (GtkWidget *obszar_rys, typAktor *aktor, int nOdlegl)
{
   GdkGC *gc;

   aktor->x += nOdlegl;
   if (aktor->x > obszar_rys->allocation.width) {
      aktor->x = 0;
   }

   /* --- Używamy następnego rysunku w sekwencji --- */
   aktor->sekw++;

   /* --- Jeśli jesteśmy na końcu sekwencji, używamy 0 --- */
   if (aktor->sekw >= aktor->nSekwencje) {
      aktor->sekw = 0;
   }

   /* --- Pobieramy kolor pierwszoplanowy --- */
   gc = obszar_rys->style->fg_gc[GTK_STATE_NORMAL];

   if (bUzyjMaski) {

      /* --- Ustawiamy przycinanie duszków --- */
      gdk_gc_set_clip_mask (gc, aktor->duszki[aktor->sekw].maska);

      /* --- Ustwiamy punkt początkowy przycinania --- */
```

```
        gdk_gc_set_clip_origin (gc, aktor->x, aktor->y);
    }

    /* --- Kopiujemy odpowiednio przyciętą piksmapę do okna --- */
    gdk_draw_pixmap (piksmapa,
        obszar_rys->style->fg_gc[GTK_STATE_NORMAL],
        aktor->duszki[aktor->sekw].piksmapa,
        0, 0,
        aktor->x, aktor->y,
        aktor->szerok, aktor->wysok);

    if (bUzyjMaski) {

        /* --- Czyścimy maskę przycinania --- */
        gdk_gc_set_clip_mask (gc, NULL);
    }
}
```

Pełny kod źródłowy

Poniżej umieszczamy pełny program, demonstrujący animację. Zawiera on kilka animowanych sekwencji, które przesuwają się przez okno programu. Większość duszków składa się z kilku kolejnych piksmap, dzięki którym wydają się być w ruchu. Chodzący mężczyzna stawia kroki, a rowerzysta podczas jazdy kręci pedałami. Duszek piłki jest przykładowym rysunkiem zawierającym wewnątrz przezroczyste obszary - kiedy piłka porusza się nad tłem i innymi duszkami, można przez nią dojrzeć inne obiekty.

```
/*
 * Autor: Eric Harlow
 *
 * Tworzenie aplikacji w Linuksie
 * Demonstracja duszków
 */

#include <gtk/gtk.h>
#include <time.h>

#include "pan.h"
#include "pani.h"
#include "pilka.h"
#include "auto.h"
```

```
#include "policja.h"
#include "rower.h"

#define DLUG_LINII 20
#define DLUG_PRZERWY 15

/*
 * Struktura z danymi duszka
 */
typedef struct {

   char **dane_xpm;
   GdkPixmap *piksmapa;
   GdkBitmap *maska;

} typDuszek;

/*
 * Struktura aktora.  Aktor składa się z jednego
 * lub kilku duszków.
 */
typedef struct {

   int sekw;
   int x;
   int y;
   int wysok;
   int szerok;
   int nSekwencje;
   typDuszek *duszki;

} typAktor;

/*
 * Chodzący mężczyzna
 */
typDuszek duszek_pan[] =  {
   { xpm_stoi, NULL, NULL },
   { xpm_idzie1, NULL, NULL },
   { xpm_stoi, NULL, NULL },
   { xpm_idzie2, NULL, NULL },
   { NULL, NULL, NULL }
```

```
};

/*
 * Mężczyzna na rowerze
 */
typDuszek duszek_rower[] = {
    { xpm_rower1, NULL, NULL },
    { xpm_rower2, NULL, NULL },
    { xpm_rower3, NULL, NULL },
    { NULL, NULL, NULL }
};

/*
 * Chodząca kobieta
 */
typDuszek duszek_pani[] =  {
    { xpm_pani, NULL, NULL },
    { xpm_paniidzie1, NULL, NULL },
    { xpm_panistoi, NULL, NULL },
    { xpm_paniidzie2, NULL, NULL },
    { NULL, NULL, NULL }
};

/*
 * Samochód policyjny
 */
typDuszek duszek_policja[] = {
    { xpm_policja1, NULL, NULL },
    { xpm_policja2, NULL, NULL },
    { NULL, NULL, NULL }
};

/*
 * Częściowo przezroczysta piłka
 */
typDuszek duszek_pilka[] = {
    { xpm_pilka1, NULL, NULL },
    { NULL, NULL, NULL }
};

/*
```

```
 * Samochód
 */
typDuszek duszek_auto[] = {
   { xpm_auto1, NULL, NULL },
   { xpm_auto1, NULL, NULL },
   { NULL, NULL, NULL }
};

/*
 * Oto gwiazdy naszego filmu
 */
typAktor pan;
typAktor rower;
typAktor pani;
typAktor policja;
typAktor pilka1;
typAktor auto1;

/* --- Drugoplanowa piksmapa dla obszaru rysunkowego --- */
static GdkPixmap *piksmapa = NULL;

/* --- Znacznik używania maski --- */
static int bUzyjMaski = TRUE;

/*
 * Prototypy.
 */
void PobierzSzerIWys (gchar **xpm, int *szerok, int *wysok);

/*
 * LadujPiksmapy
 *
 * Ładuje piksmapy aktorów, pobiera informacje o duszkach
 * z danych xpm i inicjuje dane animacji aktorów.
 */
void LadujPiksmapy (GtkWidget *kontrolka, typAktor *aktor,
                    typDuszek *duszki)
{

   int i = 0;

   /* --- Pobieramy każdą klatkę --- */
```

```
    while (duszki[i].dane_xpm) {

      /* --- Zamieniamy dane xpm na piksmapę i maskę --- */
      duszki[i].piksmapa = gdk_pixmap_create_from_xpm_d (
                    kontrolka->window,
                    &duszki[i].maska,
                    NULL,
                    duszki[i].dane_xpm);
      i++;
    }

    /* --- Pobieramy wysokość i szerokość duszka --- */
    PobierzSzerIWys (duszki[0].dane_xpm, &aktor->szerok, &aktor->wysok);

    /* --- Inicjacja danych duszka --- */
    aktor->sekw = 0;
    aktor->nSekwencje = i;
    aktor->duszki = duszki;
}

/*
 * KlatkaAnimacji
 *
 * Przechodzi do następnego duszka w sekwencji
 * i rysuje go przy użyciu maski.
 */
KlatkaAnimacji (GtkWidget *obszar_rys, typAktor *aktor, int nOdlegl)
{
    GdkGC *gc;

    aktor->x += nOdlegl;
    if (aktor->x > obszar_rys->allocation.width) {
      aktor->x = 0;
    }

    /* --- Używamy następnego rysunku w sekwencji --- */
    aktor->sekw++;

    /* --- Jeśli jesteśmy na końcu sekwencji, używamy 0 --- */
    if (aktor->sekw >= aktor->nSekwencje) {
      aktor->sekw = 0;
    }
```

```
    /* --- Pobieramy kolor pierwszoplanowy --- */
    gc = obszar_rys->style->fg_gc[GTK_STATE_NORMAL];

    if (bUzyjMaski) {

      /* --- Ustawiamy przycinanie duszków --- */
      gdk_gc_set_clip_mask (gc, aktor->duszki[aktor->sekw].maska);

      /* --- Ustwiamy punkt początkowy przycinania --- */
      gdk_gc_set_clip_origin (gc, aktor->x, aktor->y);
    }

    /* --- Kopiujemy odpowiednio przyciętą piksmapę do okna --- */
    gdk_draw_pixmap (piksmapa,
        obszar_rys->style->fg_gc[GTK_STATE_NORMAL],
        aktor->duszki[aktor->sekw].piksmapa,
        0, 0,
        aktor->x, aktor->y,
        aktor->szerok, aktor->wysok);

    if (bUzyjMaski) {

      /* --- Czyścimy maskę przycinania --- */
      gdk_gc_set_clip_mask (gc, NULL);
    }
}

/*
 * Przerysuj
 *
 * dane - kontrolka do przerysowania
 */
gint Przerysuj (gpointer dane)
{
    GtkWidget*    obszar_rys = (GtkWidget *) dane;
    GdkRectangle  uakt_prostokat;
    static przesuniecie = 0;
    int nGora;

    /* --- czyścimy piskmapę (rysunek drugoplanowy) --- */
    gdk_draw_rectangle (piksmapa,
            obszar_rys->style->black_gc,
```

```
        TRUE,
        0, 0,
        obszar_rys->allocation.width,
        obszar_rys->allocation.height);

/* --- Rysujemy ulicę --- */
gdk_draw_rectangle (piksmapa,
        obszar_rys->style->white_gc,
        TRUE,
        50, 0,
        100,
        obszar_rys->allocation.height);

/*
 * Rysujemy linie na jezdni
 */

/* --- Ustalamy, gdzie powinna być pierwsza linia --- */
przesuniecie++;
if ((przesuniecie - DLUG_PRZERWY) >= 0) {
   przesuniecie -= (DLUG_LINII + DLUG_PRZERWY);
}

/* --- Rysujemy wszystkie linie na jezdni --- */
nGora = przesuniecie;
do {
   gdk_draw_rectangle (piksmapa,
        obszar_rys->style->black_gc,
        TRUE,
        100, nGora, 5, DLUG_LINII);
   nGora += DLUG_LINII + DLUG_PRZERWY;
} while (nGora < obszar_rys->allocation.height);

/* --- Rysujemy wszystkie duszki --- */
KlatkaAnimacji (obszar_rys, &rower, 3);
KlatkaAnimacji (obszar_rys, &pan, 2);
KlatkaAnimacji (obszar_rys, &pani, 2);
KlatkaAnimacji (obszar_rys, &policja, 0);
KlatkaAnimacji (obszar_rys, &pilka1, 2);
KlatkaAnimacji (obszar_rys, &auto1, 3);

/* --- Trzeba uaktualnić cały ekran --- */
```

```
    uakt_prostokat.x = 0;
    uakt_prostokat.y = 0;
    uakt_prostokat.width = obszar_rys->allocation.width;
    uakt_prostokat.height = obszar_rys->allocation.height;

    /* --- Uaktualniamy go --- */
    gtk_widget_draw (obszar_rys, &uakt_prostokat);
}

/*
 * configure_event
 *
 * Tworzy nową drugoplanową piksmapę o odpowiednich
 * rozmiarach. Wywoływana jest przy każdej zmianie
 * rozmiarów okna. Musimy zwolnić przydzielone
 * zasoby.
 */
static gint configure_event (GtkWidget *kontrolka,
                             GdkEventConfigure *zdarzenie) {
    /* --- Zwalniamy drugoplanową piksmapę,
           jeśli już ją przydzieliliśmy --- */
    if (piksmapa) {
        gdk_pixmap_unref (piksmapa);
    }

    /* --- Tworzymy piksmapę o nowych rozmiarach --- */
    piksmapa = gdk_pixmap_new (kontrolka->window,
                kontrolka->allocation.width,
                kontrolka->allocation.height,
                -1);

    return TRUE;
}

/*
 * expose_event
 *
 * Wywoływana wtedy, kiedy zostanie odsłonięte
 * okno, albo po wywołaniu procedury gdk_widget_draw.
 * Kopiuje drugoplanową piksmapę do okna.
 */
```

```
gint expose_event (GtkWidget *kontrolka, GdkEventExpose *zdarzenie)
{

   /* --- Kopiujemy piksmapę do okna --- */
   gdk_draw_pixmap (kontrolka->window,
        kontrolka->style->fg_gc[GTK_WIDGET_STATE (kontrolka)],
        piksmapa,
        zdarzenie->area.x, zdarzenie->area.y,
        zdarzenie->area.x, zdarzenie->area.y,
        zdarzenie->area.width, zdarzenie->area.height);

   return FALSE;
}

/*
 * zamknij
 *
 * Koniec programu
 */
void zamknij ()
{
   gtk_exit (0);
}
/*
 * PobierzSzerIWys
 *
 * Pobiera szerokość i wysokość z danych xpm
 *
 */
void PobierzSzerIWys (gchar **xpm, int *szerok, int *wysok)
{

     sscanf (xpm [0], "%d %d", szerok, wysok);
}

/*
 * main
 *
 * Program zaczyna się tutaj
 */
int main (int argc, char *argv[])
```

```
{
   GtkWidget *okno;
   GtkWidget *obszar_rys;
   GtkWidget *xpole;

   /* --- Inicjacja GTK --- */
   gtk_init (&argc, &argv);

   if (argc > 1) {
      bUzyjMaski = FALSE;
   }

   /* --- Tworzymy okno najwyższego poziomu --- */
   okno = gtk_window_new (GTK_WINDOW_TOPLEVEL);

   xpole = gtk_hbox_new (FALSE, 0);
   gtk_container_add (GTK_CONTAINER (okno), xpole);
   gtk_widget_show (xpole);

   gtk_signal_connect (GTK_OBJECT (okno), "destroy",
            GTK_SIGNAL_FUNC (zamknij), NULL);

   /* --- Tworzymy obszar rysunkowy  --- */
   obszar_rys = gtk_drawing_area_new ();
   gtk_drawing_area_size (GTK_DRAWING_AREA (obszar_rys), 200, 300);
   gtk_box_pack_start (GTK_BOX (xpole), obszar_rys, TRUE, TRUE, 0);

   gtk_widget_show (obszar_rys);

   /* --- Sygnały używane do obsługi drugoplanowej piksmapy --- */
   gtk_signal_connect (GTK_OBJECT (obszar_rys), "expose_event",
            (GtkSignalFunc) expose_event, NULL);
   gtk_signal_connect (GTK_OBJECT(obszar_rys),"configure_event",
            (GtkSignalFunc) configure_event, NULL);

   /* --- Pokazujemy okno --- */
   gtk_widget_show (okno);

   /* --- Przerysowujemy co pewien czas --- */
   gtk_timeout_add (100, Przerysuj, obszar_rys);

   /* --- Ładujemy wszystkie duszki --- */
   LadujPiksmapy (okno, &pan, duszek_pan);
```

```
    LadujPiksmapy (okno, &rower, duszek_rower);
    LadujPiksmapy (okno, &pani, duszek_pani);
    LadujPiksmapy (okno, &policja, duszek_policja);
    LadujPiksmapy (okno, &pilka1, duszek_pilka);
    LadujPiksmapy (okno, &auto1, duszek_auto);

    /* --- Rozmieszczamy duszki --- */
    rower.x = 30;
    rower.y = 60;

    pan.x = 50;
    pan.y = 60;

    pan.x = 60;
    pan.y = 60;

    policja.x = 60;
    policja.y = 90;

    pilka1.x = 0;
    pilka1.y = 90;

    auto1.x = 0;
    auto1.y = 120;

    /* --- Wywołujemy główną pętlę GTK --- */
    gtk_main ();

    return 0;
}
```

Gry wideo

Możliwości GDK wykraczają poza prostą animację. Możemy wykorzystać wiedzę zdobytą w poprzednim rozdziale, aby stworzyć ramy gry opartej na popularnym w latach osiemdziesiątych Defenderze. W grze tej gracz latał nad powierzchnią planety, usiłując powstrzymać obcych najeźdźców przed porwaniem swoich ludzi. Obcy próbowali uprowadzić ludzi z powierzchni i donieść ich na górę ekranu, gdzie ulegali mutacji. Mutacja powodowała śmierć człowieka i powstanie groźniejszego obcego.

Autor musiał odtworzyć grę z pamięci (nie zdołał znaleźć salonu gier, w którym ciągle stałyby antyczne automaty). Konieczne było także wyeliminowanie pewnych cech oryginału (ze względu na ograniczoną objętość książki). Program nie

nalicza punktów, brakuje także niektórych obcych. Celem jest zademonstrowanie możliwości GDK w krótkiej grze, którą łatwo można rozszerzyć (co, jak zwykle, pozostawiamy jako ćwiczenie dla czytelników).

Gry oparte na GTK+/GDK

W przykładowym programie wykorzystamy wiedzę o duszkach i animacji, wyniesioną z lektury poprzedniego rozdziału, aby stworzyć szkielet gry. Animacja w Defenderze przypomina animację w poprzednim przykładzie, ale tym razem jednym z duszków steruje gracz, a komputer kontroluje pozostałe duszki oraz oddziaływania pomiędzy wszystkimi obiektami w grze. Jak łatwo się domyślić, możemy użyć GTK+ i GDK do stworzenia bardziej wypracowanych gier, niż Defender.

Oryginalna gra jest dużo bardziej skomplikowania, ale poniższy przykład pokazuje, że w GTK+ można napisać grę wideo nie obciążającą zbytnio procesora, zwłaszcza na nowszym sprzęcie (innymi słowy, na Pentium 133 gra pracowała szybko, wykorzystując procesor w niewielkim stopniu. Prawdopodobnie nie działałaby zbyt dobrze na 386 albo 486SX-16).

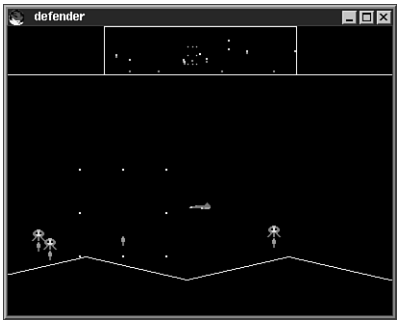
Elementy gry

Podczas tworzenia gry należy rozważyć kilka czynników: sterowanie, grafikę, sztuczną inteligencję, efekty specjalne, sprawdzanie kolizji, poziomy trudności oraz naliczanie punktów. Z pewnością chcielibyśmy umieścić te elementy w grze. W przykładowym programie nie będziemy jednak zajmować się wszystkimi tymi czynnikami; na przykład w naszej wersji Defendera nie będzie punktowania ani poziomów trudności. Oczywiście, w prawdziwej grze wzrastający poziom trudności jest niezbędny. Mało urozmaicona gra, która nie stanowi wyzwania, szybko kończy swój żywot na półce z przecenionym oprogramowaniem.

Wejście

Klawiatura jest urządzeniem wejściowym, które na pewno posiadają wszyscy użytkownicy komputera, dlatego grą będziemy sterować za pośrednictwem klawiszy. W grze istnieje możliwość poruszania się (w czterech kierunkach) oraz oddawania strzałów do obcych (patrz rysunek 13.3). Gracz mógłby poruszać statkiem przy pomocy klawiszy kursora i strzelać klawiszem spacji, ale zaprogramowanie tego jest trudniejsze, niż można by przypuszczać. Pisanie gier różni się od pisania innych aplikacji-musimy w każdej chwili wiedzieć, które klawisze są wciśnięte. Gracz może na przykład przytrzymywać klawisz kierunku, jednocześnie uderzając spację, aby zestrzelić obce statki. W domyślnym trybie GTK+ nie da się tego przeprowadzić, ponieważ do aplikacji wysyłany jest tylko kod ostatnio naciśniętego klawisza. Jeśli użytkownik przytrzyma klawisz, wówczas

klawisz zacznie się powtarzać. Klawisze, które były wciśnięte przed naciśnięciem nowego klawisza, zostaną zignorowane.



Rysunek 13.3. Lądowniki porywają ludzi – jeden z lądowników eksplodował, upuszczając człowieka.

Sprawdzanie klawiszy

Można uporać się z problemem klawiszy, używając sygnałów key_press_event i key_press_release. Po wciśnięciu klawisza wysyłany jest sygnał key_press_event, a po jego zwolnieniu – key_press_release. Mówiąc ściśle, sygnał key_press_release domyślnie nie jest wysyłany, więc zwykłe zdefiniowanie funkcji zwrotnej nie zadziała. Aby otrzymywać informacje o zwolnieniu klawiszy, musimy wywołać funkcję gtk_widget_set_events ze znacznikiem GDK_KEY_RELEASE_MASK. Dzięki temu sygnał key_release_ event będzie wysyłany do okna. GTK+ odfiltrowuje sygnały o mniejszym znaczeniu, więc musimy jawnie zażądać ich przekazywania.

Powtarzanie klawiszy

Powtarzanie klawiszy jest w naszym programie zupełnie zbędne, chociaż jedynym efektem przytrzymywania wielu klawiszy byłoby niepotrzebne obciążenie, związane z przetwarzaniem nietypowych kombinacji klawiszy. Funkcja gdk_key_repeat_disable wyłącza powtarzanie klawiszy w aplikacji. Po wywołaniu tej funkcji każdy naciśnięty klawisz wysyła pojedynczy sygnał key_press_event, nawet jeśli zostanie przytrzymany. Funkcja gdk_key_repeat_disable ma jednak efekt globalny – wpływa nie tylko na grę, ale na wszystkie działające aplikacje. Aby temu zapobiec, funkcja gdk_key_repeat_disable jest wywoływana z procedury obsługi sygnału focus_in_event, który wskazuje, że gra otrzymała ognisko. Gdy ognisko opuści grę, funkcja zwrotna dla sygnału focus_out_event wywołuje funkcję gdk_key_repeat_restore, aby przywrócić powtarzanie klawiszy; dzięki temu

gracz może spokojnie pracować w arkuszu kalkulacyjnym, kiedy do pokoju wejdzie szef.

Grafika

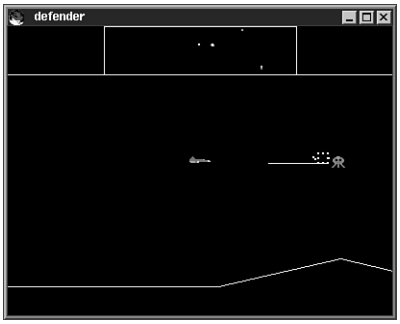
Gra posiada kilka elementów graficznych: ekran radaru, który pokazuje wszystkie jednostki w grze, główny ekran, w którym rozgrywa się większość wydarzeń, duszki dla każdej jednostki oraz kilka efektów specjalnych. Każda jednostka jest piksmapą, tworzoną z danych xpm umieszczonych w kodzie aplikacji. Gra jest podwójnie buforowana, dzięki czemu animacja jest nienaganna. W każdej klatce gry przerysowywany jest cały ekran. Najpierw czyścimy drugoplanową piksmapę, następnie rysujemy na niej aktualnie widoczne jednostki, wreszcie rysujemy radar, który pokazuje rozmieszczenie wszystkich obiektów. Kiedy rysowanie dobiegnie końca, cały rysunek jest kopiowany z drugoplanowej piksmapy do kontrolki obszaru rysunkowego.

Rysunki na głównym ekranie są piksmapami, natomiast radar jest tworzony z kolorowych punktów. Każdy kolor reprezentuje inną jednostkę, dzięki czemu po odrobinie treningu łatwo jest rozróżnić poszczególne obiekty. Radar pokazuje nawet pociski, wystrzeliwane przez obcych oraz eksplozje.

Sztuczna inteligencja (AI)

Każda jednostka w grze posiada pewne zadania. Sztuczna inteligencja jest dość prymitywna, ale może stanowić wyzwanie – zwłaszcza, jeśli dodamy więcej wrogich jednostek. Przegrana byłaby wówczas dość prawdopodobna, gdyby w programie nie wyłączono sprawdzania kolizji gracza.

Gra zawiera tylko dwie jednostki obdarzone sztuczną inteligencją, które dążą do osiągnięcia specyficznych celów. Lądowniki, które porywają ludzi, poruszają się poziomo, dopóki kogoś nie odszukają. Kiedy znajdą się bezpośrednio nad człowiekiem, wówczas opuszczają się w dół i próbują wynieść człowieka na górę ekranu, gdzie mogą się zamienić w mutantów (patrz rysunek 13.4). Mutanci polują na gracza-poruszają się w sposób do pewnego stopnia losowy, ale powoli zmierzają w stronę gracza.



Rysunek 13.4. Gracz strzela do mutantów – jeden został zabity, a drugi zginie za chwilę.

Sztuczna inteligencja mutanta powoduje jego ruch w osi x na jeden z czterech sposobów. Mutant może odsunąć się od gracza, pozostać w aktualnej pozycji, przysunąć się do gracza albo wykonać długi skok w stronę gracza. W połowie przypadków mutant będzie poruszał się w stronę gracza. Może czasem polecieć w przeciwnym kierunku, ponieważ jego ruch jest do pewnego stopnia chaotyczny, ale w dłuższym przedziale czasu będzie zbliżał się do gracza. Ruch w osi y jest podobny, ale zachodzi tylko wtedy, kiedy gracz jest w polu widzenia mutanta; w przeciwnym razie ruch ten jest przypadkowy. Ruchy jednostek są określane przez funkcję ModulAI. Każda wroga jednostka może oddać strzał do gracza, jeśli ten znajduje się w jej zasięgu. Mutanci mają naturalnie większą szansę oddania strzału.

Wewnętrzne struktury gry

Program musi przechowywać stan wszystkich jednostek oraz ich położenie w świecie gry. Najważniejszymi elementami są duszki oraz góry. Duszki poruszają się nieprzerwanie, a góry są nieruchome, chociaż również zdają się poruszać, kiedy gracz leci do przodu. Ich pozycja (względem gracza) zmienia się, więc góry rysowane na ekranie również zmieniają się w trakcie poziomego lotu gracza. Ponieważ świat jest „zawinięty", kod obliczający odległości i kierunki pomiędzy jednostkami jest dość skomplikowany. Aby uprościć program, wszystkie jednostki używają tej samej struktury danych, choć nie wszystkie wykorzystują każde jej pole.

Struktury danych

Dwoma podstawowymi strukturami są typDuszek i typJednostka. Struktura typDuszek zawiera piksmapę duszka, razem z maską, wysokością i szerokością. Struktury typJednostka zawierają informacje o wszystkich jednostkach w grze. Gracz, obcy, pociski, a nawet eksplozje są opisane przez strukturę typJednostka. Przechowuje ona typ jednostki (LADOWNIK, MUTANT, POCISK, WYBUCH itd.), współrzędne jednostki, prędkość, czas życia oraz prawdopodobieństwo oddania strzału do gracza. Prawdopodobnie winni jesteśmy kilka wyjaśnień, dotyczących czasu życia poszczególnych jednostek.

Czas życia większości jednostek jest nieskończony, o ile nie zostaną zestrzelone, ale niektóre obiekty istnieją tylko przez określony czas. Na przykład wystrzelone przez obcych pociski po chwili znikają, a więc musimy przypisać im czas życia, który ulega zmniejszeniu, kiedy pocisk się porusza. Kiedy czas życia osiągnie zero, pocisk zniknie z ekranu. Wybuchy przypominają pod tym względem pociski – tak naprawdę składają się z ośmiu małych duszków, rozrzucanych w kilku kierunkach i obdarzonych skończonym czasem życia. Wszystkie jednostki,

z wyjątkiem gracza, są przechowywane na łączonej liście (GList *). Lista ta zawiera więc obcych, pociski oraz wszystkie osiem fragmentów każdego wybuchu.

Góry

Góry są generowane po to, aby widać było ruch gracza, kiedy na ekranie nie ma żadnych obcych ani ludzi. Ich rola sprowadza się więc do punktu odniesienia, który pomaga zasymulować poruszanie się gracza. Generacja gór polega na utworzeniu losowo rozmieszczonych szczytów (x, y), które następnie używane są do stworzenia listy punktów, opisujących rozmieszczenie wzniesień i dolin.

Obliczenia

Ponieważ gra rozgrywa się w wirtualnym świecie, rozciągającym się od 0 do X_ZAKRES i z powrotem do 0, podczas obliczania odległości i kierunków konieczne jest przeprowadzenie pewnych dodatkowych kroków. Załóżmy, że X_ZAKRES wynosi 2000. Jeśli współrzędna x gracza jest równa 1999, a wrogiej jednostki 1, wówczas odległość pomiędzy nimi wynosi nie 1998, ale 2, ponieważ świat zawija się od 1999 z powrotem do 0.

Typy danych

Teraz, kiedy wyjaśniliśmy już, co powinien robić program, przyjrzyjmy się jego kodowi. Najpierw opiszemy typy danych i wykorzystywaną grafikę, a następnie zamieścimy kod implementujący grę.

defender.h

```
#include <stdlib.h>

/*
 * Kierunki ruchu gracza
 */
enum {
   RUCH_LEWO,
   RUCH_PRAWO,
   RUCH_GORA,
   RUCH_DOL
};

/*
 * Bohaterowie gry
 */
enum {
```

```
 GRACZ,              /* --- gracz --- */
 OSOBA,              /* --- ludzie na powierzchni --- */
 LADOWNIK,           /* --- obcy próbujący uprowadzić ludzi --- */
 MUTANT,             /* --- obcy, którzy wynieśli ludzi na górę
                          ekranu i ulegli mutacji --- */
 POCISK,             /* --- obcy strzelający do gracza --- */
 LASER,              /* --- gracz strzelający do obcych --- */
 WYBUCH              /* --- ktoś zrobił "bum" --- */
};

/*
 * Struktura danych używana przez wszystkie jednostki w grze
 */
typedef struct jednostka {

  int bZniszcz;                 /* --- Skazany na zagładę --- */
  int kierunek;                 /* --- Dokąd zmierza jednostka? --- */
  int typ;                      /* --- Typ jednostki --- */
  float pctStrzal;              /* --- Prawdopodobieństwo oddania strzału --- */
  float x;                      /* --- Położenie --- */
  float y;                      /* --- Położenie --- */
  float vx;                     /* --- Prędkość --- */
  float vy;                     /* --- Prędkość --- */
  int zycie;                    /* --- Pozostałe życia --- */
  struct jednostka *przylacz;   /* --- Przyłączone jednostki --- */

} typJednostka;

typedef struct {

   int x;
   int y;

} typPunkt;

typedef struct {

   typPunkt poczatek;
   typPunkt szczyt;
   typPunkt koniec;

} typGora;
```

```
/*
 * Duszki, przy pomocy których rysujemy jednostki na ekranie.
 */
typedef struct {

   char **dane_xpm;              /* --- Pierwotne dane xpm --- */
   GdkPixmap *piksmapa;          /* --- Piksmapa --- */
   GdkBitmap *maska;             /* --- Maska piksmapy --- */
   int wysok;                    /* --- Wysosokość duszka --- */
   int szerok;                   /* --- Szerokość duszka --- */

} typDuszek;
```

mutant.h

Plik mutant.h zawiera rysunki wszystkich jednostek występujących w grze.

```
/*
 * Rysunki wszystkich jednostek
 * występujących w Defenderze
 */

static char * xpm_pocisk[] = {
"2 2 2 1",
"    c None",
"X  c #ffffff",
"XX",
"XX",
};

static char * xpm_mutant[] = {
"16 12 4 1",
"    c None",
"b   c #8888ff",
"r   c #FF3366",
"G  c #00AA00",
"     bbbr     ",
"   rrbbbrrr   ",
"  GrrbbbrrrG  ",
" GGr rr rGG ",
" GGr rr rGG ",
```

```
"  GGrrrrrrrrGG  ",
"   GGrrrrrrGG   ",
"    GGGrrGGG    ",
"    GG rr GG    ",
"   GG  rr  GG   ",
"  GG   rr   GG  ",
" GG    rr    GG",
};

static char * xpm_ladownik[] = {
"16 12 4 1",
"     c None",
"y    c #ffaa00",
"g    c #88FF88",
"G  c #009900",
"      yyyy      ",
"    GGyyyyGG    ",
"   GGGGggGGGG   ",
"  GGG  gg  GGG  ",
"  GGG  gg  GGG  ",
"  GGGgggggggGGG  ",
"   GGGggggGGG   ",
"    GGGGGGGG    ",
"    GG GG GG    ",
"   GG  GG  GG   ",
"  GG   GG   GG  ",
" GG    GG    GG ",
};

static char * xpm_statek1 [] = {
"22 7 4 1",
"  c None",
"x c #777777",
"p c #ff66ff",
"o c #cccccc",
"   x              ",
"  xxx             ",
" xxxxx            ",
"xxxxxxxxxxxxxxpp      ",
"xxxxxxxxxxxxxxxxxxxxoo  ",
```

```
"ppppppxxooxxxxxxxxxooo",
" pppppxxoo         "
};

static char * xpm_statek2 [] = {
"22 7 4 1",
"  c None",
"x c #777777",
"p c #ff66ff",
"o c #cccccc",
"               x     ",
"              xxx   ",
"             xxxxx ",
"      ppxxxxxxxxxxxxx",
"  ooxxxxxxxxxxxxxxxxxx",
"oooxxxxxxxxxooxxppppppp",
"         ooxxppppp ",
};

static char * xpm_osoba[] = {
"6 10 4 1",
[357]
"                c None",
"y               c #ffaa00",
"p               c #CC00CC",
"P               c #FF44FF",
"  pp  ",
"  yP  ",
" yPPP ",
" PPPP ",
" PPPP ",
" PPPP ",
"  pp  ",
"  pp  ",
"  pp  ",
"  pp  ",
};
```

animacja.c

Kod w pliku animacja.c nie zawiera żadnych algorytmów używanych przez grę. Zajmuje się tworzeniem okien i przetwarzaniem zdarzeń. Kryjąca się za nim logika mogłaby być częścią dowolnej gry; akurat w tym przypadku mamy do czynienia z klonem Defendera.

```
/*
 * Autor: Eric Harlow
 * Plik: animacja.c
 *
 * Tworzenie aplikacji w Linuksie
 *
 * Gra "Defender"
 */

#include <gtk/gtk.h>
#include <gdk/gdkkeysyms.h>
#include "defender.h"

/* --- Drugoplanowa piksmapa dla obszaru rysunkowego  --- */
GdkPixmap *piksmapa = NULL;

/*
 * WyswietlDuszka
 *
 * Wyświetla duszka na obszarze rysunkowym w punkcie o
 * określonych współrzędnych. Używamy maski, aby nie
 * rysować niewidocznego obszaru - powinien pozostać
 * przezroczysty.
 *
 * obszar_rys - gdzie należy narysować duszka
 * duszek - duszek do narysowania
 * x, y - pozycja rysowania duszka
 */
void WyswietlDuszka (GtkWidget *obszar_rys, typDuszek *duszek, int x, int y)
{
   GdkGC *gc;

   /* --- Pobieramy kontekst gc zwykłego stylu --- */
   gc = obszar_rys->style->fg_gc[GTK_STATE_NORMAL];

   /* --- Ustawiamy maskę przycinania i punkt początkowy --- */
```

```
    gdk_gc_set_clip_mask (gc, duszek->maska);
    gdk_gc_set_clip_origin (gc, x, y);

    /* --- Kopiujemy piksmapę do drugoplanowej piksmapy --- */
    gdk_draw_pixmap (piksmapa,
        obszar_rys->style->fg_gc[GTK_STATE_NORMAL],
        duszek->piksmapa,
        0, 0, x, y,
        duszek->szerok, duszek->wysok);

    /* --- Zerujemy niepotrzebną już maskę przycinania. --- */
    gdk_gc_set_clip_mask (gc, NULL);
}

/*
 * Przerysuj
 *
 * dane - kontrolka do przerysowania
 */
gint Przerysuj (gpointer dane)
{
    GtkWidget*    obszar_rys = (GtkWidget *) dane;
    GdkRectangle  uakt_prostokat;

    /* --- Rysujemy okno gry na drugim planie --- */
    RysujEkran (piksmapa, obszar_rys);

    /* --- Kopiujemy drugoplanową piksmapę na ekran --- */
    uakt_prostokat.x = 0;
    uakt_prostokat.y = 0;
    uakt_prostokat.width = obszar_rys->allocation.width;
    uakt_prostokat.height = obszar_rys->allocation.height;

    gtk_widget_draw (obszar_rys, &uakt_prostokat);

    return (TRUE);
}

/*
 * configure_event
 *
 * Tworzy nową drugoplanową piksmapę o odpowiednich
```

```
 * rozmiarach. Wywoływana przy każdej zmianie rozmiarów
 * okna. Musimy zwolnić przydzielone zasoby.
 */
static gint configure_event (GtkWidget *kontrolka,
                             GdkEventConfigure *zdarzenie)
{
   /* --- Zwalniamy drugoplanową piksmapę,
          jeśli już ją utworzyliśmy --- */
   if (piksmapa) {
      gdk_pixmap_unref (piksmapa);
   }

   /* --- Tworzymy piksmapę o nowych rozmiarach --- */
   piksmapa = gdk_pixmap_new (kontrolka->window,
              kontrolka->allocation.width,
              kontrolka->allocation.height,
              -1);

   return TRUE;
}

/*
 * OtrzymanoOgnisko
 *
 * Wywoływana wtedy, kiedy okno gry otrzyma ognisko. Jest
 * potrzebna, ponieważ klawisze powtarzają się i blokują inne
 * klawisze wciśnięte w tym samym momencie. Jedynym sposobem
 * uniknięcia tego efektu jest wyłączenie powtarzania klawiszy
 * i samodzielna obsługa zdarzeń "wciśnięty" i "zwolniony".
 * Zmiana ta jesr globalna: dotyczy wszystkich aplikacji,
 * więc powinniśmy dokonywać jej tylko wtedy, kiedy okno
 * gry otrzyma ognisko.
 */
static gint OtrzymanoOgnisko (GtkWidget *kontrolka, gpointer dane)
{
   gdk_key_repeat_disable ();
   return (FALSE);
}

/*
 * UtraconoOgnisko
```

```
 *
 * Patrz OtrzymanoOgnisko.  Ponieważ zmiana jest globalna,
 * powinniśmy przywrócić powtarzanie klawiszy, aby inne
 * aplikacje działały poprawnie.
 */
static gint UtraconoOgnisko (GtkWidget *kontrolka, gpointer dane)
{
   gdk_key_repeat_restore ();
   return (FALSE);
}

/*
 * NacisniecieKlawisza
 *
 * Użytkownik nacisnął klawisz. Dodajemy go do listy aktualnie
 * wciśniętych klawiszy.
 */
static gint NacisniecieKlawisza (GtkWidget *kontrolka, GdkEventKey *zdarzenie)
{

   DodajKlawisz (zdarzenie);
   return (FALSE);
}

/*
 * ZwolnienieKlawisza
 *
 * Użytkownik zwolnił klawisz. Usuwamy go z listy aktualnie
 * wciśniętych klawiszy.
 */
static gint ZwolnienieKlawisza (GtkWidget *kontrolka, GdkEventKey *zdarzenie)
{
   UsunKlawisz (zdarzenie);
   return (FALSE);
}

/*
 * expose_event
 *
 * Wywoływana po odsłonięciu okna albo po wywołaniu
 * procedury gdk_widget_draw. Kopuje drugoplanową piksmapę
```

```
 * do okna.
 */
gint expose_event (GtkWidget *kontrolka, GdkEventExpose *zdarzenie)
{

   /* --- Kopiujemy piksmapę do okna --- */
   gdk_draw_pixmap (kontrolka->window,
        kontrolka->style->fg_gc[GTK_WIDGET_STATE (kontrolka)],
        piksmapa,
        zdarzenie->area.x, zdarzenie->area.y,
        zdarzenie->area.x, zdarzenie->area.y,
        zdarzenie->area.width, zdarzenie->area.height);

   return FALSE;
}

/*
 * PobierzPioro
 *
 * Zwraca pióro utworzone na podstawie przekazanego
 * koloru GdkColor. "Pióro" (po prostu GdkGC) jest
 * tworzone i zwracane, gotowe do użycia.
 */
GdkGC *PobierzPioro (GdkColor *c)
{
   GdkGC *gc;

   /* --- Tworzymy kontekst gc --- */
   gc = gdk_gc_new (piksmapa);

   /* --- Ustawiamy pierwszy plan na określony kolor --- */
   gdk_gc_set_foreground (gc, c);

   /* --- Zwracamy pióro --- */
   return (gc);
}

/*
 * NowyKolor
 *
 * Tworzymy i przydzielamy GdkColor na podstawie
 * określonych parametrów.
```

```
 */
GdkColor *NowyKolor (long czerwona, long zielona, long niebieska)
{
   /* --- Przydzielamy miejsce na kolor --- */
   GdkColor *c = (GdkColor *) g_malloc (sizeof (GdkColor));

   /* --- Wypełniamy składowe koloru --- */
   c->red = czerwona;
   c->green = zielona;
   c->blue = niebieska;

   gdk_color_alloc (gdk_colormap_get_system (), c);

   return (c);
}

/*
 * zamknij
 *
 * Kończymy aplikację
 */
void zamknij ()
{
   gtk_exit (0);
}

/*
 * main
 *
 * Program zaczyna się tutaj.
 */
int main (int argc, char *argv[])
{
   GtkWidget *okno;
   GtkWidget *obszar_rys;
   GtkWidget *xpole;

   ZacznijGre ();

   gtk_init (&argc, &argv);

   okno = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

```
xpole = gtk_hbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (okno), xpole);
gtk_widget_show (xpole);

gtk_signal_connect (GTK_OBJECT (okno), "destroy",
        GTK_SIGNAL_FUNC (zamknij), NULL);

/* --- Tworzymy obszar rysunkowy  --- */
obszar_rys = gtk_drawing_area_new ();
gtk_drawing_area_size (GTK_DRAWING_AREA (obszar_rys), 400,
                PobierzWysokoscOkna ());
gtk_box_pack_start (GTK_BOX (xpole), obszar_rys, TRUE, TRUE, 0);

gtk_widget_show (obszar_rys);

gtk_widget_set_events (okno, GDK_KEY_RELEASE_MASK);

/* --- Sygnały używane do obsługi drugoplanowej piksmapy --- */
gtk_signal_connect (GTK_OBJECT (obszar_rys), "expose_event",
        (GtkSignalFunc) expose_event, NULL);
gtk_signal_connect (GTK_OBJECT (obszar_rys), "configure_event",
        (GtkSignalFunc) configure_event, NULL);

/* --- Sygnały do obsługi ogniska --- */
gtk_signal_connect (GTK_OBJECT (okno), "focus_in_event",
        (GtkSignalFunc) OtrzymanoOgnisko, NULL);
gtk_signal_connect (GTK_OBJECT (okno), "focus_out_event",
        (GtkSignalFunc) UtraconoOgnisko, NULL);

/* --- Sygnały naciśnięcia klawisza --- */
gtk_signal_connect (GTK_OBJECT (okno), "key_press_event",
        (GtkSignalFunc) NacisniecieKlawisza, NULL);
gtk_signal_connect (GTK_OBJECT (okno), "key_release_event",
        (GtkSignalFunc) ZwolnienieKlawisza, NULL);

/* --- Pokazujemy okno --- */
gtk_widget_show (okno);

/* --- Przerysowujemy co 1/20 sekundy --- */
gtk_timeout_add (50, Przerysuj, obszar_rys);

LadujRysunki (okno);
```

```
    /* --- Wywołujemy główną pętlę gtk --- */
    gtk_main ();

    return 0;
}

/*
 * PobierzSzerIWys
 *
 * Pobiera wysokość i szerokość piksmapy z danych xpm.
 */
void PobierzSzerIWys (gchar **xpm, int *szerok, int *wysok)
{

     sscanf (xpm [0], "%d %d", szerok, wysok);
}
```

klawisze.c

Plik klawisze.c zajmuje się obsługą wciśniętych klawiszy i wywołuje odpowiednie funkcje poruszające statek i oddające strzały, w zależności od naciśniętego klawisza.

```
/*
 * Autor: Eric Harlow
 * Plik: klawisze.c
 * Tworzenie aplikacji GUI w Linuksie
 */

#include <gtk/gtk.h>
#include <gdk/gdkkeysyms.h>
#include "defender.h"
#include "prototypy.h"

/*
 * Ruch
 */
int klawiszLewo = 0;
int klawiszPrawo = 0;
int klawiszGora = 0;
int klawiszDol = 0;

/*
```

```
 * Strzelanie
 */
int klawiszStrzal = 0;

/*
 * DodajKlawisz
 *
 * Dodajemy klawisz do listy klawiszy, których
 * naciskanie będziemy sprawdzać. Sprawdzamy tylko
 * klika klawiszy, ignorując całą resztę.
 */
void DodajKlawisz (GdkEventKey *zdarzenie)
{

   switch (zdarzenie->keyval) {

      /* --- Strzałka w lewo --- */
      case GDK_Left:
         klawiszLewo = TRUE;
         break;

      /* --- Strzałka w prawo --- */
      case GDK_Right:
         klawiszPrawo = TRUE;
         break;

      /* --- Strzałka w górę --- */
      case GDK_Up:
         klawiszGora = TRUE;
         break;

      /* --- Strzałka w dół --- */
      case GDK_Down:
         klawiszDol = TRUE;
         break;

      /* --- Spacja --- */
      case ' ':
         klawiszStrzal = TRUE;
         break;

      /* --- Ignorujemy resztę --- */
```

```
        default:
            break;
    }
}

/*
 * UsunKlawisz
 *
 * Jeśli sprawdzany klawisz zostanie zwolniony,
 * zerujemy znacznik, który wskazuje, że jest
 * naciśnięty.
 */
void UsunKlawisz(GdkEventKey *zdarzenie)
{
    switch (zdarzenie->keyval) {

        case GDK_Left:
            klawiszLewo = FALSE;
            break;

        case GDK_Right:
            klawiszPrawo = FALSE;
            break;

        case GDK_Up:
            klawiszGora = FALSE;
            break;

        case GDK_Down:
            klawiszDol = FALSE;
            break;

        case ' ':
            klawiszStrzal = FALSE;
            break;

        default:
            break;
    }
}

/*
```

```
 * ObsluzWcisnieteKlawisze
 *
 * Kiedy należy przesunąć wszystkie jednostki,
 * procedura ta jest wywoływana, aby poruszyć
 * gracza/oddać strzał w zależności od
 * aktualnie wciśniętych klawiszy.
 */
void ObsluzWcisnieteKlawisze ()
{

   /*
    * Przesuwamy statek w różnych kierunkach
    */
   if (klawiszLewo) {
      RuchGracza (RUCH_LEWO);
   }

   if (klawiszPrawo) {
      RuchGracza (RUCH_PRAWO);
   }
   if (klawiszGora) {
      RuchGracza (RUCH_GORA);
   }

   if (klawiszDol) {
      RuchGracza (RUCH_DOL);
   }

   /*
    * Strzelamy
    *
    * Po oddaniu strzału zerujemy wskaźnik wciśnięcia
    * klawisza, aby gracz musiał naciskać spację po
    * każdym strzale. W przeciwnym przypadku mógłby
    * po prostu przytrzymać klawisz.
    */
   if (klawiszStrzal) {

      /* --- Strzał! --- */
      StrzalGracza ();

      /* --- Jedno wciśnięcie klawisza = jeden strzał  --- */
```

```
            klawiszStrzal = FALSE;
        }
    }
```

defender.c

Plik ten jest głównym modułem, zawierającym logikę gry i funkcje rysujące. Nie zajmuje się interfejsem użytkownika, tylko kreśleniem obszaru rysunkowego, który tworzy pole gry.

```
/*
 * Plik: defender.c
 * Autor: Eric Harlow
 *
 * przypomina grę Defender z lat osiemdziesiątych
 * (choć nie jest to pełna wersja)
 *
 */

#include <gtk/gtk.h>
#include "defender.h"
#include "prototypy.h"
#include "mutant.h"
#include <math.h>

/*
 * Ilu ma być ludzi i obcych na początku gry?
 */
#define POCZ_OBCY 10
#define POCZ_LUDZIE 10

/*
 * Poniżej znajdują się stałe używane przy obliczaniu
 * przyspieszenia gracza. Rezultatem tarcia jest
 * zwolnienie ruchu. MAKS_PRED oznacza maksymalną
 * prędkość, jaką może osiągnąć gracz.
 */
#define TARCIE .5
#define MAKS_PREDK 16
#define PRZYSPIESZENIE 3.5
```

```
/*
 * Strzały z lasera również można konfigurować. Promień
 * lasera ma prędkość (szybkość przemieszczania się) oraz
 * długość (efekt głównie wizualny).
 */
#define DLUG_LASERA 60
#define PREDK_LASERA 60

/*
 * Zakres pola gry
 */
#define ZAKRES_X 2000

/*
 * Jak wysokie są szczyty gór?
 */
#define MAKS_SZCZYT 150

/*
 * Liczba generowanych szczytów górskich
 */
#define LICZBA_SZCZYTOW 10

/*
 * --- Agresywność
 *
 * Mutanty są dużo agresywniejsze od lądowników.
 */
#define LADOWNIK_AGRESJA 3
#define MUTANT_AGRESJA 6

/*
 * W trakcie eksplozji korzystamy z ośmiu duszków,
 * rozchodzących się gwiaździście z miejsca wybuchu.
 * Poniżej znajduje się tych osiem kierunków.
 */
int x_wyb[] = {1, 1, 0, -1, -1, -1, 0, 1};
int y_wyb[] = {0, 1, 1, 1, 0, -1, -1, -1};

/*
 * Jak duży jest ekran radaru?
 */
```

```
#define SZEROK_RADARU 200
#define WYSOK_RADARU 50

/*
 * Rozmiary pozostałych elementów ekranu
 */
#define WYSOK_OKNA 220
#define WYSOK_SPODU 30
#define WYSOK_OSOBY (WYSOK_RADARU+WYSOK_OKNA+7)

/*
 * Kolory używane do rysowanie
 */
GdkGC *pioroZielone = NULL;
GdkGC *pioroBiale = NULL;
GdkGC *pioroFioletowe = NULL;
GdkGC *pioroCzerwone = NULL;

/*
 * Zmienne używane podczas obliczeń
 */
int nRegulacjaStatku;
int nRegulacjaWzgledna;
int nSzerokOkna;

/*
 * Definiujemy duszki bohaterów gry
 */
typDuszek duszek_osoba[1]   = { { xpm_osoba,   NULL, NULL, 0, 0 } };
typDuszek duszek_statek1[1]  = { { xpm_statek1,  NULL, NULL, 0, 0 } };
typDuszek duszek_statek2[1]  = { { xpm_statek2,  NULL, NULL, 0, 0 } };
typDuszek duszek_ladownik[1] = { { xpm_ladownik, NULL, NULL, 0, 0 } };
typDuszek duszek_mutant[1]  = { { xpm_mutant,  NULL, NULL, 0, 0 } };
typDuszek duszek_pocisk[1]  = { { xpm_pocisk,   NULL, NULL, 0, 0 } };

/*
 * Statek może być skierowany w lewo lub w prawo (duszek_statek1,
 * duszek_statek2) - ale do rysowania wykorzystujemy wskaźnik
 * duszek_statek. Jeśli statek zmieni kierunek lotu, należy
 * odpowiednio zmodyfikować wskaźnik. */
typDuszek *duszek_statek = duszek_statek1;
```

```
/*
 * Lista szczytów górskich.
 */
GList *listaTerenu = NULL;

/*
 * Wszystkie jednostki w grze.
 */
GList *listaJednostek = NULL;

/*
 * Gracz
 */
typJednostka *gracz = NULL;

/*
 * Prototypy
 */
GdkColor *NowyKolor (long czerwona, long zielona, long niebieska);
GdkGC *PobierzPioro (GdkColor *c);
void PobierzSzeriWys (gchar **xpm, int *szerok, int *wysok);
void WyswietlDuszka (GtkWidget *obszar_rys, typDuszek *duszek, int x, int y);

/*
 * JednostkaX
 *
 * Przekształca współrzędną X jednostki na współrzędną
 * ekranową, liczoną względem położenia gracza.
 */
int JednostkaX (typJednostka *jednostka)
{
   int xPoz;

   /* --- Regulujemy -x- jeśli przekroczy zakres --- */

   if (jednostka->x < 0) {
      jednostka->x += ZAKRES_X;
   } else if (jednostka->x > ZAKRES_X) {
      jednostka->x -= ZAKRES_X;
   }

   /* --- Relatywizujemy współrzędną --- */
```

```
    xPoz = (int) (jednostka->x - nRegulacjaWzgledna);

    /* --- Ponownie regulujemy -x- jeśli przekroczy
           zakres --- */
    if (xPoz < 0) xPoz += ZAKRES_X;
    if (xPoz > ZAKRES_X) xPoz -= ZAKRES_X;

    return (xPoz);
}

/*
 * EkranX
 *
 * Pobieramy współrzędną -x- w grze i przekształcamy ją
 * we współrzędną -x- na ekranie.
 */
int EkranX (int x)
{
    int xPoz;

    /* --- Regulujemy -x- jeśli przekroczy zakres --- */
    if (x < 0) {

       x += ZAKRES_X;

    } else if (x > ZAKRES_X) {

       x -= ZAKRES_X;
    }

    /* --- Współrzędna -x- jest absolutna --- */
    xPoz = (int) (x - nRegulacjaWzgledna);

    /* --- Ponownie regulujemy -x- jeśli
           przekroczy zakres --- */
    if (xPoz < (-(ZAKRES_X - nSzerokOkna) / 2)) {

       xPoz += ZAKRES_X;
    } else if (xPoz > ((ZAKRES_X - nSzerokOkna) / 2)) {

       xPoz -= ZAKRES_X;
    }

    return (xPoz);
```

```
}

/*
 * GraX
 *
 * Pobieramy współrzędną -x- na ekranie i
 * przekształcamy ją na współrzędną -x- w grze.
 */
int GraX (int x)
{
   int xPoz;

   /* --- Współrzędna względem gracza --- */
   xPoz = (int) (x + nRegulacjaWzgledna);

   /* --- Upewniamy się, że nie przekroczyliśmy
          zakresu --- */
   if (xPoz < 0) xPoz += ZAKRES_X;
   if (xPoz > ZAKRES_X) xPoz -= ZAKRES_X;

   return (xPoz);
}

/*
 * Ruch
 *
 * Przesuwa jednostkę w kierunku x o vx (prędkość w osi x) i
 * w kierunku y o vy.
 */
void Ruch (typJednostka *jednostka)
{

   /* --- Przesuwamy jednostkę --- */
   jednostka->y += jednostka->vy;
   jednostka->x += jednostka->vx;

   /* --- ...ale utrzymujemy ją w granicach pola gry --- */
   if (jednostka->x < 0) jednostka->x += ZAKRES_X;
   if (jednostka->x > ZAKRES_X) jednostka->x -= ZAKRES_X;
}
/*
 * LadujPiksmapy
```

```
 *
 * Ładuje rysunki do duszków.
 */
void LadujPiksmapy (GtkWidget *kontrolka, typDuszek *duszki)
{

   /* --- Tworzymy piksmapę z danych xpm --- */
   duszki->piksmapa = gdk_pixmap_create_from_xpm_d (
                        kontrolka->window,
                        &duszki->maska,
                        NULL,
                        duszki->dane_xpm);

   /* --- Pobieramy szerokość i wysokość --- */
   PobierzSzerIWys (duszki->dane_xpm,
                &duszki->szerok,
                &duszki->wysok);

}

/*
 * LadujRysunki
 *
 * Ładuje rysunki, aby umożliwić wyświetlenie ich w oknie
 * gry i skonfigurowanie kolorów.
 */
void LadujRysunki (GtkWidget *okno)
{
   /* --- Ładujemy rysunki --- */
   LadujPiksmapy (okno, duszek_osoba);
   LadujPiksmapy (okno, duszek_statek1);
   LadujPiksmapy (okno, duszek_statek2);
   LadujPiksmapy (okno, duszek_ladownik);
   LadujPiksmapy (okno, duszek_mutant);
   LadujPiksmapy (okno, duszek_pocisk);

   /* --- Pobieramy zdefiniowane kolory --- */
   pioroCzerwone = PobierzPioro (NowyKolor (0xffff, 0x8888, 0x8888));
   pioroZielone = PobierzPioro (NowyKolor (0, 0xffff, 0));
   pioroFioletowe = PobierzPioro (NowyKolor (0xffff, 0, 0xffff));
   pioroBiale = PobierzPioro (NowyKolor (0xffff, 0xffff, 0xffff));
```

```
}

/*
 * UtworzGracza
 *
 * Tworzymy strukturę typJednostka dla gracza i
 * inicjujemy jej wartości.
 */
typJednostka *UtworzGracza ()
{
   /* --- Przydzielamy pamięć --- */
   gracz = g_malloc (sizeof (typJednostka));

   /* --- Inicjujemy dane gracza --- */
   gracz->bZniszcz = FALSE;
   gracz->kierunek = 1;
   gracz->typ = GRACZ;
   gracz->x = 0;
   gracz->y = 150;
   gracz->vx = 0;
   gracz->vy = 0;
   gracz->przylacz = NULL;

   /* --- Zwracamy obiekt --- */
   return (gracz);
}

/*
 * StrzalGracza
 *
 * Gracz otworzył ogień! Tworzymy strzał laserowy
 * i dodajemy go do globalnej listy jednostek.
 */
void StrzalGracza ()
{
   typJednostka *laser;

   /* --- Tworzymy laser --- */
   laser = (typJednostka *) g_malloc (sizeof (typJednostka));

   /* --- Kierunek jest taki sam, jak kierunek ruchu statku --- */
   laser->kierunek = gracz->kierunek;
```

```
    laser->typ = LASER;

    /*
     * Umieszczamy początkową pozycję lasera przed dziobem
     * statku.
     */
    if (laser->kierunek > 0) {
        laser->x = gracz->x + (duszek_statek->szerok / 2);
    } else {
        laser->x = gracz->x - (duszek_statek->szerok / 2);
    }
    laser->y = gracz->y + 4;

    laser->vx = PREDK_LASERA * gracz->kierunek;
    laser->vy = 0;
    laser->przylacz = NULL;
    laser->bZniszcz = 0;

    /* --- Promień laser istnieje przez dwa ruchy --- */
    laser->zycie = 2;

    /* --- Dodajemy laser do listy jednostek --- */
    listaJednostek = g_list_append (listaJednostek, laser);
}

/*
 * RuchGracza
 *
 * Przesuwa gracza w określonym kierunku.
 */
void RuchGracza (int kierunek)
{
    switch (kierunek) {

        case RUCH_GORA:
            gracz->y -= 3;
            break;

        case RUCH_DOL:
            gracz->y += 3;
            break;
```

```
        case RUCH_LEWO:
          /*
           * Upewniamy się, że dziób statku jest
           * zwrócony we właściwym kierunku.
           */
          duszek_statek = duszek_statek2;
          gracz->kierunek = -1;

          /* --- Przyspieszamy statek --- */
          gracz->vx -= PRZYSPIESZENIE;
          break;

        case RUCH_PRAWO:
          /*
           * Upewniamy się, że dziób statku jest
           * zwrócony we właściwym kierunku.
           */
          duszek_statek = duszek_statek1;
          gracz->kierunek = 1;

          /* --- Przyspieszamy statek --- */
          gracz->vx += PRZYSPIESZENIE;
          break;
    }
}

/*
 * DodajTarcie
 *
 * Zwalniamy stopniowo statek i upewniamy się, że
 * gracz nie przekroczy prędkości światła, trzymając
 * wciśnięty klawisz.
 */
void DodajTarcie ()
{
    /* --- Zwalniamy statek --- */
    if (gracz->vx > TARCIE) {
        gracz->vx -= TARCIE;
    } else if (gracz->vx < -TARCIE) {
        gracz->vx += TARCIE;
    } else {
```

```
        /* --- Prędkość mniejsza od tarcia,
               zatrzymujemy statek --- */
        gracz->vx = 0;
    }

    /* --- Nie pozwalamy na przekroczenie maksymalnej
           prędkości  --- */
    if (gracz->vx > MAKS_PREDK) gracz->vx = MAKS_PREDK;
    if (gracz->vx < -MAKS_PREDK) gracz->vx = -MAKS_PREDK;
}

/*
 * UtworzOsobe
 *
 * Tworzymy ludzi na planecie.
 */
typJednostka *UtworzOsobe ()
{
    typJednostka *osoba;

    /* --- Przydzielamy pamięć --- */
    osoba = g_malloc (sizeof (typJednostka));

    /* --- Inicjujemy osobę --- */
    osoba->bZniszcz = FALSE;
    osoba->kierunek = 0;
    osoba->typ = OSOBA;
    osoba->x = rand () % ZAKRES_X;
    osoba->y = WYSOK_OSOBY;
    osoba->vx = 0;
    osoba->vy = 0;
    osoba->przylacz = NULL;

    return (osoba);
}

/*
 * RozmiescLudzi
 *
 * Tworzy osoby i rozmieszcza je losowo na ekranie
 */
void RozmiescLudzi ()
```

```
{
   int i;
   typJednostka *osoba;

   /* --- Tworzymy wszystkich ludzi --- */
   for (i = 0; i < POCZ_LUDZIE; i++) {

      /* --- Tworzymy osobę --- */
      osoba = UtworzOsobe ();

      /* --- Dodajemy ją do listy jednostek --- */
      listaJednostek = g_list_append (listaJednostek, osoba);
   }
}

/*
 * UtworzObcego
 *
 * Tworzy obcy lądownik
 */
typJednostka *UtworzObcego ()
{
   typJednostka *obcy;

   /* --- Przydzielamy pamięć --- */
   obcy = g_malloc (sizeof (typJednostka));

   /* --- Inicjujemy strukturę --- */
   obcy->bZniszcz = FALSE;
   obcy->pctStrzal = LADOWNIK_AGRESJA;
   obcy->typ = LADOWNIK;
   obcy->x = rand () % ZAKRES_X;
   obcy->y = rand () % 50 + WYSOK_RADARU;
   obcy->vx = 0;
   obcy->vy = 0;
   obcy->przylacz = NULL;

   return (obcy);
}

/*
 * UtworzPocisk
```

```
 *
 * Obcy strzelają pociskami. Pociski posiadają stały
 * kierunek ruchu i znikają po upływie pewnego czasu.
 */
typJednostka *UtworzPocisk (typJednostka *obcy, typJednostka *gracz)
{
   float fdlug;
   typJednostka *pocisk;

   /* --- Przydzielamy pamięć --- */
   pocisk = (typJednostka *) g_malloc (sizeof (typJednostka));

   /* --- Inicjujemy strukturę --- */
   pocisk->bZniszcz = FALSE;
   pocisk->pctStrzal = 0;
   pocisk->typ = POCISK;
   pocisk->x = obcy->x;
   pocisk->y = obcy->y;

   /* --- Obliczamy prędkość pocisku --- */
   pocisk->vx = (float) OdleglosciPomiedzy (pocisk, gracz) *
                        Kierunek (pocisk, gracz);
   pocisk->vy = (float) (gracz->y - obcy->y);

   /*
    * Regulujemy prędkość pocisku
    */
   fdlug = sqrt (pocisk->vx * pocisk->vx + pocisk->vy * pocisk->vy);
   if (fdlug < .1) fdlug = .1;
   fdlug /= 3;
   pocisk->vx /= fdlug;
   pocisk->vy /= fdlug;

   pocisk->przylacz = NULL;

   /*
    * --- Pocisk posiada czas życia. Kiedy czas życia spadnie
    *     do zera, pocisk znika.
    */
   pocisk->zycie = 60;

   return (pocisk);
```

```
}

/*
 * DodajWybuch
 *
 * Zniszczono jednostkę - tworzymy wybuch w miejscu, gdzie
 * znajdowała się jednostka.
 */
void DodajWybuch (typJednostka *jednostka)
{
   typJednostka *frag;
   int i;

   /* --- Tworzymy osiem fragmentów --- */
   for (i = 0; i < 8; i++) {

      /* --- Tworzymy fragment wybuchu --- */
      frag = (typJednostka *) g_malloc (sizeof (typJednostka));

      /* --- Inicjujemy fragment --- */
      frag->bZniszcz = FALSE;
      frag->pctStrzal = 0;
      frag->typ = WYBUCH;
      frag->x = jednostka->x;
      frag->y = jednostka->y;

      /* --- Nadajemy mu sporą prędkość... --- */
      frag->vx = x_wyb[i] * 5;
      frag->vy = y_wyb[i] * 5;

      frag->przylacz = NULL;

      /* --- ...i krótki czas życia --- */
      frag->zycie = 20;

      /* --- Dodajemy go do listy jednostek --- */
      listaJednostek = g_list_append (listaJednostek, frag);
   }
}

/*
 * RozmiescObcych
 *
```

```
 * Tworzymy obcych
 */
void RozmiescObcych ()
{
   int i;
   typJednostka *obcy;

   /* --- Tworzymy wszystkich obcych --- */
   for (i = 0; i < POCZ_OBCY; i++) {

      /* --- Tworzymy jednego obcego --- */
      obcy = UtworzObcego ();

      /* --- Dodajemy go do listy jednostek --- */
      listaJednostek = g_list_append (listaJednostek, obcy);
   }
}

/*
 * OdlegloscPomiedzy
 *
 * Oblicza odległość pomiędzy dwoma jednostkami, korzystając
 * tylko ze współrzędnych x. Odległość nie jest po prostu
 * różnicą pomiędzy współrzędnymi x jednostek - ponieważ
 * świat jest zawinięty, musimy uwzględnić fakt, że
 * odległość pomiędzy 1 i ZAKRES_X to nie (ZAKRES_X-1) - 1.
 */
int OdlegloscPomiedzy (typJednostka *u1, typJednostka *u2)
{
   int nOdleglosc;
   int nOdleglosc2;

   /* --- Która współrzędna jest większa? --- */
   if (u1->x < u2->x) {

      /* --- Obliczamy odległość w obie strony --- */
      nOdleglosc = u2->x - u1->x;
      nOdleglosc2 = ZAKRES_X + u1->x - u2->x;

   } else {
      /* --- Obliczamy odległość w obie strony --- */
      nOdleglosc = u1->x - u2->x;
```

```
      nOdleglosc2 = ZAKRES_X + u2->x - u1->x;
   }

   /* --- Wybieramy mniejszą odległość --- */
   return ((nOdleglosc < nOdleglosc2) ? nOdleglosc : nOdleglosc2);
}

/*
 * Kierunek
 *
 * Jaki jest kierunek -x-, w którym powinna poruszać się
 * jednostka, aby zbliżyć się do innej jednostki?
 * Może to być -1, 1 albo 0.
 */
int Kierunek (typJednostka *u1, typJednostka *u2)
{
   int nOdleglosc;
   int nOdleglosc2;

   if (u1->x < u2->x) {

      /* --- Odległość w obu kierunkach --- */
      nOdleglosc = u2->x - u1->x;
      nOdleglosc2 = ZAKRES_X + u1->x - u2->x;

   } else {
      /* --- Odległość w obu kierunkach --- */
      nOdleglosc2 = u1->x - u2->x;
      nOdleglosc = ZAKRES_X + u2->x - u1->x;
   }

   /* --- Kierunek zależy od mniejszej odległości --- */
   return ((nOdleglosc < nOdleglosc2) ? 1 : -1);
}

/*
 * ProbaStrzalu
 *
 * Obcy strzela do gracza.
 */
void ProbaStrzalu (typJednostka *jednostka)
{
```

```
    typJednostka *pocisk;

    /* --- Czy obcy jest dostatecznie blisko, aby oddać strzał? --- */
    if (OdlegloscPomiedzy (gracz, jednostka) < (nSzerokOkna / 2)) {

      /* --- Losowe prawdopodobieństwo, określone przez
           odpowiedni parametr jednostki --- */
      if ((rand () % 100) < jednostka->pctStrzal) {

        /* --- Tworzymy pocisk zmierzający w stronę gracza --- */
        pocisk = UtworzPocisk (jednostka, gracz);

        /* --- Dodajemy go do listy jednostek --- */
        listaJednostek = g_list_append (listaJednostek, pocisk);
      }
    }
}

/*
 * ModulAI
 *
 * Zawiera algorytmy ruchu wszystkich jednostek. Niektóre
 * jednostki (lądowniki) szukają ludzi, żeby unieść ich
 * w górę. Mutanci polują na gracza. Pociski po prostu
 * lecą, dopóki nie upłynie czas ich życia itd.
 */
void ModulAI (typJednostka *jednostka)
{
    typJednostka *tmcz;
    int najlepszaOdl = 50000;
    typJednostka *najblizsza = NULL;
    GList *wezel;

    /*
     * Algorytm AI obcego lądownika
     */
    if (jednostka->typ == LADOWNIK) {

      /* --- Jeśli obcy porwał człowieka... --- */
      if (jednostka->przylacz) {

        /* --- Przesuwa się do góry --- */
```

```
    najblizsza = jednostka->przylacz;
    jednostka->y -= .5;
    najblizsza->y -= .5;

    /* --- Jeśli dotarł na górę ekranu --- */
    if (jednostka->y - (duszek_ladownik[0].wysok / 2) <
       WYSOK_RADARU) {

       /* --- Stapia się z człowiekiem --- */
       jednostka->y = WYSOK_RADARU +
                  (duszek_ladownik[0].wysok / 2);
       jednostka->typ = MUTANT;
       jednostka->pctStrzal = MUTANT_AGRESJA;
       jednostka->przylacz = NULL;
       najblizsza->bZniszcz = TRUE;
    }

    return;
}

/* --- Czy w pobliżu są ludzie do uprowadzenia? --- */
for (wezel = listaJednostek; wezel; wezel = wezel->next) {

    tmcz = (typJednostka *) wezel->data;
    if (tmcz->typ == OSOBA && tmcz->przylacz == NULL) {

       /* --- Szukamy najbliższej osoby --- */
       if (OdlegloscPomiedzy (jednostka, tmcz) <
         najlepszaOdl) {
          najblizsza = tmcz;
          najlepszaOdl = OdlegloscPomiedzy (jednostka,
                                  tmcz);
       }
    }
}

/* --- Znaleźliśmy cel --- */
if (najlepszaOdl <= 1) {

    /* --- Ustawiamy się nad człowiekiem --- */
    jednostka->vx = 0;
    jednostka->x = najblizsza->x;
```

```
      /*
       * --- Sprawdzamy, czy można się połączyć ---
       */

      if ((jednostka->y + (duszek_ladownik[0].wysok / 2) + .8) <
         (najblizsza->y - (duszek_osoba[0].wysok / 2))) {

         /* --- Obniżamy się. --- */
         jednostka->y += .5;

      } else if ((jednostka->y + (duszek_ladownik[0].wysok / 2))
            > (najblizsza->y - (duszek_osoba[0].wysok / 2))) {

         jednostka->y -= .5;
      } else {

         /* --- Porywamy człowieka --- */
         jednostka->przylacz = najblizsza;
         najblizsza->przylacz = jednostka;
         najblizsza->zycie = 20;
      }

   /* --- Czy jest jakiś człowiek w rozsądnej odległości? --- */
   } else if (najlepszaOdl < 20) {

      /* --- Lecimy w jego kierunku --- */
      jednostka->vx = Kierunek (jednostka, najblizsza);
      jednostka->x += jednostka->vx;
   } else {

      /*
       * --- Nie ma ludzi w pobliżu. Poruszamy się losowo.
       */
      if (jednostka->vx == 0) {
         if ((rand () % 2) == 0) {
            jednostka->vx = 1;
         } else {
            jednostka->vx = -1;
         }
      }
      jednostka->x += jednostka->vx;
   }
```

```
    /*
     * Sprawdzamy, czy warto oddać strzał.
     */
    ProbaStrzalu (jednostka);

/*
 * Algorytm AI mutanta
 */
} else if (jednostka->typ == MUTANT) {

    /*
     * --- Mutanci poruszają się losowo, ale zmierzają
     *     z wolna w kierunku gracza.
     */
    jednostka->vx = Kierunek (jednostka, gracz) *
                        ((rand () % 4) + 1);
    jednostka->vy = rand () % 5 - 2;

    /*
     * Jeśli gracz jest w zasięgu, zmierzamy w jego stronę
     * w osi -y-.
     */
    if (OdleglosciPomiedzy (jednostka, gracz) < 200) {
       if (jednostka->y < gracz->y) jednostka->vy++;
       if (jednostka->y > gracz->y) jednostka->vy--;
    }

    /* --- Przesuwamy jednostkę --- */
    Ruch (jednostka);

    /* --- Próbujemy strzelić --- */
    ProbaStrzalu (jednostka);

/*
 * --- Pociski i wybuchy
 */
} else if ((jednostka->typ == POCISK) ||
           (jednostka->typ == WYBUCH)) {

    /* --- Jednostki te mają określony czas życia
           Zmniejszamy go. --- */
    jednostka->zycie --;
```

```
        /* --- Przesuwamy jednostkę. --- */
        Ruch (jednostka);

        /* --- Kiedy upłynie czas życia, niszczymy je --- */
        if (jednostka->zycie <= 0) {
           jednostka->bZniszcz = TRUE;
        }

     /*
      * Alogorytm AI osoby.
      */
     } else if (jednostka->typ == OSOBA) {

        /*
         * Osoba porusza się samodzielnie tylko wtedy, kiedy
         * spada na ziemię, ponieważ został zestrzelony
         * niosący ją obcy.
         */
        if (jednostka->przylacz==NULL && jednostka->y < WYSOK_OSOBY) {

           /* --- Przesuwamy osobę w dół --- */
           jednostka->y += 2;
        }
     }
  }

  /*
   * PobierzDuszka
   *
   * Zwraca duszka danej jednostki
   */
  typDuszek *PobierzDuszka (typJednostka *jednostka)
  {
     typDuszek *duszek;

     /* --- Pobieramy duszka --- */
     switch (jednostka->typ) {

        case GRACZ:
           duszek = duszek_statek;
           break;
```

```
        case OSOBA:
            duszek = duszek_osoba;
            break;

        case LADOWNIK:
            duszek = duszek_ladownik;
            break;

        case MUTANT:
            duszek = duszek_mutant;
            break;

        case POCISK:
        case WYBUCH:
            duszek = duszek_pocisk;
            break;

        default:
            duszek = NULL;
            break;
    }
    return (duszek);
}

/*
 * KtosPomiedzy
 *
 * Czy jakaś jednostka znajduje się pomiędzy tymi
 * współrzędnymi? Funkcja ta używana jest podczas
 * obliczania, czy ktoś został trafiony.
 */
typJednostka *KtosPomiedzy (int x1, int y1, int x2, int y2)
{
    GList *wezel;
    typJednostka *jednostka;
    typJednostka *najblizszaJednostka = NULL;
    int najblizszaX;
    typDuszek *duszek;
    int ekranX;

    /* --- Sprawdzamy wszystkie jednostki --- */
    for (wezel = listaJednostek; wezel; wezel = wezel->next) {
```

```
        jednostka = (typJednostka *) wezel->data;

        /* --- Jeśli jest to czarny charakter --- */
        if ((jednostka->typ == LADOWNIK) ||
            (jednostka->typ == OSOBA) ||
            (jednostka->typ == MUTANT)) {

            /* --- Pobieramy duszka i położenie na ekranie --- */
            ekranX = JednostkaX (jednostka);
            duszek = PobierzDuszka (jednostka);

            /* --- Jeśli mamy duszka --- */
            if (duszek) {

                /* --- Jeśli jest w określonym zakresie -x- --- */
                if ((ekranX >= x1 && ekranX <= x2) ||
                    (ekranX <= x1 && ekranX >= x2)) {

                    /* --- i w określonym zakresie -y- --- */
                    if ((jednostka->y - (duszek->wysok / 2) < y1) &&
                        jednostka->y + (duszek->wysok / 2) > y1) {

                        /*
                         * Nie znaleźliśmy jednostki, albo ta jest
                         * bliżej.
                         */
                        if ((najblizszaJednostka == NULL) ||
                            (abs (x1 - ekranX) <
                                    abs (x1 - najblizszaX))) {

                            /* --- Ta jednostka jest najbliższa
                                   spośród sprawdzonych do tej
                                   pory --- */
                            najblizszaJednostka = jednostka;
                            najblizszaX = ekranX;
                        }
                    }
                }
            }
        }
    }
```

```
    return (najblizszaJednostka);
}

/*
 * JednostkaGora
 *
 * Oblicza maksymalny pułap jednostki na podstawie
 * ekranu radaru i wysokości duszka.
 */

int JednostkaGora (typJednostka *jednostka)
{
   typDuszek *duszek;

   /* --- Pobieramy duszka --- */
   duszek = PobierzDuszka (jednostka);

   /* --- Dodajemy pół wysokości duszka do wysokości
       ekranu radaru --- */
   return (WYSOK_RADARU + (duszek[0].wysok / 2));
}

/*
 * JednostkaDol
 *
 * Obliczamy minimalny pułap jednostki na ekranie,
 * częściowo na podstawie rozmiarów jednostki.
 */
int JednostkaDol (typJednostka *jednostka)
{
   typDuszek *duszek;

   duszek = PobierzDuszka (jednostka);
   return (PobierzWysokoscOkna () - (duszek[0].wysok / 2));
}

/*
 *
 */
void RegulujWysokDuszka (typJednostka *jednostka)
{
   typDuszek *duszek;
```

```
    int nGora;
    int nDol;

    /* --- Pobieramy duszka jednostki --- */
    duszek = PobierzDuszka (jednostka);
    if (duszek == NULL) return;

    /* --- Obliczamy dolny i górny pułap jednostki --- */
    nGora = JednostkaGora (jednostka);
    nDol = JednostkaDol (jednostka);

    /* --- Nie pozwalamy jej na przemieszczenie się
           zbyt nisko lub zbyt wysoko --- */
    if (jednostka->y < nGora) {
       jednostka->y = nGora;
    } else if (jednostka->y > nDol) {
       jednostka->y = nDol;
    }
}

/*
 * RysujInneJednostki
 *
 * Wyświetla wszystkie jednostki na ekranie. Najpierw
 * musimy przesunąć wszystkie jednostki w nowe położenia.
 * Część tego zadania wykonuje moduł AI.
 *
 */
void RysujInneJednostki (GdkPixmap *piksmapa, GtkWidget *obszar_rys)
{
    typJednostka *jednostka;
    typJednostka *jednostkaTrafiona;
    GList *wezel;
    int xPoz;
    int xPozKoniec;
    typDuszek *duszek;

    /* --- Dla każdej jednostki na liście --- */
    for (wezel = listaJednostek; wezel; wezel = wezel->next) {

       /* --- Pobieramy jednostkę --- */
       jednostka = (typJednostka *) wezel->data;
```

```
    /*
     * --- Wywołujemy moduł AI, aby ją przesunąć ---
     */
    ModulAI (jednostka);

    /*
     * Jeśli jednostka została zniszczona przez moduł
     * AI, nie rysujemy jej.
     */
    if (jednostka->bZniszcz) {
       continue;
    }

    /*
     * Jeśli jednostka nie ma duszka, nie możemy
     * jej teraz narysować.
     */

    duszek = PobierzDuszka (jednostka);
    if (duszek == NULL) continue;

    /* --- W jakim kierunku zmierza jednostka? --- */
    xPoz = JednostkaX (jednostka);

    /* --- Upewniamy się, że jednostka nie ucieknie
          poza pole gry --- */
    RegulujWysokDuszka (jednostka);

    /* --- Wreszcie rysujemy jednostkę --- */
    WyswietlDuszka (obszar_rys, duszek,
             (int) (xPoz - duszek[0].szerok / 2),
             (int) (jednostka->y - duszek[0].wysok / 2));
}

/*
 * --- Kiedy wszystko jest już narysowanie, odpalamy laser
 */

for (wezel = listaJednostek; wezel; wezel = wezel->next) {

    jednostka = (typJednostka *) wezel->data;

    /* --- Jeśli jest to laser --- */
```

```
        if (jednostka->typ == LASER) {

            /* --- Pobieramy położenie początkowe i końcowe --- */
            xPoz = EkranX ((int) jednostka->x);
            xPozKoniec = xPoz + DLUG_LASERA * jednostka->kierunek;

            /* --- Sprawdzamy, czy w coś trafiliśmy --- */
            jednostkaTrafiona = KtosPomiedzy ((int) xPoz,
                        (int) jednostka->y,
                        (int) xPozKoniec, (int) jednostka->y);
            if (jednostkaTrafiona) {

               /* --- Trafiliśmy w coś --- */

               /* --- Laser porusza się tylko dotąd --- */
               xPozKoniec = JednostkaX (jednostkaTrafiona);

               /* --- Niszczymy jednostkę --- */
               jednostkaTrafiona->bZniszcz = TRUE;
               jednostka->bZniszcz = TRUE;

               /* --- Efekty specjalne zniszczenia --- */
               DodajWybuch (jednostkaTrafiona);
            }

            /* --- Rysujemy laser --- */
            gdk_draw_line (piksmapa, pioroBiale,
                        xPoz, jednostka->y,
                        xPozKoniec,
                        jednostka->y);

            /* --- Pobieramy rzeczywiste współrzędne lasera --- */
            jednostka->x = GraX (xPozKoniec);

            /* --- Jeśli laser jest za daleko... --- */
            if (OdlegloscPomiedzy (jednostka, gracz) > nSzerokOkna / 2) {

               /* --- ...niszczymy go --- */
               jednostka->bZniszcz = TRUE;
            }
         }
      }
   }
```

```
/*
 * Funkcje obsługujące teren gry
 *
 */

/*
 * PorownajGory
 *
 * Funkcja porównująca dla sortowania szczytów górskich
 */
gint PorownajGory (typGora *m1, typGora *m2)
{
   return (m1->poczatek.x - m2->poczatek.x);
}

/*
 * DodajGore
 *
 * Dodaje szczyt górski do listy przechowującej wszystkie
 * szczyty.
 */
GList *DodajGore (GList *listaGor, int xszczyt, int yszczyt)
{
   typGora *wezel;

   wezel = (typGora *) g_malloc (sizeof (typGora));
   wezel->poczatek.x = xszczyt - yszczyt;
   wezel->poczatek.y = 0;
   wezel->szczyt.x = xszczyt;
   wezel->szczyt.y = yszczyt;
   wezel->koniec.x = xszczyt + yszczyt;
   wezel->koniec.y = 0;

   return (g_list_insert_sorted (listaGor, wezel, PorownajGory));
}

/*
 * DodajPunkt
 *
 * Dodaje szczyt górski w (x, y)
 */
```

```
GList *DodajPunkt (GList *listaTerenu, int x, int y)
{
   typPunkt *p;

   /* --- Przydzielamy pamieć --- */
   p = (typPunkt *) g_malloc (sizeof (typPunkt));

   /* --- Inicjujemy punkt --- */
   p->x = x;
   p->y = y;

   /* --- Dodajemy punkt do listy --- */
   listaTerenu = g_list_append (listaTerenu, p);

   return (listaTerenu);
}

/*
 * GenerujTeren
 *
 * Tworzy losowo rozmieszczone szczyty górskie, które służą
 * do generowania terenu.
 */
void GenerujTeren ()
{
   int xszczyt;
   int yszczyt;
   GList *listaGor = NULL;
   GList *wezel;
   typGora *poprzGora;
   typGora *gora;
   int i;

   /* --- Obliczamy szczyty --- */
   for (i = 0; i < LICZBA_SZCZYTOW; i++) {
      xszczyt = rand () % ZAKRES_X;
      yszczyt = rand () % MAKS_SZCZYT;

      listaGor = DodajGore (listaGor, xszczyt, yszczyt);
   }

   poprzGora = NULL;
```

```
listaTerenu = DodajPunkt (listaTerenu, 0, 0);

/* --- Obliczamy linie na podstawie listy szczytów --- */
for (wezel = listaGor; wezel; wezel = wezel->next) {

   gora = (typGora *) wezel->data;

   /* --- Pierwsza góra --- */
   if (poprzGora == NULL) {
      listaTerenu = DodajPunkt (listaTerenu, gora->poczatek.x,
                        gora->poczatek.y);
      listaTerenu = DodajPunkt (listaTerenu, gora->szczyt.x,
                        gora->szczyt.y);
      poprzGora = gora;

   /* --- Linie nie mogą się krzyżować --- */
   } else if (poprzGora->koniec.x < gora->poczatek.x) {

      listaTerenu = DodajPunkt (listaTerenu,
                        poprzGora->koniec.x,
                        poprzGora->koniec.y);
      listaTerenu = DodajPunkt (listaTerenu, gora->poczatek.x,
                        gora->poczatek.y);
      listaTerenu = DodajPunkt (listaTerenu, gora->szczyt.x,
                        gora->szczyt.y);
      poprzGora = gora;

   /* --- Poprzednia góra przykrywa obecną --- */
   } else if (poprzGora->koniec.x > gora->koniec.x) {

      /* --- Na razie nic nie robimy --- */
   } else {

      /* --- Góry się przecinają --- */
      listaTerenu = DodajPunkt (listaTerenu,
               (poprzGora->koniec.x + gora->poczatek.x) / 2,
               (poprzGora->koniec.x - gora->poczatek.x) / 2);
      listaTerenu = DodajPunkt (listaTerenu, gora->szczyt.x,
                        gora->szczyt.y);
      poprzGora = gora;
   }
}
```

```
    listaTerenu = DodajPunkt (listaTerenu, poprzGora->koniec.x,
                              poprzGora->koniec.y);
    listaTerenu = DodajPunkt (listaTerenu, ZAKRES_X, 0);
}

void ZacznijGre ()
{
    listaJednostek = NULL;

    /* --- Tworzymy statek gracza --- */
    gracz = UtworzGracza ();

    /* --- Generujemy mapę --- */
    GenerujTeren ();

    /* --- Rozmieszczamy ludzi --- */
    RozmiescLudzi ();

    /* --- Rozmieszczamy obcych --- */
    RozmiescObcych ();
}

/*
 * RysujZboczeGory
 *
 * Rysuje góry w tle gry.
 */
void RysujZboczeGory (GdkPixmap *piksmapa,
                      typPunkt *ostatniPt,
                      typPunkt *pt,
                      int nGora,
                      int nDol)
{
    int x1;
    int x2;
    int nPoczatek;
    int nKoniec;
    int nWysok;

    nWysok = nDol - nGora;
    nPoczatek = gracz->x - (nSzerokOkna / 2);
    nKoniec = gracz->x + (nSzerokOkna / 2);
```

```
    x1 = EkranX (ostatniPt->x);
    x2 = EkranX (pt->x);

    if ((x2 < 0) ||
       (x1 > nSzerokOkna)) {

       /* ---  Nic nie robimy --- */

    } else {

       /* --- Rysujemy szczyt --- */
       gdk_draw_line (piksmapa, pioroBiale,
                EkranX (ostatniPt->x),
                (int) (nDol - ((ostatniPt->y * nWysok) / MAKS_SZCZYT)),
                EkranX (pt->x),
                (int) (nDol - ((pt->y * nWysok) / MAKS_SZCZYT)));
    }
}

void RysujGory (GdkPixmap *piksmapa,
                GtkWidget *obszar_rys,
                int nGora, int nDol)
{
    typPunkt *ostatniPt;
    typPunkt *pt;
    GList *wezel;

    ostatniPt = NULL;
    /* --- Jaki jest punkt widzenia? --- */
    for (wezel = listaTerenu; wezel; wezel = wezel->next) {

       /* --- Pobieramy punkt. --- */
       pt = (typPunkt *) wezel->data;

       if (ostatniPt) {

            RysujZboczeGory (piksmapa, ostatniPt, pt, nGora, nDol);
       }
       ostatniPt = pt;
    }
}

void ObliczRegulacje (GtkWidget *obszar_rys)
```

```
{
   nRegulacjaStatku = (obszar_rys->allocation.width / 2);
   nRegulacjaWzgledna = gracz->x - nRegulacjaStatku;
}

/*
 * RysujWszystkieJednostki
 *
 * Rysuje wszystkie jednostki
 */
void RysujWszystkieJednostki (GdkPixmap *piksmapa,
            GtkWidget *obszar_rys)
{
   /*
    * Przesuwamy i wyświetlamy gracza
    */
   DodajTarcie ();
   Ruch (gracz);

   /* --- Utrzymujemy go w granicach pola gry --- */
   RegulujWysokDuszka (gracz);

   WyswietlDuszka (obszar_rys, duszek_statek,
               nRegulacjaStatku - (duszek_statek->szerok / 2),
               (int) gracz->y - (duszek_statek->wysok / 2));
   /*
    * Przesuwamy i wyświetlamy pozostałe jednostki.
    */
   RysujInneJednostki (piksmapa, obszar_rys);
}

/*
 * RysujRadar
 *
 * Rysujemy wszystkie jednostki na ekranie radaru. Oczywiście,
 * najpierw należy narysować ekran radaru, a następnie
 * przekształcić rzeczywiste położenia jednostek na ich
 * położenia "radarowe". Rysujemy na ekranie małe punkty
 * w różnych kolorach, odzwierciedlających typ jednostki.
 */
void RysujRadar (GdkPixmap *piksmapa, GtkWidget *obszar_rys)
```

```
{
   int   nPozostalo;
   GList *wezel;
   typJednostka *jednostka;
   int x, y;
   int nMin;
   int nowex, nowey;

   /* --- Pobieramy współrzędne radarowe --- */
   nPozostalo = (obszar_rys->allocation.width - SZEROK_RADARU) / 2;
   nMin = gracz->x - (ZAKRES_X / 2);

   /* --- Czyścimy prostokąt --- */
   gdk_draw_rectangle (piksmapa, obszar_rys->style->white_gc,
                       FALSE,
                       nPozostalo, 0, SZEROK_RADARU, WYSOK_RADARU);

   /*
    * --- Wyświetlamy wszystkie jednostki
    */
   for (wezel = listaJednostek; wezel; wezel = wezel->next) {

      jednostka = (typJednostka *) wezel->data;
      x = jednostka->x;
      y = jednostka->y;

      if (x > gracz->x + (ZAKRES_X / 2)) {

         x -= ZAKRES_X;

      } else if (x < gracz->x - (ZAKRES_X / 2)) {

         x += ZAKRES_X;
      }
      /* --- Zamieniamy -x- na współrzędne radarowe --- */
      nowex = (x - nMin) * SZEROK_RADARU / ZAKRES_X;

      /* --- Zamieniamy -y- na współrzędne radarowe --- */
      nowey = ((y - WYSOK_RADARU) * (WYSOK_RADARU - 6) /
            WYSOK_OKNA) + 2;

      switch (jednostka->typ) {
```

```
            case OSOBA:
               gdk_draw_rectangle (piksmapa, pioroFioletowe,
                     TRUE, nPozostalo + nowex-1, nowey-1, 2, 2);
               break;

            case LADOWNIK:
               gdk_draw_rectangle (piksmapa, pioroZielone,
                     TRUE, nPozostalo + nowex-1, nowey-1, 2, 2);
               break;

            case MUTANT:
               gdk_draw_rectangle (piksmapa, pioroCzerwone,
                     TRUE, nPozostalo + nowex-1, nowey-1, 2, 2);
               break;

            case POCISK:
            case WYBUCH:
               gdk_draw_rectangle (piksmapa, pioroBiale,
                     TRUE, nPozostalo + nowex, nowey, 1, 1);
               break;
         }
      }

      /*
       * --- Umieszczamy na radarze także gracza.
       */

      /* --- Zamieniamy -x- na współrzędne radarowe --- */
      nowex = (gracz->x - nMin) * SZEROK_RADARU / ZAKRES_X;

      /* --- Zamieniamy -y- na współrzędne radarowe --- */
      nowey = ((gracz->y - WYSOK_RADARU) * (WYSOK_RADARU - 6) /
   WYSOK_OKNA) + 2;

      gdk_draw_rectangle (piksmapa, pioroBiale, TRUE,
                  nPozostalo + nowex-1, nowey-1, 2, 2);
   }

   /*
    * RysujEkran
    *
    * Procedura ta wykonuje bardzo wiele czynności. Rysuje tło
    * i wszystkie jednostki, a także radar. Jest to w istocie
```

```
 * główna pętla gry, wywoływana co pewien czas, określony
 * przez częstotliwość czasomierza.
 */
void RysujEkran (GdkPixmap *piksmapa, GtkWidget *obszar_rys)
{

   /* --- Przesuwamy gracza w zależności od klawiszy --- */
   ObsluzWcisnieteKlawisze ();

   /* --- Pobieramy szerokość okna --- */
   nSzerokOkna = obszar_rys->allocation.width;

   /* --- Obliczamy przesunięcie gracza --- */
   ObliczRegulacje (obszar_rys);

   /* --- czyścimy piksmapę (drugoplanowy bufor) --- */
   gdk_draw_rectangle (piksmapa,
         obszar_rys->style->black_gc,
         TRUE,
         0, 0,
         obszar_rys->allocation.width,
         obszar_rys->allocation.height);

   /* --- Rysujemy górne obramowanie i ekran radaru --- */
   gdk_draw_line (piksmapa, obszar_rys->style->white_gc,
            0, WYSOK_RADARU,
            obszar_rys->allocation.width,
            WYSOK_RADARU);

   /* --- Wysokie góry... --- */
   RysujGory (piksmapa, obszar_rys,
            obszar_rys->allocation.height - 65,
            obszar_rys->allocation.height - WYSOK_SPODU);

   /* --- Rysujemy jednostki --- */
   RysujWszystkieJednostki (piksmapa, obszar_rys);

   /* --- Rysujemy jednostki na radarze --- */
   RysujRadar (piksmapa, obszar_rys);

   /* --- Sprawdzamy kolizje --- */
   SprawdzenieKolizji ();
```

```
    /* --- Usuwamy zniszczone jednostki --- */
    ZwolnijZniszczoneJedn ();

}

/*
 * SprawdzenieKolizji
 *
 * Czy gracz zderzył się z jakąś jednostką? Sprawdzamy
 * kolizję, ale niczego nie robimy. W końcu nie jest
 * to prawdziwa gra, więc demonstrujemy tylko, jak
 * można to zrobić.
 */
void SprawdzenieKolizji ()
{
    GList *wezel;
    typDuszek *duszek;
    int gracz_x;
    int jednostka_x;
    typJednostka *jednostka;

    /* --- Sprawdzamy, czy gracz z czymś się zderzył --- */
    for (wezel = listaJednostek; wezel; wezel = wezel->next) {

        /* --- Pobieramy jednostkę --- */
        jednostka = (typJednostka *) wezel->data;

        /* --- Pobieramy duszka jednostki --- */
        duszek = PobierzDuszka (jednostka);

        if (duszek == NULL) continue;

        /* --- Eliminujemy sytuacje, w których
               nie doszło do kolizji --- */
        if (gracz->y + (duszek_statek->wysok / 2) <
            jednostka->y - (duszek->wysok / 2)) continue;

        if (gracz->y - (duszek_statek->wysok / 2) >
            jednostka->y + (duszek->wysok / 2)) continue;

        gracz_x = JednostkaX (gracz);
        jednostka_x = JednostkaX (jednostka);
```

```
        if (gracz_x + (duszek_statek->szerok / 2) <
           jednostka_x - (duszek->szerok / 2)) continue;

        if (gracz_x - (duszek_statek->szerok / 2) >
           jednostka_x + (duszek->szerok / 2)) continue;

        /* --- Żaden z powyższych warunków nie jest
              spełniony, więc gracz z czymś się zderzył --- */
    }
}

/*
 * PobierzWysokoscOkna
 *
 * Zwraca wysokość okna gry.
 */
int PobierzWysokoscOkna ()
{

    return (WYSOK_RADARU + WYSOK_OKNA + WYSOK_SPODU);
}

/*
 * ZwolnijZniszczoneJedn
 *
 * Jednostki są najpierw oznaczane jako zniszczone. Procedura
 * ta przegląda listę i usuwa wszystkie zaznaczone jednostki,
 * zwalniając zajmowaną przez nie pamięć.
 */
void ZwolnijZniszczoneJedn ()
{
    GList *wezel;
    GList *nastepny;
    typJednostka *jednostka;

    wezel = listaJednostek;
    while (wezel) {

        nastepny = wezel->next;

        jednostka = (typJednostka *) wezel->data;

        /* --- Jeśli jednostka ma zostać zniszczona --- */
```

```
        if (jednostka->bZniszcz) {

          /* --- Usuwamy ją z listy jednostek --- */
          listaJednostek = g_list_remove (listaJednostek,
                                          jednostka);

          /* --- Jeśli jednostka była połączona z inną --- */
          if (jednostka->przylacz) {

            /* --- Usuwamy połączenie z tamtej jednostki --- */
            jednostka->przylacz->przylacz = NULL;
          }
          g_free (jednostka);
        }

        wezel = nastepny;
      }
    }
```

Podsumowanie

GTK+ wraz z GDK można wykorzystać do tworzenia animacji i gier wideo. Wydajność GTK+ jest wystarczająca dla prostych gier, nawet na niezbyt szybkich komputerach. Bardziej złożone gry wideo wymagają szybszego sprzętu i prawdopodobnie szybszej biblioteki.