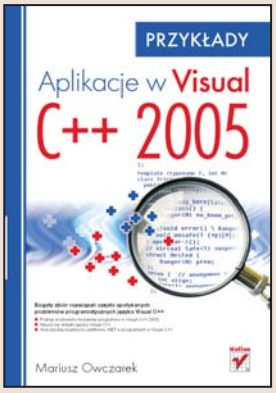
Aplikacje w Visual C++ 2005. Przykłady

Autor: Mariusz Owczarek
ISBN: 978-83-246-0875-1
Format: B5, stron: 216



Bogaty zbiór rozwiązań często spotykanych problemów programistycznych języka Visual C++

- Poznaj środowisko tworzenia programów w Visual C++ 2005
- Naucz się składni języka Visual C++
- Wykorzystaj możliwości platformy .NET w programach w Visual C++

W kontekście programowania przy użyciu platformy .NET zwykle mówi się o językach Visual Basic i C#. Większość stron internetowych, artykułów i książek dotyczących .NET zawiera kod napisany właśnie w nich. Co mają zrobić programiści, którzy od lat używają C++ i wcale nie chcą rezygnować z jego licznych zalet? Czy jedynym wyborem jest nauka nowego języka lub pozostanie poza światem programowania dla .NET?

Książka „Aplikacje w Visual C++ 2005. Przykłady" zawiera dziesiątki krótkich zadań wraz z rozwiązaniami, dzięki którym błyskawicznie zaczniesz wykorzystywać możliwości platformy .NET w programach pisanych w języku C++. Poznasz środowisko Visual C++ 2005 Express Edition oraz podstawowe składniki aplikacji pisanych w Visual C++. Nauczysz się między innymi obsługiwać dane, przetwarzać pliki, korzystać z wątków oraz łączyć aplikacje z internetem. Dowiesz się także, jak wykonać wiele innych operacji niezbędnych w codziennej pracy programisty.

- Środowisko Visual C++ 2005 Express Edition
- Elementy aplikacji języka Visual C++
- Przetwarzanie i wyświetlanie danych
- Praca z plikami
- Używanie okien dialogowych
- Programy wielowątkowe
- Stosowanie grafiki w aplikacjach
- Tworzenie programów używających sieci
- Składnia języka Visual C++ w pigułce

Przyspiesz wykonywanie codziennych zadań programistycznych, stosując sprawdzone rozwiązania

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 11.

Grafika w aplikacjach Visual C++

Obiekt Graphics — kartka do rysowania

Większości komponentów wizualnych VC++ zawiera właściwość `Graphics`, dzięki której można rysować, wypisywać teksty i umieszczać na nich grafiki w postaci bitmapy. Możesz pomyśleć o właściwości `Graphics` jako o kartce czy płótnie, na którym można tworzyć grafikę.

Na początku po utworzeniu komponentu jego właściwość `Graphics` jest „pusta". Przed rozpoczęciem rysowania trzeba stworzyć obiekt typu `Graphics` i „podpiąć" do tej właściwości. Dopiero na tym obiekcie można rysować. Do tworzenia obiektu `Graphics` służy metoda `CreateGraphics()` wywoływana na komponencie, na którym chcemy rysować.

Oprócz podłoża do rysowania niezbędne są także obiekty klasy `Pen` i `Brush`, a do wyświetlania tekstu także obiekt opisujący czcionkę typu `Font`.

`Pen` — pióro, jakim rysujemy.

`Brush` — (pędzel) rodzaj wypełnienia rysowanych obiektów (kolor, deseń). Mamy dwa rodzaje pędzli: `SolidBrush` to pędzel jednokolorowy, `TextureBrush` zaś wypełnia obiekty deseniem z podanej bitmapy.

`Font` — określa czcionkę do rysowania napisów.

Zasadę rysowania najlepiej wyjaśni prosty przykład.

Przykład 11.1.

Po naciśnięciu przycisku narysuj ukośną niebieską linię na oknie aplikacji.

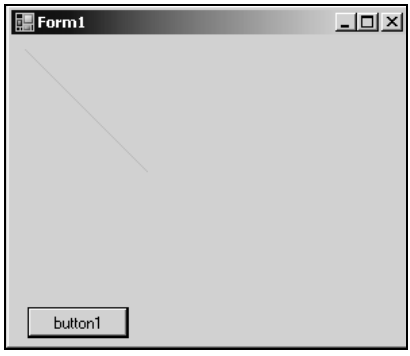
Rozwiązanie

Do nowego projektu aplikacji wstaw przycisk `Button`.

Po naciśnięciu przycisku należy utworzyć obiekt typu `Graphics` dla głównego okna aplikacji, a następnie obiekt pióra `Pen`. Ponieważ metoda `button1_Click()` jest metodą klasy reprezentującej główne okno, odwołujemy się do tego okna za pomocą wskaźnika `this`. Teraz można już rysować po oknie, korzystając z metody obiektu `Graphics` rysującej linie. Oto kod metody, którą należy przypisać do zdarzenia `Click`:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Pen^ pioro = gcnew Pen(System::Drawing::Color::Aqua);
             g1->DrawLine(pioro,10,10,100,100);
         }
```

Po naciśnięciu przycisku na oknie pojawi się linia jak na rysunku 11.1.



Rysunek 11.1. *Rysowanie na oknie aplikacji*

Zauważ, że pióro `Pen` i pędzel `Brush` nie są właściwościami obiektu `Graphics`, ale oddzielnymi obiektami.

Obiekt `Graphics` posiada wiele metod służących do rysowania punktów, linii, figur, a nawet wyświetlania całych bitmap z plików. Zestawienie metod klasy `Graphics` podaje tabela 11.1.

Tabela 11.1. *Wybrane metody rysujące obiektu Graphics*

Metoda	Działanie
`DrawLine(Pen^ pi,int x1,int y1,int x2,int y2)`	Rysuje linie o początku w *x1*, *y1* i końcu w *x2*, *y2*, używając pióra *pi*.
`DrawArc(Pen^ pi,float x,float y,float szer, float wys,float kąt_start,float kąt)`	Wycinek okręgu lub elipsy rysowany piórem *pi*, środek w (*x*, *y*), łuk mieści się w prostokącie o wymiarach *szer* i *wys*, zaczyna się od kąta *kąt_start* i biegnie przez *kąt* stopni.

Tabela 11.1. *Wybrane metody rysujące obiektu Graphics (ciag dalszy)*

Metoda	Działanie
`DrawBezier(Pen^ pi,float x1,float y1,float x2, float y2,float x3,float y3,float x4,float y4)`	Rysuje krzywą sklejaną Beziera przez cztery podane punkty.
`DrawBeziers(Pen^ pi, array<System::Drawing::PointF>^ punkty)`	Rysuje serię krzywych Beziera zgodnie z podaną tablicą punktów.
`DrawClosedCurve(Pen^ pi, array<System::Drawing::PointF>^ punkty)`	Rysuje zamkniętą krzywą sklejaną zgodnie z podaną tablicą punktów.
`DrawCurve(Pen^ pi, array<System::Drawing::PointF>^ punkty)`	Rysuje krzywą sklejaną (spline) między punktami podanymi w tabeli.
`DrawEllipse(Pen^ pi,float x, float y, float szer, float wys)`	Rysuje elipsę mieszczącą się w prostokącie określonym współrzędnymi oraz wysokością i szerokością.
`DrawIcon(Icon^ ikona, int x, int y)`	Rysuje ikonę w podanych współrzędnych.
`DrawImage(Image^ obraz,int x,int y)`	Rysuje bitmapę *obraz* we współrzędnych (*x*, *y*).
`DrawPie(Pen^ pi,float x,float y,float szer, float wys,float kąt_start,float kąt)`	Rysuje wycinek koła piórem *pi*, środek w (*x*, *y*), wycinek mieści się w prostokącie o wymiarach *szer* i *wys*, zaczyna się od kąta *kąt_start* i biegnie przez *kąt* stopni.
`DrawPolygon(Pen^ pi, array<System::Drawing::PointF>^ punkty)`	Tworzy zamknięty wielobok o wierzchołkach określonych w tabeli *punkty*.
`DrawRectangle(Pen^ pi,float x,float y, float szer,float wys)`	Rysuje prostokąt o lewym górnym rogu w punkcie (*x*, *y*) i bokach o długości *szer* i *wys*.
`DrawString(String^ tekst,Font^ czcionka, Brush^ pędzel, float x,float y)`	Wypisuje tekst *tekst* z lewym górnym rogiem w punkcie (*x*, *y*), używając podanej czcionki i pędzla.
`FillClosedCurve(Brush^ pędzel, array<System::Drawing::PointF>^ punkty)`	Tworzy wypełnioną pędzlem *pędzel* zamkniętą krzywą sklejaną (spline) określona punktami *punkty*.
`FillEllipse(Brush^ pędzel, float x, float y, float szer, float wys)`	Rysuje elipsę w prostokącie określonym współrzędnymi (*x*, *y*) oraz wysokością i szerokością wypełniona pędzlem *pędzel*.
`FillPie(Brush^ pędzel, float x,float y, float szer,float wys,float kąt_start,float kąt)`	Maluje wypełniony pędzlem *pędzel* wycinek koła, środek w (*x*, *y*), wycinek mieści się w prostokącie o wymiarach *szer* i *wys*, zaczyna się od kąta *kąt_start* i biegnie przez *kąt* stopni.
`FillPolygon(Brush^ pędzel, array<System::Drawing::PointF>^ punkty)`	Tworzy wypełniony pędzlem *pędzel* wielokąt o wierzchołkach w tabeli *punkty*.
`FillRectangle(Brush^ pędzel, float x,float y, float szer,float wys)`	Rysuje wypełniony pędzlem *pędzel* prostokąt o lewym górnym rogu w punkcie (*x*, *y*) oraz podanej szerokości i wysokości.
`Clear(Color kolor)`	Czyści całą powierzchnię rysunku i wypełnia ją kolorem *kolor*.

Ponieważ metod jest dużo i każda ma kilka postaci, tabela 11.1 podaje tylko po jednej postaci każdej metody, aby możliwe było zestawienie wszystkich.

Przykład 11.2.

Wyświetl w oknie programu tekst podany w polu tekstowym. Nie używaj komponentu `Label`.

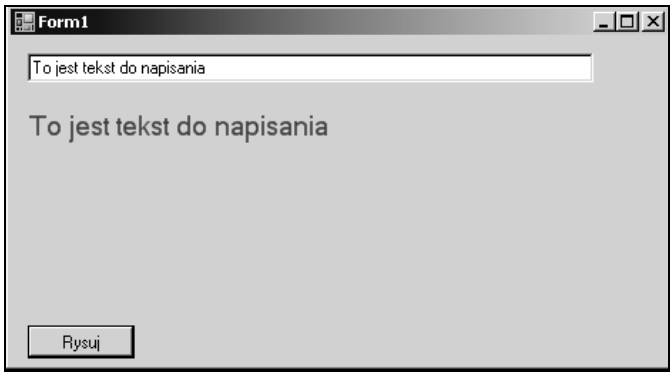
Rozwiązanie

Utwórz nowy projekt aplikacji i wstaw do okna pole tekstowe `TextBox` oraz przycisk `Button`.

Po naciśnięciu przycisku zostanie utworzony pędzel typu `SolidBrush` (jednokolorowy) i wypisany tekst z pola.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             g1->Clear(System::Drawing::Color::FromName("Control"));
             SolidBrush^ pedzel = gcnew SolidBrush(System::Drawing::Color::DarkGreen);
             System::Drawing::Font^ czcionka =
             gcnew System::Drawing::Font(System::Drawing::FontFamily::GenericSansSerif,
               14, FontStyle::Regular);
             g1->DrawString(textBox1->Text,czcionka,pedzel,10,50);
         }
```

Wynik działania aplikacji przedstawia rysunek 11.2.



Rysunek 11.2. *Rysowanie tekstu w oknie aplikacji*

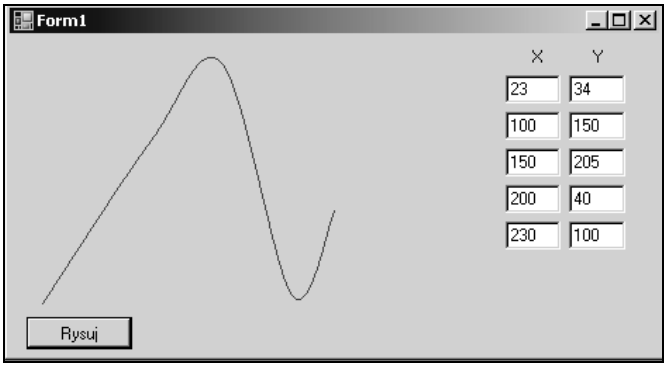
Przykład 11.3.

Po naciśnięciu przycisku wyświetl na formularzu wykres ze współrzędnych podanych w polach tekstowych.

Rozwiązanie

Wstaw do okna aplikacji dziesięć pól tekstowych `TextBox`, dwie etykiety `Label` i przycisk `Button`.

Powiększ wymiary okna, a pod właściwości `Text` etykiet i przycisku podstaw odpowiednie teksty, tak aby całość wyglądała jak na rysunku 11.3 (na razie bez wykresu i wpisanych współrzędnych).



Rysunek 11.3. *Aplikacja do rysowania wykresów*

Po naciśnięciu przycisku *Rysuj* współrzędne będą pobierane z pól tekstowych i przekazywane do tablicy obiektów `System::Drawing::PointF`, które reprezentują punkty na płaszczyźnie. Następnie narysujemy wykres za pomocą metody `DrawCurve()`. Pod zdarzenie `Click` przycisku podepnij podaną metodę:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             g1->Clear(System::Drawing::Color::FromName("Control"));
             array<System::Drawing::PointF>^ punkty = gcnew array
               <System::Drawing::PointF>(5);
             punkty[0].X=Convert::ToSingle(textBox1->Text);
             punkty[0].Y=(this->Height-30)-Convert::ToSingle(textBox2->Text);
             punkty[1].X=Convert::ToSingle(textBox3->Text);
             punkty[1].Y=(this->Height-30)-Convert::ToSingle(textBox4->Text);
             punkty[2].X=Convert::ToSingle(textBox5->Text);
             punkty[2].Y=(this->Height-30)-Convert::ToSingle(textBox6->Text);
             punkty[3].X=Convert::ToSingle(textBox7->Text);
             punkty[3].Y=(this->Height-30)-Convert::ToSingle(textBox8->Text);
             punkty[4].X=Convert::ToSingle(textBox9->Text);
             punkty[4].Y=(this->Height-30)-Convert::ToSingle(textBox10->Text);
             Pen^ pioro1 = gcnew Pen(System::Drawing::Color::DarkGreen);
             g1->DrawCurve(pioro1,punkty);
         }
```

Przykład 11.4.

Po każdym naciśnięciu przycisku wyświetl w oknie prostokąt o losowych współrzędnych, losowych wymiarach i losowym kolorze.

Rozwiązanie

Do nowego projektu aplikacji wstaw przycisk `Button`.

Aby wygenerować współrzędne i kolor prostokątów, będziemy potrzebować generatora liczb losowych. Jest to obiekt typu `Random`, który trzeba utworzyć, a następnie można pobierać z niego liczby, używając metody `Next()` tego obiektu. Po wygenerowaniu liczb prostokąt rysujemy za pomocą metody `FillRectangle()`. Kolor pędzla definiujemy za pomocą trzech liczb losowych określających składowe RGB. Oto metoda zdarzenia `Click` przycisku:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Random^ liczba_los = gcnew Random();
             System::Int32 x=liczba_los->Next(this->Width);
             System::Int32 y=liczba_los->Next(this->Height);
             System::Int32 wys=liczba_los->Next(this->Height);
             System::Int32 szer=liczba_los->Next(this->Width);

             SolidBrush^ pedzel = gcnew SolidBrush(
               System::Drawing::Color::FromArgb(liczba_los->Next(255),
               liczba_los->Next(255), liczba_los->Next(255)));
             g1->FillRectangle(pedzel,x,y,wys,szer);
         }
```

Przykład 11.5.

Wyświetl w oknie plik *rysunek.jpg* z lewym górnym rogiem w punkcie (100, 100).

Rozwiązanie

Wstaw do aplikacji przycisk `Button`.

Aby wyświetlić obrazek, najpierw utworzymy obiekt `Image` zawierający plik *rysunek.jpg* (plik o takiej nazwie trzeba wcześniej umieścić w folderze programu), a następnie wyświetlimy go w oknie, używając metody `DrawImage()`.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Image^ obrazek = Image::FromFile("rysunek.jpg");
             g1->DrawImage(obrazek,100,100);
         }
```

Pióro Pen

Używany już przez nas obiekt `Pen`, czyli pióro, daje wiele możliwości. Za pośrednictwem właściwości tego obiektu można sterować zarówno grubością linii, jak i rysować linie przerywane, wybierać styl rozpoczęcia i zakończenia linii itp. Najważniejsze właściwości przedstawia tabela 11.2.

Przykład 11.6.

Wyświetl w oknie kilka linii rysowanych różnymi stylami.

Rozwiązanie

Utwórz nowy projekt aplikacji i wstaw do niego przycisk `Button`.

Po naciśnięciu przycisku będziemy zmieniać parametry pióra i rysować nim kolejne linie. Oto metoda dla zdarzenia `Click` przycisku:

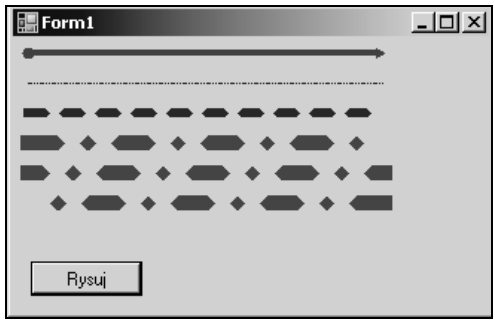
Tabela 11.2. *Niektóre właściwości obiektu Pen*

Właściwość	Znaczenie
`Width`	Szerokość pióra w punktach.
`Brush`	Pędzel, jakim rysowana jest linia.
`Color`	Kolor linii.
`StartCap`	Rodzaj znaczka dodawanego na rozpoczęcie linii. Wartości tej właściwości są typu `System::Drawing::Drawing2D::LineCap`, oto kilka z nich: `System::Drawing::Drawing2D::LineCap::Square` — linia z kwadratowym zakończeniem, `System::Drawing::Drawing2D::LineCap::ArrowAnchor` — linia ze strzałką, `System::Drawing::Drawing2D::LineCap::Triangle` — linia z trójkątem.
`EndCap`	Rodzaj zakończenia linii, wartości takie, jak w `StartCap`.
`DashStyle`	Ustawia linię przerywaną. Wartości to: `System::Drawing::Drawing2D::DashStyle::Dash` — linia z kresek, `System::Drawing::Drawing2D::DashStyle::DashDot` — linia składająca się z kresek i kropek, `System::Drawing::Drawing2D::DashStyle::DashDotDot` — linia z kreski i dwóch kropek, `System::Drawing::Drawing2D::DashStyle::Dot` — linia z kropek, `System::Drawing::Drawing2D::DashStyle::Solid` — linia ciągła.
`DashCap`	Ustawia rodzaj zakończenia elementów linii przerywanej. Wartości to: `System::Drawing::Drawing2D::DashCap::Flat` — zakończenie prostokątne, `System::Drawing::Drawing2D::DashCap::Triangle` — zakończenie trójkątne, `System::Drawing::Drawing2D::DashCap::Round` — zakończenie zaokrąglone.
`DashOffset`	Przesunięcie wzoru kreskowanej linii względem początku linii.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Pen^ pioro1 = gcnew Pen(System::Drawing::Color::DarkGreen);
             pioro1->StartCap = System::Drawing::Drawing2D::LineCap::RoundAnchor;
             pioro1->EndCap = System::Drawing::Drawing2D::LineCap::ArrowAnchor;
             pioro1->Width=4;
             g1->DrawLine(pioro1,10,10,250,10);

             pioro1->StartCap = System::Drawing::Drawing2D::LineCap::Square;
             pioro1->EndCap = System::Drawing::Drawing2D::LineCap::Square;
             pioro1->DashStyle = System::Drawing::Drawing2D::DashStyle::DashDotDot;
             pioro1->DashStyle = System::Drawing::Drawing2D::DashStyle::DashDotDot;
             pioro1->Color=System::Drawing::Color::DarkRed;
             pioro1->Width=1;
             g1->DrawLine(pioro1,10,30,250,30);

             pioro1->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash;
             pioro1->Width=6;
             pioro1->DashCap = System::Drawing::Drawing2D::DashCap::Triangle;
             g1->DrawLine(pioro1,10,50,250,50);
```

```
        pioro1->DashStyle = System::Drawing::Drawing2D::DashStyle::DashDot;
        pioro1->Width=10;
        pioro1->Color=System::Drawing::Color::Blue;
        pioro1->DashCap = System::Drawing::Drawing2D::DashCap::Triangle;
        g1->DrawLine(pioro1,10,70,250,70);

        pioro1->DashOffset = 1;
        g1->DrawLine(pioro1,10,90,250,90);

        pioro1->DashOffset = 2;
        g1->DrawLine(pioro1,10,110,250,110);
    }
```

Efekt po naciśnięciu przycisku przedstawia rysunek 11.4.



Rysunek 11.4. *Działanie różnych rodzajów piór*

Pędzle zwykłe i teksturowane

Pędzle są obiektami dwóch klas potomnych klasy `Brush`, mianowicie `SolidBrush` i `TextureBrush`. Pierwszy z nich to pędzel jednokolorowy, natomiast drugi maluje za pomocą podanych bitmap (tekstur) i daje znacznie większe możliwości.

Kolor pędzla `SolidBrush` ustawiamy, używając właściwości `Color` tego obiektu.

Właściwości pędzla `TextureBrush` przedstawia tabela 11.3.

Przykład 11.7.

Narysuj na oknie aplikacji dwa wycinki kół, jeden wypełniony jednym kolorem i drugi wypełniony teksturą pobraną z pliku *rysunek.jpg* w ułożeniu kafelkowym.

Rozwiązanie

Utwórz nowy projekt aplikacji i wstaw do niego przycisk `Button`.

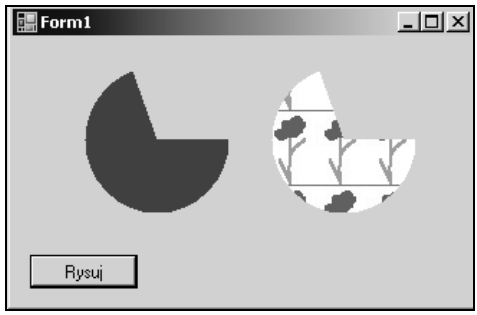
Po naciśnięciu tego przycisku utworzymy dwa pędzle: jednokolorowy i teksturowany, a następnie wyświetlimy wycinki kół za pomocą metody `FillPie()`. Aby program działał, potrzebny jest rysunek *rysunek.jpg* w folderze programu.

Tabela 11.3. *Właściwości pędzla TextureBrush*

Właściwość	Znaczenie
`Image`	Określa bitmapę (teksturę) pędzla.
`WrapMode`	Sposób wyświetlania tekstury: `System::Drawing::Drawing2D::WrapMode::Tile` — tekstura wyświetlana jest sąsiadująco (jak kafelki), `System::Drawing::Drawing2D::WrapMode::Clamp` — tekstura wyświetlana jest jako pojedyncza, `System::Drawing::Drawing2D::WrapMode::TileFlipX` — wyświetlanie sąsiadująco tekstury obróconej względem osi *X*, `System::Drawing::Drawing2D::WrapMode::TileFlipY` — jak wyżej, ale względem osi *Y*, `System::Drawing::Drawing2D::WrapMode::TileFlipXY` — jak wyżej, ale obrócenie względem obu osi.
`Transform`	Określa transformacje, jakim może zostać poddana tekstura przed wyświetleniem. Może to być skalowanie, obracanie lub pochylanie obrazu.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Image^ obrazek = Image::FromFile("rysunek.jpg");
             SolidBrush^ pedz = gcnew SolidBrush(System::Drawing::Color::DarkGreen);
             TextureBrush^ pedz_text = gcnew TextureBrush(obrazek);
             pedz_text->WrapMode=System::Drawing::Drawing2D::WrapMode::Tile;
             g1->FillPie(pedz,50,20,100,100,0,250);
             g1->FillPie(pedz_text,180,20,100,100,0,250);
         }
```

Po uruchomieniu i naciśnięciu przycisku aplikacja może wyglądać tak, jak na rysunku 11.5 (efekt jest zależny od zawartości rysunku użytego do teksturowania).



Rysunek 11.5. *Malowanie pędzlem zwykłym i teksturowanym*

Kafelkowe układanie tekstury jest opcją domyślną, dlatego linię

```
pedz_text->WrapMode=System::Drawing::Drawing2D::WrapMode::Tile;
```

można pominąć i nie zmieni to efektu działania programu.

Zajmiemy się teraz właściwością `Transform` umożliwiającą transformacje tekstury przed jej wyświetleniem. Właściwość ta może przybierać wartości typu `System::Drawing::Drawing2D::Matrix`, które są *macierzami transformacji*. Obiekt `Matrix` jest macierzą 3×3, a każda liczba w macierzy oznacza rodzaj transformacji obiektu. Strukturą macierzy nie będziemy się tu zajmować, ponieważ do podstawowych operacji na tekturach jej znajomość nie jest potrzebna. Operacje na macierzy można wykonywać za pomocą metod klasy `Matrix`. Tabela 11.4 przedstawia niektóre z nich.

Tabela 11.4. *Metody transformujące obiektu Matrix (macierzy transformacji)*

Metoda	Znaczenie
`Rotate(Single kąt)`	Obraca teksturę o *kąt* stopni w prawo.
`RotateAt(Single kąt, PointF punkt)`	Obraca teksturę o *kąt* stopni, przy czym środek obrotu znajduje się w punkcie *punkt*.
`Scale(Single skalaX, Single skalaY)`	Skaluje teksturę; *skalaX* i *skalaY* to skale w kierunkach osi *X* i *Y*. Wartość 1 oznacza oryginalny rozmiar.
`Shear(Single wspX, Single wspY)`	Transformacja polegająca na obrocie płaszczyzny rysunku wokół osi *X* lub *Y*.
`Translate(Single odlX, Single odlY)`	Przesuwa teksturę o *odlX* i *odlY* punktów w kierunku odpowiednich osi.

Przykład 11.8.

Narysuj kwadrat wypełniony teksturą z pliku *rysunek.jpg* w ułożeniu kafelkowym, tekstura ma być obrócona o 20 stopni i przesunięta o 10 punktów w kierunku osi *Y*.

Rozwiązanie

Do nowego projektu aplikacji wstaw przycisk `Button`.

Do zdarzenia `Click` przycisku przypiszemy metodę, w której utworzymy macierz transformacji, w tej macierzy zapiszemy odpowiednie transformacje za pomocą metod `Rotate()` i `Translate()`. Następnie podstawimy tę macierz do właściwości `Transform` pędzla, którym pomalujemy kwadrat.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1=this->CreateGraphics();
             Image^ obrazek = Image::FromFile("rysunek.jpg");
             System::Drawing::Drawing2D::Matrix^ macierz =
               gcnew System::Drawing::Drawing2D::Matrix();
             macierz->Rotate(20);
             macierz->Translate(0,10);
             SolidBrush^ pedz =
               gcnew SolidBrush(System::Drawing::Color::DarkGreen);
             TextureBrush^ pedz_text = gcnew TextureBrush(obrazek);
             pedz_text->WrapMode=System::Drawing::Drawing2D::WrapMode::Tile;
             pedz_text->Transform=macierz;
             g1->FillRectangle(pedz_text,180,20,100,100);
         }
```

Rysowanie pojedynczych punktów — obiekt Bitmap

Jak może zauważyłeś, na obiekcie `Graphics` nie da się wykonywać operacji na pojedynczych punktach obrazu. Aby rysować pojedyncze piksele, należy stworzyć obiekt typu `Bitmap`, na nim rysować piksele za pomocą dostępnej w nim metody `SetPixel()`, a następnie wyświetlić cały obiekt `Bitmap` na obiekcie `Graphics` za pomocą metody `DrawImage()`.

Przykład 11.9.

Narysuj na oknie aplikacji dwieście punktów o losowych współrzędnych i w losowych kolorach.

Rozwiązanie

W oknie aplikacji umieść przycisk `Button`.

Po kliknięciu przycisku utworzymy obiekt `Bitmap`, na którym w pętli `for` zostanie umieszczonych dwieście punktów. Następnie ten obiekt zostanie wyświetlony na obiekcie `Graphics` okna aplikacji, czyli punkty pojawią się na tym oknie. Metoda `DrawImage()` wymaga argumentu typu `Image`, a ponieważ obiekt `bitmapa` jest typu `Bitmap`, trzeba zastosować konwersję typów. Oto odpowiedni kod:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             System::Int32 wspx;
             System::Int32 wspy;
             System::Drawing::Color kolor;
             Graphics^ g1=this->CreateGraphics();
             Random^ liczba_los = gcnew Random();
             Bitmap^ bitmapa = gcnew Bitmap(this->Width,this->Height);
             for (System::Int32 i=0;i<200;i++) {
                wspx=liczba_los->Next(this->Width);
                wspy=liczba_los->Next(this->Height);
                kolor=System::Drawing::Color::FromArgb(liczba_los->Next(255),
                  liczba_los->Next(255), liczba_los->Next(255));
                bitmapa->SetPixel(wspx,wspy,kolor);
             }
             g1->DrawImage(dynamic_cast<Image^>(bitmapa),10,10);
         }
```

Rysowanie trwałe — odświeżanie rysunku

System Windows nie zapamiętuje zawartości okien po ich narysowaniu. Związane jest to z oszczędnością pamięci — dlatego narysowane obiekty znikają na przykład po przykryciu okna rysunku przez inne okno. Kiedy okno jest wyświetlane, powtórnie

jest odświeżane i wszystkie kontrolki są malowane jeszcze raz. Generowane jest wtedy zdarzenie `Paint`. Aby rysunek był w dalszym ciągu widoczny w oknie, należy go również odświeżyć. W praktyce oznacza to, że metoda obsługująca zdarzenie `Paint` powinna rysować rysunek jeszcze raz.

Przykład 11.10.

Stwórz aplikację z dwoma polami wyboru `CheckBox`, niech pierwsze pole włącza i wyłącza rysunek kwadratu w oknie, a drugie — rysunek koła. Rysunek ma być zachowywany po przykryciu okna przez inne okno.

Rozwiązanie

Utwórz aplikację z dwoma polami wyboru. We właściwość `Text` pierwszego pola wyboru wpisz „Kwadrat", a drugiego — „Koło".

Najpierw napisz metodę rysującą koło i kwadrat podanego koloru. Metodę umieść w klasie `Form1`, podobnie jak metody obsługujące zdarzenia.

```
private: System::Void rysuj_kwadrat(System::Drawing::Color kolor) {
             Graphics^ g1=this->CreateGraphics();
             Pen^ pioro = gcnew Pen(kolor);
             g1->DrawRectangle(pioro,10,10,150,150);
             delete g1;
             delete pioro;
         }
private: System::Void rysuj_kolo(System::Drawing::Color kolor) {
             Graphics^ g1=this->CreateGraphics();
             Pen^ pioro = gcnew Pen(kolor);
             g1->DrawEllipse(pioro,20,20,130,130);
             delete g1;
             delete pioro;
         }
```

Teraz czas na obsługę zdarzenia `Paint`. W widoku budowy okna aplikacji (zakładka *Form1.h [Design])* kliknij budowane okno aplikacji, a następnie w prawym panelu przełącz się na widok zdarzeń (ikona błyskawicy). Teraz znajdź zdarzenie `Paint` i kliknij je dwukrotnie. Zostaniesz przeniesiony do kodu aplikacji, gdzie utworzy się metoda `Form1_Paint()`. Metoda ta będzie rysowała figury, gdy będą zaznaczone odpowiednie pola. Uzupełnij ją jak niżej.

```
private: System::Void Form1_Paint(System::Object^  sender,
  System::Windows::Forms::PaintEventArgs^  e) {
             if (checkBox1->Checked)
                rysuj_kwadrat(System::Drawing::Color::DarkBlue);
             if (checkBox2->Checked)
                rysuj_kolo(System::Drawing::Color::DarkBlue);
         }
```

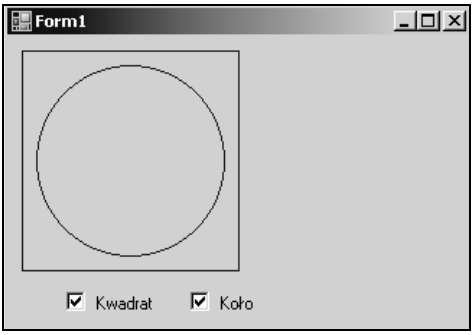
Na końcu zaprogramuj reakcję na zaznaczenie lub wyczyszczenie pól wyboru. Kliknij pole „Kwadrat" dwukrotnie i uzupełnij powstałą metodę, która będzie obsługiwała zmianę stanu zaznaczenia pola pierwszego:

```
private: System::Void checkBox1_CheckedChanged(System::Object^  sender,
  System::EventArgs^  e) {
            if (checkBox1->Checked)
               rysuj_kwadrat(System::Drawing::Color::DarkBlue);
            else
               rysuj_kwadrat(System::Drawing::Color::FromName("Control"));
         }
```

Tak samo dla pola drugiego — „Koło”:

```
private: System::Void checkBox2_CheckedChanged(System::Object^  sender,
  System::EventArgs^  e) {
            if (checkBox2->Checked)
               rysuj_kolo(System::Drawing::Color::DarkBlue);
            else
               rysuj_kolo(System::Drawing::Color::FromName("Control"));
         }
```

Uruchom i wypróbuj działanie programu. Zaznaczenie pól powoduje wyświetlenie figur, które nie znikają po zakryciu innym oknem. Wygląd aplikacji przedstawia rysunek 11.6.



Rysunek 11.6. *Aplikacja z trwałym rysowaniem*

Animacje

Animacje najłatwiej uzyskać, wykorzystując komponent `Timer`. Jednym ze sposobów jest wycieranie rysunku z poprzedniego kroku animacji i rysowanie nowego za każdym razem, kiedy generowane jest zdarzenie `Tick` komponentu `Timer`.

Przykład 11.11.

Napisz program wyświetlający animowany prostokąt w oknie. Prostokąt powinien się poruszać w górę i w dół przez całą wysokość okna.

Rozwiązanie

Utwórz nowy projekt aplikacji i wstaw do niego komponent `Timer` oraz dwa przyciski. Przyciski będą służyły do rozpoczęcia i zatrzymania ruchu prostokąta. We właściwość `Text` pierwszego przycisku wstaw „Start”, a drugiego — „Stop”.

Najpierw musimy utworzyć dwa pola klasy, które będą określały aktualną współrzędną *y* prostokąta i krok jego przesunięcia. Pola te zadeklaruj w klasie `Form1`, tak jak deklarujemy metody:

```
private: System::Int16 krok;
private: System::Int16 y;
```

Na pasku niewidocznych komponentów na dole zakładki projektowania aplikacji zaznacz komponent `Timer` i przejdź do jego zdarzeń w prawym panelu. Kliknij dwukrotnie zdarzenie `Tick`, tworząc metodę uruchamianą czasomierzem. W tej metodzie będą rysowane dwa prostokąty; jeden w kolorze formularza spowoduje wytarcie prostokąta z poprzedniego kroku, następnie będzie zmieniana współrzędna i rysowany nowy przesunięty prostokąt. Metodę zmodyfikuj jak niżej.

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
             Graphics^ g1 = this->CreateGraphics();
             SolidBrush^ pedzel = gcnew SolidBrush(System::Drawing::Color::Aquamarine);
             SolidBrush^ pedz_kas =
               gcnew SolidBrush(System::Drawing::Color::FromName("Control"));
             g1->FillRectangle(pedz_kas,10,y,100,10);
             y=y+krok;
             g1->FillRectangle(pedzel,10,y,100,10);
             if ((y>100)||(y<1))
                krok=-krok;
             delete g1;
         }
```

Zmienną `krok`, która określa przesunięcie prostokąta w jednej klatce animacji, zainicjalizuj w chwili tworzenia okna. Wykorzystamy do tego zdarzenie `FormLoad`.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e) {
                krok=2;
          }
```

Pozostało jeszcze zaprogramowanie metod wywoływanych przy naciśnięciu przycisków. Pierwszy przycisk będzie uruchamiać `Timer`, a drugi zatrzymywać.

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             krok=4;
             timer1->Start();
         }
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
             timer1->Stop();
         }
```

Po uruchomieniu programu i naciśnięciu przycisku *Start* prostokąt będzie się poruszał w górę i w dół okna. Szybkość jego poruszania się zależy od właściwości `Interval` komponentu `Timer`.