Podręcznik programisty WPF 4.5!

Adam Nathan

WPF 4.5

KSIĘGA EKSPERTA

SAMS

Helion

Tytuł oryginału: WPF 4.5 Unleashed

Tłumaczenie: Paweł Gonera (wstęp, rozdz. 1 – 14), Ireneusz Jakóbik (rozdz. 15 – 22, dodatek)

ISBN: 978-83-283-0141-2

Authorized translation from the English language edition, entitled: WPF 4.5 UNLEASHED, ISBN 0672336979; by Adam Nathan; published by Pearson Education, Inc, publishing as SAMS Publishing.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wpf4ke.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wpf4ke
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Część VI Tematy zaawansowane

ROZDZIAŁ 4.

Sterowanie rozmiarem i pozycją elementów oraz transformacje

W tym rozdziale:

- Kontrolowanie rozmiaru
- Kontrolowanie pozycji
- Stosowanie transformacji

Podczas budowania aplikacji WPF jednym z pierwszych zadań do wykonania jest ustawienie zestawu kontrolek na powierzchni aplikacji. Sterowanie wymiarami i pozycją kontrolek (jak również innych elementów) jest tworzeniem **układu**. WPF zawiera rozbudowaną infrastrukturę wspierającą bogaty w możliwości system układu.

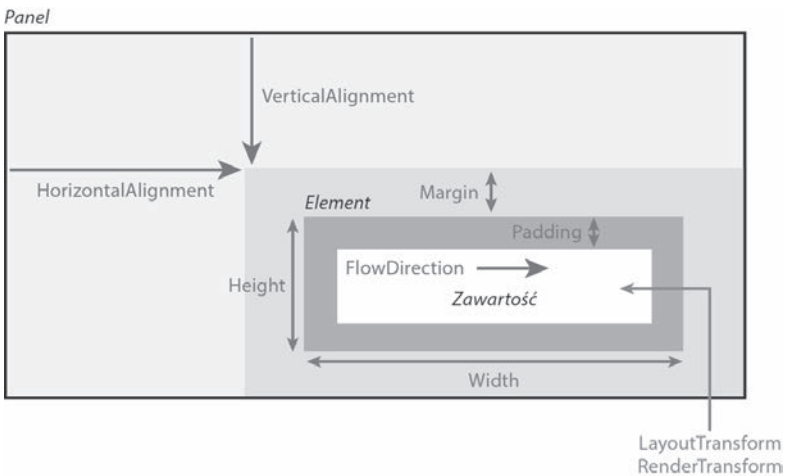
Układy w WPF bazują na interakcjach pomiędzy elementami nadrzędnymi a podrzędnymi. Elementy nadrzędne i ich elementy potomne współpracują ze sobą w celu określenia końcowej wielkości i położenia. Choć to element nadrzędny ostatecznie wskazuje elementowi podrzędnemu miejsce, w którym powinien być narysowany, oraz miejsce, jakie może zająć, to mimo wszystko jest to współpraca, a nie dyktatura. Obiekty nadrzędne *odpytują* podrzędne o ilość miejsca, jakiego potrzebują, i na tej podstawie podejmują końcową decyzję.

Elementy nadrzędne wspierające rozmieszczanie wielu elementów podrzędnych są nazywane **panelami** i dziedziczą po klasie `System.Windows.Controls.Panel`. Wszystkie elementy biorące udział w procesie tworzenia układu (zarówno elementy nadrzędne, jak i podrzędne) dziedziczą po `System.Windows.UIElement`.

Ponieważ układy w WPF są tak obszernym i ważnym tematem, w książce tej poświęcone im są trzy rozdziały:

- Rozdział 4., „Sterowanie rozmiarem i pozycją elementów oraz transformacje”
- Rozdział 5., „Układy z panelami”
- Rozdział 21., „Układy z niestandardowymi panelami”

W tym rozdziale skupię się na elementach podrzędnych, pokazując standardowe sposoby kontrolowania układu pomiędzy elementami podrzędnymi. Aspekt ten jest kontrolowany za pomocą kilku właściwości (których większość jest przedstawiona na rysunku 4.1) dla dowolnego elementu wewnątrz dowolnego panelu. Właściwości związane z wymiarami to Margin, Padding, Height i Width, a związane z położeniem — VerticalAlignment, HorizontalAlignment i FlowDirection. Ponadto elementy mogą mieć dodane do siebie transformacje (LayoutTransform, RenderTransform), które mogą wpływać zarówno na rozmiary, jak i położenie.



Rysunek 4.1. Główne właściwości układu elementów podrzędnych przedstawiane w tym rozdziale

W następnym rozdziale będziemy kontynuować temat, analizując różne panele dostępne w WPF, z których każdy ustawia elementy podrzędne na inny sposób. Tworzenie niestandardowych paneli jest zaawansowanym tematem, przedstawionym w końcowej części książki.

Kontrolowanie rozmiaru

Za każdym razem, gdy następuje wyznaczenie układu (na przykład zmieniony został rozmiar okna), elementy podrzędne przesyłają nadrzędnemu panelowi ich oczekiwane wymiary. Elementy WPF zwykle *dopasowują się do zawartości*, czyli próbują być wystarczająco duże, aby zmieścić całą zawartość, ale nie większe (nawet w `Window` działa w ten sposób, o ile jawnie ustawimy właściwość `SizeToContent`, jak zostało to zrobione w poprzednim rozdziale). Na rozmiary te można wpływać w konkretnych obiektach potomnych za pomocą kilku prostych właściwości.

Właściwości Height i Width

Wszystkie obiekty `FrameworkElement` mają proste właściwości `Height` oraz `Width` (typu `double`), jak również właściwości `MinHeight`, `MaxHeight`, `MinWidth` i `MaxWidth`, które pozwalają określić zakres akceptowalnych wartości. Wszystkie te właściwości można łatwo ustawić w elementach za pomocą kodu proceduralnego lub poprzez XAML.

Elementy w naturalny sposób pozostają możliwie małe, więc jeżeli użyjemy `MinHeight` lub `MinWidth`, będą miały podaną wysokość i szerokość, o ile ich zawartość nie wymusi większych rozmiarów. Dodatkowo wymiary te mogą być ograniczone za pomocą `MaxHeight` oraz `MaxWidth` (o ile wartości te są większe niż ich odpowiedniki zaczynające się od `Min`). Jeżeli jawnie określimy wielkość za pomocą `Height` i `Width` i jednocześnie użyjemy odpowiadających im właściwości `Min` lub `Max`, to wartości `Height` i `Width` będą miały pierwszeństwo, o ile mieszczą się w zakresie od `Min` do `Max`. Domyślną wartością `MinHeight` i `MinWidth` jest `0`, a domyślną wartością `MaxHeight` i `MaxWidth` jest `Double.PositiveInfinity` (która może być ustawiona w XAML za pomocą `"Infinity"`).

Ostrzeżenie

Unikaj jawnego ustawiania wymiarów!

Nadawanie kontrolkom jawnych wymiarów, szczególnie dziedziczących po `ContentControls`, takich jak `Button` i `Label`, niesie ze sobą ryzyko obcinania tekstu, gdy użytkownik zmieni ustawienia czcionki systemowej lub jeżeli tekst zostanie przetłumaczony na inny język. Dlatego warto unikać ustawiania na sztywno wielkości, o ile nie jest to absolutnie niezbędne. Na szczęście jawne ustawianie wielkości jest rzadko potrzebne — dzięki zastosowaniu paneli opisanych w następnym rozdziale.

SCHODZIMY GŁĘBIEJ

Specjalna długość `"Auto"`

Właściwości `Height` i `Width` klasy `FrameworkElement` mają domyślną wartość `Double.NaN` (gdzie `NaN` pochodzi od określenia *nie liczba*), co oznacza, że element będzie tak duży, jak potrzebuje jego zawartość. Ustawienie to może być również określone jawnie w XAML za pomocą `"NaN"` (wielkość liter ma znaczenie) lub preferowanego `"Auto"` (gdzie wielkość liter nie ma znaczenia) dzięki konwerterowi typów `LengthConverter` dołączonemu do tych właściwości. Aby sprawdzić, czy właściwości te są automatycznie wymiarowane, można użyć statycznej metody `Double.IsNaN`.

Aby nie było to tak proste, `FrameworkElement` zawiera kilka innych właściwości związanych z wielkością:

- `DesiredSize` (dziedziczący po `UIElement`),
- `RenderSize` (dziedziczący po `UIElement`),
- `ActualHeight` oraz `ActualWidth`.

W przeciwieństwie do pozostałych sześciu właściwości, które są *wejściem* do procesu określania układu, są to właściwości tylko do odczytu, reprezentujące *wyjście* z procesu tworzenia układu. Wartość `DesiredSize` dla elementu jest obliczana, gdy tworzony jest układ, na bazie wartości innych właściwości (takich jak wspomniane wcześniej właściwości `Width`, `Height`, `Min`*XXX* i `Max`*XXX*) oraz ilości miejsca udostępnianego przez element nadrzędny. Jest ona używana wewnętrznie przez panele.

`RenderSize` reprezentuje końcowy rozmiar elementu po zakończeniu tworzenia układu, a `ActualHeight` i `ActualWidth` są dokładnie takie same jak `RenderSize.Height` i `RenderSize.Width`. To prawda: zależnie od tego, czy w elemencie określone będą jawnie wielkość, zakres akceptowalnych rozmiarów, czy nic nie zostanie zdefiniowane, zachowanie elementu nadrzędnego może zmienić wynikowy rozmiar na ekranie. Te trzy właściwości są więc użyteczne, gdy chcemy w programie korzystać z wielkości elementu. Wartości pozostałych właściwości związanych z rozmiarami nie są interesujące z punktu widzenia logiki. Na przykład, gdy nie ustawimy ich jawnie, wartością `Height` i `Width` jest `Double.NaN`, niezależnie od wielkości elementu.

Wszystkie te właściwości są przedstawione w rozdziale 21.

Ostrzeżenie

Należy zachować uwagę, pisząc kod korzystający z `ActualHeight` i `ActualWidth` (lub `RenderSize`)!

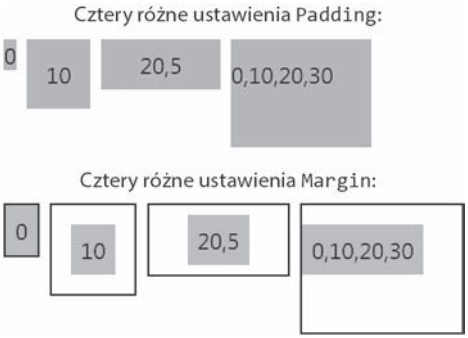
Za każdym razem, gdy zachodzi proces ładowania, aktualizowana jest wartość właściwości `RenderSize` każdego elementu (i jednocześnie `ActualHeight` oraz `ActualWidth`). Jednak tworzenie układu jest wykonywane asynchronicznie, więc nie można zawsze polegać na wartościach tych właściwości. Bezpiecznie można z nich korzystać wyłącznie w handlerze zdarzenia `LayoutUpdated` zdefiniowanego w `UIElement`.

Alternatywnie w elemencie `UIElement` zdefiniowana jest metoda `UpdateLayout`, która pozwala wymusić synchroniczne wykonanie oczekujących aktualizacji, ale należy unikać jej stosowania. Oprócz tego, że częste wywołania `UpdateLayout` mogą ograniczyć wydajność z powodu niepotrzebnego przetwarzania układu, to nie ma gwarancji, że używane elementy prawidłowo obsługują potencjalne ponowne wejście do metod związanych z układem.

Właściwości Margin i Padding

Właściwości `Margin` i `Padding` są dwoma podobnymi właściwościami, które są również związane z wielkością elementu. Wszystkie obiekty dziedziczące po `FrameworkElement` posiadają właściwość `Margin`, a wszystkie dziedziczące po `Controls` (jak również `Border`) mają właściwość `Padding`. Jedyna różnica jest taka, że `Margin` steruje ilością miejsca wokół *zewnętrznych* krawędzi elementu, natomiast `Padding` steruje ilością miejsca wokół jego *wewnętrznych* krawędzi.

Zarówno `Margin`, jak i `Padding` są typu `System.Windows.Thickness`, interesującej klasy, która może reprezentować jedną, dwie lub cztery wartości `double`. Znaczenie tych wartości jest zademonstrowane na listingu 4.1, na którym stosowane są różne ustawienia `Padding` i `Margin` do kontrolki `Label`. Drugi zestaw elementów `Label` ma przypisane wartości `Border`, ponieważ w przeciwnym razie ustawienia marginesów nie będą zauważalne. Na rysunku 4.2 pokazany jest wynik dla każdego elementu `Label`, gdyby był umieszczony na elemencie `Canvas` (panel opisany następnym rozdziale). Choć nie jest to pokazane na rysunku, `Margin` pozwala na wartości ujemne, natomiast `Padding` — nie.



Rysunek 4.2. Efekt użycia Padding i Margin

Listing 4.1. Użycie Padding i Margin z jedną wartością, dwoma lub czterema wartościami

```
<!-- PADDING: -->

<!-- 1 wartość: to samo wypełnienie dla wszystkich czterech stron: -->
<Label Padding="0" Background="Orange">0</Label>
<Label Padding="10" Background="Orange">10</Label>

<!-- 2 wartości: lewa i prawa strona uzyskują pierwszą wartość,
          góra i dół uzyskują drugą wartość: -->
<Label Padding="20,5" Background="Orange">20,5</Label>

<!-- 4 wartości: lewo,góra,prawo,dół: -->
<Label Padding="0,10,20,30" Background="Orange">0,10,20,30</Label>

<!-- MARGIN: -->

<Border BorderBrush="Black" BorderThickness="1">
 <!-- Brak marginesu: -->
  <Label Background="Aqua">0</Label>
</Border>

<Border BorderBrush="Black" BorderThickness="1">
 <!-- 1 wartość: ten sam margines dla wszystkich czterech stron: -->
  <Label Margin="10" Background="Aqua">10</Label>
</Border>

<Border BorderBrush="Black" BorderThickness="1">
 <!-- 2 wartości: lewa i prawa strona uzyskują pierwszą wartość,
          góra i dół uzyskują drugą wartość: -->
  <Label Margin="20,5" Background="Aqua">20,5</Label>
</Border>

<Border BorderBrush="Black" BorderThickness="1">
 <!-- 4 wartości: lewo,góra,prawo,dół: -->
  <Label Margin="0,10,20,30" Background="Aqua">0,10,20,30</Label>
</Border>
```

Element `Label` ma domyślną wartość `Padding` równą 5, ale może być ona zmieniona na dowolną inną. Dlatego właśnie na listingu 4.1 jawnie ustawiamy wartość `Padding` w `Label` na 0. Bez tego jawnego ustawienia wyglądałby on jak piąty element `Label` (demonstrujący niejawną wartość `Margin` równą 0), a wizualne porównanie z pozostałymi wartościami `Padding` byłoby mylące.

SCHODZIMY GŁĘBIEJ

Składnia właściwości `Thickness`

Dostępna dla właściwości `Margin` i `Padding` składnia rozdzielana przecinkami jest udostępniana przez (cóż innego mogłoby to być) konwerter typów. `System.Windows.ThicknessConverter` tworzy obiekt `Thickness` bazujący na wprowadzanym ciągu znaków. Klasa `Thickness` posiada dwa konstruktory, jeden oczekuje pojedynczej wartości `double`, a drugi czterech. Dlatego może on być używany w C# w następujący sposób:

```
myLabel.Margin = new Thickness(10);          // Identycznie jak Margin="10" w XAML
myLabel.Margin = new Thickness(20,5,20,5);   // Identycznie jak Margin="20,5" w XAML
myLabel.Margin = new Thickness(0,10,20,30); // Identycznie jak Margin="0,10,20,30" w XAML
```

Zwróć uwagę, że wygodna składnia z dwoma wartościami jest skrótem dostępnym tylko przez konwerter typów!

FAQ

Z jakich jednostek korzysta WPF?

Konwerter typów `LengthConverter` skojarzony z różnymi właściwościami długości obsługuje jednostki podawane jako `cm`, `pt`, `in` lub `px` (domyślnie).

Domyślnie wszystkie wymiary bezwzględne, takie jak liczby używane we właściwościach związanych z wymiarami, są podawane w *pikselach niezależnych od urządzenia*. Te „piksele logiczne" mają reprezentować 1/96 cala, niezależnie od ustawienia DPI ekranu. Pamiętaj, że piksele niezależne od urządzenia są zawsze podawane jako wartości `double`, więc mogą być ułamkowe.

Dokładne wymiary 1/96 cala nie są istotne, choć wymiar ten został wybrany na podstawie wyświetlacza 96 DPI, w którym piksele niezależne od urządzenia są identyczne z fizycznymi pikselami. Oczywiście zależy to również od wielkości wyświetlacza. Jeżeli aplikacja narysuje linię 1-calową na ekranie laptopa, to linia ta na pewno będzie dłuższa, gdy podłączę to tego laptopa projektor!

To, co *jest* ważne, to fakt, że wymiary te są niezależne od DPI. Jednak sama ta funkcje nie uniemożliwia zmniejszania elementów przy zwiększeniu rozdzielczości ekranu. Aby uzyskać niezależność od rozdzielczości, konieczna jest funkcja automatycznego skalowania, przedstawiona w następnym rozdziale.

Właściwość Visibility

Może wydawać się dziwne wspominanie właściwości `Visibility` (zdefiniowanej w `UIElement`) w kontekście układu, ale faktycznie ma ona na niego wpływ. Właściwość `Visibility` nie jest typu `Boolean`, ale trójstanowego typu wyliczeniowego `System.Windows.Visibility`. Jego wartości mają następujące znaczenie:

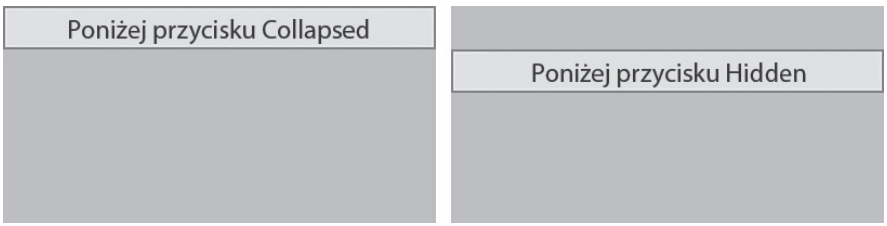
- **`Visible`** — element jest rysowany i wchodzi w skład układu,
- **`Collapsed`** — element jest niewidoczny i nie wchodzi w skład układu,
- **`Hidden`** — element jest niewidoczny, *ale wchodzi w skład układu*.

Element o widoczności `Collapsed` ma rozmiar zero, natomiast `Hidden` zachowuje swoją początkową wielkość (na przykład nie zmieniają się jego wartości `ActualHeight` i `ActualWidth`). Różnica pomiędzy `Collapsed` a `Hidden` jest pokazana na rysunku 4.3, na którym porównujemy następujący `StackPanel` z elementem `Button` o widoczności `Collapsed`:

```
<StackPanel Height="100" Background="Aqua">
  <Button Visibility="Collapsed">Przycisk Collapsed</Button>
  <Button>Poniżej przycisku Collapsed</Button>
</StackPanel>
```

z następującym elementem `StackPanel` zawierającym `Button` o widoczności `Hidden`:

```
<StackPanel Height="100" Background="Aqua">
  <Button Visibility="Hidden">Przycisk Hidden</Button>
  <Button>Poniżej przycisku Hidden</Button>
</StackPanel>
```



Rysunek 4.3. W przeciwieństwie do przycisku Collapsed przycisk Hidden nadal zajmuje miejsce

Kontrolowanie pozycji

W tym podrozdziale nie będziemy mówić o pozycjonowaniu elementów za pomocą współrzędnych (X,Y). Panele nadrzędne definiują własne, unikatowe mechanizmy pozwalające potomkom na samodzielne pozycjonowanie (za pomocą właściwości dołączanych lub po prostu kolejności dodawania do elementu nadrzędnego). Jednak kilka mechanizmów jest wspólnych dla wszystkich potomków `FrameworkElement` i właśnie one będą tematem tego podrozdziału. Mechanizmy te są związane z wyrównywaniem i koncepcją nazywaną **kierunkiem przepływu**.

Wyrównywanie

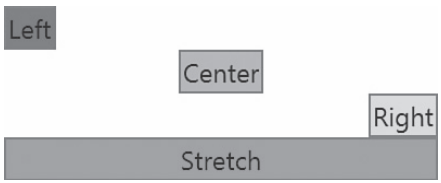
Właściwości `HorizontalAlignment` oraz `VerticalAlignment` pozwalają elementom sterować wykorzystaniem wolnego miejsca w panelu nadrzędnym. Każda właściwość posiada odpowiadający jej typ wyliczeniowy o tej samej nazwie, umieszczony w przestrzeni nazw `System.Windows`, dający nam następujące wartości:

- **`HorizontalAlignment`** — `Left`, `Center`, `Right` oraz `Stretch`,
- **`VerticalAlignment`** — `Top`, `Center`, `Bottom` oraz `Stretch`.

Domyślną wartością dla obu właściwości jest `Stretch`, choć wiele kontrolek zmienia to ustawienie w swoich stylach tematów. Efekt działania `HorizontalAlignment` można zobaczyć, umieszczając kilka elementów `Button` w `StackPanel` i nadając im różne wartości typu wyliczeniowego:

```
<StackPanel>
  <Button HorizontalAlignment="Left" Background="Red">Left</Button>
  <Button HorizontalAlignment="Center" Background="Orange">Center</Button>
  <Button HorizontalAlignment="Right" Background="Yellow">Right</Button>
  <Button HorizontalAlignment="Stretch" Background="Lime">Stretch</Button>
</StackPanel>
```

Wynik jest pokazany na rysunku 4.4.



Rysunek 4.4. Efekt działania HorizontalAlignment dla elementów Button wewnątrz StackPanel

Te dwie właściwości są użyteczne wyłącznie w przypadku, gdy panel nadrzędny daje potomkom więcej miejsca, niż potrzebują. Na przykład dodanie wartości `VerticalAlignment` do elementów wewnątrz `StackPanel` użytego na rysunku 4.4 nie spowoduje różnicy, ponieważ każdy element ma dokładnie taką wysokość, jaka jest potrzebna (nie mniej, nie więcej).

SCHODZIMY GŁĘBIEJ

Interakcja pomiędzy wyrównaniem `Stretch` a jawnym ustawieniem rozmiaru elementu

Gdy element korzysta z wyrównania `Stretch` (poziomo lub pionowo), to jawne ustawienie właściwości `Width` lub `Height` ma wyższy priorytet. Również `MaxWidth` i `MaxHeight` mają wyższy priorytet, ale wyłącznie wtedy, gdy te wartości są mniejsze niż uzyskane na podstawie rozciągnięcia. Podobnie `MinWidth` i `MinHeight` mają wyższy priorytet, ale wyłącznie wtedy, gdy te wartości są *większe* niż uzyskane na podstawie rozciągnięcia. Gdy wartość `Stretch` jest użyta w kontekście ograniczenia wielkości elementu, działa jak wyrównanie `Center` (lub `Left`, jeżeli element jest zbyt duży, aby mógł być wycentrowany w elemencie nadrzędnym).

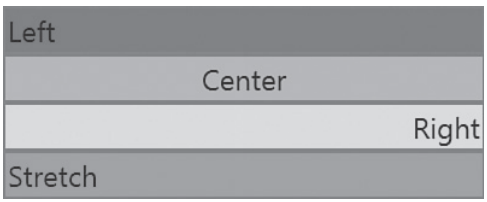
Wyrównanie zawartości

W klasie `Control` oprócz właściwości `HorizontalAlignment` i `VerticalAlignment` zdefiniowane są również właściwości `HorizontalContentAlignment` i `VerticalContentAlignment`. Właściwości te określają, w jaki sposób można sterować wypełnianiem przestrzeni przez zawartość wewnątrz kontrolek (dlatego relacja pomiędzy wyrównaniem i wyrównaniem zawartości jest podobna jak relacja między `Margin` a `Padding`).

Właściwości wyrównania zawartości korzystają z tych samych typów wyliczeniowych co odpowiednie właściwości wyrównania, więc zapewniają te same opcje. Jednak domyślną wartością dla `HorizontalContentAlignment` jest `Left`, a domyślną wartością dla `VerticalContentAlignment` jest `Top`. Ale nie dzieje się tak w przypadku poprzednio pokazanych kontrolek `Button`, ponieważ ich styl tematu nadpisuje te ustawienia (przypomnij sobie kolejność priorytetów dla dostawców wartości właściwości z poprzedniego rozdziału — wartości domyślne mają najniższy priorytet i są przebijane przez style).

Na rysunku 4.5 pokazany jest efekt użycia `HorizontalContentAlignment`. Jest to poprzedni fragment XAML — ma tylko zmienioną nazwę właściwości:

```
<StackPanel>
  <Button HorizontalContentAlignment="Left" Background="Red">Left</Button>
  <Button HorizontalContentAlignment="Center" Background="Orange">Center</Button>
  <Button HorizontalContentAlignment="Right" Background="Yellow">Right</Button>
  <Button HorizontalContentAlignment="Stretch" Background="Lime">Stretch</Button>
</StackPanel>
```



Rysunek 4.5. Efekt działania HorizontalContentAlignment dla elementów Button wewnątrz StackPanel

Na rysunku 4.5 element `Button` z `HorizontalContentAlignment="Stretch"` może wyglądać inaczej, niż się spodziewamy. Wewnętrzny `TextBlock` jest rzeczywiście rozciągnięty, ale `TextBlock` nie jest prawdziwym elementem `Control` (lecz tylko `FrameworkElement`), więc nie ma tej samej notacji dla rozciągania wewnętrznego tekstu.

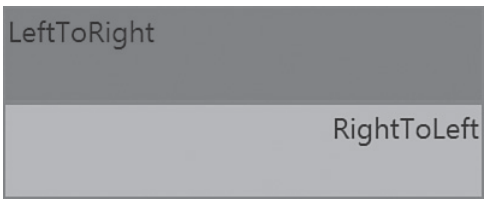
Właściwość FlowDirection

Właściwość `FlowDirection` jest właściwością `FrameworkElement` (i kilku innych klas), która może odwrócić sposób, w jaki przepływa wewnętrzna zawartość elementu. Odnosi się to do niektórych paneli i ich pozycjonowania potomków, jak również do sposobu, w jaki zawartość jest wyrównywana wewnątrz kontrolek potomnych. Właściwość ta jest typu `System.Windows.FlowDirection`, który zawiera dwie wartości: `LeftToRight` (domyślny dla `FrameworkElement`) oraz `RightToLeft`.

W założeniach właściwość `FlowDirection` powinna być ustawiona na `RightToLeft`, gdy bieżące ustawienia regionalne wskazują na język, który jest czytany od prawej do lewej. Powoduje to zamianę znaczenia „w lewo" i „w prawo" w przypadku ustawień takich jak wyrównywanie zawartości. Można to zademonstrować za pomocą poniższego kodu XAML, w którym elementy `Button` wymuszają wyrównanie zawartości na `Top` i `Left`, a następnie stosowane są dwie wartości `FlowDirection`:

```
<StackPanel>
  <Button FlowDirection="LeftToRight"
          HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
          Height="40" Background="Red">LeftToRight</Button>
  <Button FlowDirection="RightToLeft"
          HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
          Height="40" Background="Orange">RightToLeft</Button>
</StackPanel>
```

Wynik jest pokazany na rysunku 4.6.



Rysunek 4.6. Efekt działania FlowDirection dla elementów Button z wyrównaniem zawartości Top i Left

Zwróć uwagę, że `FlowDirection` nie wpływa na kolejność liter w elementach `Button`. Litery angielskie zawsze są ułożone od lewej do prawej, a arabskie zawsze od prawej do lewej. Jednak `FlowDirection` odwraca ten porządek dla pozostałych elementów interfejsu użytkownika, które zwykle muszą pasować do kierunku liter.

Właściwość `FlowDirection` musi być jawnie ustawiona tak, aby pasowała do bieżących ustawień regionalnych (może to być wykonane w głównym elemencie). Powinien to być element procesu lokalizacji.

Stosowanie transformacji

WPF zawiera kilka wbudowanych transformacji 2D (dziedziczących po `System.Windows.Media.Transform`), które pozwalają na zmianę rozmiaru i położenia elementów niezależnie od przedstawionych wcześniej właściwości. Niektóre pozwalają zmieniać elementy w bardziej egzotyczne sposoby, na przykład obracając je lub wykrzywiając.

Wszystkie klasy `FrameworkElement` mają dwie właściwości typu `Transform`, które mogą być używane do stosowania tych transformacji:

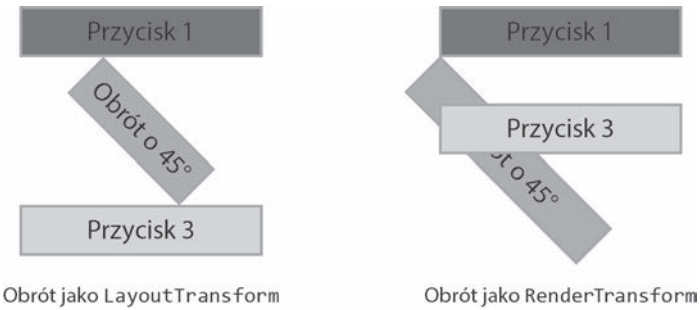
- `LayoutTransform`, która jest stosowana *przed* ułożeniem elementu,
- `RenderTransform` (dziedziczone po `UIElement`), która jest stosowana *po* zakończeniu procesu określania układu (bezpośrednio przed narysowaniem elementu).

FAQ

Jak korzystać z transformacji 3D?

W Silverlight oraz XAML dla aplikacji Windows Store elementy mają właściwość `Projection`, która pozwala na łatwe wykonywanie transformacji perspektywy. Jednak elementy WPF nie mają tej właściwości. Zamiast tego konieczne jest użycie funkcji 3D opisanych w rozdziale 16., „Grafika trójwymiarowa".

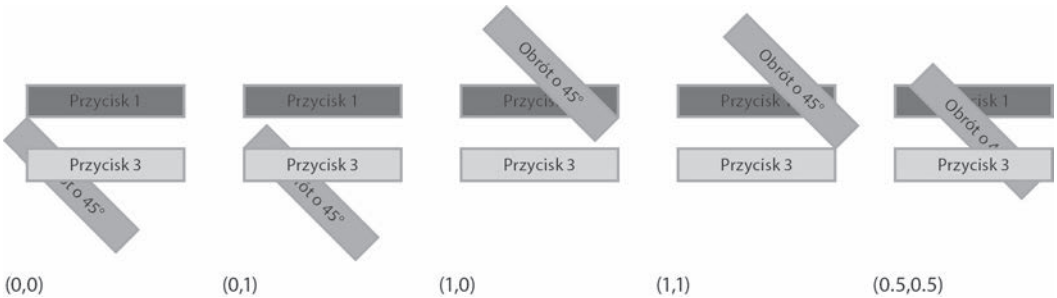
Na rysunku 4.7 zademonstrowana jest różnica pomiędzy stosowaniem transformacji o nazwie `Rotate➥Transform` jako `LayoutTransform` a użyciem transformacji `RenderTransform`. W obu przypadkach transformacja jest stosowana w drugim z trzech kolejnych elementów `Button` ze `StackPanel`. Gdy jest wykorzystywana jako `LayoutTransform`, trzeci element `Button` jest odpychany, a szerokość drugiego elementu `Button` nie jest ograniczana przez szerokość `StackPanel`. Gdy jest ona stosowana jako `RenderTransform`, trzeci element `Button` jest umieszczany tak, jakby drugi element `Button` nie był obracany, a szerokość drugiego elementu `Button` *jest* szerokością `StackPanel`.



Rysunek 4.7. Różnica pomiędzy LayoutTransform a RenderTransform widoczna dla środkowego elementu Button w StackPanel

Klasa `UIElement` posiada również wygodną właściwość `RenderTransformOrigin`, reprezentującą punkt startowy transformacji (punkt, który pozostaje nieruchomy). W przypadku transformacji `RotateTransform` użytej na rysunku 4.7 punktem początkowym dla przycisku jest lewy górny narożnik, wokół którego obracany jest element. Z kolei transformacje `LayoutTransform` nie mają punktu początkowego, ponieważ położenie transformowanego elementu jest określane przez zasady tworzenia układu panelu nadrzędnego.

Właściwość `RenderTransformOrigin` może mieć przypisany obiekt `System.Windows.Point`, którego domyślną wartością jest (0,0). Reprezentuje to lewy górny narożnik, jak jest to pokazane na rysunku 4.7. Punkt początkowy (0,1) reprezentuje lewy dolny narożnik, (1,0) górny prawy, a (1,1) to prawy dolny narożnik. Aby ustawić punkt początkowy poza granicami elementu, należy użyć wartości większej od 1, a aby ustawić go wewnątrz elementu, należy użyć wartości ułamkowych. Dlatego (0.5,0.5) reprezentuje środek obiektu. Na rysunku 4.8 przedstawiono pięć często używanych punktów początkowych używanych dla `RenderTransform` z rysunku 4.7.



Rysunek 4.8. Pięć często używanych wartości RenderTransformOrigin wykorzystanych dla obracanego elementu Button z rysunku 4.7

Dzięki `System.Windows.PointConverter` wartość `RenderTransformOrigin` może być zdefiniowana w XAML za pomocą dwóch liczb oddzielonych przecinkami (bez nawiasów). Na przykład obiekt `Button` obracany wokół swojego środka, tak jak po prawej stronie rysunku 4.8, może być utworzony w następujący sposób:

```
<Button RenderTransformOrigin="0.5,0.5" Background="Orange">
<Button.RenderTransform>
  <RotateTransform Angle="45"/>
</Button.RenderTransform>
  Obrót o 45°
</Button>
```

Można się zastanawiać, dlaczego ktoś chciałby mieć obracający się przycisk w aplikacji! Faktycznie, transformacja taka wygląda głupio w przypadku standardowych kontrolek z domyślnym stylem. Może to mieć większy sens dla aplikacji intensywnie korzystającej z tematów, ale nawet przy domyślnym stylu kontrolek transformacje mogą stanowić przyjemny element wykończeniowy, szczególnie gdy będą używane z animacjami.

W tym podrozdziale przedstawię pięć wbudowanych transformacji 2D, znajdujących się w przestrzeni nazw `System.Windows.Media`:

- `RotateTransform`
- `ScaleTransform`
- `SkewTransform`
- `TranslateTransform`
- `MatrixTransform`

Transformacja RotateTransform

Transformacja `RotateTransform`, zademonstrowana w poprzednim punkcie, pozwala na obracanie elementu zgodnie z wartościami trzech właściwości typu `double`:

- **`Angle`** — kąt obrotu, wyrażany w stopniach (domyślna wartość = 0),
- **`CenterX`** — środek obrotu poziomo (domyślna wartość = 0),
- **`CenterY`** — środek obrotu pionowo (domyślna wartość = 0).

Domyślnie punkt (CenterX,CenterY) o współrzędnych (0,0) reprezentuje lewy górny narożnik. Właściwości CenterX i CenterY są przydatne wyłącznie wtedy, gdy transformacja RotateTransform jest używana jako RenderTransform, ponieważ gdy jest używana jako LayoutTransform, pozycja jest i tak określana przez panel nadrzędny.

FAQ

Jaka jest różnica pomiędzy użyciem właściwości CenterX i CenterY w transformacji takiej jak RotateTransform a użyciem właściwości RenderTransformOrigin z UIElement?

Gdy transformacja jest stosowana do UIElement, właściwości CenterX i CenterY wydają się początkowo nadmiarowe, ponieważ istnieje właściwość RenderTransformOrigin. Oba mechanizmy sterują punktem początkowym transformacji i oba działają tylko w przypadku, gdy transformacja jest używana jako RenderTransform.

Jednak CenterX i CenterY pozwalają na bezwzględne pozycjonowanie punktu początkowego w przeciwieństwie do względnego pozycjonowania w RenderTransformOrigin. Ich wartości są podawane z użyciem pikseli niezależnych od urządzenia, więc prawy górny narożnik elementu o długości 20 może być wskazany przez podanie 20 w CenterX i 0 w CenterY, a nie punktu (1,0). Dodatkowo, gdy łączone jest wiele transformacji RenderTransform (co zostanie opisane w dalszej części rozdziału), właściwości CenterX i CenterY w poszczególnych transformacjach pozwalają na precyzyjne sterowanie procesem. Na koniec pojedyncze wartości double w CenterX i CenterY są łatwiejsze do użycia przy wiązaniu danych niż wartość Point w RenderTransformOrigin.

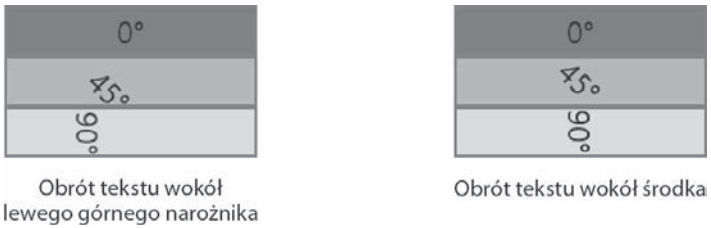
Z tych powodów właściwość RenderTransformOrigin jest zwykle mniej użyteczna niż CenterX i CenterY. W przypadku często wykorzystywanego obrotu elementów wokół ich środka względna wartość (0.5,0.5) właściwości RenderTransformOrigin jest łatwa do wpisania w XAML, natomiast wykonanie tego samego za pomocą CenterX i CenterY wymaga napisania kodu proceduralnego obliczającego bezwzględne przesunięcie.

Zwróć uwagę, że możliwe jest używanie RenderTransformOrigin dla elementu jednocześnie z CenterX oraz CenterY dla transformacji. W tym przypadku dwie wartości X i Y są ze sobą łączone, aby wyliczyć ostateczny punkt początkowy.

Na rysunkach 4.7 i 4.8 pokazane są obrócone elementy Button, natomiast na rysunku 4.9 widać, co się stanie, gdy transformacja RotateTransform będzie użyta jako RenderTransform *na wewnętrznej zawartości* elementu Button przy dwóch różnych wartościach RenderTransformOrigin. Aby to osiągnąć, należy zamienić prosty napis wewnątrz elementu Button na jawnie zdefiniowany element TextBlock:

```
<Button Background="Orange">
  <TextBlock RenderTransformOrigin="0.5,0.5">
  <TextBlock.RenderTransform>
    <RotateTransform Angle="45"/>
  </TextBlock.RenderTransform>
    45°
  </TextBlock>
</Button>
```

Rysunek 4.9.
Użycie RotateTransform na zawartości elementów Button w StackPanel



Element `TextBlock` wewnątrz `Button` po lewej stronie rysunku 4.9 może nie wydawać się obrócony wokół lewego górnego narożnika, ponieważ elementy `TextBlock` są nieco większe niż tekst. Gdy nadamy w elemencie `TextBlock` wartość właściwości `Background`, obrót zacznie mieć sens. Jest to pokazane na rysunku 4.10.

Rysunek 4.10.
Wewnętrzne elementy TextBlock ze zmienionym tłem, obrócone wokół lewego górnego narożnika

Transformacja `RotateTransform` posiada konstruktory z parametrami, które pozwalają na podanie kąta lub kąta oraz środka obrotu, co upraszcza tworzenie transformacji w kodzie proceduralnym.

Transformacja ScaleTransform

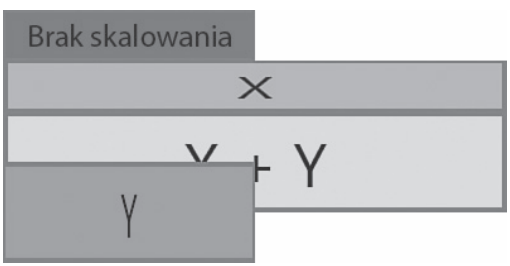
Transformacja `ScaleTransform` pozwala na powiększanie lub zmniejszanie elementu w pionie, poziomie lub w obu kierunkach. Transformacja ta posiada cztery proste właściwości typu `double`:

- **`ScaleX`** — mnożnik dla szerokości elementu (wartość domyślna = 1),
- **`ScaleY`** — mnożnik dla wysokości elementu (wartość domyślna = 1),
- **`CenterX`** — środek dla skalowania w poziomie (domyślna wartość = 0),
- **`CenterY`** — środek dla skalowania w pionie (domyślna wartość = 0).

Wartość `0.5` w `ScaleX` powoduje zmniejszenie elementu o połowę, natomiast wartość `2` w `ScaleX` powiększa jego szerokość. Właściwości `CenterX` oraz `CenterY` działają w ten sam sposób co w przypadku `RotateTransform`.

Na listingu 4.2 stosujemy `ScaleTransform` do trzech elementów `Button` w `StackPanel`, demonstrując możliwość niezależnego skalowania ich wysokości lub szerokości. Wynik jest pokazany na rysunku 4.11.

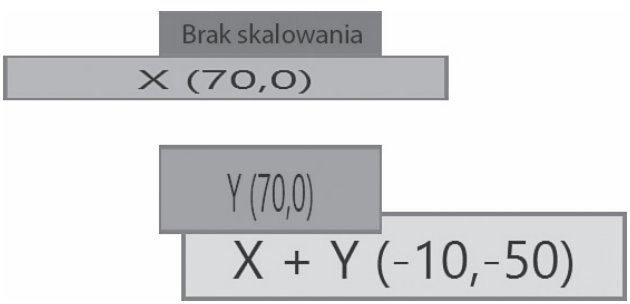
Rysunek 4.11.
Skalowanie elementów Button z listingu 4.2



Listing 4.2. Użycie ScaleTransform do elementów Button w StackPanel

```
<StackPanel Width="100">
  <Button Background="Red">Brak skalowania</Button>
  <Button Background="Orange">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2"/>
  </Button.RenderTransform>
    X</Button>
  <Button Background="Yellow">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2" ScaleY="2"/>
  </Button.RenderTransform>
    X + Y</Button>
  <Button Background="Lime">
  <Button.RenderTransform>
    <ScaleTransform ScaleY="2"/>
  </Button.RenderTransform>
    Y</Button>
</StackPanel>
```

Na rysunku 4.12 przedstawione są te same elementy `Button` z listingu 4.2 (i rysunku 4.11), ale z jawnie ustawionymi wartościami `CenterX` oraz `CenterY`. Punkt reprezentowany przez każdą parę jest wyświetlany w każdym z elementów `Button`. Zwróć uwagę, że zielony element `Button` (trzeci od góry) nie jest przesunięty w prawo tak jak pomarańczowy (drugi od góry), pomimo że użyta jest ta sama wartość `CenterX`, równa 70. Dzieje się tak, ponieważ `CenterX` ma znaczenie tylko wtedy, gdy `ScaleX` ma wartość inną niż 1, a `CenterY` ma znaczenie tylko wtedy, gdy `ScaleY` jest wartością inną niż 1.



Rysunek 4.12. Przyciski z listingu 4.2, ale z jawnie podanymi środkami skalowania

Tak jak w przypadku innych transformacji klasa `ScaleTransform` posiada kilka konstruktorów z parametrami ułatwiającymi tworzenie obiektów w kodzie proceduralnym.

SCHODZIMY GŁĘBIEJ

Interakcja pomiędzy `ScaleTransform` a wyrównaniem `Stretch`

Gdy użyjemy `ScaleTransform` jako `LayoutTransform` na elemencie, który jest już rozciągnięty w wymiarach skalowania, da to efekty tylko wtedy, gdy poziom skalowania będzie większy niż naturalna wielkość rozciągniętego już elementu.

FAQ

Jak transformacje takie jak `ScaleTransform` wpływają na właściwości `ActualHeight` i `ActualWidth` w `FrameworkElement` lub właściwość `RenderSize` z `UIElement`?

Użycie transformacji na `FrameworkElement` nigdy nie zmienia wartości tych właściwości. Dzieje się tak, gdy jest ona użyta jako `RenderTransform` lub `LayoutTransform`. Dlatego z powodu transformacji właściwości te mogą przekłamywać faktyczną wielkość elementów na ekranie. Na przykład wszystkie elementy `Button` na rysunkach 4.11 oraz 4.12 mają te same wartości `ActualHeight`, `ActualWidth` oraz `RenderSize`.

Te „kłamstwa" mogą Cię zaskoczyć, ale są one w dobrej wierze. Dyskusyjne jest, czy takie wartości powinny być prezentowane dla niektórych transformacji. Co ważniejsze, zadaniem transformacji jest zmiana wyglądu elementu bez „wiedzy" samego elementu. Dając elementom „wrażenie", że są rysowane normalnie, pozwala to na dołączenie dowolnej kontrolki i jej transformację bez specjalnego traktowania.

FAQ

W jaki sposób `ScaleTransform` wpływa na `Margin` oraz `Padding`?

Właściwość `Padding` jest skalowana wraz z resztą zawartości (ponieważ `Padding` jest wewnętrznym składnikiem elementu), ale właściwość `Margin` nie jest skalowana. Podobnie jak `ActualHeight` i `ActualWidth`, numeryczna właściwość `Padding` nie zmienia się pomimo wizualnego skalowania.

Transformacja SkewTransform

Dzięki transformacji `SkewTransform` możemy pochylić element w sposób definiowany za pomocą czterech właściwości typu `double`:

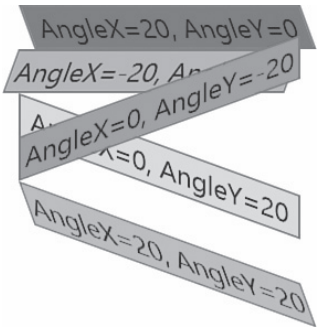
- **`AngleX`** — stopień pochylenia w poziomie (wartość domyślna = 0),
- **`AngleY`** — stopień pochylenia w pionie (wartość domyślna = 0),
- **`CenterX`** — środek dla pochylenia w poziomie (domyślna wartość = 0),
- **`CenterY`** — środek dla pochylenia w pionie (domyślna wartość = 0).

Właściwości te działają bardzo podobnie do poprzednich transformacji. Na rysunku 4.13 przedstawiony jest wynik użycia transformacji `SkewTransform` zastosowanej jako `RenderTransform` na kilku elementach `Button` przy użyciu domyślnego środka pochylania w lewym górnym rogu.

Transformacja TranslateTransform

Dzięki translacji `TranslateTransform` możemy przesunąć element zgodnie z dwoma właściwościami typu `double`:

- **`X`** — wartość przesunięcia w poziomie (wartość domyślna = 0),
- **`Y`** — wartość przesunięcia w pionie (wartość domyślna = 0).



Rysunek 4.13. Zastosowanie SkewTransform do elementów Button w StackPanel

Translacja `TranslateTransform` nie działa, gdy zostanie użyta jako `LayoutTransform`, ale zastosowanie jej jako `RenderTransform` pozwala na „szturchnięcie" elementu w wybraną stronę. Najbardziej prawdopodobne jest, że wykonane to będzie w sposób dynamiczny, w zależności od akcji użytkownika (lub animacji). Jeśli chodzi o wszystkie panele opisane w następnym rozdziale, raczej nie będzie potrzebna transformacja `TranslateTransform` do utworzenia statycznego interfejsu użytkownika.

Transformacja MatrixTransform

Translacja `MatrixTransform` jest mechanizmem niskiego poziomu, który może być używany do tworzenia własnych transformacji 2D. Posiada ona jedną właściwość `Matrix` (typu `System.Windows.Media.Matrix`) reprezentującą macierz transformacji afinicznej o rozmiarach 3×3. Jeżeli nie jesteś miłośnikiem algebry liniowej, powinieneś wiedzieć, że wszystkie poprzednie transformacje (lub dowolna ich kombinacja) mogą być wyrażone za pomocą `MatrixTransform`.

Macierz 3×3 zawiera następujące wartości:

$$\begin{bmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{bmatrix}$$

Wartości ostatniej kolumny są stałe, ale pozostałe sześć wartości może być ustawiane we właściwości typu `Matrix` (o pokazanych nazwach) lub poprzez konstruktor z sześcioma parametrami odpowiadającymi wartościom z macierzy w kolejności wierszy.

SCHODZIMY GŁĘBIEJ

Konwerter typów dla MatrixTransform

Transformacja `MatrixTransform` jest jedyną transformacją, która ma zdefiniowany konwerter typów, co pozwala nam użyć go jako prostego ciągu znaków w XAML (ten konwerter typów nosi nazwę `TransformConverter` i jest skojarzony z abstrakcyjną klasą `Transform`, ale obsługuje wyłącznie `MatrixTransform`). Na przykład można przesunąć element `Button` o 10 jednostek w prawo i 20 jednostek w dół za pomocą następującej składni:

```
<Button RenderTransform="1,0,0,1,10,20" />
```

ciąg dalszy

Lista ta reprezentuje wartości `M11`, `M12`, `M21`, `M22`, `OffsetX` oraz `OffsetY`. Wartości `1`, `0`, `0`, `1`, `0`, `0` tworzą macierz jednostkową (oznaczającą brak transformacji), więc aby transformacja `MatrixTransform` działała tak jak `TranslateTransform`, wystarczy użyć macierzy jednostkowej, a następnie dodać wartości `OffsetX` i `OffsetY` jako wartości `X` i `Y` w `TranslateTransform`. Skalowanie może być wykonane przez użycie pierwszej i czwartej wartości (`1` w macierzy jednostkowej) jako `ScaleX` i `ScaleY`. Obroty i pochylanie są bardziej skomplikowane, ponieważ wymagają użycia sinusów, cosinusów i kątów wyrażonych w radianach.

Jeżeli potrafisz korzystać z notacji macierzowej, to ten spójny (choć mniej czytelny) sposób reprezentowania transformacji pozwoli Ci zaoszczędzić dużo czasu, gdy wpisujesz XAML ręcznie.

Łączenie transformacji

Łączenie wielu transformacji, na przykład obracanie elementu w czasie jednoczesnego skalowania, może być wykonane na kilka różnych sposobów. Można jednocześnie użyć `LayoutTransform` oraz `RenderTransform`. Można również użyć reprezentacji `MatrixTransform` do uzyskania połączonego efektu. Najlepiej jednak skorzystać z możliwości klasy `TransgormGroup`.

`TransformGroup` jest kolejną klasą dziedziczącą po `Transform` (więc może być użyta wszędzie tam, gdzie stosowane są transformacje), a jej zadaniem jest łączenie podrzędnych obiektów `Transform`. W kodzie proceduralnym można dodać transformacje do kolekcji `Children`, natomiast zapis w XAML wygląda następująco:

```
<Button>
<Button.RenderTransform>
  <TransformGroup>
    <RotateTransform Angle="45"/>
    <ScaleTransform ScaleX="5" ScaleY="1"/>
    <SkewTransform AngleX="30"/>
  </TransformGroup>
</Button.RenderTransform>
  OK
</Button>
```

Na rysunku 4.14 pokazany jest wynik tych trzech transformacji zastosowanych dla elementu `Button`.

Rysunek 4.14. Element Button, który został obrócony, przeskalowany i pochylony

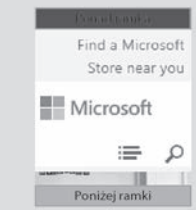
Dla zapewnienia maksymalnej wydajności WPF wylicza połączoną transformację na podstawie podrzędnych elementów z `TransformGroup`, a następnie stosuje tę wynikową (tak jak w przypadku samodzielnego użycia `MatrixTransform`). Zwróć uwagę, że do `TransformGroup` można dodać wiele instancji tej samej transformacji. Na przykład użycie dwóch osobnych instancji `RotateTransform` o kącie 45° daje obrót o 90°.

Ostrzeżenie

Nie wszystkie obiekty `FrameworkElement` obsługują transformacje!

Elementy zawierające treść, która nie pochodzi z WPF, nie obsługują transformacji, mimo że posiadają właściwości `LayoutTransform` i `RenderTransform`. Na przykład element `HwndHost` używany do osadzania zawartości GDI i opisany w rozdziale 19., „Współpraca z technologiami innymi niż WPF", nie obsługuje ich. Element `Frame`, kontrolka pozwalająca na osadzenie HTML (opisana w rozdziale 9., „Kontrolki zawartości"), obsługuje je, o ile nie zawiera kodu HTML. W przeciwnym razie `ScaleTransform` może być używana do zmiany wielkości, ale wewnętrzna zawartość nie podlega skalowaniu.

Na rysunku 4.15 przedstawiony jest `StackPanel` zawierający kilka elementów `Button` oraz element `Frame` zawierający stronę WWW (ograniczony do wielkości 100×100). Gdy cały `StackPanel` jest obracany i skalowany, element `Frame` próbuje się skalować, ale się nie obraca. W efekcie tego ukrywa większość obróconych przycisków.



Normalny `StackPanel`

`StackPanel` z `RotateTransform` oraz `ScaleTransform`

Rysunek 4.15. Element Frame z zawartością HTML odpowiada na ScaleTransform, ale ignoruje pozostałe transformacje

Podsumowanie

Na tym kończymy przegląd właściwości układu, z których elementy potomne mogą korzystać w celu zmiany wyglądu na ekranie. W tym rozdziale zapoznaliśmy się również z pierwszymi elementami interfejsu użytkownika, które nie występują w Win32 ani Windows Forms: obróconymi i pochylonymi kontrolkami!

Jednak najważniejszym elementem układów są nadrzędne panele. W tym rozdziale dla uproszczenia korzystaliśmy wyłącznie ze `StackPanel`, ale w następnym wprowadzimy obok niego również pozostałe panele.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Doskonały przewodnik dla programistów WPF!

Windows Presentation Foundation (WPF) to nowoczesna technologia, pozwalająca tworzyć zaawansowane aplikacje dla systemu Windows. Dzięki jej możliwościom sprawnie i bezproblemowo poradzisz sobie z każdym zadaniem — niezależnie od tego, czy chcesz stworzyć tradycyjną aplikację albo niesamowity interfejs 3D, czy połączyć animacje i multimedia. Co więcej, WPF sprawdza się świetnie zarówno na tradycyjnych komputerach, jak i na urządzeniach mobilnych oraz dużych ekranach telewizorów. Brzmi zachęcająco? Przekonaj się sam!

Jeśli sięgniesz po tę książkę, zdobędziesz kompletne źródło informacji na temat Windows Presentation Foundation. Na samym początku zapoznasz się z historią WPF, a następnie przejdziesz do odkrywania tajników XAML oraz podstaw tworzenia oprogramowania. Kolejne rozdziały zawierają bezcenną wiedzę na temat układów okien, zdarzeń, struktury aplikacji, wykorzystania pulpitu systemu Windows oraz używania kontrolek. Ponadto dowiesz się stąd, jak wiązać dane, korzystać ze stylów, skórek, z szablonów i tematów oraz pracować z materiałami multimedialnymi. A potem zdobędziesz zaawansowaną wiedzę na temat grafiki 3D, animacji oraz współpracy z technologiami innymi niż WPF. Książka ta jest wspaniałym kompendium wiedzy na temat WPF w wersji 4.5.

Dzięki tej książce:

- poznasz platformę WPF w wersji 4.5
- wykorzystasz jej potencjał przy tworzeniu aplikacji dla systemu Windows
- przekonasz się, jak pracować z materiałami multimedialnymi
- zobaczysz pola, na których możliwa jest współpraca z innymi technologiami
- opanujesz potencjał WPF

Adam Nathan— guru WPF, główny architekt oprogramowania w Startup Business Group firmy Microsoft. Wcześniej główny programista i architekt w serwisie Popfly (pierwszy projekt firmy Microsoft oparty na Silverlight). Jego książki zdobyły ogromną popularność i są powszechnie uznawane za lekturę obowiązkową.

Helion

35064 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0141-2

9 788328 301412

cena: 129,00 zł