helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C#. Tworzenie aplikacji graficznych w .NET 3.0

Autor: Krzysztof Rychlicki-Kicior
ISBN: 978-83-246-1076-1
Format: B5, stron: 304

Poznaj techniki budowania interfejsów użytkownika dla aplikacji Windows

- Opanuj język C# i wykorzystaj możliwości programowania obiektowego
- Zaprojektuj interfejs użytkownika, wykorzystując język XAML
- Zaimplementuj mechanizmy obsługi plików i wymiany danych

Najnowsza wersja platformy .NET, oznaczona numerem 3.0, zawiera wiele usprawnień, dzięki którym tworzenie aplikacji z interfejsem graficznym stało się zdecydowanie prostsze. Część platformy o nazwie Windows Presentation Foundation (WPF) wraz z przeznaczonym wyłącznie do projektowania interfejsów użytkownika językiem XAML pozwala na całkowite oddzielenie warstwy prezentacji aplikacji od warstwy logiki i danych. Wykorzystując podstawowy język programowania platformy .NET – C# – można niemal błyskawicznie stworzyć aplikację z graficznym interfejsem użytkownika.

Książka „C#. Tworzenie aplikacji graficznych w .NET 3.0" opisuje ten właśnie język w kontekście pisania programów wyposażonych w interfejs graficzny zbudowany w oparciu o WPF. Czytając ją, dowiesz się, jak zainstalować i skonfigurować narzędzia do pracy. Poznasz język C#, zasady programowania obiektowego i najważniejsze klasy platformy .NET. Nauczysz się konstruować dokumenty XAML i tworzyć aplikacje WPF. Przeczytasz o komponentach wizualnych, zdarzeniach i programowaniu operacji graficznych. Znajdziesz tu również informacje o obsłudze plików, połączeniach z bazami danych oraz komunikacji sieciowej.

- Pobieranie i instalacja narzędzi
- Podstawowe elementy języka C#
- Programowanie obiektowe
- Konstruowanie dokumentów XAML
- Hierarchia klas komponentów wizualnych
- Obsługa zdarzeń
- Geometria 2D
- Operacje graficzne
- Tworzenie animacji
- Korzystanie z szablonów
- Obsługa plików i danych
- Przetwarzanie dokumentów XML
- Połączenia z siecią

Poznaj najnowszą wersję narzędzia, które zrewolucjonizowało proces tworzenia oprogramowania dla systemu Windows

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

Spis treści

Rozdział 14.

Animacje

W rozdziale tym dowiesz się, jak dokonać animacji właściwości. Do tego celu będzie Ci potrzebna również wiedza na temat triggerów. Mechanizm animacji daje ogromne możliwości, jednak jak w przypadku wszystkich udogodnień, należy pamiętać, że wymaga on szybkiego komputera!

Triggery

Trigger jest mechanizmem, który pozwala na wykonanie pewnej ściśle określonej czynności, gdy jest spełniony dokładnie określony warunek. Wyróżniamy kilka rodzajów triggerów. Najpierw zapoznajmy się z pierwszym z nich — triggerem właściwości.

Trigger właściwości jest reprezentowany przez klasę `System.Windows.Trigger`. Najczęściej jest wykorzystywany i tworzony w kodzie XAML, podobnie jak pozostałe triggery. Trigger służy do zmiany właściwości (za pomocą obiektu `Setter`) w sytuacji, gdy inna właściwość ma ściśle określoną wartość. Zapoznajmy się z pierwszym przykładem. Pole tekstowe przybierze zielony kolor, gdy użytkownik wprowadzi tajne hasło. Uściślając — gdy w polu tekstowym znajdzie się pewna wartość, nastąpi zmiana właściwości `Background`. XAML pozwala na wyrażenie tego w poniższy sposób:

```
<Window.Resources>
 <Style x:Key="triggery" TargetType="{x:Type TextBox}">
  <Style.Triggers>
   <Trigger Property="Text" Value="tajnehaslo">
    <Setter Property="Background" Value="Lime"/>
   </Trigger>
  </Style.Triggers>
 </Style>
</Window.Resources>
<StackPanel>
 <TextBox Text="Tu wpisz hasło" Style="{StaticResource triggery}" />
</StackPanel>
```

Kluczowa część kodu jest zadeklarowana w stylu znajdującym się w zasobach okna. Wynika to z faktu, że triggery właściwości mogą być tworzone **jedynie** w deklaracjach stylów. Wszystkie triggery muszą znaleźć się w kolekcji `Style.Triggers`. Jak widać, każdy `Trigger` składa się z trzech kluczowych elementów:

- właściwości, która ma być obserwowana,
- wartości, jaką ma przyjąć ta właściwość w celu wykonania triggera,
- grupy obiektów klasy `Setter`, które dokonują żądanych zmian.

Mechanizm triggerów automatycznie przywraca poprzednie wartości właściwości, które uległy zmianie w wyniku działania obiektów klasy `Setter`. Po wprowadzeniu tajnego hasła, gdy kolor zmieni się z białego na zielony, usunięcie choćby jednej litery spowoduje powrót do białego tła.

Oczywiście, podobny efekt można uzyskać za pomocą tradycyjnych zdarzeń. Pomijając jednak fakt, że nie wszystkie właściwości mają odpowiednie zdarzenia informujące o ich zmianie, kod potrzebny do osiągnięcia tego samego efektu jest po prostu dłuższy i mniej zwarty (rozdzielony na kod XAML i C#):

```
<StackPanel>
 <TextBox Name="txtBox" Text="Tu wpisz tekst" TextChanged="zmiana" />
</StackPanel>
void zmiana(object o, RoutedEventArgs rea)
{
 if (txtBox.Text == "tajnehaslo")
  txtBox.Background = Brushes.Lime;
 else
  txtBox.Background = Brushes.White;
}
```

Pomijając fakt, że kod jest podzielony na dwie części, można łatwo sobie wyobrazić, co by było, gdybyśmy chcieli dokonać zmiany większej liczby właściwości! Do triggera należałoby dodać jedynie jeden obiekt `Setter`, zaś w drugim przypadku znacznego rozszerzenia wymagałby kod metody.

Idąc tropem triggerów właściwości, dojdziemy do innego interesującego mechanizmu — triggerów zdarzeń. Sens ich działania jest identyczny; inny jest jedynie element źródłowy, powodujący wywołanie obiektów `Setter`; jest nim zdarzenie (a nie zmiana właściwości, choć w gruncie rzeczy zmiana właściwości też jest zdarzeniem).

Można zastanawiać się, po co właściwie wprowadzać taką konstrukcję, skoro stanowi ona ewidentny przykład dublowania funkcjonalności zdarzeń? Powód jest jeden, ale bardzo istotny. Są nim animacje — a konkretnie animowane właściwości, które są omawiane w następnym podrozdziale. Na razie zajmiemy się zatem schematem samego zdarzenia:

```
<Style x:Key="triggery" TargetType="{x:Type TextBox}">
 <Style.Triggers>
  <Trigger Property="Text" Value="tajnehaslo">
   <Setter Property="Background" Value="Lime"/>
  </Trigger>
```

```
  <EventTrigger RoutedEvent="PreviewMouseDown">
   <!-- tu znajdzie się treść animacji -->
  </EventTrigger>
 </Style.Triggers>
</Style>
```

Najważniejszym atrybutem obiektu `EventTrigger` jest `RoutedEvent` — określa on zdarzenie, które wywoła trigger. Co ważne, triggery zdarzeń mogą być umieszczane nie tylko w deklaracji stylu, ale także w kolekcji `Triggers`, którą posiada każda kontrolka:

```
<TextBox>
 <TextBox.Triggers>
  <EventTrigger RoutedEvent="PreviewMouseDown">
   <!-- tu znajdzie się treść animacji -->
  </EventTrigger>
 </TextBox.Triggers>
</TextBox>
```

Ostatnim typem zdarzeń (choć nie ostatnim w ogóle), który chciałbym omówić w tym momencie, jest `MultiTrigger`. Jest to trigger właściwości, przy czym aby zostały wykonane założenia zapisane w obiektach `Setter`, muszą zajść wszystkie warunki określone w obiekcie `MultiTrigger`, a nie tylko jeden.

Warunki pojedynczego triggera były zapisywane jako atrybuty. Pozostawienie takiej formy dla większej liczby warunków mijało się z celem, dlatego utworzono specjalną właściwość klasy `MultiTrigger` — `Conditions` (przechowującą obiekty klasy `Condition`). Zobaczmy, jak wygląda to w praktyce, rozszerzając nasz przykład. W celu uzyskania efektu zielonego tła dodamy konieczność wskazania pola tekstowego kursorem — innymi słowy, właściwość `IsMouseOver` musi mieć wartość `true`:

```
<Window.Resources>
 <Style x:Key="triggery" TargetType="{x:Type TextBox}">
  <Style.Triggers>
   <MultiTrigger>
    <MultiTrigger.Conditions>
     <Condition Property="IsMouseOver" Value="True"/>
     <Condition Property="Text" Value="tajnehaslo"/>
    </MultiTrigger.Conditions>
    <Setter Property="Background" Value="Lime"/>
   </MultiTrigger>
  </Style.Triggers>
 </Style>
</Window.Resources>
<StackPanel>
 <TextBox Name="txtBox" Text="Tu wpisz tekst" Style="{StaticResource triggery}"/>
</StackPanel>
```

Obiekty klasy `Condition` przyjmują identyczne atrybuty jak w przypadku zwykłego obiektu `Trigger`. Obiekt klasy `Setter` ustawiamy poza blokiem warunków. W deklaracji pola tekstowego nie musimy nic zmieniać — wszelkie zmiany zachodzą bowiem jedynie w obrębie stylu.

Animacje

Na wstępie pozwolę sobie od razu zaznaczyć — tytułowe animacje nie mają ścisłego związku z animacjami wykonywanymi w programie Flash, tudzież z animowanymi plikami w formacie GIF lub jakimikolwiek innymi formatami animacji, chociaż można odnaleźć wiele elementów wspólnych między nimi. Istotą tego zagadnienia w WPF jest animowanie różnych obiektów, a nie animacja sama w sobie. Dokładnie jest to animacja właściwości, czyli płynna zmiana wartości właściwości w określonym czasie. Wykonanie tej czynności powoduje jednak często efekty w postaci faktycznych animacji — czyli np. ruchu niektórych elementów znajdujących się w oknie.

Istnieje kilka rodzajów animacji. Najprostszym z nich jest animacja wartości liczbowych. Istotą takiej animacji jest określenie wartości początkowej i końcowej. Nie można zapomnieć też o ustawieniu czasu, w jakim dochodzi do wykonania animacji. Od niego zależy, czy animacja będzie przebiegała płynnie, czy nie. Nie wszystkie typy danych obsługują animację (choć podstawowe, takie jak liczby czy łańcuchy znaków, nie mają z tym problemu), natomiast istnieje możliwość stworzenia własnego sposobu animacji. Najpierw zajmiemy się jednak budową najprostszej animacji i sposobami jej wywołania.

Najważniejszą częścią każdej animacji jest obiekt zawierający w nazwie słowo `Animation`. Oprócz tego na początku nazwy znajduje się typ animacji, co w rezultacie daje efekt w postaci nazwy np. `DoubleAnimation`. Zajmijmy się więc tą najprostszą z animacji, operującą na liczbach niecałkowitych.

Przed chwilą wymieniłem najistotniejsze elementy takiej animacji; czas pokazać, w jaki sposób są one reprezentowane w kodzie XAML. Właściwości określające początkową i końcową wartość to `From` i `To` (od i do). Ze względu na typ danych muszą to być wartości typu `Double`. Trzecią kluczową właściwością jest `Duration` — okres trwania animacji. Jest on wyrażany za pomocą znacznika czasu (nie mam jednak na myśli znacznika w rozumieniu języka XML). Ma on postać `h:m:s`, gdzie `h` to liczba godzin, `m` — liczba minut, a `s` — liczba sekund (wszystkie trzy wartości mogą być niecałkowite). W kodzie C# tego rodzaju konstrukcja jest reprezentowana przez klasę `TimeSpan` (przedział czasowy). Aby zilustrować opisane mechanizmy, utwórzmy prosty przykład — przycisk powiększający swoją szerokość trzykrotnie. Zaczniemy od obiektu animacji:

```
<DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty="Width"
From="50" To="150" Duration="0:0:5"/>
```

Stosujemy wartość `50` dla właściwości `From`, aby rozpocząć animację od tej samej wartości, jaka jest zadeklarowana dla przycisku domyślnie (w kodzie XAML). Wyznaczenie czasu na 5 sekund i szerokości na 150 pikseli powoduje, że co sekundę przyciskowi przybędzie 20 pikseli z prawej strony. Za pomocą właściwości dołączanych klasy `Storyboard` określamy obiekt docelowy i animowaną właściwość. Zanim jednak powiem, w jakich sytuacjach należy określać pierwszą z tych właściwości, należy zauważyć, że skoro korzystamy z właściwości dołączanych klasy `Storyboard`, to obiekt animacji jest umieszczony właśnie w znaczniku `Storyboard`. Tak rzeczywiście się dzieje; za pomocą właściwości tej klasy określamy właściwość i obiekt, z którymi jest związana animacja.

Obiekt `Storyboard` nie jest ostatnim znacznikiem, który poznajemy w tym miejscu. Do powiązania tego obiektu z triggerem zdarzenia należy wykorzystać obiekt `BeginStoryboard`. Całość animacji wygląda zatem następująco:

```
<Button Name="przycisk" Width="50" Height="70">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
   <BeginStoryboard>
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk"
Storyboard.TargetProperty="Width" From="50" To="150" Duration="0:0:5"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
 </Button.Triggers>
</Button>
<!--Tuż po kliknięciu przycisku rozpocznie się animacja, która po 5 sekundach doprowadzi
do poszerzenia przycisku na szerokość całego okna (przy założeniu, że okno ma 150 pikseli szerokości).-->
```

Wszystkie animacje udostępniają inne ciekawe właściwości, które pozwalają na bardziej precyzyjną kontrolę nad nimi:

`public RepeatBehavior RepeatBehavior` — określa sposób powtarzania animacji. Należy podać liczbę powtórzeń (z sufiksem `x`, np. `2x`), aby uzyskać *n*-krotność powtórzeń, wartość `Forever`, aby animacja była powtarzana nieskończenie wiele razy, lub wartość typu `TimeSpan` (czyli przedział czasowy), aby animacja była powtarzana przez określony czas (przy ustawieniu tej wartości na `0:0:6` i czasie trwania animacji wynoszącym 3 sekundy, animacja zostanie wykonana dwa razy). W pierwszym przypadku można podać wartość niecałkowitą (np. `0.3x`), aby animacja była wykonana tylko w części.

`public bool AutoReverse` — określa, czy animacja (przy wielokrotnym wykonywaniu) ma być odtwarzana od początku (wartość domyślna — `False`), czy ma cofać się od stanu końcowego do początkowego (`True` — w naszym przykładzie przy tej właściwości ustawionej na `False` szerokość przycisku po osiągnięciu wartości 150 pikseli zostanie ponownie ustawiona na 50 pikseli, natomiast przy właściwości równej `True` szerokość będzie stopniowo zmniejszana od 150 do 50 pikseli). Liczba wykonań określona we właściwości `RepeatBehavior` dotyczy całkowitych wykonań animacji — wykonanie animacji i wycofanie do stanu początkowego jest traktowane jako **jedno** wykonanie animacji.

`public TimeSpan BeginTime` — określa opóźnienie, z jakim rozpocznie się wykonywanie animacji. Domyślnie jest to wartość `0:0:0`, czyli brak opóźnienia. Można też ustawić wartość `null` (w kodzie XAML `{x:Null}`) — wtedy animacja nie zostanie wykonana.

`public FillBehavior FillBehavior` — określa sposób zachowania animowanego obiektu po zakończeniu animacji. Dla wartości `Stop` obiekt powraca do stanu sprzed animacji, natomiast dla wartości `HoldEnd` (domyślnej) zatrzymywany jest stan z końca animacji.

Z ostatnią właściwością jest związany jeszcze jeden istotny fakt. Gdy po zakończeniu animacji będziemy próbowali zmienić wartość animowanej właściwości, zwyczajna zmiana następującej animacji:

```
<DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty="Width"
From="50" To="150" Duration="0:0:5"/>
```

(np. w postaci przypisania) nie przyniesie rezultatu:

```
void zmien(object o, RoutedEventArgs rea)
{
 przycisk.Width = 200.0;
} // kod metody przypisany do innego przycisku
```

Kliknięcie tego innego przycisku, a zatem wywołanie metody, nie ma żadnego wpływu na wartość właściwości `Width`. Wynika to z faktu, że domyślne ustawienie właściwości `FillBehavior` przechowuje wartość animacji nawet po zakończeniu jej działania. Aby rozwiązać ten problem, należy ustawić właściwość `FillBehavior` na `Stop`. Wtedy animacja nie będzie miała wpływu na właściwość po zakończeniu działania i zmiana się powiedzie. Co jednak zrobić, jeśli zależy nam, aby animacja była typu `HoldEnd`, a z drugiej strony musimy zmienić wartość tej właściwości? Wystarczy, że po jej zakończeniu (najlepiej przed pożądaną zmianą wartości) usuniemy animację z obiektu `Storyboard`. Wystarczy nadać temu obiektowi nazwę:

```
<Storyboard Name="story">
```

i wywołać następującą metodę:

```
void zmien(object o, RoutedEventArgs rea)
{
 story.Remove(przycisk);
 przycisk.Width = 250.0;
}
```

Ostatnią interesującą właściwością jest `SpeedRatio`, czyli współczynnik, za pomocą którego możemy przyspieszyć lub spowolnić wykonywanie animacji. Domyślnie współczynnik ten ma wartość `1`, wartość mniejsza od 1 oznacza spowolnienie, a większa — przyspieszenie.

Po zapoznaniu się z pierwszym przykładem możemy przejść do omówienia różnych typów animacji; każdy z nich zostanie poparty typowym przykładem.

Animacje podstawowe (From / To / By)

Pierwszy przykład, z którym zapoznaliśmy się przed chwilą, jest najprostszym, podstawowym rodzajem animacji. Nazwa pochodzi od trzech właściwości, które odgrywają kluczową rolę w działaniu tej animacji (właściwość `By` określa całkowitą zmianę wartości właściwości, jaka zachodzi od początku do końca animacji — w naszym przykładzie miałaby ona wartość `100`). WPF udostępnia 16 rodzajów animacji (do podanych nazw należy dodać słowo `Animation`):

- `Byte`, `Int16`, `Int32`, `Int64`, `Single`, `Double`, `Decimal` — dokonuje zmiany zwykłej wartości liczbowej (całkowitej lub niecałkowitej).
- `Color` — wykonuje płynne przejście od jednego do drugiego koloru (w kodzie XAML można je określić jedynie za pomocą nazw — aplikacja sama znajdzie odpowiedni sposób przejścia).
- `Point` — wykonuje płynne przejście od jednego do drugiego punktu.
- `Quaternion` — wykonuje płynne przejście kwaternionu (obiektu reprezentującego obrót względem pewnego punktu w przestrzeni trójwymiarowej) między podanymi wartościami.
- `Rect` — dokonuje zmiany położenia i rozmiaru prostokąta.
- `Rotation3D` — animuje oś obrotu (w przestrzeni trójwymiarowej).
- `Size`, `Thickness` — animują rozmiar i grubość obramowań.
- `Vector3D` — animuje wartość i kierunek wektora trójwymiarowego.
- `Vector` — animuje wartość i kierunek wektora normalnego (dwuwymiarowego).

Przy tego rodzaju animacjach do określenia sposobu animacji można wykorzystywać różnego rodzaju kombinacje właściwości `From`, `To` i `By` (nigdy nie wykorzystujemy wszystkich trzech właściwości naraz, gdyż nie ma to sensu). W przykładzie zastosowaliśmy połączenie właściwości `From` i `To`; pozostałe możliwe warianty są uwzględnione na poniższej liście:

- tylko właściwość `From` — animacja rozpoczyna się od wartości określonej we właściwości `From` aż do wartości, którą właściwość posiada domyślnie zadeklarowaną,
- tylko właściwość `To` — animacja rozpoczyna się od wartości domyślnie zadeklarowanej do wartości określonej we właściwości `To`,
- właściwości `From` i `By` — animacja rozpoczyna się od wartości określonej we właściwości `From`, a kończy wraz z wartością będącą sumą właściwości `From` i `By` — dla `From=100` i `By=300` animacja zakończy się wraz z wartością `400`,
- właściwości `To` i `By` nie tworzą kombinacji, według której jest tworzona animacja — właściwość `By` jest ignorowana.

Animacje z użyciem klatek kluczowych

Powyższy sposób animowania jest niezwykle przyjazny dla programisty — nie trzeba się troszczyć o sposób przechodzenia od jednej wartości do drugiej. Z drugiej strony nie mamy możliwości, aby dokładnie kontrolować proces przebiegu animacji — np. zmieniać pośrednie stany, w jakich znajduje się obiekt w trakcie animacji. Dzięki animacji z użyciem klatek kluczowych możemy, właśnie za ich pomocą, definiować poszczególne stany obiektu w ściśle określonych momentach czasowych, dzięki czemu zachowanie obiektów może być nieco bardziej złożone.

Animacje klatkowe obsługują wszystkie typy obsługiwane przez animacje podstawowe, dodatkowo wprowadzając cztery nowe typy (w ich przypadku nazwa klasy składa się z nazwy typu i słów `AnimationUsingKeyFrames`):

- `Boolean` — animuje klatki pomiędzy dwoma wartościami logicznymi (z uwagi na istnienie tylko dwóch wartości logicznych zmiany między klatkami są nagłe — nie ma płynnych przejść).
- `Matrix` — animuje transformację macierzy, która odpowiada za przekształcenie dwuwymiarowej przestrzeni.
- `Object` — animuje klatki pomiędzy wartościami różnych typów; z tego względu zmiany są dokonywane nagle (bez płynnego przejścia, podobnie jak w przypadku typu `Boolean`).
- `String` — animuje klatki zawierające łańcuchy znaków.

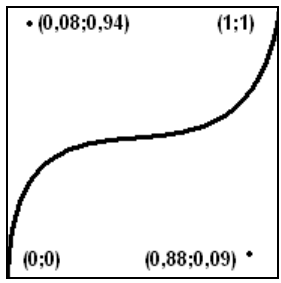
Najistotniejsze znaczenie z nowo wprowadzonych powyższych typów ma `String` — bardzo często istnieje konieczność wykorzystania w animacji zwykłych tekstów, a ten rodzaj animacji to udostępnia. Zanim zajmiemy się przykładami, warto zapoznać się z inną ciekawą cechą tego rodzaju animacji — możliwością wyboru typu animacji klatkowej.

Typy animacji klatkowych

Istnieją trzy typy animacji klatkowych, które można stosować do różnych rodzajów animacji:

- Dyskretny — zmiany między klatkami następują w sposób nagły, nie występuje płynne przejście. Można go stosować dla wszystkich typów animacji, jednak efekty działania tych animacji zazwyczaj są niezadowalające.
- Linearny — zmiany następują w sposób płynny; animacja wygląda bardziej realnie niż w przypadku typu dyskretnego. Typ linearny można stosować jedynie w przypadku typów podstawowych, tj. z pominięciem typów wprowadzonych do animacji klatek kluczowych (`Boolean`, `Matrix`, `Object` i `String`).
- Krzywoliniowy — zmiany następują w sposób złożony, z wykorzystaniem krzywej Béziera (dokładny opis znajduje się poniżej). Można go stosować w tych samych przypadkach co typ linearny.

Najbardziej interesujący wydaje się oczywiście typ krzywoliniowy. Szybkość zmian w takiej animacji jest wykonywana na podstawie krzywej Béziera, którą definiuje programista. Krzywą Béziera definiuje się za pomocą tzw. punktów kontrolnych (poza punktem początkowym i końcowym, jak w przypadku każdego odcinka). W przypadku tej animacji punkty początkowy i końcowy są zdefiniowane odgórnie — lewy dolny (0,0) i prawy górny (1,1). Zadaniem programisty jest zdefiniowanie punktów (w odniesieniu do dwóch istniejących w każdym przypadku), aby na ich podstawie została utworzona krzywa Béziera, która będzie miała wpływ na szybkość odtwarzanej animacji. Przykład takiej krzywej, wraz z zaznaczonymi punktami znajduje się na rysunku 14.1.



Rysunek 14.1. *Krzywa Béziera (ze stałymi punktami (0;0) i (1;1))*

W momencie, gdy krzywa zbliża się do linii poziomej, szybkość animacji maleje; gdy krzywa zaczyna przypominać linię pionową, szybkość wzrasta. Krzywą deklaruje się dla każdej klatki animacji; początek i koniec krzywej oznaczają początek i koniec odtwarzania klatki.

W niniejszym podrozdziale rozpatrzymy animację klatkową na przykładzie klasy `StringAnimationUsingKeyFrames`. Napis *WPF najlepsze jest* będzie pojawiał się w następujący sposób: najpierw pojedynczo będą wyświetlane litery *WPF*, a następnie pojawią się kolejne wyrazy napisu.

Każda animacja klatkowa składa się z wielu elementów, a nie tak jak w przypadku animacji podstawowej — tylko z jednego znacznika. Zawiera ona szereg obiektów klasy o nazwie zgodnej ze schematem `<typ_animacji><typ_klatki>KeyFrame`. W naszym przykładzie wykorzystamy animację dyskretną i związaną z tekstem, więc nazwa klasy zastosowanych klatek to `DiscreteStringKeyFrame`.

Każda z klas-klatek zawiera dwie kluczowe właściwości — `KeyTime`, określającą czas, w którym animowana właściwość ma przyjąć stosowną wartość, i `Value` — definiującą tę wartość. Dodatkowo w animacji krzywoliniowej należy zdefiniować punkty kontrolne krzywej Béziera, co można zrobić za pomocą właściwości `KeySpline`. Przechowuje ona kolekcję punktów, które w języku XAML deklaruje się w następujący sposób:

```
KeySpline="x1,y1 x2,y2"
```

Należy też pamiętać, że układ współrzędnych stosowany przy deklarowaniu punktów dla tej właściwości jest identyczny z kartezjańskim, różny zaś od znanego i stosowanego przy określaniu współrzędnych na ekranie ((0;0) stanowi lewy dolny róg prostokąta, w którym możemy określać współrzędne, a nie lewy górny).

Przykład wykorzystuje przycisk znany z poprzedniego fragmentu kodu; animacja będzie związana z właściwością `Content` przycisku, tak więc będzie modyfikowała tekst na nim wyświetlany:

```
<Button Name="przycisk" Width="200" Height="70">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
   <BeginStoryboard>
    <Storyboard x:Name="story">
     <StringAnimationUsingKeyFrames Storyboard.TargetName="przycisk" Storyboard.
     TargetProperty="Content" Duration ="0:0:5">
      <DiscreteStringKeyFrame KeyTime="0:0:0.5" Value="W"/>
```

```
        <DiscreteStringKeyFrame KeyTime="0:0:1" Value="WP"/>
        <DiscreteStringKeyFrame KeyTime="0:0:1.5" Value="WPF"/>
        <DiscreteStringKeyFrame KeyTime="0:0:2.5" Value="WPF najlepsze"/>
        <DiscreteStringKeyFrame KeyTime="0:0:3.5" Value="WPF najlepsze jest!"/>
       </StringAnimationUsingKeyFrames>
      </Storyboard>
     </BeginStoryboard>
    </EventTrigger>
   </Button.Triggers>
  </Button>
```

Pierwsze trzy litery pojawiają się w tempie dwa razy szybszym od wyrazów, które są wyświetlane później. Warto również przetestować powyższy kod z uwzględnieniem właściwości `FillBehavior`, `RepeatBehavior` i `AutoReverse`. Odstępy czasowe pozostają bez zmian (czyli w przypadku odwrócenia animacji dwa pierwsze słowa będą znikać w odstępie sekundy, a litery w tempie dwa razy szybszym). Zachowanie tego typu animacji jest podobne do linearnej, gdyż można powiedzieć, że animacja linearna jest animacją dyskretną o niezmiernie dużej liczbie klatek, generowanych automatycznie na bieżąco przez aplikację — stąd podobieństwo w działaniu obu animacji.

Przykład animacji krzywoliniowej zaprezentuję, wykorzystując ponownie właściwość `Width`, ponieważ działa ona tylko na właściwościach typów, które obowiązują w animacji podstawowej. Do tej animacji wykorzystamy krzywą Béziera, zaprezentowaną na rysunku 14.1. Kod animacji wygląda następująco:

```
<Button Name="przycisk" Width="50" Height="70">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
   <BeginStoryboard>
    <Storyboard x:Name="story">
     <DoubleAnimationUsingKeyFrames Storyboard.TargetName="przycisk" Storyboard.
     TargetProperty="Width" Duration="0:0:5">
      <SplineDoubleKeyFrame KeyTime="0:0:5" Value="250" KeySpline="0.08,0.94
      0.88,0.09"/>
     </DoubleAnimationUsingKeyFrames>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Szerokość jest animowana od wartości 50 do 250, a więc zwiększa się pięciokrotnie. Na początku zmiana szerokości jest bardzo szybka, blisko środka animacji zwalnia, po czym znowu przyspiesza. Naszą uwagę powinna skupić deklaracja właściwości `KeySpline`. Dwa punkty, pochodzące z rysunku 14.1, zostały zadeklarowane w określonej kolejności — i nie można jej zmieniać, gdyż spowodowałoby to efekt odwrotny do zamierzonego — tempo animacji byłoby wolne na początku, zaś szybkie w jej środku (nastąpiłoby przekształcenie krzywej Béziera).

Istnieje możliwość łączenia animacji różnych typów. Poniższy przykład zawiera klatki wszystkich trzech typów animacji (pozostały kod jest taki sam jak w poprzednim przykładzie, więc nie umieszczam go ponownie):

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetName="przycisk" Storyboard.
TargetProperty="Width" Duration="0:0:10">
 <LinearDoubleKeyFrame KeyTime="0:0:2" Value="100"/>
 <SplineDoubleKeyFrame KeyTime="0:0:6" Value="250" KeySpline="0.08,0.94 0.88,0.09"/>
 <DiscreteDoubleKeyFrame KeyTime="0:0:10" Value="300"/>
</DoubleAnimationUsingKeyFrames>
```

Przez pierwsze dwie sekundy animacja w stałym tempie podwaja swoją wartość. Następnie w ciągu czterech sekund zachodzą kolejno procesy przyspieszania, spowolniania i ponownie przyspieszania — jak w poprzednim przykładzie, tylko tym razem na nieco mniejszej szerokości. Następnie animacja zatrzymuje swe działanie i po czterech sekundach przycisk nagle zmienia swoją szerokość, jednocześnie wypełniając całe okno (które ma również 300 pikseli).

Jeśli nie znamy dokładnych wartości czasu, w jakich chcemy wprowadzać nowe klatki, a znamy jedynie proporcje czasowe między nimi (np. odstęp między pierwszą a drugą klatką jest dwa razy mniejszy niż między drugą a trzecią), warto podawać wartość właściwości `KeyTime` w procentach. Wtedy odstęp między klatkami jest proporcjonalny, a aby zmienić faktyczne odstępy czasowe, wystarczy manipulować właściwością `Duration` obiektu animacji. Aby uzyskać taki sam efekt jak w powyższym przykładzie, stosując wartości procentowe, należałoby wykonać małe obliczenia:

```
(2s/10s) * 100% = 20%
(6s/10s) * 100% = 60%
(10s/10s) * 100% = 100%
```

Istnieją jeszcze dwie specjalne wartości, jakie możemy wykorzystywać przy podawaniu wartości właściwości `KeyTime`. Jeśli chcemy, aby odstęp pomiędzy wszystkimi klatkami był taki sam, musimy nadać im wartość `Uniform`:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetName="przycisk" Storyboard.
TargetProperty="Width" Duration="0:0:10">
 <LinearDoubleKeyFrame KeyTime="Uniform" Value="100"/>
 <SplineDoubleKeyFrame KeyTime="Uniform" Value="250" KeySpline="0.08,0.94 0.88,0.09"/>
 <DiscreteDoubleKeyFrame KeyTime="Uniform" Value="300"/>
</DoubleAnimationUsingKeyFrames>
```

Na każdą klatkę przypada 10/3, czyli ok. 3,3 sekundy. Możliwe jest jednak łączenie wartości `Uniform` z tradycyjnymi wartościami `KeyTime`. Zanalizujmy następującą sekwencję klatek:

```
<LinearDoubleKeyFrame KeyTime="Uniform" Value="100"/>
<SplineDoubleKeyFrame KeyTime="Uniform" Value="250" KeySpline="0.08,0.94 0.88,0.09"/>
<DiscreteDoubleKeyFrame KeyTime="10%" Value="300"/>
<DiscreteDoubleKeyFrame KeyTime="50%" Value="275"/>
```

Jej działanie jest następujące — obydwa płynne przejścia następują bardzo szybko (każde z nich zajmuje `(100-90)/2)%=5%`, tak więc na każde przypada po 0,5 sekundy), gdyż pierwsza z klatek o zadeklarowanym czasie wywołania jest ustawiona bardzo wcześnie (już na początek drugiej sekundy). Następnie, po kolejnych czterech sekundach, szerokość jest minimalnie zmniejszana.

Kolejną specjalną wartością dla właściwości KeyTime jest Paced. Stosowanie jej sprawia, że animacja przebiega w jednym, stałym tempie, które jest wypadkową prędkości odtwarzania animacji wszystkich klatek. Stosowanie tej wartości ma sens tylko w przypadku klatek linearnych. Należy pamiętać, że jeśli pierwsza klatka ze wszystkich będzie miała właściwość KeyTime ustawioną na Paced, zostanie ona odtworzona od razu (efekt taki sam jak w przypadku ustawienia właściwości KeyTime=0:0:0).

Animacje z użyciem ścieżki

Animacje tego typu do animowania wykorzystują ścieżki, czyli krzywe określone przez szereg wyznaczonych przez programistę punktów. Tak jak w animacjach From, To, By pierwsze skrzypce grają właśnie te właściwości, a w animacjach klatkowych — klatki kluczowe, tak w tym przypadku najważniejszą właściwością animacji jest PathGeometry, definiująca zestaw punktów mających wpływ na wykonanie animacji. WPF udostępnia trzy rodzaje animacji ścieżkowych (nazwa składa się z typu i słów AnimationUsingPath):

- Double — animuje właściwość na podstawie jednej ze współrzędnych punktu,
- Point — animuje właściwość na podstawie obydwu współrzędnych (całego punktu),
- Matrix — animuje właściwość z wykorzystaniem transformacji macierzowej (MatrixTransform).

Skupimy się na dwóch pierwszych animacjach, gdyż są one najpopularniejsze. Najbardziej oczywistym rodzajem animacji wydaje się punktowa — przecież poszczególne fragmenty ścieżki są definiowane za pomocą punktów. W trakcie animacji między kolejnymi punktami ścieżki aktualny punkt jest pobierany i przypisywany do animowanej właściwości. Dzieje się tak np. w poniższym przykładzie:

```
<Button Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Background>
  <RadialGradientBrush x:Name="pedzel" Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
   <GradientStop Color="White" Offset="0.0"/>
   <GradientStop Color="Red" Offset="2.0"/>
  </RadialGradientBrush>
 </Button.Background>
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
   <BeginStoryboard>
    <Storyboard>
     <PointAnimationUsingPath Storyboard.TargetName="pedzel" Storyboard.TargetProperty=
     "Center" Duration="0:0:5">
      <PointAnimationUsingPath.PathGeometry>
 <PathGeometry>
  <PathFigure StartPoint="0.5,0.5">
   <LineSegment Point="0.1,0.3"/>
   <LineSegment Point="0.7,0.9"/>
  </PathFigure>
```

```
    </PathGeometry>
        </PointAnimationUsingPath.PathGeometry>
      </PointAnimationUsingPath>
     </Storyboard>
    </BeginStoryboard>
   </EventTrigger>
  </Button.Triggers>
 </Button>
```

W powyższym kodzie główne znaczenie mają dwie sekcje — pierwsza, odpowiedzialna za deklarację pędzla promienistego, i druga, związana z animacją ścieżkową. Deklaracja pędzla zawiera dwa istotne elementy — definicję właściwości `Center` (którą wykorzystujemy w animacji) oraz nazwy obiektu. Proszę zauważyć, że klasa `RadialGradientBrush` nie udostępnia właściwości `Name`, zazwyczaj stosowanej w tym celu. Musimy więc wykorzystać konstrukcję języka XAML, która da nam ten sam efekt. Jest to potrzebne, gdyż w animacji musimy podać nazwę obiektu, którego właściwość chcemy animować. Najistotniejszym obiektem jest oczywiście figura (w istocie trójkąt), po której bokach będzie animowany środek pędzla. Można oczywiście zastosować segment krzywej Béziera, dzięki czemu sposób poruszania się pędzla będzie bardziej urozmaicony.

Nieco bardziej skomplikowana jest animacja typu `Double`. Ogólne zasady działania tej animacji są takie same jak w przypadku animacji punktowej, jednak należy rozważyć jedną różnicę — animacja `Double` wykorzystuje jedną wartość, podczas gdy punkt składa się z dwóch współrzędnych. Koniecznym staje się więc określenie, która z właściwości punktu pobieranego z animacji ma być przypisywana do animowanej właściwości.

Służy do tego celu właściwość `Source`, zdefiniowana w klasie animacji `DoubleAnimationUsingPath`. Przyjmuje ona jedną z trzech wartości typu wyliczeniowego, które umożliwiają pobieranie różnych wartości z animowanego punktu:

- `X` — pobiera współrzędną X (wartość domyślna),
- `Y` — pobiera współrzędną Y,
- `Angle` — określa tangens kąta obrotu.

Pozostała treść animacji się nie zmienia, tak więc nie będzie widać zbyt dużych różnic między poprzednim a poniższym kodem:

```
<DoubleAnimationUsingPath Storyboard.TargetName="pedzel" Source="X" Storyboard.
TargetProperty="RadiusX" Duration="0:0:5">
 <DoubleAnimationUsingPath.PathGeometry>
  <PathGeometry>
   <PathFigure StartPoint="0.5,0.5">
    <LineSegment Point="0.1,0.0"/>
    <LineSegment Point="1.0,0.3"/>
   </PathFigure>
  </PathGeometry>
 </DoubleAnimationUsingPath.PathGeometry>
</DoubleAnimationUsingPath>
```

Najistotniejszą różnicą jest wykorzystanie właściwości `Source`, a także zmiana animowanej właściwości — `Center` jest typu `Point`, wobec czego zmieniliśmy ją na właściwość bardziej stosowną do sytuacji.

Wiele animacji w jednej

Istnieje możliwość umieszczenia dwóch lub większej liczby animacji w jednej. Wystarczy umieścić więcej niż jeden obiekt animacji w obiekcie `Storyboard`:

```
<Storyboard>
 <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
 "Width" From="100" To="150" Duration="0:0:10"/>
 <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
 "Height" From="100" To="150" Duration="0:0:10"/>
</Storyboard>
```

W ten sposób nasz przycisk wolno poszerza w jednakowym tempie wysokość i szerokość w tym samym czasie. Jeśli chcemy opóźnić wywołanie jednej z animacji (lub dopasować wywołania większej ich liczby), musimy skorzystać z właściwości `BeginTime`:

```
<Storyboard>
 <DoubleAnimation BeginTime="0:0:5" Storyboard.TargetName="przycisk" Storyboard.
 TargetProperty="Width" From="100" To="150" Duration="0:0:10"/>
 <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
 "Height" From="100" To="150" Duration="0:0:10"/>
</Storyboard>
```

Dzięki temu przez pierwsze pięć sekund będzie powiększana wysokość, przez kolejne pięć — zarówno wysokość, jak i szerokość, a przez ostatnie pięć — tylko szerokość.

Kontrola uruchomionej animacji

Spośród wszystkich elementów animacji nie zajmowaliśmy się dotychczas jednym — `BeginStoryboard`. W żaden sposób nie wpływał on na działanie animacji, jakie jest więc jego znaczenie? Jest to jeden ze znaczników, których zadaniem jest zarządzanie działaniem animacji. Zgodnie ze swoją nazwą, obiekt `BeginStoryboard` po prostu rozpoczyna odtwarzanie animacji. Można jednak wnioskować, że istnieją też inne znaczniki, które mają wpływ na działanie aplikacji — i takie wnioskowanie jest słuszne.

Typowym przykładem jest obiekt `PauseStoryboard`. Umożliwia on wstrzymanie wykonywania animacji, jednak bez jej przerywania — można wtedy rozpocząć odtwarzanie animacji od momentu wstrzymania. Wznowienie umożliwia obiekt `ResumeStoryboard`. Jedyny problem może stanowić połączenie wszystkich tych konstrukcji w jedną całość — ten proces prezentuje poniższy kod:

```
<Button Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
   <BeginStoryboard Name="story">
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
     "Height" From="100" To="150" Duration="0:0:10"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
   <PauseStoryboard BeginStoryboardName="story"/>
```

```
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseLeave">
   <ResumeStoryboard BeginStoryboardName="story"/>
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Przykład dokonuje powiększenia wysokości przycisku o 50 pikseli, przy czym jeśli kursor znajdzie się nad przyciskiem, wykonywanie animacji zostanie wstrzymane. Opuszczenie przycisku przez kursor spowoduje wznowienie animacji. Blok `BeginStoryboard`, z którym mieliśmy już do czynienia, został zaopatrzony w nazwę. Wymagają tego pozostałe obiekty kontrolujące animację — przecież muszą w jakiś sposób odwołać się do bloku animacji. Każdy z pozostałych obiektów kontrolujących jest umieszczony w osobnym triggerze, gdyż każdy jest związany z innym zdarzeniem. Właściwość `BeginStoryboardName` (jedyna dostępna w kodzie XAML właściwość w tych klasach) umożliwia określenie bloku początkowego animacji.

Oczywiście, obiekty znajdujące się w powyższym kodzie nie są jedynymi, które należą do „rodziny" obiektów kontrolujących. Pełne zatrzymanie animacji umożliwia obiekt `StopStoryboard`. Po jego wywołaniu nie ma sensu wykonywanie (oczywiście, to określenie oznacza wykonanie zdarzenia, które wywoła stosowny trigger) obiektu `ResumeStoryboard`. Do bardziej skomplikowanych obiektów z pewnością możemy zaliczyć `SeekStoryboard`. Dzięki niemu można zmieniać aktualną pozycję animacji. Przesunięcie określa właściwość `Offset` (typu `TimeSpan`), natomiast sposób jego obliczenia — właściwość `Origin` (przyjmuje wartość `BeginTime` (domyślną), jeśli ma być liczone od początku animacji, lub `Duration`, jeśli od końca — w takim przypadku należy podać wartość ujemną). Poniższy przykład prezentuje sytuację, w której najechanie kursorem myszy na przycisk powoduje cofnięcie animacji do początku:

```
<Button Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
   <BeginStoryboard Name="story">
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
     "Height" From="100" To="150" Duration="0:0:10"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
   <SeekStoryboard BeginStoryboardName="story" Offset="-0:0:10" Origin="Duration" />
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Animacja zaczyna odtwarzanie tuż po załadowaniu przycisku. Najechanie kursorem na przycisk powoduje cofnięcie o 10 sekund w odniesieniu do końca animacji — jest to równoznaczne z określeniem właściwości `Origin` na `BeginTime` i `Offset` na `0:0:0`.

Możemy zmieniać pozycję animacji, możemy zmieniać także szybkość jej działania. Jak pamiętamy, odpowiada za to właściwość `SpeedRatio`. Istnieje możliwość jej zmiany w trakcie działania animacji. Wystarczy skorzystać z obiektu `SetStoryboardSpeedRatio`:

```
<Button Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
   <BeginStoryboard Name="story">
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
     "Height" From="100" To="150" Duration="0:0:10"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
   <SetStoryboardSpeedRatio BeginStoryboardName="story" SpeedRatio="2.0"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseLeave">
   <SetStoryboardSpeedRatio BeginStoryboardName="story" SpeedRatio="1.0"/>
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Najechanie kursorem spowoduje dwukrotne przyspieszenie wykonywania animacji, a opuszczenie przycisku — powrót do normalnego tempa.

Przedostatnim elementem, który ma wpływ na wykonywanie animacji, jest `SkipStoryboardToFill`. Dzięki niej animacja jest przestawiana w stan końcowy (treść pozostała do końca animacji jest pomijana):

```
<Button Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
   <BeginStoryboard Name="story">
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
     "Height" From="100" To="150" Duration="0:0:10"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
   <SkipStoryboardToFill BeginStoryboardName="story"/>
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Niezależnie od aktualnego stanu animacji, najechanie myszą na przycisk spowoduje zakończenie animacji.

Dobrą praktyką jest wywołanie, naturalnie po zakończeniu wszystkich koniecznych operacji, obiektu `RemoveStoryboard`, który zwalania zasoby związane z aplikacją. Poza tym pozwala on na wykorzystywanie animowanej właściwości po zakończeniu działania aplikacji — jest to już trzeci sposób, aby osiągnąć ten efekt. Dzięki temu obiektowi wykonanie poniższego kodu będzie miało sens:

```
void odjazd(object o, MouseEventArgs rea)
{
 przycisk.Height = 200.0;
}
<Button MouseLeave="odjazd" Content="Przycisk" Width="100" Height="100" Name="przycisk">
 <Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
   <BeginStoryboard Name="story">
    <Storyboard>
     <DoubleAnimation Storyboard.TargetName="przycisk" Storyboard.TargetProperty=
     "Height" From="100" To="150" Duration="0:0:10"/>
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
   <SkipStoryboardToFill BeginStoryboardName="story"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseLeave">
   <RemoveStoryboard BeginStoryboardName="story"/>
  </EventTrigger>
 </Button.Triggers>
</Button>
```

Najechanie kursorem na przycisk spowoduje zakończenie działania animacji, ale nie zwolnienie jej zasobów. Dzieje się tak dopiero, gdy kursor opuszcza przycisk. Najpierw jest wywoływany trigger — zwolnienie alokowanych zasobów i przede wszystkim uwolnienie animowanej właściwości — a następnie zdarzenie `MouseLeave`, które prowadzi do uruchomienia metody `odjazd`. Dochodzi w niej do zmiany właściwości `Height`, co bez wprowadzenia elementu `RemoveStoryboard` nie mogłoby mieć miejsca.

Pytania testowe

1. Do określenia kilku warunków naraz dla jednego triggera wykorzystuje się klasę:
   - **a)** `EventTrigger`,
   - **b)** `EventSetter`,
   - **c)** `Trigger`,
   - **d)** `MultiTrigger`.
2. Wszystkie trzy rodzaje animacji są możliwe dla typu:
   - **a)** `Double`,
   - **b)** `Point3D`,
   - **c)** `Rect`,
   - **d)** `Quaternion`.

3. Krzywa Béziera może być zastosowana w klatce typu:

a) `Discrete`,

b) `Spline`,

c) `Linear`,

d) w żadnej z powyższych.