Rozdział 10

GDK (Graphics Drawing Kit)

GDK (*Graphics Drawing Kit*) jest niskopoziomową warstwą, znajdującą się pomiędzy GTK+ i zależnym od systemu operacyjnego interfejsem API (*Application Programming Interface*) - w przypadku Linuksa jest to Xlib. Ponieważ biblioteka GTK+ nie współpracuje bezpośrednio z interfejsem API komputera, jej przeniesienie do innego systemu jest kwestią przeniesienia GDK i GLIB. GDK udostępnia funkcje rysujące (aż do poziomu pojedynczych punktów), oraz niskopoziomowe funkcje tworzące okna i manipulujące nimi. Kontrolki GTK+ są wystarczające w wielu aplikacjach, ale jeśli zechcielibyśmy napisać program analogowego zegara, byłoby to trudne, gdybyśmy nie dysponowali możliwością narysowania tarczy zegara. Wykorzystanie kontrolki obszaru rysunkowego (*drawing area widget*) wraz z funkcjami GDK pozwoli nam na narysowanie dowolnych obiektów; nie jesteśmy więc skazani na gotowe kontrolki.

Funkcje rysujące

Pisanie aplikacji z wykorzystaniem procedur GDK jest niewiele łatwiejsze, niż posługiwanie się bezpośrednio biblioteką Xlib. Na szczęście GTK+ zawiera kontrolkę, którą można użyć w aplikacjach wymagających samodzielnego rysowania. Kontrolką obszaru rysunkowego posługujemy się w ten sam sposób, co wszystkimi innymi kontrolkami GTK+; jest ona przy tym wystarczająco elastyczna, aby stać się podstawą aplikacji wymagających użycia grafiki. Zaletą takiego podejścia jest to, że w jednej aplikacji możemy użyć zarówno GTK+, jak i GDK. GTK+ zapewnia wówczas menu, paski narzędziowe i inne kontrolki, które wspierają rysowanie w kontrolce obszaru rysunkowego. GDK udostępnia interfejs API, służący do kreślenia linii, pikseli, prostokątów, okręgów i innych figur.

Każda procedura GDK przyjmuje przynajmniej dwa parametry - obszar rysunkowy (GdkDrawable) oraz GdkDC. GdkDrawable reprezentuje obszar, na którym będzie odbywać się rysowanie, natomiast GdkDC zawiera informacje o czcionce i kolorze oraz inne parametry rysowania.

Rysowanie pikseli

GDK udostępnia procedury służące do rysowania wewnątrz okien i kontrolek (takich jak obszar rysunkowy). Najprostszą z nich jest procedura gdk_draw_point, służąca do rysowania pojedynczego punktu wewnątrz obszaru rysunkowego.

```
/* --- rysujemy punkt o współrzędnych 10, 10 --- */
gdk_draw_point (obszar_rys, gc, 10, 10);
```

W tym i w dalszych przykładach obszar_rys reprezentuje obszar, w którym odbywa się rysowanie, a gc zazwyczaj reprezentuje informacje o kolorze. W dalszej części tego rozdziału nauczymy się pobierać i ustawiać te parametry.

Rysowanie linii

Funkcja gdk_draw_line służy do rysowania linii pomiędzy dwoma punktami.

```
/* --- rysujemy ukośną linię --- */
gdk_draw_line (obszar_rys, gc, 0, 0, 10, 10);
```

Rysowanie prostokątów

Funkcja gdk_draw_rectangle rysuje prostokąt wewnątrz obszaru rysunkowego. Prostokąt może zostać wypełniony kolorem - z możliwości tej często korzysta się, aby wymazać obszar ekranu, który wymaga przerysowania. Pierwsze dwie liczby są współrzędnymi lewego górnego rogu prostokąta, a dwie ostatnie liczby określają szerokość i wysokość prostokąta. Trzeci parametr określa, czy prostokąt należy wypełnić, czy też pozostawić go pustym. Wartość TRUE nakazuje wypełnienie prostokąta.

```
/* --- rysujemy prostokąt na ekranie (0, 0), (20, 20) --- */
gdk_draw_rectangle (drawable, gc, FALSE, 0, 0, 20, 20);

/* --- rysujemy wypełniony prostokąt (10, 10), (30, 30) --- */
gdk_draw_rectangle (drawable, gc, TRUE, 10, 10, 30, 30);
```

Rysowanie wielokątów

Rysowanie bardziej skomplikowanych kształtów (na przykład ośmiokąta albo ludzkiej głowy) wymaga nakreślenia wielu linii. Zamiast wielokrot-

nie wywoływać funkcję gdk_draw_line, zazwyczaj wygodniej jest skorzystać z funkcji gdk_draw_polygon i przekazać do niej tablicę punktów, które należy połączyć liniami. Funkcja połączy także pierwszy i ostatni punkt w tablicy, jeśli jest to ten sam punkt. Można wypełnić wielokąt, ustawiając odpowiedni parametr na TRUE.

Oto jeden ze sposobów narysowania trójkąta:

```
GdkPoint punkty[4];

/* --- górny róg trójkąta --- */
punkty[0].x = 50;
punkty[0].y = 0;

/* --- dolny, lewy róg trójkąta --- */
punkty[0].x = 0;
punkty[0].y = 50;

/* --- dolny, prawy róg trójkąta --- */
punkty[0].x = 100;
punkty[0].y = 50;

/* --- zamykamy trojkąt --- */
punkty[0].x = 50;
punkty[0].y = 0;

/* --- rysujemy trójkąt (bez wypełnienia) --- */
gdk_draw_polygon (obszar_rys, gc, FALSE, punkty, 4);

/* --- rysujemy trójkąt i wypełniamy go --- */
gdk_draw_polygon (obszar_rys, gc, TRUE, punkty, 4);
```

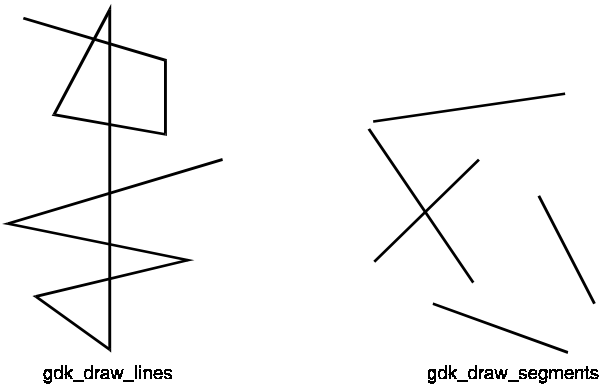
Rysowanie wielu linii

Istnieją dwie funkcje, które mogą przyspieszyć rysowanie wielu linii. Funkcja gdk_draw_lines przyjmuje listę punktów i łączy je liniami. Funkcja gdk_draw_segments przyjmuje tablicę odcinków i rysuje je jako niezależne linie. W poniższym przykładzie używamy obu funkcji, aby narysować ten sam trójkąt, który rysowaliśmy wcześniej.

```
/*
 * najpierw do narysowania trójkąta użyjemy funkcji gdk_draw_lines
 */

/* --- górny róg trójkąta --- */
```

```
punkty[0].x = 50;
punkty[0].y = 0;

/* --- dolny, lewy róg trójkąta --- */
punkty[0].x = 0;
punkty[0].y = 50;

/* --- dolny, prawy róg trójkąta --- */
punkty[0].x = 100;
punkty[0].y = 50;

/* --- zamykamy trójkąt --- */
punkty[0].x = 50;
punkty[0].y = 0;

/* --- rysujemy linie pomiędzy czterema punktami --- */
gdk_draw_lines (obszar_rys, gc, punkty, 4);

/*
 * teraz narysujemy trójkąt za pomocą funkcji gdk_draw_segments
 */

/* --- określamy odcinki trójkąta --- */
odcinek[0].x1 = 50;
odcinek[0].y1 = 0;
odcinek[1].x2 = 0;
odcinek[1].y2 = 50;

odcinek[2].x1 = 0;
odcinek[2].y1 = 50;
odcinek[3].x2 = 100;
odcinek[3].y2 = 50;

odcinek[4].x1 = 100;
odcinek[4].y1 = 50;
odcinek[5].x2 = 50;
odcinek[5].y2 = 0;

/* --- rysujemy trzy odcinki --- */
gdk_draw_segments (obszar_rys, gc, odcinek, 3);
```

Rysunek 10.1 przedstawia różnicę pomiędzy rysowaniem wielu linii i wielu odcinków.



Rysunek 10.1. Rysowanie wielu linii i odcinków.

Można także użyć funkcji gdk_draw_points, aby jednocześnie narysować wiele punktów wewnątrz kontrolki obszaru rysunkowego. Jest to szybsze, niż wielokrotne wywoływanie funkcji gdk_draw_point, ale niezależnie od użytej metody, rysowanie przez aplikację pojedynczych punktów jest dosyć wolne. Jeśli to możliwe, należy wykorzystać piksmapy.

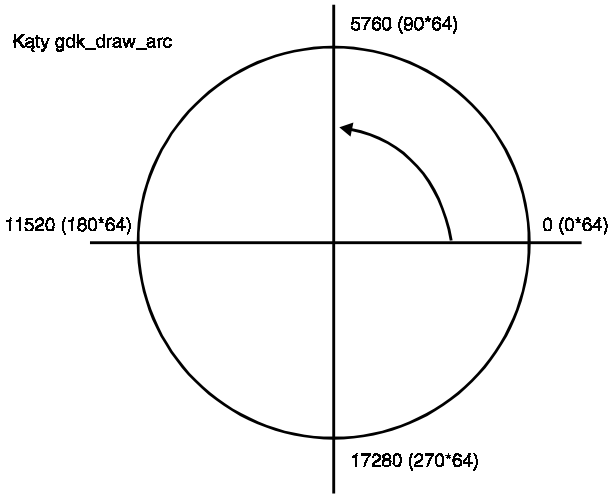
Rysowanie okręgów i łuków

Funkcja gdk_draw_arc rysuje łuk albo okrąg wewnątrz obszaru rysunkowego. Funkcja ta wymaga bardziej szczegółowego objaśnienia, niż pozostałe funkcje, ponieważ przyjmuje znacznie więcej parametrów. Prototyp funkcji jest zdefiniowany następująco:

```
gdk_draw_arc (GdkDrawable *obszar_rys, /* - obszar rysunkowy - */
        gint    *gc,              /* - dane (kolor/czcionka) - */
        gint    wypelniony,       /* - czy należy wypełniać? - */
        gint    x,                /* - lewa strona - */
        gint    y,                /* - górna strona - */
        gint    szerokosc,
        gint    wysokosc,
        gint    kat1,             /* - kąt początkowy - */
        gint    kat2);            /* - kąt końcowy - */
```

Definiowanie kąta początkowego i końcowego łuku wprowadza trochę zamieszania, ponieważ w przeciwieństwie do większości funkcji operujących na łukach, nie określa się ich w stopniach ani radianach. Kąt należy przekształcić na stopnie i pomnożyć przez 64. Kąt 0 znajduje się na godzinie trzeciej, a wzrasta przeciwnie do ruchu wskazówek zegara. Jeśli

chcielibyśmy narysować łuk od szczytu do spodu prostokąta, rysowalibyśmy go od kąta (90 * 64) do (180 * 64). Rysunek 10.2 ilustruje te rozważania.



Rysunek 10.2. gdk_draw_arc.

```
/* --- rysujemy pełny okrąg --- */
gdk_draw_arc (obszar_rys, gc, FALSE, 0, 0, 30, 30, 0, (360 * 64));

/* --- rysujemy 90-stopniowy łuk od godziny 12 do 9 --- */
gdk_draw_arc (obszar_rys, gc, FALSE, 0, 0, 30, 30, (90 * 64),
        (360 * 64));
```

Wyświetlanie tekstu

Do wyświetlania tekstu wewnątrz obszaru rysunkowego służy funkcja gdk_draw_string. Oprócz obszaru rysunkowego i GdkGC należy przekazać do niej żądaną czcionkę; pod innymi względami jest podobna do pozostałych funkcji graficznych.

```
/* --- wyświetlamy "Hej!" --- */
gdk_draw_string (obszar_rys, czcionka, gc, 50, 50, "Hej!");
```

Funkcja gdk_draw_text robi to samo, co gdk_draw_string, ale przyjmuje także dodatkowy parametr - długość łańcucha.

```
/* --- wyświetlamy "Hej!" --- */
```

```
szKomunikat = "Hej!";
gdk_draw_text (obszar_rys, czcionka, gc, 50, 50, szKomunikat,
        strlen (szKomunikat));
```

Rysowanie piksmap

Kiedy rysowanie linii i punktów okaże się niewystarczające, można skorzystać z funkcji gdk_draw_pixmap, aby skopiować kompletny rysunek do obszaru rysunkowego. Więcej informacji na temat tej przydatnej funkcji zawiera podrozdział „Usuwanie migotania".

```
/* --- Kopiujemy piksmapę 40 x 40 do obszaru rysunkowego --- */
gdk_draw_pixmap (obszar_rys, gc, piksmapa, 0, 0, 0, 0, 40, 40);
```

Kontrolka obszaru rysunkowego

Kontrolka obszaru rysunkowego jest bardzo prosta. Większość kontrolek, które omawialiśmy do tej pory była pojemnikami albo posiadała przypisane im właściwości. Obszar rysunkowy posiada bardzo nieliczne właściwości w porównaniu z innymi kontrolkami. Ma rozmiar, który można ustawić przy pomocy funkcji gtk_drawing_area_size, i to mniej więcej wszystko. Jedynym zadaniem kontrolki jest udostępnienie obszaru, na którym mogą rysować funkcje GDK. Kontrolkę tworzymy przy pomocy funkcji gtk_drawing_area_new.

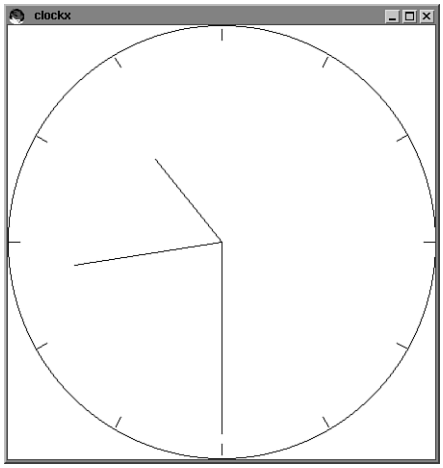
```
/* --- Tworzymy kontrolkę --- */
obszar_rys = gtk_drawing_area_new ();

/* --- ustawiamy odpowiednie rozmiary --- */
gtk_drawing_area_size (obszar_rys, 200, 200);
```

Zdarzenia w obszarze rysunkowym

Dwoma zdarzeniami, które mają znaczenie w przypadku obszaru rysunkowego, są configure_event i expose_event. Sygnał configure_event jest wysyłany po to, aby wskazać, że obszar rysunkowy został utworzony albo zmienił się jego rozmiar. Sygnał expose_event jest wysyłany wtedy, kiedy konieczne jest przerysowanie obszaru: kiedy rysowana jest kontrolka obszaru rysunkowego lub kiedy inne okno zostanie odsunięte znad obszaru (odsłaniając go). Może zostać także wysłany w odpowiedzi na żądanie przerysowania, wygenerowane przez samą aplikację.

Prosta aplikacja zegara

Aplikacja zegara ilustruje korzystanie z obszaru rysunkowego i funkcji GDK. Konieczne jest narysowanie zegara na ekranie (łuk i kilka kresek, oznaczających godziny), oraz narysowanie wskazówek zegara i uaktualnianie ich pozycji co sekundę. Zegar korzysta z czasomierza, aby co sekundę uaktualniać wyświetlany czas i odświeżać cały obszar rysunkowy. Rysunek 10.3 przedstawia cel, do którego zmierzamy.



Rysunek 10.3. Aplikacja zegara.

Oczywiste jest, że potrzebny będzie czasomierz, który co sekundę będzie rysował na zegarze aktualny czas. Aplikacja zegara ustawia czasomierz przy pomocy funkcji gtk_timeout_add, nakazując wywoływanie funkcji Przerysuj. Funkcja Przerysuj co sekundę czyści obszar rysunkowy i przerysowuje cały zegar. Oto kod:

```
/*
 * Autor: Eric Harlow
 *
 *
 */

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

int promien;
```

```
/*
 * NarysujKreske
 *
 * Rysuje kreskę na tarczy zegara. Kreski rysujemy przy
 * godzinach, aby łatwiej było odczytać czas na zegarze.
 *
 * piksmapa - obszar rysunkowy
 * gc - pióro
 * nGodzina - 1-12, przy jakiej godzinie narysować kreskę
 * cx - szerokość zegara
 * cy - wysokość zegara
 */
void NarysujKreske (GdkDrawable *piksmapa, GdkGC *gc, int nGodzina,
                    int cx, int cy)
{

   /* --- Przekształcamy godzinę na kąt w radianach  --- */
   double dRadiany = nGodzina * 3.14 / 6.0;

   /* --- Rysujemy linię od .95 * rad do 1.0 * rad --- */
   gdk_draw_line (piksmapa, gc,
                        cx+(int) ((0.95 * promien * sin (dRadiany))),
                        cy-(int) ((0.95 * promien * cos (dRadiany))),
                        cx+(int) ((1.0 * promien * sin (dRadiany))),
                        cy-(int) ((1.0 * promien * cos (dRadiany))));
}

/*
 * Przerysuj
 *
 * dane - kontrolka do przerysowania
 */
gint Przerysuj (gpointer dane)
{
   GtkWidget*obszar_rys = (GtkWidget *) dane;
   int midx, midy;
   int    nGodzina;
   float dRadiany;
   time_t teraz;
   struct tm *teraz_tm;
   GdkDrawable *rysunek;
```

```
/* --- Pobieramy okno z obszarem rysunkowym --- */
rysunek = obszar_rys->window;

/* --- Czyścimy obszar, rysując prostokąt --- */
gdk_draw_rectangle (rysunek,
                    obszar_rys->style->white_gc,
                    TRUE,
                    0, 0,
                    obszar_rys->allocation.width,
                    obszar_rys->allocation.height);

/* --- Określamy punkt środkowy --- */
midx = obszar_rys->allocation.width / 2;
midy = obszar_rys->allocation.height / 2;

/* --- Znajdujemy mniejszą wartość --- */
promien = MIN (midx, midy);

/* --- Rysujemy tarczę zegara (okrąg) --- */
gdk_draw_arc (rysunek,
              obszar_rys->style->black_gc,
              0,
              0, 0, midx + midx, midy + midy, 0, 360 * 64);

/* --- Rysujemy kreski przy godzinach --- */
for (nGodzina = 1; nGodzina <= 12; nGodzina++) {

 NarysujKreske (rysunek,
                obszar_rys->style->black_gc,
                nGodzina, midx, midy);
}

/* --- Pobieramy aktualny czas --- */
time (&teraz);

/* --- Przekształcamy czas --- */
teraz_tm = localtime (&teraz);

/*
 * --- Rysujemy wskazówkę sekundową
 */

/* --- Obliczamy radiany na podstawie liczby sekund --- */
dRadiany = teraz_tm->tm_sec * 3.14 / 30.0;
```

```
    /* --- Rysujemy wskazówkę --- */
    gdk_draw_line (rysunek, obszar_rys->style->black_gc,
                   midx, midy,
                   midx + (0.9 * promien * sin (dRadiany)),
                   midy - (0.9 * promien * cos (dRadiany)));

    /*
     * --- Rysujemy wskazówkę minutową
     */

    /* --- Obliczamy radiany na podstawie liczby minut i sekund --- */
    dRadiany = (teraz_tm->tm_min * 3.14 / 30.0) +
               (3.14 * teraz_tm->tm_sec / 1800.0);

    /* --- Rysujemy wskazówkę --- */
    gdk_draw_line (rysunek, obszar_rys->style->black_gc,
                   midx, midy,
                   midx+(int) (0.7 * promien * sin (dRadiany)),
                   midy-(int) (0.7 * promien * cos (dRadiany)));

    /*
     * --- Rysujemy wskazówkę godzinową
     */

    /* --- Przeliczamy godzinę na radiany --- */
    dRadiany = (teraz_tm->tm_hour % 12) * 3.14 / 6.0 +
            (3.14 * teraz_tm->tm_min / 360.0);

    /* --- Rysujemy wskazówkę --- */
    gdk_draw_line (rysunek, obszar_rys->style->black_gc,
                   midx, midy,
                   midx + (int) (promien * 0.5 * sin (dRadiany)),
                   midy - (int) (promien * 0.5 * cos (dRadiany)));

    return (TRUE);
}

/*
 * zamknij
 *
 * opuszczamy pętlę zdarzeń gtk
 */

void zamknij ()
```

```
{
   gtk_exit (0);
}

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

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

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

   /* --- Tworzymy pole pakujące --- */
   ypole = gtk_hbox_new (FALSE, 0);

   /* --- Dodajemy pole pakujące do okna --- */
   gtk_container_add (GTK_CONTAINER (okno), ypole);

   /* --- Uwidaczniamy pole pakujące --- */
   gtk_widget_show (ypole);

   /* --- Czekamy na sygnał "destroy" --- */
   gtk_signal_connect (GTK_OBJECT (okno), "destroy",
                       GTK_SIGNAL_FUNC (zamknij), NULL);

   /* --- Tworzymy obszar rysunkowy  --- */
   obszar_rys = gtk_drawing_area_new ();

   /* --- Ustawiamy jego rozmiary --- */
   gtk_drawing_area_size (GTK_DRAWING_AREA (obszar_rys), 200, 200);

   /* --- Dodajemy obszar rysunkowy do pola pakującego --- */
   gtk_box_pack_start (GTK_BOX (ypole), obszar_rys, TRUE, TRUE, 0);

   /* --- Uwidaczniamy obszar rysunkowy --- */
```

```
    gtk_widget_show (obszar_rys);

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

    /* --- Przerysowujemy obszar co 1000 ms (co sekundę) --- */
    gtk_timeout_add (1000, Przerysuj, (gpointer) obszar_rys);

    /* --- Uruchamiamy pętlę obsługi zdarzeń gtk --- */
    gtk_main ();

    return 0;
}
```

Z jakimi problemami będziemy musieli się uporać? Po pierwsze, zegar będzie migotał na wolnych komputerach, a także na szybszych, choć w nieco mniejszym stopniu. Migotanie jest spowodowane przez procedurę przerysowującą. Ekran komputera jest nieustannie odświeżany (wiele razy na sekundę). Jeśli sprzęt odświeży wyświetlany na monitorze obraz zaraz po wyczyszczeniu obszaru rysunkowego przez kod, wówczas przez krótki moment na ekranie pojawi się czysty (albo tylko częściowo przerysowany) obszar rysunkowy. Oczywiście, przy następnym sprzętowym odświeżeniu ekranu rysowanie kontrolki prawdopodobnie dobiegnie końca, ale będzie już za późno - ludzkie oko nie dostrzeże co prawda pustego zegara, ale zauważy migotanie. Zegar ma szansę migotać tylko raz na sekundę, kiedy jest przerysowywany. Im bardziej skomplikowany jest rysunek, tym większe prawdopodobieństwo, że nie zostanie on dokończony pomiędzy kolejnymi odświeżeniami obrazu, a użytkownik będzie widział migotanie. Jest to niepożądany efekt, wiec będziemy musieli się go pozbyć.

Drugi problem stanie się widoczny, kiedy kilkakrotnie spróbujemy przykryć zegar jakimś oknem, a następnie odsunąć je znad zegara. Za którymś razem zauważymy długie opóźnienie pomiędzy odsłonięciem zegara a uaktualnieniem wyświetlanego obrazu. Opóźnienie to nigdy nie przekracza sekundy, a spowodowane jest tym, że aplikacja nie oczekuje na sygnał expose_event, który informuje, że okno zostało odsłonięte i wymaga przerysowania. Poprawna aplikacja GDK powinna zawierać funkcję zwrotną dla sygnału expose_event i uaktualniać zegar natychmiast, nie czekając na następny sygnał od czasomierza.

Usuwanie migotania

Podstawowy problem w aplikacji polega na tym, że zegar migocze co sekundę w trakcie przerysowywania. Najwygodniejszą metodą usunięcia tego efektu jest użycie techniki znanej jako *podwójne buforowanie* (*double buffering*). W metodzie tej korzystamy z bufora, używanego jako płótno dla wszelkich operacji graficznych. Obraz na ekranie nie jest modyfikowany; rysowanie przeprowadza się w buforze. W naszym przykładzie możemy utworzyć bufor o takich samych rozmiarach jak okno, i rysować zegar w buforze, zamiast bezpośrednio w oknie. Kiedy zakończymy rysowanie zegara w buforze, skopiujemy go do okna. Ponieważ przenoszenie bufora do okna nie wymaga czyszczenia ekranu, migotanie nie będzie widoczne.

Pierwszymi krokami na drodze do eliminacji migotania podczas animacji zegara jest sprawdzanie sygnałów expose_event i configure_event. Sygnał configure_event jest wysyłany podczas tworzenia obszaru rysunkowego albo zmiany jego rozmiarów. W tym momencie możemy utworzyć drugoplanowy bufor, o rozmiarze równym obszarowi rysunkowemu. Jeśli rozmiary okna ulegną zmianie, funkcja zwrotna dla sygnału configure_event ponownie utworzy bufor o właściwym rozmiarze.

configure_event

Funkcja zwrotna dla sygnału configure_event tworzy drugoplanowy bufor, w którym będziemy rysować zegar, nie naruszając obrazu wyświetlanego na ekranie. Bufor ten będzie piksmapą; możemy utworzyć piksmapę o rozmiarach równych obszarowi rysunkowemu przy pomocy funkcji gdk_pixmap_new. Jeśli zmieni się rozmiar kontrolki obszaru rysunkowego, należy ponownie przydzielić miejsce na piksmapę. Funkcja zwrotna configure_event jest wywoływana wtedy, kiedy rozmiar kontrolki ulega zmianie, co umożliwia nam utworzenie nowej, drugoplanowej piksmapy o odpowiednio uaktualnionych rozmiarach. Ponieważ poprzednia piksmapa jest już niepotrzebna, należy wywołać funkcję gdk_pixmap_unref, aby poinformować GDK, że nie odwołujemy się już do piksmapy, więc można ją zwolnić - o ile nie odwołuje się do niej ktoś inny. Funkcja zwrotna dla sygnału expose_event wygląda więc następująco:

```
static gint configure_event (GtkWidget *kontrolka,
                             GdkEventConfigure *zdarzenie)
{
   /* --- zwalniamy bufor, jeśli już istnieje --- */
   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

Sygnał expose_event jest bardzo prosty w obsłudze. Ponieważ rysujemy zegar w drugoplanowej piksmapie, jedyną czynnością po wystąpieniu expose_event jest skopiowanie drugoplanowego rysunku do naszego okna przy pomocy funkcji gdk_draw_pixmap. Obsługa sygnału expose_ event usuwa drugi problem, związany z zegarem: opóźnienie w odświeżaniu obrazu po odsunięciu innego okna znad zegara. Teraz, kiedy okno zostanie odsunięte znad zegara, aplikacja obsłuży sygnał expose_event, kopiując ostatni obraz zegara (piksmapę) do swojego okna. Sygnał expose_event jest wysyłany wraz z dodatkowym parametrem, zawierającym dane o zdarzeniu. Pole area tego parametru określa obszar okna, który został odsłonięty i wymaga przerysowania. Przydaje się to w sytuacjach, kiedy na ekranie jest wiele nakładających się na siebie okien i jedno z nich zostanie przesunięte; wówczas najszybszą metodą jest przerysowanie tylko tego obszaru, który był ukryty pod innym oknem. W naszym przykładowym programie musielibyśmy wymusić przerysowanie całego zegara. Funkcja zwrotna dla sygnału expose_event wygląda więc następująco:

```
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;
}
```

Prawdopodobnie kilka rzeczy wymaga jeszcze wyjaśnienia. Jeśli rysujemy na drugoplanowej piksmapie, jak zamierzamy uaktualnić obraz na ekranie? Po narysowaniu zegara na piksmapie musimy spowodować, aby funkcja zwrotna expose_event skopiowała piksmapę do okna. Definiujemy w tym celu obszar, który ma zostać odsłonięty (cały obszar rysunkowy) i wywołujemy funkcję gtk_widget_draw, przekazując jej kontrolkę i obszar, który wymaga uaktualnienia.

```
/* --- definiujemy obszar, wymagający przerysowania --- */
uakt_prostokat.x = 0;
uakt_prostokat.y = 0;
uakt_prostokat.width = obszar_rys->allocation.width;
uakt_prostokat.height = obszar_rys->allocation.height;

/* --- rysujemy obszar (powoduje wywołanie expose_event) --- */
gtk_widget_draw (obszar_rys, &uakt_prostokat);
```

Poniżej znajduje się nowa wersja zegara, z zaimplementowanym podwójnym buforowaniem. Większość kodu przypomina pierwotną wersję. Zmiany są nieznaczne, ale efekty uderzające.

```
/*
 * Autor: Eric Harlow
 *
 *
 */

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

/* --- piksmapa - bufor obszaru rysunkowego --- */
static GdkPixmap *piksmapa = NULL;

int promien;

/*
 * NarysujKreske
 *
 * Rysuje kreskę na tarczy zegara. Kreski rysujemy przy
 * godzinach, aby łatwiej było odczytać czas na zegarze.
 *
 * piksmapa - obszar rysunkowy
 * gc - pióro
```

```
 * nGodzina - 1-12, przy jakiej godzinie narysować kreskę
 * cx - szerokość zegara
 * cy - wysokość zegara
 */
void NarysujKreske (GdkDrawable *piksmapa, GdkGC *gc, int nGodzina,
                    int cx, int cy)
{

   /* --- Przekształcamy godzinę na kąt w radianach  --- */
   double dRadiany = nGodzina * 3.14 / 6.0;

   /* --- Rysujemy linię od .95 * rad do 1.0 * rad --- */
   gdk_draw_line (piksmapa, gc,
                  cx+(int) ((0.95 * promien * sin (dRadiany))),
                  cy-(int) ((0.95 * promien * cos (dRadiany))),
                  cx+(int) ((1.0 * promien * sin (dRadiany))),
                  cy-(int) ((1.0 * promien * cos (dRadiany))));
}

/*
 * Przerysuj
 *
 * dane - kontrolka do przerysowania
 */
gint Przerysuj (gpointer dane)
{
   GtkWidget*obszar_rys = (GtkWidget *) dane;
   GdkRectangle uakt_prostokat;
   char bufor[88];
   int midx, midy;
   int   nGodzina;
   float dRadiany;
   time_t teraz;
   struct tm *teraz_tm;

   /* --- Czyścimy piksmapę (rysując drugoplanowy prostokąt) --- */
   gdk_draw_rectangle (piksmapa,
                       obszar_rys->style->white_gc,
                       TRUE,
                       0, 0,
                       obszar_rys->allocation.width,
                       obszar_rys->allocation.height);
```

```
/* --- Określamy punkt środkowy zegara --- */
midx = obszar_rys->allocation.width / 2;
midy = obszar_rys->allocation.height / 2;

/* --- obliczmy promień --- */
promien = MIN (midx, midy);

/* --- Rysujemy okrąg --- */
gdk_draw_arc (piksmapa,
                    obszar_rys->style->black_gc,
                    0,
                    0, 0, midx + midx, midy + midy, 0, 360 * 64);

/* --- Rysujemy kreski przy godzinach --- */
for (nGodzina = 1; nGodzina <= 12; nGodzina++) {

 NarysujKreske (piksmapa, obszar_rys->style->black_gc,
           nGodzina, midx, midy);
}

/* --- Pobieramy czas --- */
time (&teraz);
teraz_tm = localtime (&teraz);

/*
 * --- Rysujemy wskazówkę sekundową
 */

/* --- Obliczamy radiany na podstawie liczby sekund --- */
dRadiany = teraz_tm->tm_sec * 3.14 / 30.0;

/* --- Rysujemy wskazówkę --- */
gdk_draw_line (piksmapa, obszar_rys->style->black_gc,
                    midx, midy,
                    midx + (0.9 * promien * sin (dRadiany)),
                    midy - (0.9 * promien * cos (dRadiany)));

/*
 * --- Rysujemy wskazówkę minutową
 */

/* --- Obliczamy radiany na podstawie liczby minut i sekund --- */
dRadiany = (teraz_tm->tm_min * 3.14 / 30.0) +
           (3.14 * teraz_tm->tm_sec / 1800.0);
```

```
    /* --- Rysujemy wskazówkę --- */
    gdk_draw_line (piksmapa, obszar_rys->style->black_gc,
                   midx, midy,
                   midx+(int) (0.7 * promien * sin (dRadiany)),
                   midy-(int) (0.7 * promien * cos (dRadiany)));

    /*
     * --- Rysujemy wskazówkę godzinową
     */

    /* --- Przeliczamy godzinę na radiany --- */
    dRadiany = (teraz_tm->tm_hour % 12) * 3.14 / 6.0 +
               (3.14 * teraz_tm->tm_min / 360.0);

    /* --- Rysujemy wskazówkę --- */
    gdk_draw_line (piksmapa, obszar_rys->style->black_gc,
                   midx, midy,
                   midx + (int) (promien * 0.5 * sin (dRadiany)),
                   midy - (int) (promien * 0.5 * cos (dRadiany)));

    /* --- Ustawiamy prostokąt
     * --- Do okna będzie kopiowana cała piksmapa, więc
     * --- prostokąt ma rozmiary równe obszarowi okna
     */
    uakt_prostokat.x = 0;
    uakt_prostokat.y = 0;
    uakt_prostokat.width = obszar_rys->allocation.width;
    uakt_prostokat.height = obszar_rys->allocation.height;

    /* --- rysujemy obszar (powoduje wywołanie expose_event) --- */
    gtk_widget_draw (obszar_rys, &uakt_prostokat);
}

/*
 * configure_event
 *
 * tworzy nową, drugoplanową piksmapę o odpowiednich rozmiarach.
 * Jest wywoływana przy każdej zmianie rozmiarów okna. Musimy
 * zwolnić przydzielone zasoby
 */
static gint configure_event (GtkWidget *kontrolka,
                             GdkEventConfigure *zdarzenie);
{
```

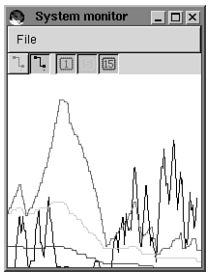
```
    /* --- zwalniamy bufor, jeśli go utworzyliśmy --- */
    if (piksmapa) {
        gdk_pixmap_unref (piksmapa);
    }

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

    return TRUE;
}

/*
 * expose_event
 *
 * Funkcja ta wywoływana jest po odsłonięciu okna albo po wywołaniu
 * funkcji 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
 *
 * Wychodzimy z aplikacji
 */

void zamknij ()
{
```

```
    gtk_exit (0);
}

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

    gtk_init (&argc, &argv);

    okno = gtk_window_new (GTK_WINDOW_TOPLEVEL);

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

    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, 200);
    gtk_box_pack_start (GTK_BOX (ypole), 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);

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

    /* --- Przerysowujemy obszar co sekundę --- */
    gtk_timeout_add (1000, Przerysuj, (gpointer) obszar_rys);
```

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

    return 0;

}
```

Monitor systemowy

Wykorzystamy ponownie niektóre z poprzednio napisanych funkcji, aby stworzyć niewielki program, który monitoruje komputer i wyświetla wykres obrazujący stan procesora oraz połączeń ethernetowych (patrz rysunek 10.4). Jedynym problemem jest pozyskanie informacji o systemie.



Rysunek 10.4. Monitor systemowy.

Wykorzystanie systemu plików /proc

Większość danych o stanie systemu Linux można odczytać z systemu plików /proc. „Pliki" w katalogu /proc można otwierać i czytać jak zwykłe pliki. Znajduje się w nich wiele informacji o komputerze - na przykład plik /proc/loadavg zawiera dane o obciążeniu procesora. Aby dowiedzieć się, do czego służą pozostałe pliki, można wpisać polecenie man proc, które wyświetli skrótową informację o wszystkich plikach w systemie /proc.

Jeśli wyświetlimy plik /proc/loadavg poleceniem cat, zobaczymy rezultaty podobne do zamieszczonych niżej:

```
[bystry@umysl /proc]$ cat loadavg
0.00 0.00 0.00 2/47 649
```

Pierwsze trzy liczby oznaczają średnie obciążenie procesora w ostatniej minucie, ostatnich 5 minutach i ostatnich 15 minutach. Jeśli wydamy polecenie w z linii poleceń Linuksa, otrzymamy mniej więcej następujące rezultaty:

```
4:15poleceniem  up  3:44,  4 users,  load average: 0.00, 0.00, 0.00
```

W powyższej linii średnie obciążenie jest określone przez ostatnie trzy liczby (Hmmm... komputer nie jest zbyt zapracowany).

Innym przykładowym plikiem w systemie /proc jest /proc/net/dev, zawierający informacje o pakietach wysyłanych i odbieranych przez wszystkie urządzenia w systemie. W komputerze autora rezultaty przeglądania pliku wyglądają następująco:

```
Inter-|   Receive
 face |packets    errs    drop    fifo    frame | packets    errs    drop    fifo    colls    car-
rier
    lo:      103       0       0       0       0         103       0       0       0        0
0
  eth0:    9600       0       0       0       0        6487       0       0       0        0
0
```

Oczywiście, przy każdym przeglądaniu pliku liczba wysłanych i odebranych pakietów będzie inna. Jeśli wpiszemy to samo polecenie trochę później, otrzymamy:

```
Inter-|   Receive
 face |packets    errs    drop    fifo    frame | packets    errs    drop    fifo    colls    car-
rier
    lo:      103       0       0       0       0         103       0       0       0        0
0
  eth0:    9600       0       0       0       0        6487       0       0       0        0
0
```

Urządzenie, które będziemy monitorować, to eth0:. Jest to karta Ethernet w komputerze; chcemy wiedzieć, jaki w danym momencie panuje na niej ruch. Możemy obliczyć aktualne natężenie ruchu, porównując bieżące dane o pakietach z danymi pochodzącymi z poprzedniego odczytu pliku.

Opis

Aplikacja korzysta z czasomierza, aby uaktualniać ekran co kilka sekund. Odczytuje statystyki z systemu plików /proc i wyświetla informacje w postaci wykresu, na którym każdy element jest zaznaczony odmien-

nym kolorem. Wykres można konfigurować przy pomocy przycisków na pasku narzędziowym.

Autor przeprowadził kilka testów i zauważył, że wykres obciążenia sieci miał tendencje do znacznych fluktuacji, co utrudniało zmierzenie rzeczywistej przepustowości. Pomocne okazało się dodanie kolejnego przycisku, który wyświetla średnią przepustowość. Średnia jest obliczana na podstawie ostatnich N próbek i daje bardziej zrównoważony wykres.

Program monitora systemowego jest podzielony na kilka modułów. W każdym z nich umieszczono specyficzny kod, który wykonuje swoją część pracy na rzecz aplikacji. Opis zaczniemy od pliku interfejs.c.

interfejs.c

interfejs.c tworzy okno aplikacji, pasek narzędziowy (z piksmapami) oraz niewielkie menu. Posiada także procedury używane przez inne funkcje, które sprawdzają stan interfejsu użytkownika. Można na przykład wywołać funkcję WcisnietoPrzyciskCPU15, aby sprawdzić, czy użytkownik wcisnął przycisk paska narzędziowego, który wyświetla obciążenie procesora w ostatnich 15 minutach. Rozwiązanie takie jest lepsze, niż używanie globalnej zmiennej pasek_cpu15 na zewnątrz pliku (rodzaj kapsułkowania danych w języku C). Zwróćmy uwagę, że przyciski paska narzędziowego mają różne kolory. Kolory przycisków odpowiadają kolorom na wykresie, co ułatwia użytkownikowi włączanie i wyłączanie różnych elementów wykresu. Tego typu wizualne wskazówki pomagają powiązać elementy wykresu z przyciskami paska narzędziowego.

```
/*
 * Interfejs przykładowej aplikacji GUI.
 *
 * Autor: Eric Harlow
 * Plik: frontend.c
 *
 */

#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <gtk/gtk.h>

/*
 * --- Prototypy funkcji
 */
GtkWidget *UtworzObszarRysunkowy ();
```

```
static void UtworzGlowneOkno ();
void UtworzPasek (GtkWidget *glowne_ypole);
void UstawPrzyciskPaska (char *szPrzycisk, int nStan);
void ZaznaczMenu (GtkWidget *kontrolka, gpointer dane);
void OdznaczMenu (GtkWidget *kontrolka, gpointer dane);
void UstawPrzyciskMenu (char *szPrzycisk, int nStan) ;
GtkWidget *UtworzKontrolkeZXpm (GtkWidget *okno, gchar **dane_xpm);
GtkWidget *UtworzElementMenu (GtkWidget *menu,
                              char *szNazwa,
                              char *szSkrot,
                              char *szPodpowiedz,
                              GtkSignalFunc funkcja,
                              gpointer dane);
GtkWidget *UtworzZaznaczalnyElement (GtkWidget *menu,
                                     char *szNazwa,
                                     GtkSignalFunc funkcja,
                                     gpointer dane);
GtkWidget *UtworzPodmenu (GtkWidget *pasek, char *szNazwa);
GtkWidget *UtworzPodmenuPaska (GtkWidget *menu, char *szNazwa);
void Przerysuj ();

/*
 * --- Zmienne globalne
 */

GtkWidget          *glowne_okno;
GtkTooltips        *podpowiedzi;
GtkAccelGroup      *grupa_skrotow;
GtkWidget          *pasek_narz;

static GtkWidget   *pasek_cpu1;
static GtkWidget   *pasek_cpu5;
static GtkWidget   *pasek_cpu15;
static GtkWidget   *pasek_siec;
static GtkWidget   *pasek_siec_srednia;

/*
 * --- Bitmapa dla przycisku "przepustowość sieci"
 */
static const gchar *xpm_siec[] = {
"16 16 4 1",
"  c None",
```

```
"B c #888888",
"              ",
"              ",
"  BBB           ",
"  BBBBBBB       ",
"  BBB   B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B  BBB  ",
"        BBBBBB  ",
"          BBB   ",
"              ",
"              ",
};

/*
 * --- Bitmap dla przycisku "średnia przepustowość sieci"
 */

static const gchar *xpm_siec_srednia[] = {
"16 16 4 1",
"  c None",
"B c #000000",
"              ",
"              ",
"  BBB           ",
"  BBBBBBB       ",
"  BBB   B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B       ",
"        B  BBB  ",
"        BBBBBB  ",
"          BBB   ",
```

```
"                ",
"                ",
};

/*
 * --- Bitmapa dla przycisku "obciążenie cpu w ostatniej minucie"
 */
static const char *xpm_cpu1[] = {
"16 16 2 1",
"  c None",
"B c #FF0000",
"                ",
"   B B B B B B  ",
"  BBBBBBBBBBBB ",
" BB           B ",
"  B     B     BB",
" BB    BB     B ",
"  B     B     BB",
" BB     B     B ",
"  B     B     BB",
" BB     B     B ",
"  B     B     BB",
" BB    BBB    B ",
"  B           BB",
"  BBBBBBBBBBBB ",
"   B B B B B B  ",
"                ",
};

/*
 * --- Bitmapa dla przycisku "obciążenie cpu w ostatnich 5 minutach"
 */
static const char *xpm_cpu5[] = {
"16 16 2 1",
"  c None",
"B c #00FF00",
"                ",
"   B B B B B B  ",
"  BBBBBBBBBBBB ",
" BB           B ",
"  B   BBBBB   BB",
```

```
" BB   B       B ",
"  B   B       BB",
" BB   BBBB    B ",
"  B       B   BB",
" BB       B   B ",
"  B   B   B   BB",
" BB    BBB    B ",
"  B           BB",
"  BBBBBBBBBBBBB ",
"   B B B B B B  ",
"                ",
};

/*
 * --- Bitmapa dla przycisku "obciążenie cpu w ostatnich 15 minutach”
 */
static const char *xpm_cpu15[] = {
"16 16 2 1",
"  c None",
"B c #0000FF",
"                ",
"   B B B B B B  ",
"  BBBBBBBBBBBBB ",
" BB           B ",
"  B  B  BBBBB BB",
" BB BB  B     B ",
"  B  B  B     BB",
" BB  B  BBBB  B ",
"  B  B      B BB",
" BB  B      B B ",
"  B  B  B   B BB",
" BB BBB  BBB  B ",
"  B           BB",
"  BBBBBBBBBBBBB ",
"   B B B B B B  ",
"                ",
};

/*
 * PrzerysujWykres
 *
```

```
 * Wywoływana wtedy, kiedy użytkownik przełącza przyciski na
 * pasku narzędziowym, aby zmienić informacje wyświetlane na wykresie.
 * Wywołuje funkcję "Przerysuj", aby wymusić przerysowanie
 * wykresu.
 */
void PrzerysujWykres (GtkWidget *kontrolka, gpointer dane)
{
   Przerysuj ();
}

/*
 * WcisnietoPrzyciskSiec
 *
 * Czy przycisk "sieć" na pasku narzędziowym jest wciśnięty?
 */
int WcisnietoPrzyciskSiec ()
{
   return (GTK_TOGGLE_BUTTON (pasek_siec)->active);
}

/*
 * WcisnietoPrzyciskSiecSrednia
 *
 * Czy przycisk "średnia sieci" na pasku narzędziowym jest wciśnięty?
 */
int WcisnietoPrzyciskSiecSrednia ()
{
   return (GTK_TOGGLE_BUTTON (pasek_siec_srednia)->active);
}

/*
 * WcisnietoPrzyciskCPU1
 *
 * Czy przycisk "CPU-1" na pasku narzędziowym jest wciśnięty?
 */
int WcisnietoPrzyciskCPU1 ()
{
   return (GTK_TOGGLE_BUTTON (pasek_cpu1)->active);
}

/*
 * WcisnietoPrzyciskCPU5
```

```
 *
 * Czy przycisk "CPU-5" na pasku narzędziowym jest wciśnięty?
 */
int WcisnietoPrzyciskCPU5 ()
{
   return (GTK_TOGGLE_BUTTON (pasek_cpu5)->active);
}

/*
 * WcisnietoPrzyciskCPU15
 *
 * Czy przycisk "CPU-15" na pasku narzędziowym jest wciśnięty?
 */
int WcisnietoPrzyciskCPU15 ()
{
   return (GTK_TOGGLE_BUTTON (pasek_cpu15)->active);
}

/*
 * KoniecProgramu
 *
 * Wyjście z programu
 */
void KoniecProgramu ()
{
   gtk_main_quit ();
}

/*
 * UtworzGlowneOkno
 *
 * Tworzy główne okno i związane z nim menu/pasek narzędziowy.
 */
static void UtworzGlowneOkno ()
{
   GtkWidget    *kontrolka;
   GtkWidget    *glowne_ypole;
   GtkWidget    *pasekmenu;
   GtkWidget    *menu;
   GtkWidget    *elmenu;

   /* --- Tworzymy główne okno --- */
```

```
glowne_okno = gtk_window_new(GTK_WINDOW_TOPLEVEL);

/* --- Nie pozwalamy na zmianę rozmiarów okna --- */
gtk_window_set_policy (GTK_WINDOW (glowne_okno),
                       FALSE, FALSE, TRUE);

/* --- Tytu³ --- */
gtk_window_set_title (GTK_WINDOW (glowne_okno),
                      "Monitor systemowy");
gtk_container_border_width (GTK_CONTAINER (glowne_okno), 0);

/* --- Tworzymy tablicę skrótów klawiszowych --- */
grupa_skrotow = gtk_accel_group_new ();
gtk_accel_group_attach (grupa_skrotow, GTK_OBJECT (glowne_okno));

/* --- Główne okno musi sprawdzać sygnał "destroy" --- */
gtk_signal_connect (GTK_OBJECT (glowne_okno), "destroy",
                    GTK_SIGNAL_FUNC(KoniecProgramu), NULL);

/* --- Tworzymy pionowe pole pakujące dla menu/paska narz. --- */
glowne_ypole = gtk_vbox_new (FALSE, 0);

/* --- Wyświetlamy pionowe pole pakujące --- */
gtk_container_add (GTK_CONTAINER (glowne_okno), glowne_ypole);

gtk_widget_show (glowne_ypole);
gtk_widget_show (glowne_okno);

/* --- Pasek menu --- */
pasekmenu = gtk_menu_bar_new ();
gtk_box_pack_start (GTK_BOX (glowne_ypole), pasekmenu,
                    FALSE, TRUE, 0);
gtk_widget_show (pasekmenu);

/* ------------------
   --- Menu Plik ---
   ------------------ */
menu = UtworzPodmenuPaska (pasekmenu, "Plik");

elmenu = UtworzElementMenu (menu, "Zakończ", "",
          "Czy istnieje bardziej wymowna opcja?",
          GTK_SIGNAL_FUNC (KoniecProgramu), "zakończ");

/* --- Tworzymy pasek narzędziowy --- */
```

```
    UtworzPasek (glowne_ypole);

    /* --- Tworzymy obszar do wyświetlania informacji --- */
    kontrolka = UtworzObszarRysunkowy ();
    gtk_box_pack_start (GTK_BOX (glowne_ypole), kontrolka,
                        TRUE, TRUE, 0);
}

/*
 * UtworzPasek
 *
 * Tworzy pasek narzędziowy, którego przyciski będą włączały i
 * wyłączały elementy wykresu w obszarze rysunkowym.
 */
void UtworzPasek (GtkWidget *glowne_ypole)
{
    /* --- Tworzymy pasek narzędziowy i dodajemy go do okna --- */
    pasek_narz = gtk_toolbar_new (GTK_ORIENTATION_HORIZONTAL,
                                  GTK_TOOLBAR_ICONS);
    gtk_box_pack_start (GTK_BOX (glowne_ypole), pasek_narz,
                        FALSE, TRUE, 0);
    gtk_widget_show (pasek_narz);

    /* --- Informacja o przepływie pakietów --- */
    pasek_siec = gtk_toolbar_append_element (GTK_TOOLBAR
(pasek_narz),
      GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
      NULL,
      "Sieć", "Sieć", "Sieć",
      UtworzKontrolkeZXpm (glowne_ypole, (gchar **) xpm_siec),
      (GtkSignalFunc) PrzerysujWykres,
      NULL);

    /* --- Informacja o średnim przepływie pakietów --- */
    pasek_siec_srednia = gtk_toolbar_append_element (
      GTK_TOOLBAR (pasek_narz),
      GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
      NULL,
      "Sieć (średnia)", " Sieć (średnia)", " Sieć (średnia)",
      UtworzKontrolkeZXpm (glowne_ypole,(gchar **) xpm_siec_srednia),
      (GtkSignalFunc) PrzerysujWykres,
      NULL);
```

```
    /* --- Niewielki odstęp --- */
    gtk_toolbar_append_space (GTK_TOOLBAR (pasek_narz));

    /* --- Informacja o wykorzystaniu CPU w ostatniej minucie --- */
    pasek_cpu1 = gtk_toolbar_append_element (GTK_TOOLBAR
(pasek_narz),
      GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
      NULL,
      "CPU 1", "CPU 1", "CPU 1",
      UtworzKontrolkeZXpm (glowne_ypole, (gchar **) xpm_cpu1),
      (GtkSignalFunc) PrzerysujWykres,
      NULL);

    /* --- Informacja o wykorzystaniu CPU w ostatnich 5 minutach -- */
    pasek_cpu5 = gtk_toolbar_append_element (GTK_TOOLBAR
(pasek_narz),
      GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
      NULL,
      "CPU 5", "CPU 5", "CPU 5",
      UtworzKontrolkeZXpm (glowne_ypole, (gchar **) xpm_cpu5),
      (GtkSignalFunc) PrzerysujWykres,
      NULL);

    /* -- Informacja o wykorzystaniu CPU w ostatnich 15 minutach -- */
    pasek_cpu15 = gtk_toolbar_append_element (
      GTK_TOOLBAR (pasek_narz),
      GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
      NULL,
      "CPU 15", "CPU 15", "CPU 15",
      UtworzKontrolkeZXpm (glowne_ypole, (gchar **) xpm_cpu15),
      (GtkSignalFunc) PrzerysujWykres,
      NULL);
}

/*
 * main
 *
 * --- Program zaczyna się tutaj
 */
int main(int argc, char *argv[])
{
    gtk_init (&argc, &argv);
```

```
    podpowiedzi = gtk_tooltips_new();

    UtworzGlowneOkno ();

    gtk_main();

    return 0;
}
```

urzadzenia.c

W pliku urzadzenia.c znajdują się procedury, które przechowują urządzenia oraz dane liczbowe, na podstawie których tworzony jest wykres. Urządzenia są przechowywane na liście GSList (patrz rozdział 2, „GLIB"). Procedury te umożliwiają przechowywanie informacji o urządzeniu w postaci nazwy urządzenia i zbioru wartości, zamiast wykorzystywania odrębnej struktury danych dla każdego urządzenia. Typ typUrzadzenie, wykorzystywany w urzadzenie.c, jest zdefiniowany w urzadzenie.h.

```
/*
 * urzadzenie.h
 * Definuje typUrzadzenie
 */
#define PRZYROST 1
#define RZECZYWISTA 2

#define MAKS_WARTOSCI 200

typedef long typStaraWartosc;

typedef struct {
   char        *sNazwa;          /* --- Nazwa urządzenia --- */
   int         nTyp;             /* --- Typ wartości: PRZYROST
                                        albo RZECZYWISTA --- */
   typStaraWartosc nOstatnia;    /* --- Ostatnia wartość
                                       (dla PRZYROSTU) --- */
   typStaraWartosc nMaks;        /* --- Wartość maksymalna --- */
   typStaraWartosc nWartosci [MAKS_WARTOSCI];
                                 /* --- Zapamiętane wartości --- */
} typUrzadzenie;

/*
 * Plik: urzadzenie.c
```

```
 * Autor: Eric Harlow
 *
 */

#include <stdio.h>
#include <strings.h>
#include <gtk/gtk.h>
#include "urzadzenie.h"

static GSList *lista_urz = NULL;

/*
 * SzukajUrzadzenia
 *
 * Szuka urządzenia na liście
 */
typUrzadzenie *SzukajUrzadzenia (char *sNazwa)
{
   GSList             *wezel;
   typUrzadzenie   *urz;

   /* --- Przechodzimy przez listę urządzeń. --- */
   for (wezel = lista_urz; wezel; wezel = wezel->next) {

      /* --- Pobieramy dane --- */
      urz = (typUrzadzenie *) wezel->data;

      /* --- Czy właśnie tego szukamy? --- */
      if (!strcmp (urz->sNazwa, sNazwa)) {
         return (urz);
      }
   }
   return (NULL);
}

/*
 * DodajUrzadzenie
 *
 * Dodaje nowe urządzenie do listy urządzeń i zwraca
 * je, aby można je było zainicjować
 */
typUrzadzenie *DodajUrzadzenie ()
{
```

```
    typUrzadzenie *urz;

    /* --- tworzymy nowe urządzenie --- */
    urz = (typUrzadzenie *) g_malloc (sizeof (typUrzadzenie));

    /* --- Dodajemy je do listy --- */
    lista_urz = g_slist_append (lista_urz, urz);

    /* --- Zwracamy nowy element listy --- */
    return (urz);
}

/*
 * UaktualnijUrzadzenie
 *
 * Uaktualnia informacje o urządzeniu.  Jeśli urządzenie obecnie
 * nie istnieje, zostanie dodane i zainicjowane.
 *

 * sNazwaUrz - nazwa urządzenia.  "eth0:", "cpu1", itd.
 * nWartosc - wartość w tym momencie.
 * nTyp - PRZYROST/RZECZYWISTA.
 *        RZECZYWISTA oznacza wartość "tak, jak jest"
 *        PRZYROST oznacza, że wartość odnosi się do poprzedniej
 *              wartości i należy obliczyć rzeczywistą wartość,
 *              mając to na względzie
 * nMaks - Jest to maksymalna wyświetlana wartość
 */
void UaktualnijUrzadzenie (char *sNazwaUrz, long nWartosc,
                           int nTyp, long nMaks)
{
    typUrzadzenie  *urz;
    int        i;

    /* --- Nie podano nazwy? Wracamy... --- */
    if (sNazwaUrz == NULL) return;

    /* --- Szukamy urządzenia według nazwy --- */
    urz = SzukajUrzadzenia (sNazwaUrz);

    /* --- Czy mamy już dane urządzenia? --- */
    if (urz) {

        /* --- Po prostu dodajemy wartość --- */
```

```
      NowaWartosc (urz, nWartosc, FALSE);
   } else {
      /* --- Nie mamy urządzenia! --- */

      /* --- Tworzymy nowe urządzenie --- */
      urz = DodajUrzadzenie ();

      /* --- Inicjujemy wartości --- */
      urz->sNazwa = strdup (sNazwaUrz);
      urz->nTyp = nTyp;
      urz->nMaks = nMaks;

      /* --- Czyścimy wartości --- */
      for (i = 0; i < MAKS_WARTOSCI; i++) {

         urz->nWartosci[i] = (typStaraWartosc) 0;
      }

      /* --- Uaktualniamy bieżącymi danymi --- */
      NowaWartosc (urz, nWartosc, TRUE);
   }
}

/*
 * UaktualnijIstniejaceUrzadzenie
 *
 * Jeśli wiemy, że urządzenie istnieje, korzystamy z tej procedury,
 * ponieważ nie wymaga wpisywania tylu parametrów (i odkładania ich
 * na stos/zdejmowania ze stosu)
 */
void UaktualnijIstniejaceUrzadzenie (char *sNazwaUrz, long nWartosc)
{
   typUrzadzenie  *urz;

   /* --- Nazwa nie może być pusta --- */
   if (sNazwaUrz == NULL) return;

   /* --- Szukamy urządzenia według nazwy --- */
   urz = SzukajUrzadzenia (sNazwaUrz);

   /* --- Jeśli urządzenie istnieje... --- */
   if (urz) {

      /* --- Dodajemy wartość --- */
```

```
      NowaWartosc (urz, nWartosc, FALSE);
   }
}

/*
 * NowaWartosc
 *
 * Procedura dodaje wartość do listy wartości, przechowywanych
 * dla danego urządzenia. Musi także przesunąć wartości o 1 za
 * każdym razem, kiedy dodajemy kolejną wartość.
 *
 * urz - urządzenie, dla którego należy dodać wartość
 * wartosc - wartość dodawana dla urządzenia
 * bInicjuj - Czy jest to pierwsza dodawana wartość?
 */
void NowaWartosc (typUrzadzenie *urz,
                  typStaraWartosc wartosc,
                   int bInicjuj)
{
   int    i;

   /* --- Przesuwamy wartości w dół --- */
   for (i = MAKS_WARTOSCI - 2; i >=0; i--) {

      urz->nWartosci[i+1] = urz->nWartosci[i];
   }
   /* --- Jeśli to NIE JEST inicjacja --- */
   if (!bInicjuj) {

      /* --- Dodajemy nową wartość na koniec --- */
      if (urz->nTyp == PRZYROST) {

         /* --- Nowa wartość to przyrost, czyli różnica
          *     pomiędzy poprzednią a bieżącą wartością
          */
         urz->nWartosci[0] = wartosc - urz->nOstatnia;
      } else {
         /* --- Zapisujemy wartość tak, jak jest --- */
         urz->nWartosci[0] = wartosc;
      }
   }

   /* --- Zapamiętujemy ostatnią wartość --- */
```

```
    urz->nOstatnia = wartosc;

    /* --- Obliczamy wartość maksymalną --- */
    if (urz->nWartosci[0] > urz->nMaks) {
       urz->nMaks = urz->nWartosci[0];
    }
}
```

sys.c

Plik sys.c zawiera procedury, które analizują pliki /proc (w celu uzyskania informacji o obciążeniu sieci i procesora). Informacje o każdym urządzeniu są uaktualniane poprzez wywołania funkcji z pliku urzadzenie.c. Można łatwo dodać inne analizatory, posługując się istniejącym kodem jako przykładem. Procedura dodawania nowego urządzenia polega na odczytaniu informacji, zanalizowaniu jej i uaktualnieniu informacji o urządzeniu (przy pomocy funkcji z pliku urzadzenie.c). Można potem odwoływać się do urządzenia poprzez uniwersalną strukturę danych.

```
/*
 * Plik: sys.c
 * Autor: Eric Harlow
 *
 */

#include <stdio.h>
#include <string.h>

#include <gtk/gtk.h>
#include "urzadzenie.h"

#define SEPARATORY " |\n"
void PobierzDaneOPakietach ();

/*
 * PobierzDaneOPakietach
 *
 * Pobiera informacje o pakietach przesyłanych przez sieć
 * Ethernet.  Dane o pakietach znajdują się w pliku /proc/net/dev,
 * jeśli komputer posiada system plików /proc.
 */
void PobierzDaneOPakietach ()
{
```

```
char *szNazwaPliku = "/proc/net/dev";
char szBufor[132];
int  nNumerLinii = 0;
FILE *wp;
char *sLeksem;
char *sNazwaUrz;
int  nNrSlowa;
long nPrzychodzace;
long nWychodzace;
typUrzadzenie *urz;

/* --- Otwieramy plik, aby pobrać informacje --- */
wp = fopen (szNazwaPliku, "r");
if (wp) {

   /* --- Dopóki są dane do odczytania --- */
   while (!feof (wp)) {

      nNumerLinii++;

      /* --- Wczytujemy linię tekstu --- */
      fgets (szBufor, sizeof (szBufor), wp);

      /* -- Po trzeciej linii są informacje o urządzeniach -- */
      if (nNumerLinii >= 3) {

         nNrSlowa = 0;
         sNazwaUrz = NULL;
         sLeksem = strtok (szBufor, SEPARATORY);
         while (sLeksem) {

            switch (nNrSlowa) {
               case 0:
                  sNazwaUrz = sLeksem;
                  break;
               case 1:
                  nPrzychodzace = atoi (sLeksem);
                  break;
               case 6:
                  nWychodzace = atoi (sLeksem);
                  break;
            }
```

```
                nNrSlowa++;
                sLeksem = strtok (NULL, SEPARATORY);
            }

            /* --- Znaleźliśmy nazwę urządzenia --- */
            if (sNazwaUrz) {

                /* --- Wyszukujemy urządzenie --- */
                urz = SzukajUrzadzenie (sNazwaUrz);

                /* --- Dodajemy je/uaktualniamy dane --- */
                if (urz) {
                    UaktualnijIstniejaceUrzadzenie (sNazwaUrz,
                        (long) (nPrzychodzace + nWychodzace));
                } else {
                    UaktualnijUrzadzenie (sNazwaUrz,
                        (long) (nPrzychodzace + nWychodzace),
                        PRZYROST, 1);
                }
            }
            szBufor[0] = (char) 0;
        }
    }

    /* --- Wszystko gotowe --- */
    fclose (wp);
  } else {

    /* --- Niestety, błąd! --- */
    printf ("Nie udało się otworzyć pliku %s.", szNazwaPliku);
  }
}

/*
 * PobierzDaneOCPU
 *
 * Odczytuje plik /proc/loadavg i analizuje zawarte w nim
 * informacje o procesorze.
 */
void PobierzDaneOCPU ()
{
  static char *szPlik = "/proc/loadavg";
  char szBufor[88];
```

```
    FILE *wp;
    float cpu1, cpu2, cpu3;
    long lcpu1, lcpu2, lcpu3;
    typUrzadzenie *urz;

    /* --- Otwieramy plik z danymi o CPU --- */
    wp = fopen (szPlik, "r");

    /* --- Jeśli otwarcie się powiodło... --- */
    if (wp) {

       fgets (szBufor, sizeof (szBufor), wp);

       /* --- Pobieramy liczby, określające obciążenie CPU --- */
       sscanf (szBufor, "%f %f %f", &cpu1, &cpu2, &cpu3);

       /* --- Portrzebny zakres to 1-100+, a nie .01 to 1.0+ --- */
       lcpu1 = cpu1 * 100;
       lcpu2 = cpu2 * 100;
       lcpu3 = cpu3 * 100;

       /* --- Wyszukujemy urządzenie --- */
       urz = SzukajUrzadzenia ("cpu1");
       if (urz) {

          /* --- Już istnieje, uaktualniamy urządzenie --- */

          UaktualnijIstniejaceUrzadzenie ("cpu1", lcpu1);
          UaktualnijIstniejaceUrzadzenie ("cpu5", lcpu2);
          UaktualnijIstniejaceUrzadzenie ("cpu15", lcpu3);
       } else {

          /* --- To pierwszy raz; tworzymy urządzenia --- */

          UaktualnijUrzadzenie ("cpu1", lcpu1, RZECZYWISTA, 100);
          UaktualnijUrzadzenie ("cpu5", lcpu2, RZECZYWISTA, 100);
          UaktualnijUrzadzenie ("cpu15", lcpu3, RZECZYWISTA, 100);
       }
       /* --- Sprzątamy. --- */
       fclose (wp);
    } else {
       printf ("Nie udało się otworzyć pliku %s.\n", szPlik);
    }
}
```

wykres.c

W pliku tym znajduje się cały kod operujący na kontrolce obszaru rysunkowego. Aktualne dane sprawdzamy za pośrednictwem nazw urządzeń, używanych w sys.c. Funkcje UaktualnijUrzadzenie i UaktualnijIstniejaceUrzadzenie tworzą urządzenie i związane z nim wartości liczbowe. Procedury w pliku wykres.c pobierają te dane, wywołując funkcje z pliku urzadzenie.c. Ponieważ plik wykres.c nie ma żadnej wiedzy na temat systemu /proc, można zmienić szczegóły implementacyjne (sys.c), gdyby zaszły zmiany w jądrze (mało prawdopodobne), albo w razie przenoszenia aplikacji do systemu operacyjnego, który nie posiada systemu plików /proc.

Wiele procedur przypomina te, które uaktualniały przykładowy zegar. Znaczącą różnicą jest jednak wykorzystanie kolorów podczas rysowania. Podczas uruchamiania aplikacji tworzymy kilka „piór" (są to w istocie struktury GdkGC), które pozwalają na rysowanie poszczególnych kolorów. Miejsce na pióra jest przydzielane na wstępie, aby przyspieszyć rysowanie (w przeciwnym przypadku musielibyśmy tworzyć nowy GdkGC przy każdym przerysowaniu ekranu). Pióra tworzymy w funkcji o nazwie PobierzPioro, dzięki czemu kod jest czytelniejszy, niż gdyby funkcja nosiła nazwę PobierzGdkGC.

```
/*
 * Plik: wykres.c
 * Autor: Eric Harlow
 *
 *
 */

#include <gtk/gtk.h>
#include "urzadzenie.h"

void RysujUrzadzenie (GtkWidget *obszar_rys, char *szName,
                      GdkGC *pen, int bSrednia);
int WcisnietoPrzyciskSiec ();
int WcisnietoPrzyciskSiecSrednia ();
int WcisnietoPrzyciskCPU15 ();
int WcisnietoPrzyciskCPU5 ();
int WcisnietoPrzyciskCPU1 ();
void Przerysuj ();

GtkWidget *obszar_rys;

typedef struct {
```

```
    GdkDrawable *piksmapa;
    GdkGC *gc;

} typGrafika;

static typGrafika *g;
static GdkGC *czarnePioro = NULL;
static GdkGC *czerwonePioro = NULL;
static GdkGC *niebieskiePioro = NULL;
static GdkGC *zielonePioro = NULL;
static GdkGC *szarePioro = NULL;

/*
 * NowaGrafika
 *
 * Tworzy nowy element danych graficznych, przechowujący piksmapę
 * i kontekst gc.
 */
typGrafika *NowaGrafika ()
{
    typGrafika *gfx;

    /* --- Przydzielamy pamięć --- */
    gfx = (typGrafika *) g_malloc (sizeof (typGrafika));

    /* --- Inicjujemy --- */
    gfx->gc = NULL;
    gfx->piksmapa = NULL;

    /* --- Zwracamy element gotowy do użycia --- */
    return (gfx);
}

/*
 * PobierzPioro
 *
 * Zwraca pióro na podstawie przekazanej struktury GdkColor
 * Pióro (po prostu GdkGC) jest tworzone i zwracane w postaci
 * gotowej do użycia.
 */
GdkGC *PobierzPioro (GdkColor *c)
{
```

```
    GdkGC *gc;

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

    /* --- Ustawiamy kolor pierwszego planu --- */
    gdk_gc_set_foreground (gc, c);

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

/*
 * NowyKolor
 *
 * Tworzymy kolor na podstawie listy parametrów i przydzielamy
 * mu miejsce
 */
GdkColor *NowyKolor (long czerwony, long zielony, long niebieski)
{
    /* --- Tworzymy strukturę koloru --- */
    GdkColor *c = (GdkColor *) g_malloc (sizeof (GdkColor));

    /* --- Wypełniamy ją --- */
    c->red = czerwony;
    c->green = zielony;
    c->blue = niebieski;

    gdk_color_alloc (gdk_colormap_get_system (), c);

    return (c);
}

/*
 * UaktualnijIPrzerysuj
 *
 * Procedura sprawdza najnowsze statystyki obciążenia sieci
 * oraz procesora i uaktualnia ekran na podstawie tych
 * informacji
 */
gint UaktualnijIPrzerysuj (gpointer dane)
{
    /* --- Pobieramy informacje o sieci --- */
```

```
    PobierzDaneOPakietach ();

    /* --- Pobieramy informacje o procesorze --- */
    PobierzDaneOCPU ();

    /* --- Przerysowujemy ekran --- */
    Przerysuj ();

    return (1);
}

/*
 * Przerysuj
 *
 * Uaktualnia ekran na podstawie najnowszych danych
 */
void Przerysuj ()
{
    GdkRectangle     uakt_prostokat;

    /* --- czyścimy piksmapę, aby móc na niej rysować --- */
    gdk_draw_rectangle (g->piksmapa,
            obszar_rys->style->white_gc,
            TRUE,
            0, 0,
            obszar_rys->allocation.width,
            obszar_rys->allocation.height);

    /* --- Jeśli użytkownik chce widzieć rzeczywiste
          obciążenie sieci ... --- */
    if (WcisnietoPrzyciskSiec ()) {
       RysujUrzadzenie (obszar_rys, "eth0:", szarePioro, 0);
    }
    /* -- Jeśli użytkownik chce widzieć średnie
         obciążenie sieci ... -- */
    if (WcisnietoPrzyciskSiecSrednia ()) {
       RysujUrzadzenie (obszar_rys, "eth0:", czarnePioro, 1);
    }
    /* -- Jeśli użytkownik chce widzieć średnie obciążenie
         procesora w ostatnich 15 minutach ... -- */
    if (WcisnietoPrzyciskCPU15 ()) {
       RysujUrzadzenie (obszar_rys, "cpu15", niebieskiePioro, 0);
    }
```

```
    /* -- Jeśli użytkownik chce widzieć średnie obciążenie
          procesora w ostatnich 5 minutach ... -- */
    if (WcisnietoPrzyciskCPU5 ()) {
       RysujUrzadzenie (obszar_rys, "cpu5", zielonePioro, 0);
    }
    /* -- Jeśli użytkownik chce widzieć średnie obciążenie
          procesora w ostatniej minucie ... -- */
    if (WcisnietoPrzyciskCPU1 ()) {
       RysujUrzadzenie (obszar_rys, "cpu1", czerwonePioro, 0);
    }

    /* --- Uaktualniamy ekran drugoplanową piksmapą --- */
    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);
}

/*
 * RysujUrzadzenie
 *
 * Rysuje wykres z informacjami o urządzeniu
 *
 * obszar_rys - kontrolka
 * szNazwa - nazwa monitorowanego urządzenia
 * pioro - GC z informacjami o kolorze
 * bSrednia - Znacznik uśredniania.  True => przeprowadzić uśrednianie
 */
void RysujUrzadzenie (GtkWidget *obszar_rys, char *szNazwa,
                      GdkGC *pioro, int bSrednia)
{
    typUrzadzenie  *urz;
    int    poprzx = 0;
    int    poprzy = 0;
    int    x = 0;
    int    y = 0;
    int    i;
    int    nOstatnia;

    /* --- Wyszukujemy urządzenie na podstawie nazwy --- */
```

```
urz = SzukajUrzadzenia (szNazwa);

/* --- Jeśli je znaleźliśmy --- */
if (urz) {

   /* --- Jeśli należy wykonać uśrednienie --- */
   if (bSrednia) {
      nOstatnia = MAKS_WARTOSCI-4;
   } else {
      nOstatnia = MAKS_WARTOSCI;
   }

   /* --- Rysujemy w poprzek kontrolki --- */
   for (i = 0;
      i < obszar_rys->allocation.width && i < nOstatnia; i++) {

      x = i;
      if (urz->nMaks != 0) {

         if (bSrednia) {

            y = ((urz->nWartosci[i] +
                  urz->nWartosci[i+1] +
                  urz->nWartosci[i+2] +
                  urz->nWartosci[i+3] +
                  urz->nWartosci[i+4]) *
                  obszar_rys->allocation.height) /
                  (urz->nMaks * 5);
         } else {
            y = (urz->nWartosci[i] *
             obszar_rys->allocation.height) / urz->nMaks;
         }
         y = obszar_rys->allocation.height - y;
      } else {
         y = 1;
      }

      if (i == 0) {
         poprzx = x;
         poprzy = y;
      }

      /* --- Rysujemy linię od poprzedniego do bieżącego
```

```
            --- punktu  --- */
          gdk_draw_line (g->piksmapa,
                         pioro,
                         poprzx, poprzy,
                         x, y);

          /* --- Następnym "poprzednim punktem" będzie bieżący -- */
          poprzx = x;
          poprzy = y;
        }
    } else {

        /* --- Nigdy nie powinno się zdarzyć! --- */
        printf ("Wskaźnik do urządzenia to NULL (%s)\n", szNazwa);
    }
}

/*
 * configure_event
 *
 * Wywoływana podczas tworzenia obszaru rysunkowego
 * i za każdym razem, kiedy jego rozmiar ulegnie zmianie.
 * Tworzy nową piksmapę-bufor o odpowiednich rozmiarach.
 */
static gint configure_event (GtkWidget *kontrolka,
                             GdkEventConfigure *zdarzenie)
{
    if (g == NULL) {
        g = NowaGrafika ();
    }

    /* --- Zwalniamy piksmapę --- */
    if (g->piksmapa) {
        gdk_pixmap_unref (g->piksmapa);
    }

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

    /* --- Jeśli jeszcze nie utworzyliśmy piór... --- */
```

```
    if (czarnePioro == NULL) {

      /* --- ...tworzymy kolorowe pióra --- */
      czarnePioro = PobierzPioro (NowyKolor (0, 0, 0));
      czerwonePioro = PobierzPioro (NowyKolor (0xffff, 0, 0));
      niebieskiePioro = PobierzPioro (NowyKolor (0, 0, 0xffff));
      zielonePioro = PobierzPioro (NowyKolor (0, 0xffff, 0));
      szarePioro = PobierzPioro (NowyKolor (0x9000, 0x9000, 0x9000));
    }

    /* --- Czyścimy obszar rysunkowy --- */
    gdk_draw_rectangle (g->piksmapa,
                        kontrolka->style->white_gc,
                        TRUE,
                        0, 0,
                        kontrolka->allocation.width,
                        kontrolka->allocation.height);

    return TRUE;
}

/*
 * expose_event
 *
 * Przerysowuje ekran przy pomocy piksmapy
 */
static gint expose_event (GtkWidget *kontrolka,
                          GdkEventExpose *zdarzenie)
{
  /* --- Kopiujemy piksmapę do okna --- */
  gdk_draw_pixmap (kontrolka->window,
          kontrolka->style->fg_gc[GTK_WIDGET_STATE (kontrolka)],
          g->piksmapa,
          zdarzenie->area.x, zdarzenie->area.y,
          zdarzenie->area.x, zdarzenie->area.y,
          zdarzenie->area.width, zdarzenie->area.height);

  return FALSE;
}

/*
 * zamknij
 *
```

```
 * Kończy działanie aplikacji
 */
void zamknij ()
{
   gtk_exit (0);
}

/*
 * UtworzObszarRysunkowy
 *
 * Tworzy kontrolkę obszaru rysunkowego i zwraca ją.
 */
GtkWidget *UtworzObszarRysunkowy ()
{
   /* --- Tworzymy obszar rysunkowy  --- */
   obszar_rys = gtk_drawing_area_new ();

   /* --- Nadajemy mu odpowiednie rozmiary --- */
   gtk_drawing_area_size (GTK_DRAWING_AREA (obszar_rys), 200, 200);

   /* --- Uwidaczniamy go --- */
   gtk_widget_show (obszar_rys);

   /* --- Musimy sprawdzaæ expose_event i configure_event --- */
   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);

   /* --- Wywołujemy funkcję co 2 sekundy --- */
   gtk_timeout_add (2000, UaktualnijIPrzerysuj, obszar_rys);

   /* --- Zwracamy kontrolkę, aby można było
         umieścić ją na ekranie --- */
   return (obszar_rys);
}
```

Podsumowanie

Połączenie możliwości GDK z kontrolką obszaru rysunkowego umożliwia tworzenie skomplikowanych, zawierających grafikę aplikacji. GDK posiada procedury pozwalające na rysowanie prostych kształtów wewnątrz kontrolki obszaru rysunkowego. Wykorzystanie w aplikacjach

techniki podwójnego buforowania pomaga wyeliminować migotanie, choć wymaga większego nakładu pracy. Aplikacje wyświetlające informacje graficzne powinny zawsze używać podwójnego buforowania, aby usunąć migotanie obrazu.