Maciej Przybylski, Paweł Wnuk, Jan Klimaszewski

Projekty

Materiały dydaktyczne, Ośrodek Kształcenia Na Odległość – OKNO

Spis treści

PROJEKT 1 – edytor tekstu (łatwy)

Wprowadzenie

Zadanie polega na napisaniu prostego edytora tekstu obliczającego dodatkowo statystykę wyrazów.

Wymagane umiejętności

- zasad tworzenia GUI,
- podstawowych komponentów wizualnych i ich właściwości,
- tworzenia i dołączania modułów,
- operacji na tablicach statycznych,
- podstaw programowania obiektowego (m.in. dziedziczenie),

na lepszą ocenę zastosowanie:

- list dynamicznych.

Funkcje podstawowe programu

- Otwieranie i zapisywanie wielu plików tekstowych.
- Obliczanie statystyki tekstu (ilość akapitów, wyrazów, liter, znaków)
- Wyświetlenie statystyki w postaci raportu.

Funkcje dodatkowe programu

- Zliczanie powtarzających się wyrazów z zastosowaniem listy dynamicznej.

Komponenty do wyświetlenia i edycji zawartości pliku tekstowego

Do wyświetlenia i edycji zawartości pliku tekstowego można użyć jednego z dwóch komponentów:

- `Memo` (na zakładce `Standard`) lub,
- `RichEdit` (na zakładce `Win32`)

To, który z nich wybierzecie do Waszego edytora, w zasadzie nie ma znaczenia. Przy pomocy obydwu można w prosty sposób zapewnić wymaganą funkcjonalność.

Oba te komponenty są w zasadzie pełnoprawnymi edytorami tekstu, i pozwalają bez pisania specjalnych procedur na edycję tekstu umieszczonego w nich, zaznaczanie i przenoszenie fragmentów tekstu, kopiowanie i wstawianie za pomocą skrótów CTRL+C – CTRL+V, itp. Na tym etapie p[rojektu należy napisać dwie procedury – odczytującą dany plik i umieszczającą jego zawartość w komponencie edycyjnym, oraz zapisującą zawartość komponentu na dysk.

Komponenty edycyjne są w stanie zarówno zapisać tekst w nich zawarty do pliku (służy do tego metoda `Strings.LoadFromFile`), jak i odczytać zawartość pliku tekstowego (za pomocą metody `Strings.SaveToFile`). Jedyne co trzeba zrobić, to przekazać tym procedurom nazwę pliku do zapisu – odczytu. A nazwę uzyskamy wykorzystując ... `OpenDialog` i `SaveDialog`**.** Inne własności komponentów `Memo` i `RichEdit` opasane zostały w podręczniku.

Komponent RichEdit jest zawansowaną wersją komponentu Memo umożliwiającą dodatkowo:

- Formatowanie tekstu,
- Wczytywanie plików w formacie RTF,
- Drukowanie,
- Wyszukiwanie fragmentów tekstu,
- Obsługę mechanizmu drag-and-drop,
- Numerowanie,
- Określenie kroju i innych parametrów czcionki, wyrównania tekstu, położenia znaków tabulacji.

Obsługa wielu plików tekstowych

Większość edytorów tekstu pozwala na jednoczesną pracę nad kilkoma dokumentami jednocześnie. Możemy wyróżnić trzy typowe podejścia do konstruowania interfejsu aplikacji:

- SDI (Single Document Interface) – każdy dokument w osobnym oknie,
- MDI (Multiple Document Interface) – wewnątrz obszaru głównego okna aplikacji znajdują się osobne okna dla każdego dokumentu (możliwy widok kilku okien na raz),
- TDI (Tabbed Document Interface) – każdy dokument otwarty w osobnej zakładce.

Dla prostego edytora tekstu proponujemy zastosowanie inerfejsu z zakładkami (TDI). Potrzebny do tego będzie komponent PageControl z palety Win32. Klikając na niego prawym przyciskiem myszy i wybierając New Page, dodamy nową zakładkę, która pojawi się w Object TreeView jako dziecko komponentu PageControl. Zakładka jest osobnym obiektem klasy TTabSheet, który może istnieć jedynie jako element komponentu PageControl.

Jednak ręczne dodawanie zakładek w trybie projektowania nie ma sensu, gdyż nie wiemy z iloma dokumentami zechce pracować użytkownik. Dlatego nauczycie się dynamicznego dodawania komponentów. Procedura, która umożliwi dodanie nowej zakładki może wyglądać na przykład tak:

```
procedure DodajZakladke(APageControl: TPageControl);
var Zakladka: TTabSheet;
begin

{ Tworzenie nowego obiektu klasy TTabSheet. Jako argument konstruktora
należy podać komponent klasy TpageControl, do którego dodajemy zakładkę.}
  Zakladka := TtabSheet.Create(APageControl);

{ Dla nowo utworzonej zakładki musimy jeszcze ustawić właściwość
PageControl, czyli wskazanie na właściciela }
  Zakladka.PageControl := ApageControl;

{ Na koniec ustawiamy tekst, który wyświetli się jako nazwa zakładki}
  Zakladka.Caption:='Nowy dokument';

end;
```

Jak łatwo zauważyć po wyjściu z procedury nie mamy wskaźnika ani referencji do nowo stworzonej zakładki. Nie jest to konieczne, gdyż dostęp do zakładek możliwy jest z poziomu komponentu PageControl. Najważniejsze własności i metody, które pozwalają na poruszanie się po zakładkach to:

- ActivePageIndex – przechowuje indeks aktywnej zakładki. Ustawiając tę własność

możemy sami wybrać aktywną zakładkę
- `PageCount` – przechowuje informację o liczbie zakładek
- `Pages[integer index]` – zwraca referencję do obiektu klasy `TTabSheet` o podanym przez nas indeksie
- `ActivePage` – zwraca referencję do aktywnej zakładki (obiektu klasy `TTabSheet`)

Jeżeli sami tworzymy obiekt, sami też musimy zadbać o jego usunięcie pod koniec pracy programu. Wyjątkowo w przypadku komponentów nie musimy o tym pamiętać. Tworząc obiekt klasy `TTabSheet` podaliśmy referencję do jego właściciela, czyli obiektu `PageControl`. Komponenty VCL, zanim same zostaną zniszczone, zadbają o to, żeby zwolnić pamięć zajmowaną przez komponenty, których są właścicielami.

Wiemy już jak stworzyć zakładki, brakuje jeszcze komponentu do obsługi tekstu, który także musimy stworzyć dynamicznie i umieścić na zakładce. Można to zrobić dodając do procedury `DodajZakladke` nową zmienną `RichEdit` klasy `TRichEdit` oraz następujący kod:

```
procedure DodajZakladke(APageControl: TPageControl);
var Zakladka: TTabSheet;
    RichEdit : TrichEdit;
begin
...
  RichEdit := TrichEdit.Create(Zakladka);
  RichEdit.Parent := Zakladka; // Konieczne jest przypisanie rodzica
  RichEdit.Align := alClient; // RichEdit rozciągnięty na całą zakładkę
{ w razie konieczności pojawią się paski przewijania}
  RichEdit.ScrollBars := ssBoth;
end;
```

Usunięcie zakładki odbywa się przez wywołanie dla niej procedury `Free`.

Nie odpowiedzieliśmy jeszcze na najważniejsze pytanie, czyli jak zarządzać kilkoma otwartymi dokumentami na raz. Poznaliście dotąd takie struktury dynamiczne jak listy jedno- dwukierunkowe, stosy i kolejki. W środowisku Borland'a nie musimy sami tworzyć takich struktur. Do dyspozycji mamy np. obiekt klasy `TObjectList`, który jak sama nazwa wskazuje jest listą obiektów. Udostępnia wszystkie podstawowe metody potrzebne do dodawania, usuwania, sortowania obiektów. Obiekt ten może być także właścicielem obiektów (własność `OwnsObjects` ustawiona na `true`), co oznacza, że sam zadba o ich usunięciu zanim zostanie zniszczony.

Z każdym dokumentem będzie związana nowa zakładka (`TabSheet`) oraz komponent do edycji (`Memo` lub `RichEdit`). Warto by także przechowywać ścieżkę dostępu do pliku, który edytujemy. Możemy stworzyć własną klasę, która będzie odpowiadała pojedynczemu dokumentowi i będzie zawierać trzy obiekty.

```
TDocument = class(TObject)

    TabSheet  : TTabSheet;
    RichEdit  : TRichEdit;
    FileName  : String;
    ...
  end;
```

Oczywiście klasa powinna jeszcze zawierać konstruktor i destruktor. W konstruktorze powinien znaleźć się kod podobny do tego, który jest w procedurze `DodajZakladke`. Obiekty tej klasy mogą być przechowywane w liście `ObjectList`. Jednak nie jest to jedyne możliwe rozwiązanie.

Można wykorzystać fakt, że komponent `PageControl` przechowuje obiekty typu `TTabSheet`. Wykorzystajmy więc możliwości jakie daje nam programowanie obiektowe i zaprojektujmy klasę `TDocument` tak, aby dziedziczyła po klasie `TTabSheet`.

```
TDocument = class(TTabSheet)
  public
    RichEdit  : TRichEdit;
    FileName  : String;
    constructor Create(APageControl: TPageControl);
    constructor CreateFromFile(APageControl: TPageControl; AFileName:
String);
    ...
  end;
```

Konstruktor takiej klasy może wyglądać np. tak:

```
constructor TDocument.Create(APageControl: TPageControl);
begin

  { Istotne jest wywołanie konstruktora klasy, po której dziedziczymy, co
ma miejsce w następującej lini }
  inherited Create(APageControl);
  PageControl := APageControl;
  Caption:='Nowy dokument';
  { słówko Self oznacza odwołanie się do nowostworzonego obiektu klasy
Tdocument}
  RichEdit := TRichEdit.Create(Self);
  RichEdit.Parent := Self;
  RichEdit.Align := alClient;
  RichEdit.WordWrap := false;
  RichEdit.ScrollBars := ssBoth;

end;
```

Powyższy konstruktor tworzy nowy, pusty dokument. Dodajmy metodę, której użyjemy do otwierania istniejącego dokumentu.

```
procedure TDocument.LoadFromFile(AFilename: String);
begin
  FileName :=  AfileName;
  RichEdit.Lines.LoadFromFile(FileName);
  {Procedura ExtractFileName pozwala na wyciągniecie samej nazwy pliku z
pełnej ścieżki dostępu. Nazwa zakładki będzie nazwą pliku.}
  Caption := ExtractFileName(FileName);
end;
```

Lista łańcuchów (StringList)

Klasa `TStringList` jest wyspecjalizowaną klasą służącą do przechowywania łańcuchów tekstowych, może być użyta do podstawienia bezpośrednio do własności `Lines` komponentów `RichEdit` i `Memo`.

Najważniejszym polem tej klasy, służącym do przechowywania łańcuchów, jest pole `Strings`. Ustawienie pierwszego elementu listy może wyglądać na przykład tak:

```
StringList1.Strings[0] := 'To jest pierwszy ciąg znaków';
```

Pole to jest własnością domyślną – jeśli podany zostanie tylko indeks elementu, zostanie przyjęte, że ten indeks dotyczy właśnie pola `Strings`. Tak więc powyższy fragment kodu jest równoważny

z następującym:

```
StringList1[0] :='To jest pierwszy ciąg znaków';
```

Obiekt tej klasy wymaga zainicjowania – używany w tym celu jest konstruktor `Create`, oraz zwolnienia pamięci – najwygodniej do tego zadania użyć metody `Free`, która jeśli obiekt był zainicjowany wywołuje destruktor obiekty, a nie powoduje błędu w przeciwnym przypadku.

Tworzenie obiektu klasy `TStringList` ilustruje poniższy przykład:

```
var
  StringList: TStringList;
begin
    StringList := TStringList.Create;
    StringList.Add('Nowy element');
   StringList.Free;
 end;
```

W przykładzie wykorzystano również metodę `Add`, która dodaje nowy element na końcu istniejącej listy.

Algorytm podziału zmiennej łańcuchowej na podłańcuchy

Przeprowadzenie większości analiz wymaganych w projekcie wymagało będzie podziału dłuższych łańcuchów na podłańcuchy. Poszczególne podłańcuchy oddzielone są określonym separatorem np. ‘ ‘, ‘.’, ‘;’ lub ‘,’.

Dostarczyliśmy Wam funkcję `Tokenize` która wykonuje właśnie to zadanie. Jest ona zapisana w pliku modułu. Jedyne co musicie zrobić to włączyć ów moduł do programu (a jak to zrobić powinniście wiedzieć z kursu). W praktyce programiści bardzo często korzystają z funkcji czy bibliotek pochodzących od osób trzecich. I aby z nich korzystać, nie trzeba rozumieć co się w środku dzieje, wystarczy wiedzieć jakich parametrów wymaga dana funkcja (procedura) i w jaki sposób zwraca wynik. Nasza funkcja `Tokenize` dzieli łańcuch s na podłańcuchy, które zwraca w postaci listy typu `TStringList`. Pierwszy argument, `s`, jest zmienną typu łańcuchowego zawierającą tekst do podziału. Separator jest przekazywany funkcji jako drugi parametr funkcji `delimiter`. Np. wywołanie funkcji `Tokenize`(‘Ala ma kota.’, ‘ ‘) zwróci listę zawierającą trzy elementy: ‘Ala’, ‘ma’ i ‘kota.’.

```
function Tokenize (s: String; delimiter: String) : TStringList;
var
  StringList: TStringList;
  l: integer; //length of s
  start,delimp : integer;
  sub : string;
begin
    StringList := TStringList.Create;
    start:=0;
    if Pos(delimiter, s)>0 then
    begin
      delimp := Pos(delimiter, s);
      repeat
        l:=Length(s);
        sub:=Copy(s,start,delimp-1);
        if sub <> '' then StringList.Add(sub);
        s:=Copy(s,delimp+1,l-delimp);
        delimp := Pos(delimiter, s);
      until delimp = 0;
      StringList.Add(s); // dodanie tego co zostało na końcu
```

```
    end
    else StringList.Add(s);
    Result:=StringList;
end;
```

Powyższa funkcja ma pewne ograniczenia – separatorem może być jeden lub więcej znaków, jednakże nie możliwe jest podzielenie poprzez kilka równorzędnych separatorów.

Prezentacja statystyki w postaci tekstowej.

W celu zaprezentowania wyników analizy statystycznej tekstu programista może wykorzystać kolejny komponent typu `TMemo`, w którym umieści wyniki w postaci tekstowej, np. tak:

Podany tekst zawiera 352 słów, 45 wyrazów, 4 zdań, 1 akapit.

Łańcuchy znakowe można do siebie dodawać, a liczbę całkowitą na jej reprezentację tekstową konwertujemy wykorzystując funkcję `IntToStr`. Tak więc powyższy napis można stworzyć przy pomocy następującego kodu:

```
procedure TForm1.Button1Click(Sender: TObject);
  var ll, ls, lz, la : integer;
begin
  ll := 352;
  ls := 45;
  lz := 4;
  la := 1;
  Memo1.Lines.Add('Podany tekst zawiera ' + IntToStr(ll) + ' słów, ' +
                  IntToStr(ls) + ' wyrazów, '+
                  IntToStr(lz) + ' zdań, ' +
                  IntToStr(la) + ' akapit.' );
end;
```

Bardziej elegancką metodą jest wykorzystanie komponentów specjalizowanych na przykład paska stanu `StatusBar`. W trakcie pracy programu dostęp do poszczególnych paneli uzyskujemy poprzez odwołanie się do poszczególnych pól obiektu *Panels* w komponencie *StatusBar*, przykładowo instrukcja:

```
StatusBar1.Panels.Items[3].Text := 'Zdań: 10';
```

Inną możliwością prezentacji wyników analizy jest skorzystanie z komponentu `StringGrid` z palety `Additional`. Przykładowo, aby w polu umieszczonym w 2 kolumnie i 3 wierszu umieścić tekst *Ala,* wykorzystujemy następujący kod:

```
StringGrid1.Cells[1, 2] := 'Ala';
```

Zarówno kolumny, jak i wiersze są numerowane od 0.

Analiza liczby znaków w wyrazie

Celem tej analizy jest ustalenie najczęściej spotykanej liczby znaków w wyrazie. Efektem analizy powinna być tabela, zawierająca w pierwszym wierszu liczbę znaków, w drugim zaś liczbę słów, które mają dokładnie tyle znaków. Sposób prezentacji wyników powinien naprowadzić nas na rozwiązanie problemu – czyli stworzenie algorytmu.

Załóżmy więc że mamy tablicę liczb całkowitych o `n` polach. Chcemy ją wypełnić danymi w ten sposób, aby pole o indeksie `i` zawierało liczbę słów w tekście które mają `i` liter. Innymi słowy, jeśli

nazwiemy naszą tablicę `TablicaWynikowa`, to jeśli element `TablicaWynikowa[3]` będzie zawierał wartość 10, to oznacza że w tekście było dokładnie 10 słów trzyliterowych.

Teraz algorytm wypełnienia tablicy danymi może wyglądać następująco:

- Dzielimy tekst na wyrazy umieszczone w liście napisów (`TStringList`). Do tego już jest gotowa procedura.
- Inicjujemy tablicę wynikową zerami.
- Dla każdego słowa:
    - Obliczamy jego długość
    - Zwiększamy w tablicy wynikowej pole o indeksie równym tej długości o jeden

Pozostaje jeszcze jeden problem: a co jeśli w tekście znajdzie się słowo dłuższe niż rozmiar tablicy wynikowej? Wtedy nie ma dla niego pola. Rozwiążemy go następująco – ostatnie pole w tablicy będzie oznaczało liczbę wyrazów o długości `n` lub większej. Tak więc docelowy algorytm będzie wyglądał następująco:

```
const max_n_znakow = 20;
type TTablica = array[1..max_n_znakow] of integer;

procedure LiczZnaki(slowa : TStringList, var TablicaWynikowa : TTablica);
var i, dl : integer;

begin
  { najpierw zainicjujmy tablicę wyników }
  for i := 1 to max_n_znakow do
    TablicaWynikowa[i] := 0;

  { następnie policzmy odpowiednie słowa }
  for i := 0 to slowa.Count-1 do
  begin
    dl := Length(slowa[i]);
    if dl > max_n_znakow then
      TablicaWynikowa[max_n_znakow] := TablicaWynikowa[max_n_znakow] + 1
    else if dl > 0 then
      TablicaWynikowa[dl] := TablicaWynikowa[dl] + 1;
  end;
end;
```

W zaprezentowanej postaci procedura jest zupełnie niezależna, lecz nic nie stoi na przeszkodzie, by uczynić ją metodą jednej z form.

Podsumowanie – plan prac

1. Przygotowanie głównego okna aplikacji, umieszczenie potrzebnych komponentów.
2. Przygotowanie klasy `TDocument` w osobnym module.
3. Dołączenie modułu z funkcją `Tokenize`.
4. Przygotowanie odpowiednich akcji oraz ikon je reprezentujących (otwarcie, zapis, statystyka).
5. Wyświetlenie raportu.
6. Testowanie programu i wprowadzenie dodatkowych zabezpieczeń.

PROJEKT 2 – RYSOWANIE KRZYWEJ BEZIERA (łatwy lub średni)

Wprowadzenie

Zadanie polega na napisaniu programu rysującego krzywą Beziera na podstawie punktów podanych przez użytkownika.

Wymagane umiejętności

- zasad tworzenia GUI,
- podstawowych komponentów wizualnych i ich właściwości,
- podstaw grafiki w środowisku Borland – płótno, pędzelek, pióro,
- tworzenia i dołączania modułów,
- tablic dynamicznych,
- podstaw programowania obiektowego,

na lepszą ocenę zastosowanie:

- list dynamicznych,
- programowania obiektowego.

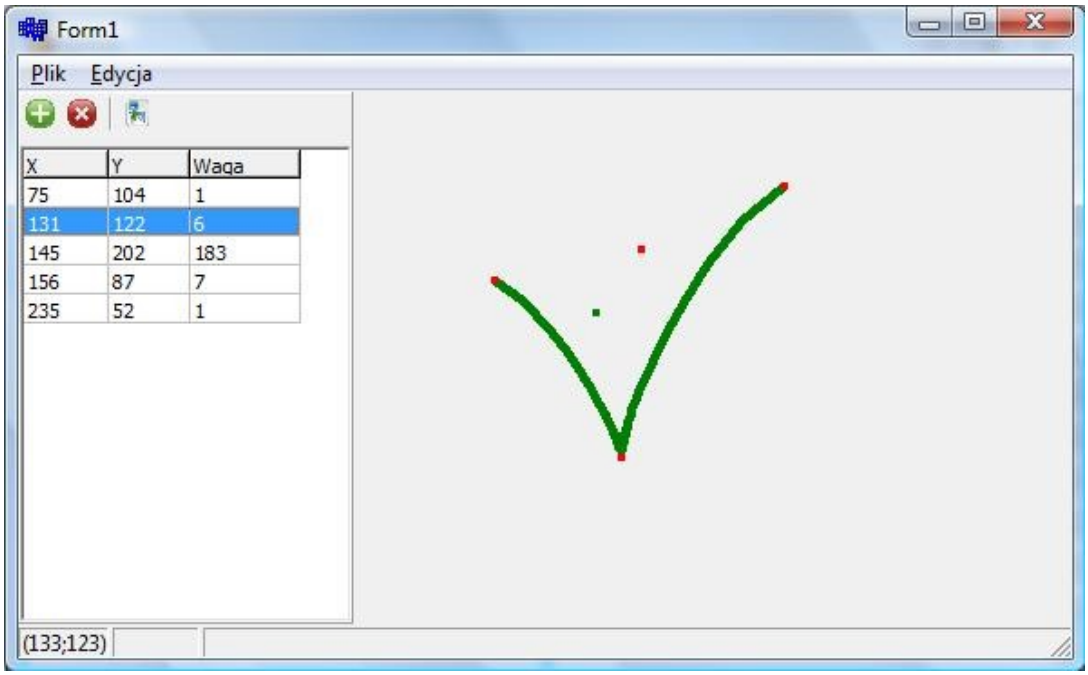
Funkcje podstawowe programu

- Dodawanie, usuwanie i edycja punktów kontrolnych poprzez ręczne wprowadzenie wartości (współrzędne x,y oraz waga).
- Dodawanie (prawy przycisk myszy), usuwanie i edycja (przesunięcie punktu – lewy przycisk myszki, zmiana wagi np. środkowy przycisk myszki) punktów kontrolnych za pomocą myszki.
- Rysowanie krzywej Beziera.
- Rysowanie punktów kontrolnych z wyróżnieniem punktu po najechaniu kursorem.
- Zapis i odczyt współrzędnych krzywej do/z pliku tekstowego.
- Wyświetlanie pozycji kursora w pasku stanu.

Funkcje dodatkowe programu

- Zmiana ustawień pędzla.
- Przechowywanie punktów w liście dynamicznej.
- Zapis obrazu do pliku graficznego.
- Tworzenie wielu krzywych.
- Własne propozycje udogodnień interfejsu

Przygotowanie głównego okna aplikacji



Rys. 1. Projekt 2 - Główne okno programu

Główne okno aplikacji (rys.1) powinno zawierać takie elementy jak:

- menu główne (`MainMenu`)
- pasek stanu (`StatusBar`)
- pasek narzędziowy (`ToolBar`)
- `StringGrid`
- `PaintBox`
- lista akcji `(ActionList)`
- lista obrazów `(ImageList)` – potrzebna do przechowania ikon

Komponent PaintBox

W celu narysowania dowolnego rysunku na `PaintBox'ie`, najprościej odwołać się do właściwości *`Canvas`*, a następnie do właściwości `Pen` oraz metody `LineTo(x:integer; y:integer)` i `MoveTo(x:integer; y:integer)`.

Metoda `LineTo` służy do rysowania prostej linii pomiędzy punktem, w którym aktualnie znajduje się „długopis” (po ang. pen) a punktem wskazanym przez współrzędne x, y. Punktem początkowym zawsze jest x = 0, y = 0. Przy czym należy pamiętać, że jest to lewy górny róg komponentu `PaintBox`, czyli oś Y jest skierowana od góry do dołu.

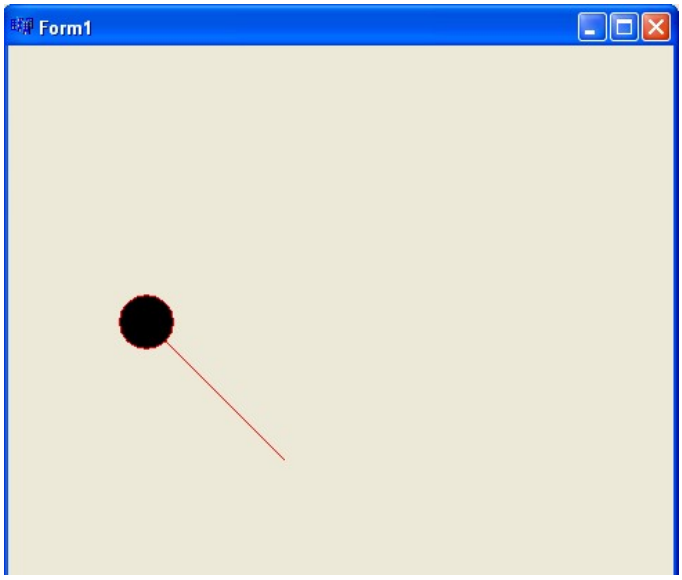
Jeśli nie chcemy rysować linii od punktu 0, 0 możemy wykorzystać metodę `MoveTo`. Służy ona do przemieszczania „długopisu” po powierzchni bez rysowania linii, a poza tym działa analogicznie do metody opisanej powyżej.

Dodatkowo w celu ułatwienia rysowania niektórych kształtów, możemy wykorzystać właściwość `Brush`, należącą do `Canvas`. Kolor rysowania ustawiamy analogicznie do koloru właściwości `Pen`. Jednakże `Brush` w odróżnieniu od `Pen` służy do rysowania gotowych kształtów. Wykorzystując odpowiednio metody `Ellipse` lub `Rectangle` właściwości `Canvas` można narysować elipsę (a więc również i koło) oraz prostokąt.

Aby pokazać jak rysowanie wygląda w praktyce napiszmy prostą procedurę rysującą linię i elipsę:

```
PaintBox1.Canvas.Pen.Color := clRed;
PaintBox1.Canvas.MoveTo(100, 200);
PaintBox1.Canvas.LineTo(200, 300);
PaintBox1.Canvas.Brush.Color := clBlack;
PaintBox1.Canvas.Ellipse(80, 180, 120, 220);
```

W efekcie otrzymamy linię koloru czerwonego narysowaną pomiędzy punktami o współrzędnych 100, 200 a 200, 300 oraz czarną elipsę wpisaną w prostokąt zdefiniowany przez dwa punkty o współrzędnych 80, 180 oraz 120, 220. Widoczne jest to na ilustracjach poniżej 2.



Rys. 2: Rysowanie na płótnie

Zdarzenia do obsługi ruchu myszy

Przy rysowaniu na płótnie `Canvas` komponentu `PaintBox`, pomocne będą zdarzenia `OnMouseDown`, `OnMouseMove` i `OnMouseUp`. Sprawdzenie, który przycisk myszy został wciśnięty można wykonać następująco:

```
if ssLeft in Shift then …;
```

Przykładowa obsługa zdarzenia `OnMouseDown`:

```
procedure TForm1.PaintBox1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
      if ssLeft in Shift then
      PaintBox1.Canvas.Ellipse(X-5, Y-5, X+5, Y+5);
end;
```

Jej działanie polega na rysowaniu niewielkich elips (w praktyce kół), w miejscu kliknięcia lewego przycisku myszy na komponencie `PaintBox1`.

Czasami istnieje potrzeba zapamiętania położenia myszki w momencie wciśnięcia przycisku na przykład, żeby móc obliczyć przesunięcie kursora wewnątrz procedury obsługującej zdarzenie `OnMouseMove`. Można w tym celu stworzyć zmienną typu `TPoint` i zadeklarować ją w pliku w sekcji private definicji klasy `TForm1`:

```
private
    { Private declarations }
      mPos :TPoint;
```

Rysowanie krzywej Beziera

Teoria dotycząca tworzenia krzywych Beziera nie jest łatwa, jednak na potrzeby projektu wykorzystamy gotowe funkcje. Funkcje te należy umieścić w osobnym module. W module tym musimy także umieścić definicję nowego typu `SWezel`, w którym będą przechowywane informacje o pojedynczym punkcie.

```
type
SWezel = record
  waga, x, y :integer;
end;
TabWezly = array of SWezel;
WskWezly = ^TabWezly;
```

Oprócz współrzędnych `x, y` punktu kontrolnego potrzebny jest jeszcze dodatkowy parametr `waga`. Domyślnie powinien wynosić on 1, ogólnie powinien być większy lub równy 0. Jego wartość określa „siłę przyciągania” krzywej do punktu kontrolnego. Tablicę dynamiczną elementów `SWezel`, podobnie jak jej rozmiar, najlepiej zadeklarować w sekcji private definicji klasy `TForm1`:

```
private
    { Private declarations }
      wezly :TabWezly;
      liczbaWezlow :integer;
```

Aby poniższe funkcje zadziałały konieczne jest jeszcze napisanie funkcji `silnia` i umieszczenie jej w tym samym module.

```
function WielomianB(i, n :integer; u :real) :real;
var
  pom :real;
begin
  if (i < 0) or (i > n) then
    pom := 0
  else
  begin
    if i = 0 then
      pom := power(1-u, n)
    else
      pom := silnia(n) * power(u, i) * power(1-u, n-i) / (silnia(i)
                                                    * silnia(n-i));
  end;

  WielomianB := pom;

end;

procedure RysujKrzywa(tablicaWezlow :WskWezly; liczbaWezlow :integer ;
Canvas:TCanvas);
var
  x, y, u, licznikX, licznikY, mianownik :real;
  i :integer;
begin
```

```
{Zmienna liczbaWezlow przechowuje ilość punktów znajdujących się w tablicy}
  if liczbaWezlow > 2 then
  begin

    Canvas.MoveTo(tablicaWezlow^[0].x,tablicaWezlow^[0].y);
    u := 0;
    repeat
      licznikX := 0;
      licznikY := 0;
      mianownik := 0;
      for i := 0 to liczbaWezlow-1 do
      begin
        licznikX := licznikX + tablicaWezlow^[i].waga*tablicaWezlow^[i].x*
                                  WielomianB(i, liczbaWezlow-1, u);
        licznikY := licznikY + tablicaWezlow^[i].waga*tablicaWezlow^[i].y*
                                  WielomianB(i, liczbaWezlow-1, u);
        mianownik := mianownik + tablicaWezlow^[i].waga*
                                  WielomianB(i, liczbaWezlow-1, u);
      end;
      if mianownik <> 0 then
      begin
        x := licznikX/mianownik;
        y := licznikY/mianownik;
      end
      else
      begin
        x := 0;
        y := 0;
      end;
      Canvas.LineTo(round(x), round(y));
      u := u+0.01;
    until u > 1;
  end;

end;
```

Procedura `RysujKrzywą` jako parametry przyjmuje wskaźnik na początek listy dynamicznej węzłów, rozmiar tablicy węzłów oraz wskaźnik do płótna, na którym krzywa ma być narysowana. Ze względu na obecność wskaźnika do zmiennej `Canvas` do modułu rysowania należy dołączyć moduł `vcl`.

Ręczna edycja wartości współrzędnych węzłów – StringGrid

Do precyzyjnej edycji wartości współrzędnych punktów kontrolnych wygodny będzie komponent `StringGrid`. Ponieważ nie wiemy z góry ile punktów wprowadzi użytkownik nie możemy od razu stworzyć tablicy dynamicznej. Dlatego wygodnym rozwiązaniem będzie przechowywanie punktów w elemencie `StringGrid`, a dopiero w momencie wywołania akcji `Rysuj` stworzenie odpowiedniej tablicy dynamicznej i przepisanie do niej wartości z komponentu `StrinGrid`. Przy dodaniu nowych punktów, można zastosować mało eleganckie, ale praktyczne rozwiązanie polegające na usunięciu poprzedniej tablicy węzłów, stworzenie nowej większej i przepisanie do niej wartości tak jak poprzednio.

Podsumowanie – plan prac

1. Stworzenie formatki głównego okna, umieszczenie niezbędnych komponentów
2. Stworzenie dodatkowego modułu z funkcjami do rysowania krzywej Beziera, w tym napisanie funkcji `silnia`.
3. Dołączenie modułu rysowania do głównego modułu programu.
4. Zadeklarowanie dodatkowych zmiennych (wskaźnik do tablicy dynamicznej, liczba

węzłów) w definicji klasy `TForm1`.
5. Zdefiniowanie akcji `dodajPunkt` i `usunPunkt` z komponentu `StringGrid`.
6. Napisanie procedury kopiującej dane z komponentu `StringGrid` do tablicy dynamicznej.
7. Zdefiniowanie akcji `Rysuj`.
8. Stworzenie dodatkowych metod do rysowania punktów kontrolnych (węzłów).
9. Obsługa zdarzeń myszy.
10. Zapis i odczyt z/do pliku.
11. Testowanie programu i wprowadzenie dodatkowych zabezpieczeń.

PROJEKT 3 – PRZEGLĄDARKA ZDJĘĆ (średni lub trudny)

Wprowadzenie

Zadanie polega na stworzeniu aplikacji pozwalającej na łatwe i szybkie przeglądanie plików graficznych JPG.

Wymagane umiejętności

- zasad tworzenia GUI,
- podstawowych komponentów wizualnych i ich właściwości,
- podstaw grafiki w środowisku Borland – płótno,
- obsługa plików graficznych – TImage, TBitmap, TJPEGImage,
- operacje na obrazie,
- tworzenia i dołączania modułów,
- podstaw programowania obiektowego,

na lepszą ocenę zastosowanie:

- operacji na tablicach wskaźników,
- programowania obiektowego.

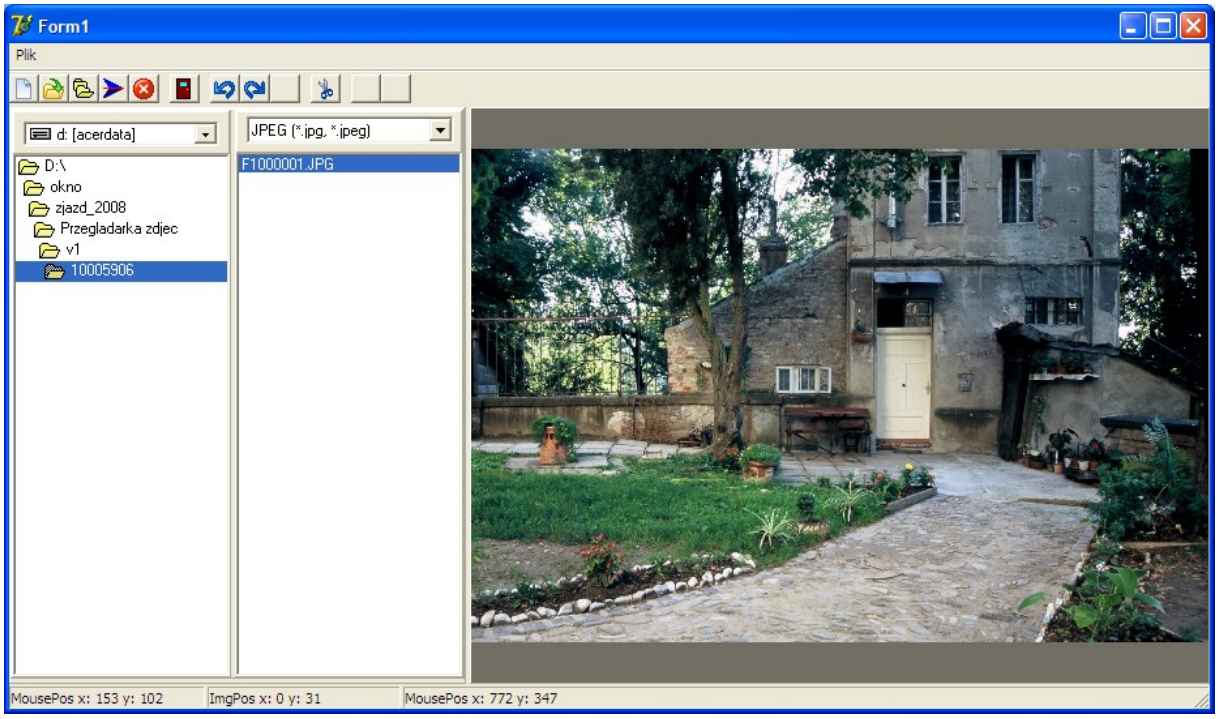
Funkcje podstawowe programu

- Przeglądanie dysków w poszukiwaniu obrazów
- Otwieranie, zapisywanie plików graficznych
- Obroty obrazu w lewo/prawo
- Skalowanie obrazu
- Kadrowanie
- Wyświetlanie pozycji kursora w pasku stanu

Funkcje dodatkowe programu

- Otwarcie wielu plików
- Wyświetlenie obrazu na pełnym ekranie
- Własne propozycje udogodnień interfejsu

Przygotowanie głównego okna aplikacji



Rys. 3: Projekt 3 - główne okno apliakcji

Główne okno aplikacji (rys.Błąd: Nie znaleziono źródła odwołania) powinno zawierać takie elementy jak:

- menu główne (`MainMenu`)
- pasek stanu (`StatusBar`)
- pasek narzędziowy (`ToolBar`)
- komponenty do otwarcia pliku (`DriveComboBox, DirectoryListBox, FileListBox, FilterComboBox`)
- `Image`
- lista akcji (`ActionList`)
- lista obrazów (`ImageList`) – potrzebna do przechowania ikon

Komponenty do przeglądania katalogów

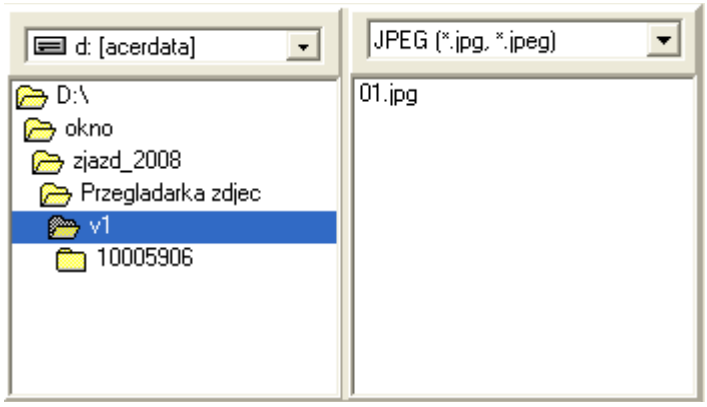
Poznaliście już okna dialogowe służące do otwierania plików. Jednak przy przeglądaniu dużej ilości plików każdorazowe używanie okna dialogowego nie jest wygodne. Istnieją komponenty, które znacznie ułatwiają przeglądania katalogów bez wykorzystania okien dialogowych. Znajdują się one w zakładce `Win3.1` i są to:

- `DriveComboBox` – wybór dysku

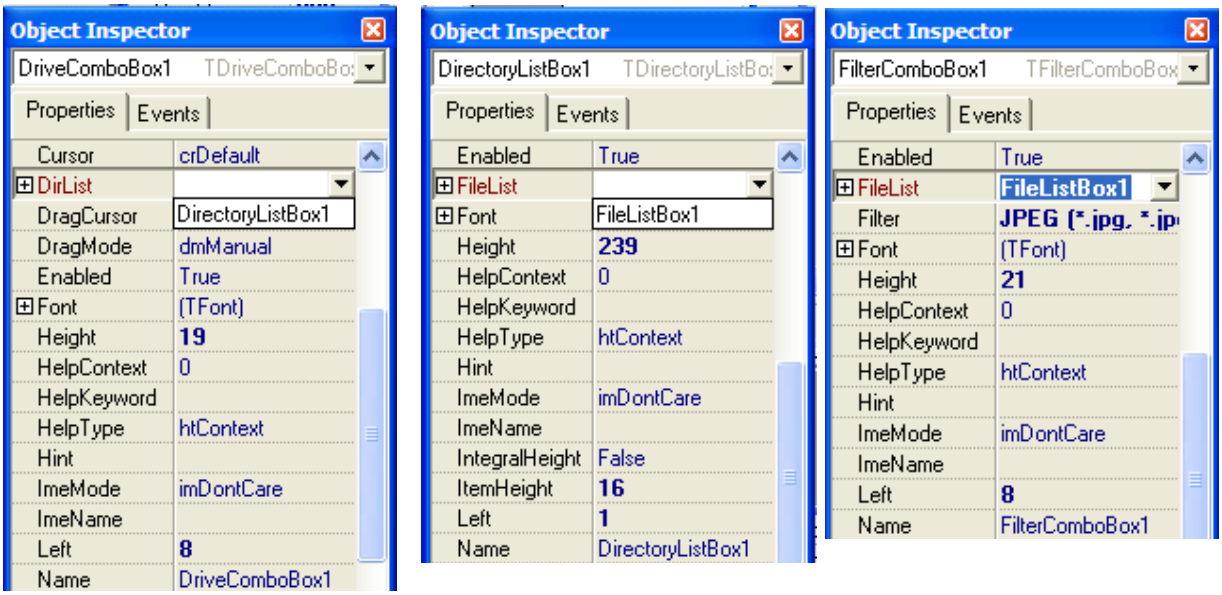
- `DirectoryListBox` – wybór katalogu

- `FileListBox` – wybór pliku

- `FilterComboBox` – wybór rozszerzenia plików



Komponenty te są przystosowane do współpracy. Po umieszczeniu wszystkich komponentów na formie musimy je ze sobą powiązać. Dla komponentu `DriveComboBox` musimy ustawić własność `DirList` w `ObjectInspectorze`, gdzie z rozwijanej listy wybieramy wcześniej umieszczony na formatce komponent `DirectoryListBox1` (jak na rysunku poniżej). Podobnie dla komponentów `DirectoryListBox1` i `FilterComboBox1` ustawiamy własność `FileList` na `FileListBox1`. Ważną własnością jest także `Filter` komponentu `FilterComboBox1`, gdzie wpisujemy wszystkie rozszerzenia, które chcemy aby były uwzględnione przy wyświetlaniu zawartości katalogów.



Komponent `FileListBox` pozwala na wybór pliku. Chcąc wykonać jakąś operację na pliku wskazanym w `FileListBox` możemy wykorzystać zdarzenie `OnClick` (po kliknięciu myszką) lub `OnChange` (po wybraniu pliku innego niż poprzednio wybrany). Pełna ścieżka dostępu do wybranego pliku przechowywana jest przez własność `FileName`.

Zawsze zanim wykonamy jakąś operację na pliku dobrze jest sprawdzić czy on istnieje, do czego służy funkcja `FileExists` z modułu `SysUtils`. Zwraca ona wartość logiczną `true` lub `false`.

Otwarcie pliku

Plik graficzny najwygodniej jest otworzyć za pomocą obiektu klasy `TJPEGImage` z modułu `jpeg`. Warto zadeklarować zmienną `JPEGImage` w sekcji private definicji klasy `TForm1`.

```
type  TForm1 = class(TForm)
```

```
  private
    JPEGImage1 :TJPEGImage;
  ...
end;
```

Obsługa kółka myszy – zdarzenie OnMouseWheel

Komponent `Image` obsługuje standardowe zdarzenia myszy, z których przydadzą się `OnMouseMove` (ruch myszy), `OnMouseUp` (puszczenie dowolnego przycisku myszy), `OnMouseDown` (wciśnięcie dowolnego przycisku myszy). Niestety ruch kółka myszy obsługiwany jest jedynie przez zdarzenie `OnMouseWheel` głównej formy (w naszym przypadku `Form1`).

Wewnątrz zdarzenia `OnMouseWheel` interesujące mogą być informacje o stanie przycisków myszy (`Shift`), wartości obrotu kółka (`WheelDelta`) oraz pozycji kursora (`MousePos`). Niestety pozycja kursora podana jest w globalnym układzie współrzędnych związanym z ekranem a nie z komponentem `Image`. Można ją jednak łatwo przeliczyć za pomocą metody `ScreenToClient`.

```
procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
  WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);
var  MousePosImg : Tpoint;
begin
        MousePosImg := Image1.ScreenToClient(MousePos);
        StatusBar1.Panels.Items[0].Text := 'x: ' + IntToStr(MousePosImg.x);
        StatusBar1.Panels.Items[1].Text := 'y: ' + IntToStr(MousePosImg.y);
        StatusBar1.Panels.Items[2].Text := 'delta: ' + IntToStr(WheelDelta);
end;
```

Procedura ta spowoduje wyświetlenie w pasku stanu pozycji kursora, w momencie obrotu kółka myszy, liczonej względem komponentu `Image1`.

Kadrowanie

Chcąc wykadrować obraz musimy mieć możliwość narysowania prostokąta określającego nowe krawędzie obrazu. Własność *Canvas* dysponuje metodami, które znacznie ułatwiają rysowanie. Kod umieszczony poniżej spowoduje narysowanie prostokąta, którego lewy górny róg znajduje się w punkcie (10,20), a prawy dolny róg w punkcie (30,30).

```
Image1.Canvas.Rectangle(10,20,30,30);
```

Zwykły prostokąt (`Rectangle`) raz narysowany zostanie dopóki nie przeładujemy ponownie obrazu, co nie jest rozwiązaniem optymalnym. Użyjemy specjalnie do tego celu stworzoną metodę `DrawFocusRect` obiektu klasy `TCanvas`, która ma tę własność, że rysując ponownie ten sam prostokąt powodujemy jego zniknięcie bez przeładowywania wcześniej załadowanego obrazu.

Rysowanie zaznaczenia najlepiej umieścić w procedurze wywoływanej zdarzeniem `OnMouseMove`. Konieczne też będzie wykorzystanie zdarzenia `OnMouseDown`, w którym zapamiętamy początkowe położenie kursora. Przydadzą nam się także dwie dodatkowe zmienne, które możemy zadeklarować jako zmienne globalne, bądź pola prywatne klasy `TForm1`. Zmienna `mousePos` typu `TPoint`, który jest strukturą składającą się z dwóch pól *X,Y* typu `Integer`, zostanie wykorzystana do zapamiętania pozycji myszy po wciśnięciu lewego przycisku. Musimy także zapamiętać pozycję poprzednio narysowanego prostokąta, do czego posłuży nam zmienna `lastRect` typu `TRect`, zawierająca pola `Left`, `Top`, `Right`, `Bottom` typu `Integer` kolejno odpowiadające lewej, górnej, prawej i dolnej krawędzi prostokąta.

```
TForm1 = class(TForm)
...
  private
    lastRect: TRect; //ostatnio narysowany prostokąt
    mousePos: TPoint; //pozycja kursora po wciśnięciu lewego przycisku
...
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  mousePos.X := X;
  mousePos.Y := Y;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var  ppoint:TPoint;
begin

  //rysujemy tylko jeżeli wciśniety jest lewy przycisk myszy
  if ssLeft in Shift then
  begin

    //przerysowanie poprzedniego prostokąta
    Image1.Canvas.DrawFocusRect(lastRect);
    // zapamiętanie nowego prostokąta
    lastRect.Left:=mousePos.X;
    lastRect.Right:=X;
    lastRect.Top:=mousePos.Y;
    lastRect.Bottom:=Y;
    //rysowanie nowego prostokąta
    Image1.Canvas.DrawFocusRect(lastRect);

  end;
end;
```

Powyższy kod umożliwia narysowanie prostokąta tylko wtedy, gdy po kliknięciu lewym przyciskiem poruszymy myszką po skosie w prawo i w dół. Należy jeszcze wprowadzić odpowiednie warunki sprawdzające, czy górny lewy róg ma współrzędne o mniejszych wartościach niż prawy dolny.

Uwaga! Niestety rysowanie na płótnie komponentu `Image` nie uda się w przypadku, gdy bezpośrednio załadujemy plik w formacie JPEG. Można jednak obejść tę przeszkodę otwierając plik za pomocą obiektu `TJPEGImage`, a następnie przerysować obraz na płótno komponentu

Image.

Przesuwanie i zoom

Do wygodnego przeglądania obrazu konieczna jest możliwość powiększenia oraz przesuwania powiększonego obrazu. Zadanie można to wykonać na dwa sposoby. Najprostsze rozwiązanie to przesuwanie oraz zmiana rozmiaru komponentu Image1 (musi mieć własność Align ustawioną na alCustom). Rozwiązanie to choć łatwe powoduje jednak miganie obrazu. Trudniejsze ale lepsze jest rozwiązanie, w którym obraz przechowujemy cały czas w obiekcie niewizualnym i przerysowywanie tylko tego fragmentu, który akurat mieści się w obrębie komponentu Image1.
Pozycja obrazu w obiekcie klasy TJPEGImage jest umowna, ponieważ tak naprawdę nie jest on fizycznie usytuowany na formie. Zakładamy, że pozycję obrazu (położenie lewego górnego rogu) przechowujemy w zmiennej imgTopLeft typu TPoint dostępnej dla procedur formy Form1. Podobnie mamy także zapamiętaną pozycję kursora myszy w zmiennej mousePos oraz skalę. W zdarzeniu OnMouseMove, który częściowo wykorzystaliśmy wcześniej, musimy napisać obsługę przesuwania, gdy wciśnięty jest prawy przycisk myszki. W procedurze tej zakładamy, że istnieje jakaś inna procedura (SkalujIPrzerysuj), która przerysowuje wybrany obszar z uwzględnieniem skali z obiektu JPEGImage na płótno komponentu Image1 (rozdziały *Kopiowanie fragmentu obrazu* oraz *Skalowanie w podręczniku*).

```
procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
...
  if ssRight in Shift then
  begin
      imgTopLeft.X:= imgTopLeft.X + X - mousePos.X;
      imgTopLeft.Y:= imgTopLeft.Y + Y - mousePos.Y;
      SkalujIPrzerysuj(JPEGImage1, zoom, imgTopLeft);
      mousePos.X := X;
      mousePos.Y := Y;

  end;
end;
```

Operację zbliżania i oddalania obrazu możemy zrealizować równie łatwo. Jak wspomnieliśmy wcześniej obsługę ruchu kółka myszy możemy umieścić jedynie w procedurze zdarzenia OnMouseWheel głównej formy.

```
procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
  WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);
var skok : Double;

begin
  skok:= 1.1; //zakładamy wartość skoku przy zoomowaniu

  {zmienna WheelDelta przyjmuje wartości dodatnie lub ujemne w zależności
od kierunku obrotu kółka myszy. Załużmy, że rozmiar będzie zmniejszany lub
zwiększany o stały współczynnik skok = 1,1}

  if WheelDelta<0 then
    zoom := zoom/skok
  else   zoom := zoom*skok;

  SkalujIPrzerysuj(JPEGImage1, zoom, imgTopLeft);

end;
```

Skalowanie jest przekształceniem, w którym zawsze jeden punkt jest stały. W naszym przypadku jest to lewy górny róg obrazu. Aplikacja byłaby znacznie przyjemniejsza w obsłudze, gdyby punktem stałym był punkt znajdujący się pod kursorem myszy. W takim przypadku sami musimy policzyć, gdzie powinien znaleźć się lewy górny róg obrazu tak, aby punkt obrazu, nad którym znajduje się kursor, po przeskalowaniu znalazł się w tym samym miejscu. Zarówno płótno komponentu `Image1` jak i obraz mają swoje układy współrzędnych. Gdy zmniejszymy skalę układu współrzędnych to wszystkie punkty znajdujące się w tym układzie zbliżą się do jego środka oprócz jednego punktu, który już tam jest. Dlatego pierwszym krokiem będzie takie przesunięcie obrazu, aby w środku jego układu współrzędnych znalazł się punkt, który ma być punktem stałym. Następnie możemy dokonać przeskalowania. W tym momencie mamy już przeskalowany obraz, ale punkt, który wcześniej był pod kursorem myszy jest teraz w lewym górnym rogu płótna komponentu `Image1`. Musimy, więc go przesunąć z powrotem w miejsce, w którym był wcześniej, czyli do punktu, w którym znajduje się kursor myszy. Napiszmy funkcję, która wykona odpowiednie obliczenia.


```
procedure ObliczPozycje(var AimgTopLeft : TPoint; mousePos: TPoint;
  zoomBef, zoomAft: Double);
begin

  {zmienna zoomBef oznacza poprzednią wartość skali, a zmienna zoomAft nową
wartość skali po ruchu kółka myszki }
  AimgTopLeft.X:= Round((AimgTopLeft.X - mousePos.X)*zoomAft/zoomBef +
mousePos.X);
  AimgTopLeft.Y:= Round((AimgTopLeft.Y - mousePos.Y)*zoomAft/zoomBef +
mousePos.Y);

end;
```

Niestety pojawia się problem, o którym wspomnieliśmy wcześniej. Ponieważ procedurę `ObliczPozycje` musimy wywołać z procedury `FormMouseWheel` to pozycja myszy jest pozycją w globalnym układzie współrzędnych, a nam potrzebna jest pozycja myszki na komponencie `Image1`.

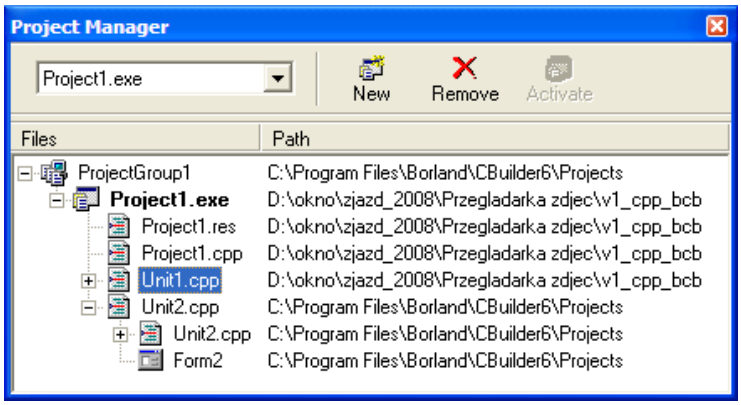
Obroty

Wykonanie obrotu obrazu wymaga dostępu do pojedynczych pikseli. Operacja ta została dokładnie opisana w podręczniku.

Wyświetlanie obrazu na pełnym ekranie

Do wyświetlenia obrazu na pełnym ekranie wykorzystamy osobne okno. W tym celu musimy z menu głównego wybrać `File | New | Form`. Pojawi się nowy moduł `Unit2`, który my

nazwaliśmy `FullScreen`. Jeżeli wszystko wykonaliśmy poprawnie w `Project Managerze` (`View | Project Manager`) powinien pojawić się taki widok.



Klikając dwukrotnie na `Form2` otworzymy nowo utworzoną formę. Na formie umieśćmy dobrze już nam znany komponent `Image` z zakładki `Additional` i ustawmy jego własność `align` na `alClient`, oraz własności `Proportional` i `Center` na `True`. Możemy zmienić kolor okna na czarny (własność `Color`), ale najważniejsze jest ustawienie własności `BorderStyle` na `bsNone` oraz własności `WindowState` na `wsMaximized`. Pierwsza własność sprawi, że forma nie będzie miała żadnej ramki, a druga spowoduje rozciągnięcie formy na cały ekran.

To co musimy jeszcze zrobić to przewidzieć w jaki sposób zamkniemy naszą formę skoro zajmuje cały ekran i nie ma ramki z przyciskami. Najprościej będzie obsłużyć zdarzenie `OnKeyPress`, gdzie po wciśnięciu klawisza *Esc* zamkniemy naszą formę.

```
procedure TForm2.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = Chr(27) then Close();
end;
```

Pozostaje jeszcze wywołanie `Form2` z poziomu głównego okna aplikacji. Możemy do tego przewidzieć osobną akcję. Oczywiście musimy dołączyć nowy moduł, w którym zadeklarowaliśmy `TForm2`.

```
procedure TForm1.PelnyEkranActExecute(Sender: TObject);
var fform: Tform2;
    JPEGImage1 : TJPEGImage;
begin

  fform:= TForm2.Create(Self);

  JPEGImage1 := TJPEGImage.Create;
  JPEGImage1.LoadFromFile('zdjecie.jpg');
  //wyświetlenie zdjęcia w obiekcie Image1 formy Form2
  fform.Image1.Picture.Assign(JPEGImage1);
  JPEGImage1.Free;
  {pokazujemy formę Form2.ShowModal oznacza, że forma Form1 nie jest
aktywna tak długo jak otwarta jest forma Form2}
  fform.ShowModal;
  fform.Free;

end;
```

Podsumowanie – plan prac

7. Przygotowanie głównego okna aplikacji, umieszczenie potrzebnych komponentów.
8. Powiązanie i skonfigurowanie komponentów odpowiedzialnych za przeglądanie plików.
9. Przygotowanie odpowiednich akcji oraz ikon je reprezentujących.
10. Zadeklarowanie zmiennych w sekcji private deklaracji klasy `TForm1`, m.in. zmienna przechowująca obraz poza ekranem, pozycja obrazu, skala.
11. Napisanie procedury otwierającej plik za pomocą `TJPEGImage`.
12. Napisanie procedury przerysowującej obraz z `TJPEGImage` na `Image1` z uwzględnieniem pozycji obrazu oraz jego skali.
13. Obsługa zdarzeń myszy w celu zapewnienia przesuwania, skalowania i kadrowania obrazu.
14. Napisanie procedur wykonujących kadrowanie i obroty.
15. Zapis zmodyfikowanego obrazu do pliku.
16. Wyświetlenie obrazu na pełnym ekranie.
17. Testowanie programu i wprowadzenie dodatkowych zabezpieczeń.