Maciej Przybylski, Jakub Możaryn

# Grafika

Materiały dydaktyczne, Ośrodek Kształcenia Na Odległość – OKNO

2009-08-14

# Spis treści

1 Pojęcia podstawowe – piórko, pędzelek, płótno....3
1.1 Figura geometryczna (TShape)....3
1.2 Komponent TPaintBox....3
1.3 Klasa TCanvas....3
1.4 Pióro (TPen)....4
1.5 Pędzel (TBrush)....4
2 Przykładowy program – rysowanie obiektów 2-wymiarowych....5
3 Obsługa plików graficznych....13
2.1 Rysunek (TImage)....13
2.2.2Komponenty niewizualne do obsługi plików graficznych....14
4. Operacje na obrazie – skalowanie, buforowanie, akceleracja....15
4.1. Kopiowanie fragmentu obrazu....15
4.2. Skalowanie....16
4.3. Operacje na pikselach – obrót, akceleracja....16
5. Przykładowy program – operacje na plikach JPEG....17

# *1 Pojęcia podstawowe – piórko, pędzelek, płótno*

## 1.1 Figura geometryczna (TShape)

Komponent `Shape` umożliwia dodawanie do formularza figur geometrycznych o różnych kształtach. Pozwala na rysowanie okręgów, elips, kwadratów i prostokątów (również z zaokrąglonymi rogami).

*Tabela 1. Właściwości komponentu TShape*

| **Właściwość** | **Opis** |
|---|---|
| `Brush` | Tzw. pędzel, wpływa na kolor i rodzaj tła. |
| `Pen` | Zmienia kolor i grubość krawędzi figur. |
| `Shape` | Określa rysowany kształt, może przyjmować następujące wartości:<br>`stCircle` - Okrąg.<br>`stEllipse` - Elipsa.<br>`stRectangle` - Prostokąt (figura domyślna).<br>`stRoundRect` - Prostokąt z zaokrąglonymi rogami.<br>`StRoundSquare` - Kwadrat z zaokrąglonymi rogami.<br>`stSquare` - Kwadrat. |

## 1.2 Komponent TPaintBox

Przy pomocy komponentu `PaintBox` można określić prostokątny obszar w postaci tzw. płótna (ang. canvas), stanowiącego miejsce w aplikacji na którym można dokonywać różne operacje graficzne 2D. Płótno to reprezentowane jest przez jego właściwość `Canvas`, będącą obiektem klasy TCanvas; klasa ta odpowiedzialna jest za większość operacji graficznych wykonywanych w środowisku Turbo C++.

## 1.3 Klasa TCanvas

Klasa `TCanvas` powala na zrealizowanie podstawowych operacji graficznych. Jej najważniejsze własności zostały zebrane w poniższej tabeli.

*Tabela 2. Główne właściwości klasy TCanvas*

| **Właściwość** | **Opis** |
|---|---|
| `Brush` | Zawiera kolor pędzla lub wzór stosowany do wypełniania figur. |
| `ClipRect` | Określa prostokątny wycinek płótna, do którego dodatkowo ograniczone jest tworzenie grafiki. Właściwość tylko do odczytu. |
| `CopyMode` | Określa sposób tworzenia grafiki w kontekście bieżącej zawartości obszaru (normalnie, inwersyjne, `xor` itd.) |
| `Font` | Określa rodzaj czcionki stosowanej przez płótno do wypisywania tekstu. |
| `Handle` | Zawiera uchwyt, stanowiący kontekst urządzenia (HDC) płótna, stosowany podczas bezpośrednich wywołań funkcji API. |
| `Pen` | Określa styl i kolor linii rysowanych na płótnie. |
| `PenPos` | Zawiera bieżącą pozycję rysowania wyrażoną przez współrzędne `x` i `y`. |
| `Pixels` | Reprezentuje poszczególne piksele płótna w postaci macierzy |

Własność `Font` jest taka sama jak dla wszystkich komponentów wizualnych. Własności `Pen` i `Brush` wymagają dokładniejszego wytłumaczenia.

## 1.4 Pióro (TPen)

Pióro definiuje obiekt, którego przeznaczeniem jest rysowanie linii. Może to być prosta linia rysowana od jednego punktu do drugiego, lub krawędź rysowana wokół prostokątów, elips i wielokątów. Dostęp do pióra, będącego obiektem klasy `TPen`, następuje poprzez właściwość `Pen` klasy `TCanvas`. Właściwości klasy `TPen` zostały przedstawione w tabeli 3.

*Tabela 3. Właściwości klasy Tpen*

| Właściwość | Opis |
|---|---|
| `Color` | Ustala kolor linii. |
| `Handle` | Zawiera kontekst urządzenia pióra (HDC). Stosowany podczas bezpośrednich odwołań do GDI. |
| `Mode` | Określa sposób w jaki linia będzie rysowana w kontekście bieżącej zawartości obszaru (normalny, inwersyjny, `xor`, itd.). |
| `Style` | Określa styl pióra. Może to być styl ciągły, kropkowy, kreskowy, kropkowo-kreskowy, czysty lub inny. |
| `Width` | Zawiera grubość linii w pikselach. |

## 1.5 Pędzel (TBrush)

Pędzel jest obiektem służącym do wypełniania wnętrza obszarów – wszystkie rysowane elipsy, prostokąty, wielokąty itp. zostaną wypełnione zgodnie z bieżącymi ustawieniami pędzla. Ustawienia te nie ograniczają się tylko do koloru, lecz obejmują również wzór („deseń") wypełnienia, bądź to w jednej z przedefiniowanych postaci, bądź też w postaci określonej przez wskazaną bitmapę. Dostęp do wypełnienia, będącego obiektem klasy `TBrush`, następuje poprzez właściwość `Brush` klasy `TCanvas`. Właściwości klasy `TBrush` zostały przedstawione w tabeli 4.

*Tabela 4. Właściwości klasy TBrush*

| Właściwość | Opis |
|---|---|
| `Bitmap` | Bitmapa określająca wzór wypełnienia. W przypadku Windows 95 bitmapa ta nie może przekroczyć rozmiaru 8×8 pikseli. |
| `Color` | Kolor wypełnienia. |
| `Handle` | Kontekst urządzenia (HDC) pędzla. Stosowany przy bezpośrednich odwołaniach do GDI. |
| `Style` | Styl pędzla (jednolity, wymazujący (*clear*) lub jeden z predefiniowanym wzorów). |

Wszelkie operacje graficzne na „płótnie" wykonywane są przez odpowiednie metody klasy `TCanvas`. Podstawowe metody klasy `TCanvas` zostały zebrane w tabeli 5.

*Tabela 5. Główne metody klasy TCanvas*

| Metoda | Opis |
|---|---|
| `Arc` | Rysuje łuk korzystając z bieżących ustawień pióra. |
| `BrushCopy` | Wyświetla bitmapę z przezroczystym tłem. |
| `CopyRect` | Kopiuje fragment obrazu na płótno. |
| `Draw` | Kopiuje obraz z pamięci na płótno. |
| `Ellipse` | Rysuje elipsę, korzystając z bieżących ustawień pióra (dla krawędzi) i pędzla (dla wypełnienia wnętrza). |
| `FloodFill` | Wypełnia obszar na płótnie zgodnie z bieżącymi ustawieniami pędzla. |
| `LineTo` | Rysuje odcinek linii prostej od bieżącego punktu do pozycji wyznaczonej przez parametry x i y. |
| `MoveTo` | Ustawia nową pozycję bieżącego punktu rysowania. |
| `Pie` | Rysuje wycinek koła – zgodnie z bieżącymi ustawieniami pióra i pędzla. |
| `Polygon` | Rysuje wielokąt na podstawie danych z tablicy punktów i wypełnia go zgodnie z bieżącymi ustawieniami pędzla. |
| `Polyline` | Rysuje linię łamaną na podstawie punktów z tablicy i bieżących ustawień pióra. `Linia nie jest automatycznie domykana`. |
| `Rectangle` | Rysuje prostokąt, korzystając z bieżących ustawień pióra (dla krawędzi) i pędzla (dla wypełnienia wnętrza). |
| `RoundRect` | Rysuje wypełniony prostokąt z zaokrąglonymi narożnikami. |
| `StretchDraw` | Kopiuje bitmapę z pamięci na płótno. Bitmapa jest rozciągana lub skracana w zależności od rozmiaru obszaru przeznaczenia. |
| `TextExtent` | Zwraca szerokość i wysokość (w pikselach) łańcucha przekazanego przez parametr `Text`. Szerokość jest obliczana na podstawie bieżącej czcionki płótna. |
| `TextHeight` | Zwraca wysokość (w pikselach) łańcucha przekazanego przez parametr `Text`. Szerokość jest obliczana na podstawie bieżącej czcionki płótna. |
| `TextOut` | Wypisuje tekst na płótnie od określonego położenia, korzystając z bieżącej czcionki. |
| `TextRect` | Wypisuje tekst w ramach ograniczonego obszaru. |

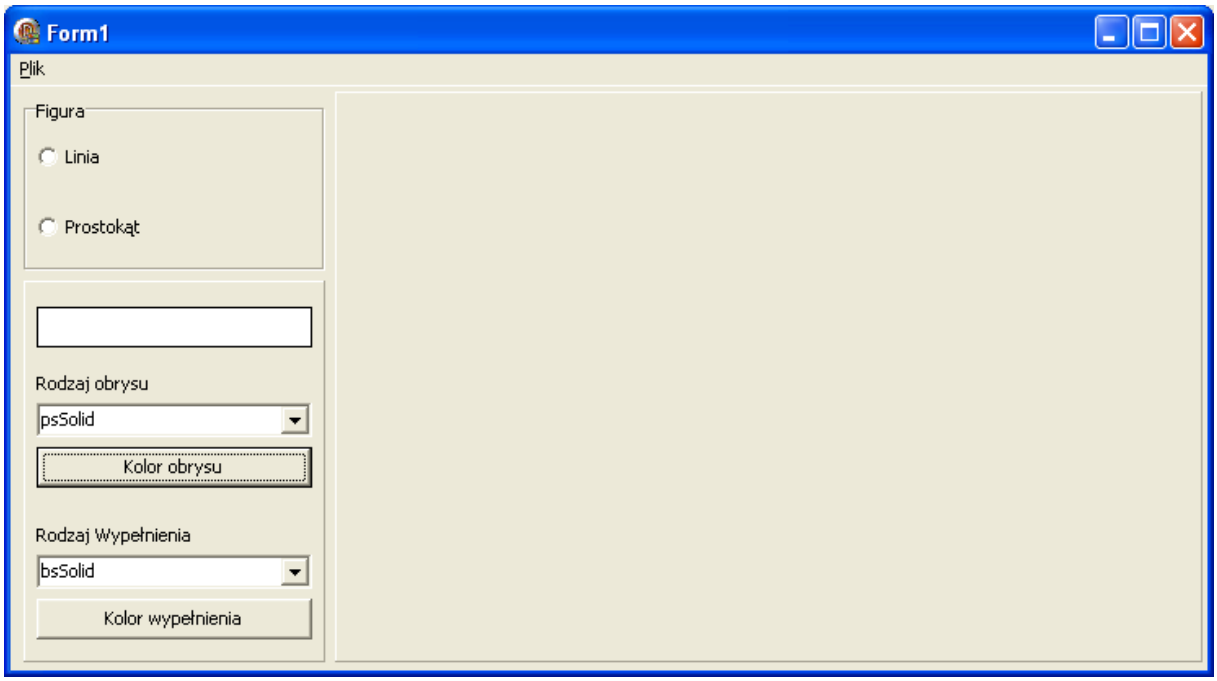
## *2 Przykładowy program – rysowanie obiektów 2-wymiarowych*

Program powinien umożliwiać rysowanie na zadanym polu prostych figur i linii. Powinien też umożliwić zapisanie uzyskanej figury w pliku, oraz odczytanie figury z pliku do programu i jej wyświetlenie. Należy wykorzystać komponenty `PaintBox, Shape, ColorDialog`.

1. Otwieramy nową aplikację *File>NewApplication*

2. Zapisujemy projekt przy pomocy *File>Save All* w katalogu gdzie będzie kompilowany nasz projekt

3. Na początku zaprojektujemy główny formularz (rys. 1).




**Rysunek 23.** Postać głównego formularza

Dodajemy odpowiednie komponenty (nie wszystkie komponenty zostały tu opisane) w kolejności

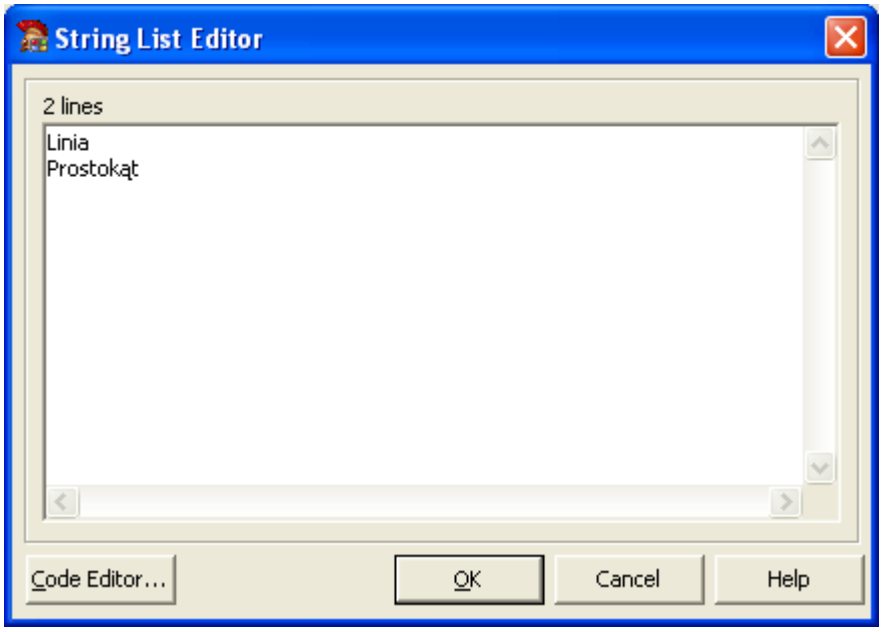
- Komponent do tworzenia głównego menu – `MainMenu` (paleta Standard)
- Panel zawierający wybór figury rysowanej na ekranie `RadioGroup` (paleta Standard).
- Pole graficzne pozwalające na podgląd wybranego obrysu i wypełnienia, oraz ich kolorów dla zadanej figury `Shape` (paleta Additional).
- Komponenty pozwalające na wybranie odpowiedniej opcji dotyczącej stylu piórka (obrysu) i stylu pędzelka (wypełnienia) `ComboBox` (paleta Standard).
- Etykiety opisujące `Label` (paleta Standard).
- Przyciski do wyboru koloru obrysu i wypełnienia `Button` (paleta Standard).
- Komponenty wybierające kolor dla obrysu i wypełnienia `ColorDialog` (paleta Dialogs).
- Panel na którym będzie znajdowało się pole do rysowania `Panel` (paleta Standard)
- Pole do rysowania figury `PaintBox` (paleta System).
- Komponent wybierający plik do zapisu rysunku `SaveDialog` (paleta Dialogs).
- Komponent wybierający plik do odczytu rysunku `OpenDialog` (paleta Dialogs).

## 4. Ustawienie podstawowych własności komponentów

### 4.1. Projektowanie panelu wyboru opcji

Dodać odpowiednie opcje do komponentu wyboru opcji `RadioGroup`

(rgDostepnosc). Opcje do tego komponentu dodaje się edytując własność Items. Korzystając z edytora kolejnych elementów (tzw. *String List Editor*) utworzyć 2 elementy określające rodzaj rysowanej figury: 'Linia', 'Prostokąt' (rys. 24).




**Rysunek 24**. Edytor panelu wyboru opcji

Ustawić listę opcji na pierwszy element, poprzez ustawienie własności ItemIndex (indeks elementu w TRadioGroup) na 0.

4.2. Projektowanie listy rozwijalnej

Należy zaprojektować listy rozwijalne ComboBox dla wyboru rodzaju obrysu (cbObrys) i wypełnienia (cbWypelnienie) figury. Opcje do tego komponentu dodaje się edytując własność *Items*. Dla każdej z list utworzyć odpowiednie elementy.

**CbObrys:** psSolid; psDash; psDot; psDashDot; psDashDotDot; psClear; psInsideFrame;

**CbWypełnienie:** bsSolid; bsCross; bsDiagCross; bsBDiagonal; bsHorizontal; bsFDiagonal; bsVertical; bsClear;

Ustawić listy opcji na pierwszy element, poprzez ustawienie własności *ItemIndex* (indeks elementu w ComboBox) na 0.

5.Wybór własności rysowanych elementów.

Napisać procedurę do obsługi przycisków służących do wyboru koloru obramowania i wypełnienia figury rysowanej na ekranie. Wybrane opcje będą automatycznie zmieniały komponent Shape, pozwalający sprawdzić wybór. Do tego wykorzystamy komponent ColorDialog i własności komponentu Shape (piórko (obrys) - Shape->Pen, pędzelek (wypełnienie) - Shape->Brush). Komponent Shape przy okazji pozwala w łatwy sposób zapamiętać ustawienia.

Dla zmiany koloru obramowania/linii

```
void TForm1::btnKolorObrysuClick(TObject *Sender)
{
//najpierw przypiszmy okienku wartość domyślną
      this->ColorDialog1->Color=this->Shape1->Pen->Color;
//ta instrukcja wykona się tylko wtedy, gdy użytkownik wciśnie w okienku OK
      if (this->ColorDialog1->Execute())
//przepisanie koloru
      this->Shape1->Pen->Color=this->ColorDialog1->Color ;
}
```

Analogicznie dla zmiany koloru wypełnienia kształtu

```
void TForm1::btnKolorWypelnieniaClick(TObject *Sender)
{
      this->ColorDialog1->Color=this->Shape1->Brush->Color;
      if (this->ColorDialog1->Execute())
      this->Shape1->Brush->Color=this->ColorDialog1->Color;
}
```

6. Wybór i zmiana właściwości rysowanych obiektów.

Napisać procedurę do obsługi wyboru rodzaju obramowania i wypełnienia, w tym celu należy obsłużyć zdarzenie *OnChange* komponentu `ComboBox`. Do podglądy wybranych opcji można wykorzystać komponent `Shape` dla którego należy zmodyfikować własność `pen->style` lub `brush->style`. Podczas wyboru opcji należy odnosić się do numeru opcji (opcje są numerowane od 0) wykorzystując własność:

`ComboBox->ItemIndex.`

Wybór stylu obramowania/linii:

```
void TForm1::cbObrysChange(TObject *Sender)
{
{sprawdź wybrany element na liście cbObrys}
     switch (cbObrys->ItemIndex)
     {
           case 0: this->Shape1->Pen->Style=psSolid; break;
           case 1: this->Shape1->Pen->Style=psDash; break;
           case 2: this->Shape1->Pen->Style=psDot; break;
           case 3: this->Shape1->Pen->Style=psDashDot; break;
           case 4: this->Shape1->Pen->Style=psDashDotDot; break;
           case 5: this->Shape1->Pen->Style=psClear; break;
           case 6: this->Shape1->Pen->Style=psInsideFrame; break;
           default: this->Shape1->Pen->Style=psSolid; break;
     }
}
```

Analogicznie wybór stylu wypełnienia:

```
void TForm1::cbWypelnienieChange(TObject *Sender)
{
      switch (cbWypelnienie->ItemIndex)
      {
            case 0: this->Shape1->Brush->Style=bsSolid; break;
            case 1: this->Shape1->Brush->Style=bsCross; break;
            case 2: this->Shape1->Brush->Style=bsDiagCross; break;
            case 3: this->Shape1->Brush->Style=bsBDiagonal; break;
            case 4: this->Shape1->Brush->Style=bsHorizontal; break;
            case 5: this->Shape1->Brush->Style=bsFDiagonal; break;
            case 6: this->Shape1->Brush->Style=bsVertical; break;
            case 7: this->Shape1->Brush->Style=bsClear; break;
            default: this->Shape1->Brush->Style=bsSolid; break;
     }
}
```

7. Inicjacja zmiennych.

W kolejnym kroku należy napisać procedurę inicjalizującą nasz program. W tym wypadku obsługujemy zdarzenie *OnShow* formularza `Form`. Wcześniej należy zadeklarować obiekt o nazwie `Tlo` typu `TBitmap` w którym będzie przechowywany nasz rysunek i z którego ewentualnie zapisywać się go będzie do pliku. Należy także zadeklarować współrzędne punktu (`xp,yp`) z którego zaczyna się rysowanie danej figury po najechaniu na odpowiednie miejsce kursorem i naciśnięciu w tym miejscu lewego przycisku myszki. Ważne jest także zadeklarowanie flagi (zmiennej logicznej), która będzie mówiła, czy rysujemy, czy tylko poruszamy kursorem po ekranie.

Deklaracje zmiennych w programie (plik Unit1.h)

```
public: //User declarations
    TBitmap *Tlo;
    int xp;
    int yp;
    bool Rysowanie;
```

Procedura inicjacyjna (zdarzenie *OnShow* formularza `Form`)

```
void TForm1::FormShow(TObject *Sender)
{
//Ustalimy, jaka wartość typu obrysu i wypełnienia będzie przyjmowana
jako domyślna
         this->cbObrys->ItemIndex=0;
         this->cbWypelnienie->ItemIndex=0;
//tworzenie obiektu pamiętającego nasz rysunek
         this->Tlo=new TBitmap();
//ustalenie jego wielkości na odpowiadającą przeznaczonej do rysowania
części okna (komponent TPaintBox)
         this->Tlo->Height=this->PaintBox1->Height;
         this->Tlo->Width=this->PaintBox1->Width;
}
```

8. Rysowanie na płótnie.

Należy napisać własną procedurę która na płótnie (obiekt klasy `TCanvas`) będącym jej argumentem będzie rysowała kształt zdefiniowany punktem początku (`xp,yp`) i punktem końca (`xk,yk`). Wykorzystane zostaną tutaj następujące metody klasy `Tcanvas` (tabela 6):

*Tabela 6. Metody klasy TCanvas*

| **Metoda** | **Opis** |
|---|---|
| `Canvas.MoveTo(xp,yp)` | Ustawienie domyślnego punktu początku rysowania |
| `Canvas.LineTo(xk,yk)` | Rysowanie linii pomiędzy domyślnym punktem (ustawionym przez MoveTo), a punktem określonym przez współrzędne xk,yk |
| `Canvas.Rectangle(xp,yp,xk,yk)` | Rysowanie prostokąta pomiędzy punktami (`xp,yp`) - górny lewy róg, (`xk,yk`) - dolny prawy róg |

Deklaracja procedury

```
public:
...
void Rysuj(TCanvas *Canvas, int xp, int yp, int xk, int yk);
```

Treść procedury

```
void TForm1::Rysuj(TCanvas *Canvas, int xp, int yp, int xk, int yk)
{
//Ustawienie własności rysowanej figury, przechowywane we własnościach
komponentu Shape

  Canvas->Pen=this->Shape1->Pen;
  Canvas->Brush=this->Shape1->Brush;
  switch (this->rgFigura->ItemIndex)
  {
    case 0: //Rysowanie linii
      {
                Canvas->MoveTo(xp,yp);
                Canvas->LineTo(xk,yk);
      } break;
    case 1: //Rysowanie prostokata
      {
                Canvas->Rectangle(xp,yp,xk,yk);
      } break;
    default:
      {
                Canvas->MoveTo(xp,yp);
                Canvas->LineTo(xk,yk);
      } break;
  }
}
```

9. Obsługa zdarzeń związanych z myszką.

Następnie piszemy procedurę służącą do obsługi zdarzeń związanych z ruchem kursora i zachowaniem myszki, Wykorzystywane zdarzenia to zdarzenia formularza takie jak:

*OnMouseDown* - naciśnięcie lewego przycisku w określonym punkcie, pozwala wykorzystać współrzędne (X,Y) punktu na który wskazywał kursor, gdy przycisk został wciśnięty.

*OnMouseMove* - ruch myszki z wciśniętym lewym przyciskiem, zwracana jest informacja o współrzędnych kolejnych punktów na ekranie (X,Y).

*OnMouseUp* - Puszczenie lewego przycisku myszki, pozwala wykorzystać współrzędne (X,Y) punktu na który wskazywał kursor, gdy przycisk został puszczony.

Należy zwrócić uwagę na to jak wykorzystywana jest zmienna logiczna `Rysowanie`, zadeklarowana na początku programu (pkt.7), mówiąca o tym czy nasz rysunek jest rysowany

na ekranie. Ustawiana jest ona na wartość `true`, w przypadku gdy lewy przycisk myszki został wciśnięty (*OnMouseDown*). Dopiero wtedy ruszając myszką można cokolwiek narysować na ekranie (*OnMouseMove*). Zmienna `Rysowanie` przyjmuje wartość `false`, w przypadku puszczenia lewego przycisku (*OnMouseUp*).

Należy zwrócić uwagę jak rozwiązano sposób rysowania. Wykorzystywana jest tu funkcja `Rysuj` która rysuje tymczasową figurę na ekranie, natomiast ostateczną figurę najpierw w obiekcie `TBitmap` nie pokazywanym na ekranie. Jest to tzw. bitmapa pozaekranowa (ang. *offscreen bitmap*), Korzystania z bitmap pozaekranowych sprowadza się do trzech zasadniczych zagadnień:

- Utworzenie bitmapy w pamięci.
- Wykonanie rysunku w bitmapie pamięciowej.
- Skopiowanie bitmapy z pamięci na ekran (po zatwierdzeniu wszystkich zmian).

Przy każdym tymczasowym narysowaniu zmazywana jest poprzednia zmiana i jest pokazywany ostatni zapisany rysunek z obiektu `TBitmap`. Do pokazania tego obiektu na ekranie wykorzystywana jest metoda do wyświetlania na określonym płótnie na ekranie (`PaintBox->Canvas`) całej mapy bitowej (jeśli ma ona rozmiar tego płótna) `PaintBox->Canvas->Draw(0,0,Tlo)`. Dopiero po puszczeniu przycisku podczas rysowania zmiany są wprowadzone na stałe.

Obsługa zdarzenia *OnMouseDown* dla komponentu `PaintBox`

```
void TForm1::PaintBox1MouseDown(TObject *Sender, TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
       this->xp=X;
       this->yp=Y;
//Zaznaczenie, że będziemy rysować figurę tymczasową
       this->Rysowanie=true;
}
```

Obsługa zdarzenia *OnMouseMove* dla komponentu `PaintBox`

```
void TForm1::PaintBox1MouseMove(TObject *Sender, TShiftState Shift, int X,
int Y)
{
 if (this->Rysowanie=true)
 {
//Rysujemy bitmapę zawierającą nasz rysunek
        this->PaintBox1->Canvas->Draw(0,0,Tlo);
//Dodajemy dodatkową figurę, ale nie na bitmapę, tylko na PaintBox
        this->Rysuj(this->PaintBox1->Canvas,this->xp,this->yp,X,Y);
 }
}
```

Obsługa zdarzenia *OnMouseUp* dla komponentu `PaintBox`

```
void TForm1::PaintBox1MouseUp(TObject *Sender, TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
//rysowanie
  if (this->Rysowanie==true)
   {
//Dodajemy dodatkową figurę na bitmapę
        this->Rysuj(this->Tlo->Canvas,this->xp,this->yp,X,Y);
//Rysujemy bitmapę zawierającą nasz rysunek
        this->PaintBox1->Canvas->Draw(0,0,this->Tlo);
   }
//Zaznaczenie, że nie trzeba już rysować figury tymczasowej
```

```
  this->Rysowanie=false;
//Przyjęcie punktu koncowego za poczatkowy
  this->xp=X;
  this->yp=Y;
}
```

10. Zapisywanie i odczytywanie bitmap z plików.

Program można uzupełnić o procedury służące do zapisywania i czytania z plików .bmp bitmap. W tym celu wykorzystywane mogą być metody obiektu klasy `TBitmap` oraz okna dialogowe do odczytu i zapisu pliku OpenDialog, SaveDialog.

`Tbitmap::LoadFromFile(string nazwaPliku)` – zapis do pliku .bmp

`Tbitmap::SaveToFile(string nazwaPliku)` – odczyt z pliku .bmp

Odpowiednie procedury można zrealizować, projektując menu główne programu. W tym celu należy wykorzystać komponent `MainMenu` i dodać do niego elementy: *'Plik' > 'Wczytaj z pliku', 'Zapisz do pliku'*.

10.1. Zapisywanie bitmapy do pliku ('Zapisz do pliku')

```
void TForm1::Zapiszbmpdopliku1Click(TObject *Sender)
{
if (this->SaveDialog1->Execute())
   {
     this->Tlo->SaveToFile(this->SaveDialog1->FileName);
   }
}
```

10.2. Odczytywanie bitmapy z pliku ('Wczytaj z pliku')

```
void TForm1::Wczytajbmpzpliku1Click(TObject *Sender)
{
if (this->OpenDialog1->Execute())
   {
     this->Tlo->LoadFromFile(this->OpenDialog1->FileName);
   }
}
```

10.3 Zabezpieczenia

Podczas działania aplikacji trzeba zabezpieczyć się przed utraceniem informacji o wyglądzie pola do rysowania w momencie przysłonięcia go przez okno innej aplikacji. Można to zrobić obsługując zdarzenie *OnPaint* komponentu `PaintBox` i pisząc procedurę, która będzie przerysowywała rysunek.

```
void TForm1::PaintBox1Paint(TObject *Sender)
{
      this->PaintBox1->Canvas->Draw(0,0,this->Tlo);
}
```

# 3 Obsługa plików graficznych

## 2.1 Rysunek (TImage)

Podstawowym komponentem, który pozwala na wyświetlenie zawartości pliku graficznego jest `TImage` z zakładki `Additional`. Zakładając, że nasz komponent nazywa się `Image1` otwarcie pliku odbywa się przez wywołanie:

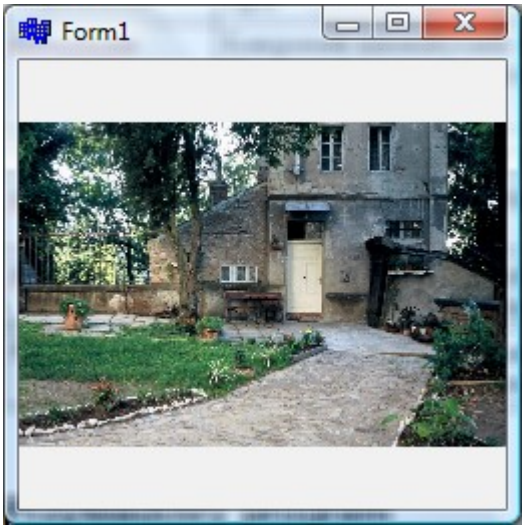
```
Image1->Picture->Bitmap->LoadFromFile(„filename.bmp”);
```

**Uwaga!** Jeżeli chcemy otworzyć plik JPEG musimy wcześniej dołączyć moduł **`Jpeg`**.

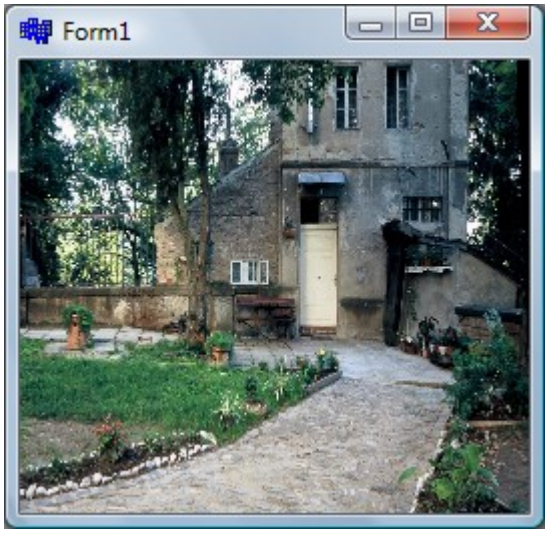
Podstawowe własności komponentu klasy `TImage` zostały zestawione w tabeli 7.

*Tabela 7. Właściwości klasy TImage*

| **Właściwości** | **Opis** |
|---|---|
| `AutoSize` | Komponent automatycznie dopasowuje swój rozmiar do rozmiaru załadowanego pliku (Uwaga! W przypadku, gdy własność `Align` jest ustawiona np. na `alClient`, rozmiar komponentu nie zmieni się) |
| `Center` | Ładowany obraz zostanie wyświetlony na środku komponentu |
| `Picture` | Określa własności wyświetlanego rysunku i pozwala załadować plik w trybie projektowania formy (`Design Mode`) |
| `Proportional` | W przypadku, gdy obraz ładowany ma rozmiar większy niż komponent `Image` obraz zostanie wyświetlony proporcjonalnie tak, aby cały się zmieścił wewnątrz komponentu `Image` (rys. 25) |
| `Stretch` | W przypadku, gdy obraz ładowany ma inny rozmiar niż komponent `Image` obraz zostanie rozciągnięty lub zmniejszony tak, aby zajmował cały obszar komponentu `Image` (rys. 26) |




***Rysunek 25.*** *Komponent TImage z ustawionymi własnościami Center i Proportional*




***Rysunek 26.*** *Komponent TImage z ustawioną własnością Stretch*

Zmiany rozmiaru i położenia komponentu `Timage` z poziomu kodu dokonujemy za pomocą metody `SetBounds(int Left, int Top, int Right, int Bottom)`, gdzie

argumentami są kolejno pozycje krawędzi lewej, górnej, prawej i dolnej.

### 2.2.2 Komponenty niewizualne do obsługi plików graficznych

Otwieranie plików graficznych za pomocą `Image` nie jest jedynym sposobem. W module **Jpeg** zadeklarowana jest klasa `TJPEGImage`, która udostępnia metody `LoadFromFile` i `SaveToFile`. Przed zapisaniem obrazu do pliku powinniśmy ustawić własność `CompressionQuality` nadając jej wartość od 1 do 100, gdzie 1 oznacza najmocniejszą kompresję, czyli najgorszą jakość obrazu.

Wadą obiektów klasy `TJPEGImage` jest brak płótna (`Canvas`), na którym możliwe byłoby rysowanie. W przeciwieństwie do `TJPEGImage` obiekty klasy `TBitmap` mają płótno, co może się okazać bardzo przydatne. `TBitmap` pozwala na otwieranie i zapisywanie nieskompresowanych obrazów (map bitowych) do pliku. Przepisanie zawartości między tymi dwoma typami obiektów jest bardzo proste i odbywa się za pomocą jednej metody `Assign`.

Obiekty klas `TJPEGImage` i `TBitmap` są komponentami niewizualnymi, co oznacza, że otwarcie pliku za pomocą jednego tych komponentów nie spowoduje jego wyświetlenia na ekranie. Chcąc wyświetlić obraz musimy skorzystać z komponentu `TImage`, lub innego dowolnego komponentu zawierającego płótno (`TCanvas`). Poniżej został umieszczony przykładowy kod, który otwiera plik JPEG, kopiuje jego zawartość do obiektu klasy `TBitmap`, a następnie przerysowuje obraz na płótno komponentu `Image1` (który uprzednio umieściliśmy na formie) za pomocą metody `Draw`.

```
void TForm1::OtworzIPrzerysuj()
{
  TJPEGImage *JPEGImage1;
  TBitmap  *Bitmap1;
  JPEGImage = new TJPEGImage();
  Bitmap1 = new TBitmap();
  JPEGImage1->LoadFromFile("obrazek.jpg");
  //kopiujemy zawartość JPEGImage1 do Bitmap1
  Bitmap1->Assign(JPEGImage1);
  //przerysujmy obraz z JPEGImage1 na płótno Image1, zaczynając od lewego
górnego rogu płótna - punkt (0,0)
  Image1->Canvas->Draw(0, 0 , JPEGImage1 );
  //możemy to samo zrobić z obrazem przechowywanym w Bitmap1
  Image1->Canvas->Draw(0, 0 , Bitmap1 );
  delete JPEGImage1;
  delete Bitmap1;
}
```

Jeżeli oryginalny obraz ma rozmiar większy niż komponent `Image1` przerysowany zostanie tylko jego fragment. Możemy jednak użyć metody płótna `StretchDraw`, która wpasuje cały kopiowany obraz w płótno komponentu `Image1`. Niestety jeżeli proporcje obrazu są inne niż proporcje płótna obraz zostanie zniekształcony.

```
// w prostokąt płótna o rozmiarach Image1.ClientRect zostanie wrysowany
obraz przechowywany przez JPEGImage1
Image1->Canvas->StretchDraw(Image1->ClientRect, JPEGImage1);
```

# 4. Operacje na obrazie – skalowanie, buforowanie, akceleracja

## 4.1. Kopiowanie fragmentu obrazu

Wiemy już jak otworzyć plik graficzny nie wyświetlając go na ekranie (komponent `TJPEGImage` lub `TBitmap`). Załóżmy, że mamy obiekt klasy `TImage` mniejszy od oryginalnego obrazu, którego fragment chcemy wyświetlić. Wykorzystując metodę `CopyRect` możemy wyciąć z oryginalnego obrazu fragment o rozmiarze komponentu `TImage` i przerysować go na ten komponent (rys. 27).


**Rysunek 27.** Przerysowanie fragmentu obrazu za pomocą metody CopyRect

Poniżej umieszczony został kod procedury, która jako jako parametry przyjmuje wskaźnik do obiektu `TJPEGImage` oraz element `TPoint`, zawierający współrzędne górnego lewego rogu kopiowanego obszaru.

```
void TForm1::Kopiuj( TJPEGImage *JPEGImage, TPoint imgTopLeft)
{
  TBitmap *Bitmap1;
  Trect  sourceRect;
  // sourceRect to prostokątny obszar, z którego przekopiujemy
  // fragment obrazu. Jego rozmiary będą identyczne jak rozmiary
  // płótna Image1. Zakładamy jednak, że obraz
  // będzie przesunięty, a położenie jego lewego górnego rogu
  // zapisane jest w zmiennej  AimgTopLeft.
  sourceRect.Left= imgTopLeft.x;
  sourceRect.Top= imgTopLeft.y;
  sourceRect.Right= Image1->ClientRect->Right + imgTopLeft.x;
  sourceRect.Bottom= Image1->ClientRect->Bottom + imgTopLeft.y;
  // Ponieważ TJPEGImage nie posiada płótna TCanvas trzeba najpierw
  // przekopiować obraz z JPEGImage do tymczasowego obiektu klasy TBitmap
  Bitmap1 = new TBitmap();
  Bitmap1->Assign(JPEGImage);
  Image1->Canvas->CopyRect(Image1->ClientRect, Bitmap1->Canvas,
sourceRect);
  delete Bitmap1;
}
```

Powinniśmy jeszcze wspomnieć, że `sourceRect` nie musi mieścić się cały wewnątrz obszaru obrazu `Bitmap1`. Skopiowany zostanie tylko ten fragment obrazu, który pokrywa się z naszym wybranym obszarem `sourceRect`, co jest znacznym ułatwieniem.

## *4.2. Skalowanie*

Opisana w poprzednim rozdziale metoda `StretchDraw` może być wykorzystana do skalowania obrazu. Poniżej został umieszczony przykładowy kod przerysowujący obraz z komponentu `TJPEGImage` na komponent `TBitmap` jednocześnie zmieniając jego rozmiar o współczynnik skali `skala`.

```
  TBitmap *Bitmap1;
  TJPEGImage *JPEGImage1;
  double  skala;


  ...
  //musimy sami zadbać o proporcjonalne rozmiary
  Bitmap1->Width =  int(JPEGImage1->Width * skala);
  Bitmap1->Height =  int(JPEGImage1->Height * skala);
  Bitmap1->Canvas->StretchDraw(Rect(0,0, Bitmap1->Width, Bitmap1->Height),
    JPEGImage1 );
  ...
```

## 4.3. Operacje na pikselach – obrót, akceleracja

Zadaniem nieco trudniejszym jest obrócenie obrazu. Będziemy potrzebowali dwóch obiektów typu `TBitmap`. Do pierwszego skopiujemy obraz z obiektu typu `TJPEGImage` (musimy to zrobić, ponieważ klasa `TJPEGImage` nie ma płótna – `Canvas`). Drugi obiekt będzie naszym obróconym obrazem. Płótno daje dostęp do każdego piksela, więc naszym zadaniem jest odpowiednie przepisanie wartości pikseli z obrazu oryginalnego do obrazu obróconego.

```
void TForm1::ObrocWPrawo( TJPEGImage *AJPEGImage)
{
  int i,j;
  TBitmap  *Bitmap1, *Bitmap2;
  Bitmap1 = new TBitmap();
  Bitmap2 = new TBitmap();
  Bitmap1->Assign(AJPEGImage);
  Bitmap2->Width=Bitmap1->Height;
  Bitmap2->Height=Bitmap1->Width;
  for (int i=0; i<Bitmap1.Width;i++)
     for(int j=0; j<Bitmap1.Height; j++)
     Bitmap2->Canvas->Pixels[Bitmap2->Width - j][i]= Bitmap1->Canvas-
>Pixels[i][j];
  AJPEGImage->Assign(Bitmap2);
  delete Bitmap1;
  delete Bitmap2;
}
```

Przedstawiony sposób jest prosty jednak obrócenie obrazu o dużych rozmiarach jest czasochłonne. Ambitniejsi programiści z pewnością będą chcieli ten czas skrócić. Do szybkich operacji na obrazie przeznaczona jest własność `ScanLine` obiektów klasy `TBitmap`. Zwraca ona wskaźnik do początku jednolitego obszaru pamięci odpowiadającemu pojedynczemu wierszowi pikseli w obrazie. Pamiętać należy, że piksel w obrazie kolorowym zajmuje trzy bajty (po jednym dla każdego z kanałów RGB), dlatego zdefiniujemy nowy typ `TByteTriple`, który będzie trójelementową tablicą bajtów.

```
void TForm1::ObrocWLewo(TJPEGImage *AJPEGImage)
{
  //definiujemy nowy typ, który będzie trójelementową tablicą bajtów
```

```
  typedef  Byte TByteTriple[3];
  TByteTriple *PixelNew, *PixelOld;

  Graphics::TBitmap  *Bitmap1, *Bitmap2;

  Bitmap1 = new Graphics::TBitmap();
  Bitmap2 = new Graphics::TBitmap();

  Bitmap1->Assign(AJPEGImage);
  Bitmap2->Width =Bitmap1->Height;
  Bitmap2->Height =Bitmap1->Width;

//koniecznie musimy ustawić rozmiar piksela na 24 bity czyli 3 bajty
  Bitmap1->PixelFormat=pf24bit;
  Bitmap2->PixelFormat=pf24bit;

  for(int j=0; j<Bitmap2->Height; j++)
  {

  //pobieramy wskaźnik do pierwszego piksela j-tej lini nowego obrazu
    PixelNew = (TByteTriple *) Bitmap2->ScanLine[j];

    for(int i=0; i<Bitmap1->Height; i++)
    {
  //pobieramy wskaźnik do pierwszego piksela i-tej lini starego obrazu
      PixelOld= (TByteTriple *)Bitmap1->ScanLine[i];

  //przepisujemy zawartość 3 bajtów pamięci, ze starego obrazu do
nowego
      PixelNew[i][0] = PixelOld[Bitmap1->Width-j-1][0];
      PixelNew[i][1] = PixelOld[Bitmap1->Width-j-1][1];
      PixelNew[i][2] = PixelOld[Bitmap1->Width-j-1][2];

    }
  }

  AJPEGImage->Assign(Bitmap2);

  delete(Bitmap1);
  delete(Bitmap2);
}
```

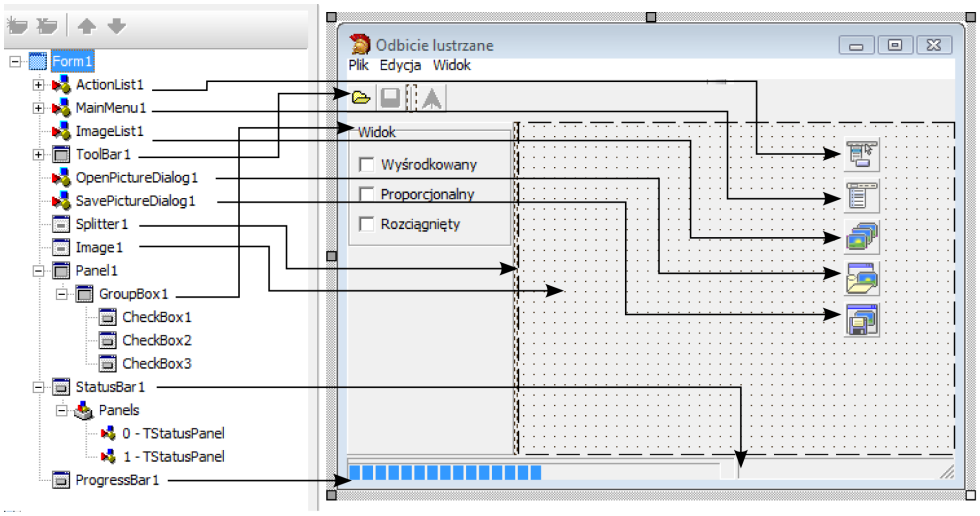
## 5. Przykładowy program – operacje na plikach JPEG

Program ma pozwalać na otwarcie wybranego pliku JPEG, wykonanie jego odbicia lustrzanego i zapis zmienionego obrazu do pliku z zadanym poziomem kompresji.

**1.** Otwieramy nową aplikację *File>NewApplication*

**2.** Zapisujemy projekt przy pomocy *File>Save All* w katalogu gdzie będzie kompilowany nasz projekt

**3.** Na początku zaprojektujemy główny formularz.



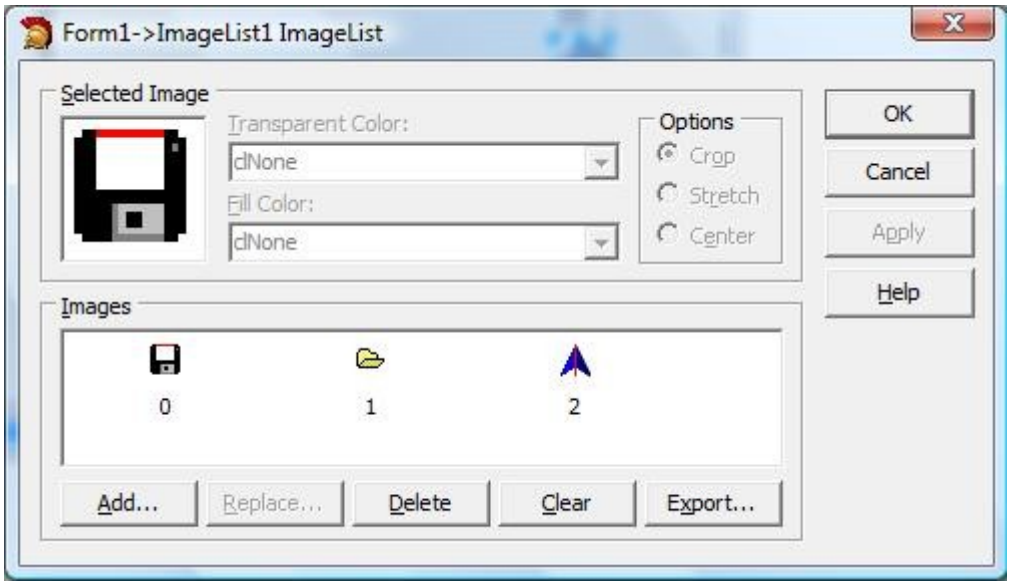

**Rysunek 28.** Przykład "Odbicie lustrzane" - główny formularz

Potrzebne będą następujące komponenty:

- Lista akcji (`Standard > ActionList`)
- Menu główne (`Standard > MainMenu`)
- Pasek narzędziowy (`Win32 > ToolBar`)
- W liście obrazów (`Win32 > ImageList`) będą przechowywane ikony, które przydadzą się do udekorowania paska narzędzi
- Panel (`Standard > Panel`), na którym umieścimy `GroupBox` zawierający opcje wyświetlania obrazu na komponencie `Image`. Ustawiamy jego własność `Align` na `alLeft`, a minimalną szerokość panelu `MinWidth` (`Constraints > MinWidth`) na wartość 120.
- Po umieszczeniu panelu wstawiamy obiekt `Splitter` (`Additional > Splitter`), dzięki czemu użytkownik będzie mógł dostosować szerokość panelu kosztem znajdującego się obok obiektu `Image`.
- Następnie umieszczamy obiekt, który pozwoli na wyświetlenie obrazu (`Additional > Image`). Ustawiamy jego własność `Align` na `alClient`. Komponent zajmie cały dostępny obszar na prawo od `Splittera`.
- Na wcześniej przygotowanym panelu umieszczamy obiekt `GroupBox` (zakładka Standard), a na nim trzy komponenty typu `CheckBox` (zakładka Standard). Na tym etapie nie zmieniamy jeszcze wyświetlanych nazw, zadanie ułatwi nam lista akcji.
- U dołu okna umieszczamy komponent `StatusBar` (zakładka `Win32`)
- Ponieważ operacja odbicia lustrzanego przy dużych plikach może potrwać, dodamy jeszcze pasek postępu (`Win32 > ProgressBar`). Nie daje się on umieścić na komponencie `StatusBar`, ale jest na to sposób, który zostanie omówiony w dalszej części.
- Na koniec dodamy jeszcze okna dialogowe do otwarcia i zapisu pliku graficznego (`Dialogs > OpenPictureDialog` i `SavePictureDialog`)

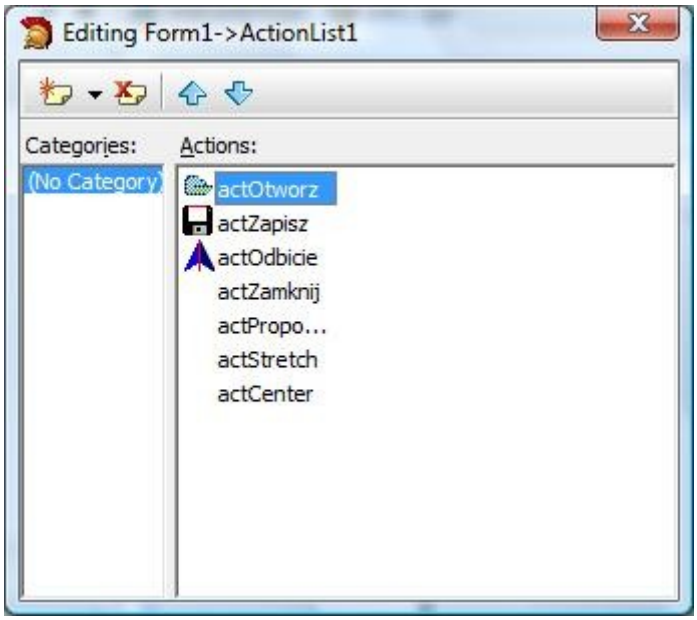
4. Kiedy mamy już wszystkie komponenty umieszczone na formularzu możemy przygotować ikony, które wykorzystamy na pasku narzędziowym. Klikamy dwukrotnie na komponent `ImageList1`. Pojawi się okno takie jak pokazane na rysunku 29. Klikając przycisk `Add...` mamy możliwość dodania ikon na przykład z katalogu *C:\Program Files\Common Files\Borland Shared\Images\Buttons*. Przydadzą się z pewnością ikony do takich akcji jak otwieranie, zapisywanie

pliku i oczywiście odbicie lustrzane.




**Rysunek 29.** Przykład "Odbicie lustrzane" - ImageList1

5. Przygotowanie listy akcji. Najpierw klikamy raz na komponent `ActionList1` i ustawiamy własność Images na `ImageList1`, dzięki czemu będziemy mogli przypisać akcjom ikony. Następnie klikamy dwukrotnie na komponent `ActionList1`; otworzy się okno edycji akcji (rys.30).



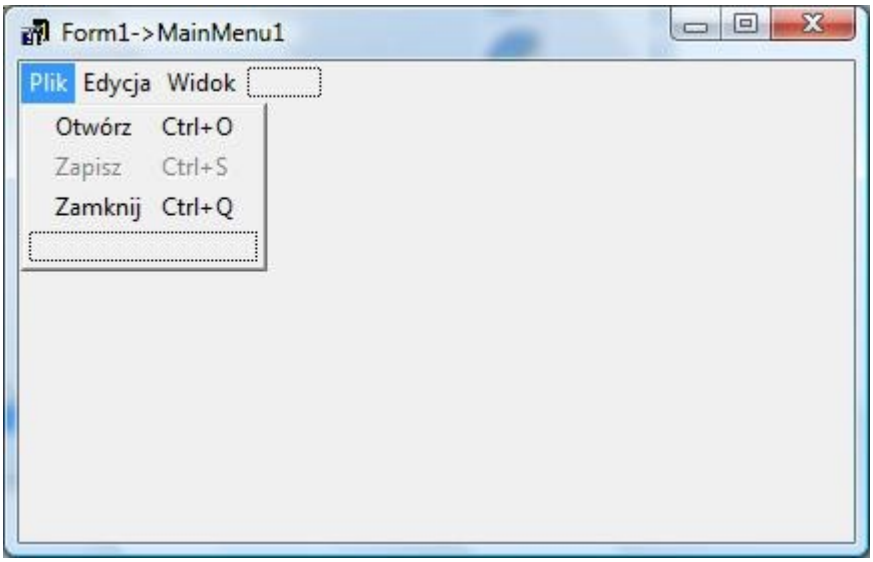

**Rysunek 30:** Przykład "Odbicie lustrzane" - ActionList1

Tworzymy 7 akcji i ustawiamy ich własności:

- `Name = acnOtworz; Caption = Otwórz;` dla `ImageIndex` wybieramy numer odpowiedniej ikony; przypisujemy skrót klawiszowy `ShortCut = Ctrl+O`
- `Name = acnZapisz; Caption = Zapisz;` dla ImageIndex wybieramy numer odpowiedniej ikony; własność `Enabled` ustawiamy na `false`, ponieważ chcemy, żeby była ona aktywna dopiero po zmodyfikowaniu pliku; przypisujemy skrót klawiszowy `ShortCut = Ctrl+S`

- `Name = acnOdbicie; Caption = Odbicie;` dla `ImageIndex` wybieramy numer odpowiedniej ikony; własność `Enabled` ustawiamy na `false`, ponieważ chcemy, żeby była ona aktywna dopiero po załadowaniu pliku; przypisujemy skrót klawiszowy `ShortCut = Ctrl+M`
- `Name = acnZamknij; Caption = Zamknij;` przypisujemy skrót klawiszowy `ShortCut = Ctrl + Q`
- `Name = acnProportional; Caption = Proporcjonalny; AutoCheck = true`, ponieważ akcja ta będzie odpowiadać za przełączanie własności logicznej `Proportional` komponentu `Image1`, chcemy aby ta akcja po każdorazowym wywołaniu zmieniała swoją własność `Checked`, tak jak komponent `CheckBox`
- `Name = acnStretch; Caption = Rozciągnięty; AutoCheck = true`, ponieważ akcja ta będzie odpowiadać za przełączanie własności logicznej `Stretch` komponentu `Image1`
- `Name = acnCenter; Caption = Wyśrodkowany; AutoCheck = true`, ponieważ akcja ta będzie odpowiadać za przełączanie własności logicznej `Center` komponentu `Image1`

6. Projektowanie menu głównego. Klikamy dwukrotnie na ikonę komponentu `MainMenu`; pojawi się okno edytora (rys. 31).




**Rysunek 31.** Przykład "Odbicie lustrzane" - MainMenu1

W prostokątne wykropkowane pole wpisujemy nazwę menu np. `Plik`. Pojawi się podmenu, tutaj podobnie klikamy na wykropkowany prostokąt, ale nie wpisujemy nazwy sami tylko ustawiamy własność `Action` wybierając jedną z wcześniej przygotowanych akcji. Podobnie postępujemy dla pozostałych akcji, przy czym akcję `acnObicie` umieszczamy w menu `Edycja`, a akcje związane z własnościami komponentu `Image` na przykład w menu `Widok`.

7. Projektowanie paska narzędziowego. Ustawiamy własność `Images` na `ImageList1`. Klikamy prawym przyciskiem i z menu podręcznego wybieramy `New Button`. Klikamy na nowo stworzony przycisk i ustawiamy jego własność `Action` na `acnOtworz`. Podobnie postępujemy dla `acnZapisz` i `acnOdbicie` umieszczając między nimi separator (menu podręczne > `New Separator`).

8. Wcześniej na panelu 1 umieściliśmy `GroupBox` wraz z 3 komponentami `CheckBox`. Zmieńmy własność `Caption` komponentu `GroupBox` na `Widok`. Teraz powiążemy kolejno komponenty

CheckBox z akcjami. Klikamy CheckBox1 i ustawiamy jego własność Action na acnCenter, podobnie postępujemy dla pozostałych komponentów CheckBox przypisując akcje acnProportional i acnStretch. Wyświetlane nazwy na formularzu powinny zmienić się automatycznie.

9. Przygotowanie paska stanu. Klikamy dwukrotnie na komponent StatusBar1 i tworzymy dwa panele. Ustawmy szerokość pierwszego panelu (Width) na podobną do szerokości paska postępu (ProgressBar1).

10. Dodatkowe pola i metody klasy TForm1:: Otwieramy plik Unit1.h.

Ponieważ nasz program ma wykonywać operacje na plikach JPEG, musimy dołączyć odpowiedni plik nagłówkowy:

```
#include <Jpeg.hpp>
```

W definicji klasy TForm1 odnajdujemy sekcję private i deklarujemy dwa dodatkowe pola:

- Zmienną będącą obiektem klasy TJPEGImage, w którym będziemy przechowywali otwarty plik,

```
TJPEGImage *JPEGImage;
```

- pole typu logicznego zmodyfikowany, w którym będziemy przechowywać informację o tym czy plik został zmodyfikowany.

```
bool zmodyfikowany;
```

Deklarujemy także dwie nowe metody:

- modyfikacja(bool mo), która zadba o ustawienie odpowiednich komponentów, po tym jak obraz zostanie zmodyfikowany
- pokazObraz(), która będzie odpowiadać za przerysowanie obrazu z obiektu JPEGImage na komponent Image1.

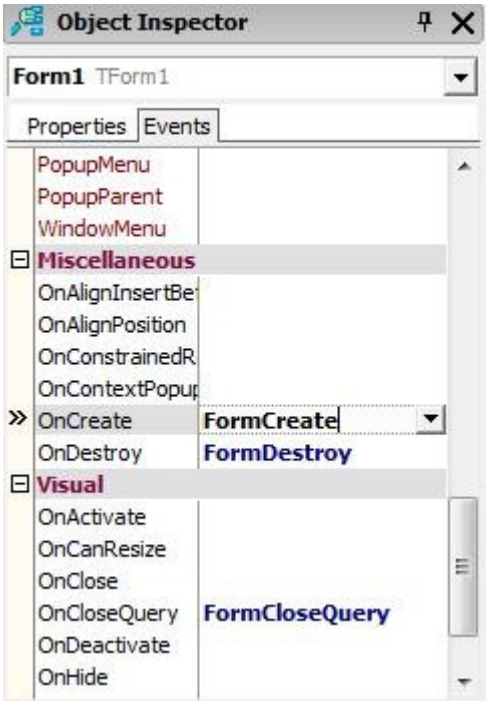
```
private:    // User declarations
      TJPEGImage *JPEGImage1;
      bool zmodyfikowany;
      void modyfikacja(bool mo);
      void pokazObraz();
```

Następnie umieszczamy definicje naszych metod:

```
     void TForm1::modyfikacja(bool mo)
     {
      zmodyfikowany = mo;

      //Po zmodyfikowaniu obrazu informujemy o tym użytkownika
      //umieszczając stosowny napis na panelu paska stanu
      if(mo) StatusBar1->Panels->Items[1]->Text = "Plik zmodyfikowany";
      else StatusBar1->Panels->Items[1]->Text = "";

      //Akcja actZapisz powinna być aktywna tylko wtedy, kiedy obraz został
      //zmodyfikowany
      actZapisz->Enabled = zmodyfikowany;
     }

     void TForm1::pokazObraz()
     {
      //Jeżeli obiekt JPEGImage1 nie jest pusty to możemy przerysować obraz
      //na komponent Image1 i aktywować akcję actOdbicie
      if(!JPEGImage1->Empty){
```

```
        Image1->Picture->Assign(JPEGImage1);
        actOdbicie->Enabled = true;
    }
    else actOdbicie->Enabled = false;
}
```

11. Zdarzenia formularza. Klikamy na `Form1` i wchodzimy do zakładki `Events` w inspektorze obiektów (`Object Inspector`) (rys. 32).




**Rysunek 32.** Przykład "Odbicie lustrzane" - Zdarzenia komponentu Form1

Zdefiniujemy trzy zdarzenia `OnCreate`, `OnDestroy`, `OnCloseQuery`. Klikamy dwukrotnie na wolne pole obok zdarzenia `OnCreate`. Zostaliśmy automatycznie przeniesieni do pliku `Unit1.pas`, gdzie jest już przygotowany odpowiedni nagłówek procedury. Zdarzenie `OnCreate` wywoływane jest przy starcie programu. Jest to dobre miejsce do zainicjowania niektórych zmiennych. Umieszczamy tam następujący kod:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
//Tworzymy obiekt klasy TJPEGImage, który będzie przechowywał obraz
 JPEGImage1 = new TJPEGImage;

//Tutaj umieszczamy komponent ProgressBar1 na pasku stanu StatusBar1,
//czego nie mogliśmy zrobić wcześniej w trybie projektowania
formularza.
//Sposób ten polega na zmianie rodzica komponentu ProgressBar1, którym
//wcześniej był formularz Form1.
 ProgressBar1->Parent = StatusBar1;

//Ustawiamy pozycję paska postępu, która jes teraz liczona względem
//komponentu StatusBar1
 ProgressBar1->Left = 3;
 ProgressBar1->Top = 3;
 ProgressBar1->Position = 0;

//Inicjujemy pole zmodyfikowany wywołując wcześniej zdefiniowaną
procedurę
 modyfikacja(false);
```

```
//Inicjujemy własności Checked akcji odpowiadających ustawieniom
//komponentu Image1
 actCenter->Checked = Image1->Center;
 actProportional->Checked = Image1->Proportional;
 actStretch->Checked = Image1->Stretch;
}
```

Zdarzenie `OnDestroy` wywoływanym przy niszczeniu obiektu `Form1` dbamy o usunięcie obiektów, które sami stworzyliśmy, czyli `JPEGImage1`.

```
void TForm1::FormDestroy(TObject *Sender)
{
      delete(JPEGImage);
}
```

Zdarzenie `OnCloseQuery` jest bardziej rozbudowane. Metoda wywoływana przez to zdarzenie powinna posiadać referencję do zmiennej logicznej `CanClose`. Od ustawienia tej zmiennej zależy czy nasza aplikacja zostanie zamknięta czy też nie. Zdarzenie to jest wywoływane przy próbie zamknięcia formularza `Form1`. Jest to idealne miejsce do zapytania użytkownika, czy na pewno chce opuścić program, bez zapisywania zmian.

```
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool
&CanClose)
{
//Pomocnicza zmienna, przechowująca informację o tym jaki przycisk
//nacisnął użytkownik odpowiadając na pytanie zawarte w komunikacie
//wyświetlonym za pomocą metody MessageBox
 int mb_return;

//Domyślnie ustawiamy CanClose na true
 CanClose = true;

//Jeżeli nasz obraz został zmodyfikowany pytamy użytkownika, czy chce
//zapisać zmiany przed wyjściem z programu
 if (zmodyfikowany) {
       mb_return = Application->MessageBox("Czy chcesz zapisać zmiany
do pliku?", "Wyjście z programu", MB_YESNOCANCEL | MB_ICONQUESTION);

//Jeżeli użytkownik chce zapisać zmiany wywołujemy akcję actZapisz
       if(mb_return==ID_YES) actZapisz->Execute();

//Jeżeli zostanie wciśnięty przycisk CANCEL nie pozwalamy na
//zamknięcie programu
       if(mb_return==ID_CANCEL) CanClose = false;
 }
}
```

12. Definiowanie akcji. Dla każdej akcji musimy zdefiniować zdarzenie `OnExecute`. Klikamy dwukrotnie na komponent `ActionList1`. Wybieramy akcję i otwieramy zakładkę zdarzenia (`Events`) w inspektorze obiektów (`Object Inspector`). Klikamy dwukrotnie na puste pole obok zdarzenia `OnExecute`, co spowoduje utworzenie odpowiedniej metody.

```
void __fastcall TForm1::actOtworzExecute(TObject *Sender)
{
 if(OpenPictureDialog1->Execute())
 {
//Jeżeli użytkownik wskaże poprawny plik możemy go otworzyć i załdować
//do zmiennej JPEGImage1
       JPEGImage1->LoadFromFile(OpenPictureDialog1->FileName);

//Na górnej belce programu wypiszemy nazwę otwartego pliku
```

```
        this->Caption = "Odbicie lustrzane - " +
                          OpenPictureDialog1->FileName;

//Na koniec wyświetlamy obraz na komponencie Image1
        PokazObraz();
 }
}

void __fastcall TForm1::actZamknijExecute(TObject *Sender)
{
 this->Close();
}

void __fastcall TForm1::actCenterExecute(TObject *Sender)
{
//Ponieważ akcja actCenter ma zaznaczoną własność AutoCheck,
//po każdym wywołaniu akcji actCenter zmienia się jej pole Checked,
//które przepisujemy do odpowiedniej własności komponentu Image1
 Image1->Center = actCenter->Checked;
}

void __fastcall TForm1::actStretchExecute(TObject *Sender)
{
 Image1->Stretch = actStretch->Checked;
}

void __fastcall TForm1::actProportionalExecute(TObject *Sender)
{
 Image1->Proportional = actProportional->Checked;;
}
```

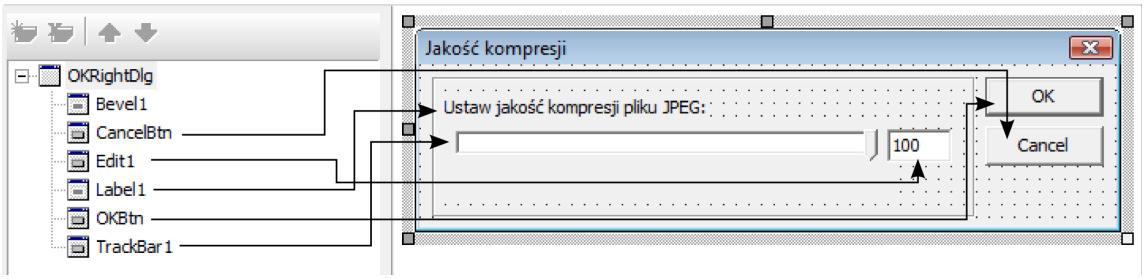
Najwazniejsza akcja, czyli odbicie lustrzane obrazu.

```
void __fastcall TForm1::actOdbicieExecute(TObject *Sender)
{
 Graphics::TBitmap *bitmap1 = new Graphics::TBitmap;

 //Obiekt klasy TJPEGImage nie ma płótna, dlatego wykorzystujemy
 // dodatkowo obiekt klasy TBitmap
 bitmap1->Assign(JPEGImage1);

 //Pomocnicza zmienna przechowująca kolor pojedynczego piksela
 TColor c;

 //Ustawiamy wartość maksymalną paska postępu
 ProgressBar1->Max = bitmap1->Height;

 for(int i=0; i<bitmap1->Height; i++)
 {
       //Przy każdym cyklu pętli zmieniamy pozycję paska postępu.
       ProgressBar1->Position = i;

       //Zamieniamy miejscami piksele z początku wiersza, z końcowymi,
       //dochodząc tylko do połowy!
       for(int j=0; j<bitmap1->Width/2; j++)
       {
             c = bitmap1->Canvas->Pixels[j][i];

             bitmap1->Canvas->Pixels[j][i] =
                   bitmap1->Canvas->Pixels[bitmap1->Width-j][i];
```

```
                bitmap1->Canvas->Pixels[bitmap1->Width-j][i] = c;
            }
        }

        //Zerujemy położenie paska postępu
        ProgressBar1->Position = 0;

        //Przepisujemy odbity obraz do komponentu JPEGImage1
        JPEGImage1->Assign(bitmap1);

        //Wyświetlamy zmodyfikowany obraz na komponencie Image1
        pokazObraz();

        //Odnotowujemy, że obraz został zmodyfikowany
        modyfikacja(true);

        //Usuwamy pomocniczą bitmapę
        delete bitmap1;
      }
```

13. Ustawienie jakości kompresji. W założeniach programu była możliwość zapisania obrazu do pliku z wybraną jakością kompresji. Najwygodniej dla użytkownika, będzie jeżeli pytanie o jakość kompresji pojawi się w momencie zapisywania pliku, a najlepiej na sam koniec po wyborze nazwy pliku. Do tego będzie nam potrzebne specjalne okno dialogowe, które musimy stworzyć.

Wybieramy *File* > *New* > *Other...* > *Delphi Files* > *Standard Dialog (Vertical)*. Na formularzu umieszczamy dodatkowo komponenty `TrackBar`, `Edit` i `Label` (rys. 33).




**Rysunek 33.** Przykład "Odbicie lustrzane" - Projekt okno dialogowego do ustawiania kompresji obrazu

Nasze okno dialogowe nazywa się `OKRighDlg` i jest klasy `TOKRightDlg`. W deklaracji klasy dodamy dwie zmienne:

- `OK` typu `bool`, której wartość będzie zależeć od tego czy okno zaostało zamknięte za pomocą przycisku `OK` czy `CANCEL`.
- `kompresja` typu całkowitego, w której będziemy przechowywać wartość jakości kompresji (od 1 do 100)

Deklaracje tych zmiennych umieszczamy w sekcji public.

```
public:
      bool OK;
      int kompresja;
```

Zmieniamy własności komponentu `TrackBar1`:

- `Min = 1,`

- Max = 100,
- Position = 100
- TickStyle = tsNone
- Frequency = 1

Pole edycyjne Edit1 będzie pełniło rolę informacyjną, dlatego blokujemy możliwość edycji ustawiając własność ReadOnly na true i Enabled na false.Pozostało jedynie zdefiniowanie odpowiednich zdarzeń.

Zdarzenie OnChange komponentu TrackBar1:

```
void __fastcall TOKRightDlg::TrackBar1Change(TObject *Sender)
{
 //Aby umożliwić użytkownikowi precyzyjne ustawienie wartości
 //wyświetlamy ją dodatkowo na komponencie Edit1
 Edit1->Text = IntToStr(TrackBar1->Position);
}
```

Zdarzenie OnClick przycisku OKBtn:

```
void __fastcall TOKRightDlg::OKBtnClick(TObject *Sender)
{
 //Przed zamknięciem okna dialogowego ustawiamy odpowiednie zmienne
 OK = true;
 kompresja = TrackBar1->Position;
 this->Close();
}
```

Zdarzenie OnClick przycisku CancelBtn

```
void __fastcall TOKRightDlg::CancelBtnClick(TObject *Sender)
{
 OK = false;
 this->Close();
}
```

Mając przygotowane okno dialogowe możemy zdefiniować akcję actZapisz. Najpierw jednak musimy dołączyć nowy plik nagłówkowy wpisując w Unit1.h #include "dlgKompresja.h" zakładając, że nasz nowy moduł nazwaliśmy dlgKompresja.

```
void __fastcall TForm1::actZapiszExecute(TObject *Sender)
{
 if(SavePictureDialog1->Execute())
 {
       //Tworzymy obiekt klasy  TOKRightDlg, czyli okno dialogowe
kompresji
       TOKRightDlg *kompresja = new TOKRightDlg(this);

       //Wyświetlamy okno dialogowe modalnie, czyli blokując główne
       //okno  aplikacji
       kompresja->ShowModal();

       if(kompresja->OK)
       {
             //Jeżeli okno dialogowe zostało zamknięte poprawnie,
             //możemy dokonać zapisu obrazu do pliku, wcześniej
             //ustawiając jakość kompresji
             JPEGImage1->CompressionQuality = kompresja->kompresja;
```

```
                JPEGImage1->SaveToFile(SavePictureDialog1->FileName);

                //Po zapisie zmian do pliku resetujemy zmienną
zmodyfikowano
                modyfikacja(false);
        }

        //Pamiętajmy o usunięciu obiektu, który sami stworzyliśmy
        delete kompresja;
  }

}
```

14. Na koniec warto ustawić własność *Options* > *ofFileMustExists* komponentu `OpenPictureDialog1` na `true`, co nie pozwoli na otwarcie pliku, który nie istnieje. Ponieważ nasz program obsługuje tylko pliki JPEG, powinniśmy we własności `Filter` okien dialogowych pozostawić tylko rozszerzenia *.jpeg i *.jpg.