<http://msdn2.microsoft.com/en-us/magazine/cc500570.aspx>

**Silverlight**

**Tworzenie zaawansowanych animacji trójwymiarowych w Silverlight 2.0**

Declan Brennan

**Kod dostępny do pobrania:** [SilverlightAnimation2008\_04.exe](http://download.microsoft.com/download/f/2/7/f279e71e-efb0-4155-873d-5554a0608523/SilverlightAnimation2008_04.exe) (228 KB)
[Przejrzyj kod w trybie on-line](http://msdn2.microsoft.com/en-us/magazine/cc501070.aspx)

|  |  |
| --- | --- |
| zagadnienia omówione w artykule:* Podstawy XAML
* Tworzenie elementów w XAML
* Jak poskładać wielościan
* Emulowanie obliczeń DirectX
 | artykuł dotyczy następujących technologii: Silverlight |

 Spis treści

[Zastosowanie XAML](#_Toc198550034)

[Kilka przykładów XAML](#_Toc198550035)

[Wskazówki dotyczące pracy z XAML](#_Toc198550036)

[Jak poskładać wielościan](#_Toc198550037)

[Emulowanie obliczeń DirectX](#_Toc198550038)

[Więcej do odkrycia](#_Toc198550039)

Jeśli cały szum, wywołany w ciągu ostatnich kilku miesięcy wokół technologii Silverlight™, jakimś dziwnym zrządzeniem losu ominął cię, pozwól, że szybko naświetlę sytuację — Silverlight to opracowana przez Microsoft nowa wtyczka współpracująca z różnymi przeglądarkami internetowymi, pozwalająca na korzystanie z funkcjonalności Microsoft®.NET Framework w obszarach, które do tej pory zarezerwowane były wyłącznie dla aplikacji Flash i apletów Java. W Silverlight wbudowano wiele użytecznych funkcji. Technologia ta udostępnia odchudzoną funkcjonalność biblioteki .NET Framework 3.5, w tym między innymi obsługę XML i XAML, kolekcji generycznych, usług sieciowych i LINQ. Pozwala na korzystanie z całej gamy języków zgodnych z .NET, jednak w tym artykule ograniczymy się do języka C#.

Sądzę, że najlepszym sposobem zapoznania się z nową technologią jest pobawienie się nią. Dlatego, gdy pojawił się Silverlight 1.1 w wersji alfa i zobaczyłem fascynującą prezentację Tima Sneatha na konferencji IMT w Dublinie, zdecydowałem się przygotować niewielką aplikację edukacyjną, demonstrującą, w jaki sposób z płaskiego szablonu można poskładać różne trójwymiarowe kształty (wielościany). Silverlight domyślnie nie obsługuje animacji 3D, dlatego na potrzeby aplikacji musiałem przygotować emulację bibliotek DirectX® do obliczeń trójwymiarowych.

Wielościan to obiekt trójwymiarowy o płaskich ścianach. W mojej aplikacji zająłem się wielościanami foremnymi i półforemnymi, zwanymi także odpowiednio platońskimi i archimedesowymi. Ściany takich wielościanów są wielokątami foremnymi (o wszystkich bokach tej samej długości), takimi jak trójkąt równoboczny czy kwadrat. Wielościany te są także wypukłe, tzn. nie mają żadnych odstających elementów. Jak można wywnioskować z nazw wywodzących się z antycznej Grecji, obiekty te od dawna fascynują ludzkość. Dużo więcej informacji na ich temat można znaleźć w witrynie George’a Harta pod adresem [georgehart.com/virtual-polyhedra/vp.html](http://georgehart.com/virtual-polyhedra/vp.html).

Działającą aplikację można zobaczyć na **ilustracji 1.** lub pod adresem [picturespice.com/ps/Polyhedra/](http://picturespice.com/ps/Polyhedra/). Aplikacja pozwala na wybranie bryły (wielościanu) poprzez wskazanie jej myszą. W prawym górnym rogu okna wyświetlane są informacje na temat wybranej bryły. Prezentowana jest także animacja przejścia od płaskiego szablonu do wielościanu. Kliknięcie przycisku Cycle powoduje automatyczne przechodzenie pomiędzy poszczególnymi bryłami.

Ilustracja 1. **Demonstracja wielościanów w Silverlight**

# Zastosowanie XAML

Podobnie jak inne aplikacje Silverlight, aplikacja Polyhedra intensywnie korzysta z języka XAML, który jest językiem definiowania treści. W pewnym sensie język ten jest odpowiednikiem języka HTML, ale jest bardziej elastyczny. Idąc dalej — o ile możliwe jest utworzenie strony HTML wyłącznie w oparciu o obiektowy model dokumentu HTML (DOM), stosowanie takiej metody rzadko ma sens — strony takie bardzo wolno się inicjalizują, a ich przygotowanie jest czasochłonne. Prawie zawsze lepiej jest jak najwięcej treści strony umieścić w kodzie HTML i korzystając z JavaScript i DOM dodać uzupełnienia tam, gdzie potrzebna jest większa elastyczność.

Bardzo podobna sytuacja ma miejsce w przypadku XAML. Najszybszym sposobem tworzenia treści jest generowanie kodu XAML i — tam, gdzie jest to niezbędne — uzupełnianie go z wykorzystaniem języka zgodnego z .NET, takiego jak C#, i interfejsu Silverlight Media API. Kod XAML można tworzyć ręcznie lub z wykorzystaniem oprogramowania projektowego takiego jak Expression BlendTM, albo generować za pomocą aplikacji uruchamianej w ramach procesu projektowania, albo nawet dynamicznie generować na serwerze. Tworzenie aplikacji Silverlight może wymagać zmiany podejścia do zagadnień programistycznych. Typowy programista C# może zagalopować się, próbując w swoim naturalnym środowisku kodować funkcjonalność, którą najlepiej byłoby zawrzeć w XAML.

Wdawanie się w szczegóły na temat XAML byłoby wyjściem poza ramy tematyczne tego artykułu. Na szczęście Charles Petzold szczegółowo opisał XAML w swojej książce pod tytułem „Applications= Code+Markup”.

Oto napisany w XAML odpowiednik przykładu „Hello World”, jaki przywykliśmy analizować, ucząc się nowych języków programowania:

Kopiuj kod

<UserControl x:Class="Polyhedra.Page"

 xmlns="http://schemas.microsoft.com/client/2007"

 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

 Width="400" Height="300">

 <Grid x:Name="LayoutRoot" Background="White">

 <TextBlock>Hello World</TextBlock>

 </Grid>

</UserControl>

Elementem-korzeniem jest UserControl. Wewnątrz tego elementu znajduje się element Grid, a z kolei wewnątrz tego elementu znajduje się element TextBlock z tekstem „Hello World”.

Przyjrzyjmy się atrybutom elementu UserControl. Bez wdawania się w szczegóły możemy powiedzieć, że zdefiniowany jest tu odpowiednik klasy z kodem schowanym dla pliku XAML. Instancja tej klasy jest tworzona podczas procesu parsowania i ładowania kodu XAML. W konstruktorze tej klasy można umieścić kod inicjalizujący różne elementy, natomiast do zdefiniowania bardziej wyrafinowanych zachowań może być konieczne wykorzystanie procedur obsługi zdarzeń. Tu uwidacznia się główna zaleta Silverlight. Procedury obsługi zdarzeń mogą być przypisywane do różnych obiektów XAML i implementowane w dowolnym, zgodnym z .NET języku programowania, który — w odróżnieniu od JavaScript — jest kompilowany, co daje wiele nowych możliwości, których uzyskanie do tej pory nie było możliwe.

Ponownie sięgnijmy do analogii z HTML. Elementy stron HTML są często grupowane w elementach DIV w celu ustalenia ich pozycji na stronie. Analogicznie, w XAML kształty są grupowane wewnątrz elementów Canvas (lub innych elementów, takich jak Grid, które są odmianą elementów Canvas). Tak jak w przypadku stron HTML elementy DIV mogą być zagnieżdżane, tak samo w XAML można zagnieżdżać elementy Canvas. Większość elementów HTML ma kształt prostokątny. Natomiast w XAML występuje duża różnorodność kształtów, takich jak TextBlock, Rectangle, Polygon, Ellipse czy bardzo elastyczny kształt Path, pozwalający na tworzenie kształtów zdefiniowanych przez użytkownika. Elementy HTML są identyfikowane za pomocą atrybutu ID, którego odpowiednikiem w XAML jest x:Name, przy czym x jest aliasem przestrzeni nazw XAML.

XAML pozwala na znacznie więcej niż tylko tworzenie statycznych stron. Jedną z najważniejszych jego funkcji jest możliwość zastosowania scenopisów (storyboard) do opisu animacji początkowego układu elementów opisanego w XAML (podobna technologia, nazywana HTML+Time, została wprowadzona w HTML w przeglądarce Internet Explorer® 5.0). Animować można na przykład zmiany koloru, widoczności czy przezroczystości obiektu. W powiązaniu z transformacjami scenopisy mogą także powodować obracanie, skalowanie i przesuwanie obiektów.

Scenopisy pozwalają na tworzenie dowolnych efektów animacyjnych prawie bez pisania konwencjonalnego kodu. W powiązaniu z wyzwalaczami (trigger) teoretycznie pozwalają na automatyczne uruchamianie animacji w odpowiedzi na zdarzenia takie jak MouseEnter. Jednakże w Silverlight w wersji 2.0 Beta z marca 2008 roku, jedynym zdarzeniem obsługiwanym przez wyzwalacze było zdarzenie Loaded. W przypadku innych zdarzeń konieczne jest napisanie niewielkiej ilości kodu łączącego, mającego postać procedury obsługi zdarzenia. Podejrzewam, że wkrótce problem ten zostanie rozwiązany (o ile już go nie rozwiązano).

Ciekawą cechą scenopisów Silverlight jest to, że są one definiowane w odniesieniu do czasu, a nie liczby klatek. Złożone zachowania można w prosty sposób zaimplementować, stosując wiele niezależnych scenopisów, powiązanych z różnymi momentami czasowymi i zdarzeniami. W końcu w świecie rzeczywistym jednostką czasu nie jest klatka — efekt uboczny sposobu zapisywania ruchomych obrazów na taśmie celuloidowej. Gdy niezależne obiekty mogą być implementowane z wykorzystaniem niezależnych zachowań, życie staje się prostsze.

# Kilka przykładów XAML

Animacja składania wielościanu, widoczna w środkowej części aplikacji Polyhedra, jest generowana przez kod napisany w języku C#. Jednak większość pozostałej części interfejsu jest zdefiniowana w XAML — w tym krąg przykładowych wielościanów, różnorodne sposoby akcentowania aktualnie wybranego wielościanu czy przycisk Cycle, którego aktywność sygnalizowana jest przez animowaną, obracającą się strzałkę.

Przeanalizujmy kilka fragmentów pliku Page.xaml, aby zobaczyć, jak uzyskać poszczególne efekty animacyjne. Zaczniemy od przycisku Cycle, którego kod przedstawiono na **ilustracji 2.** Zwróćmy uwagę na element Path o nazwie Cycle. Jego atrybut Data definiuje serię operacji kodowanych za pomocą pojedynczych liter: M jak move (przesunięcie), A jak arc (łuk), L jak Line (linia). Ponieważ jest to dość prosty symbol, po prostu narysowałem na kartce papieru kształt, jaki chciałem uzyskać, i na podstawie rysunku ręcznie przygotowałem listę wymaganych operacji. W większości wypadków szybciej pracuje się, korzystając z narzędzia takiego jak Expression Design.

 Ilustracja 2. Przycisk Cycle

Kopiuj kod

<Canvas x:Name="CycleButton"

 Canvas.Top="250"

 MouseLeftButtonDown="CycleButtonLeftMouseDown" >

 <Path x:Name="Cycle" Stroke="#000033"

 Fill="#FFB47C0D" Canvas.Left="10" Canvas.Top="10"

 Data="M 25,35 A 10,10 180 1 0 25,15 L 25,20 L 12.5,10 L 25,0 L 25,

 5 A 10,10 180 1 1 25,45 Z"

 Width="50" Height="50">

 <Path.RenderTransform>

 <RotateTransform x:Name="CycleRotate" Angle="0"

 CenterX="25" CenterY="25"/>

 </Path.RenderTransform>

 <Path.Resources>

 <Storyboard x:Name="CycleLatched">

 <DoubleAnimation Storyboard.TargetName="CycleRotate"

 Storyboard.TargetProperty="Angle" From="360" To="0"

 Duration="00:00:02" RepeatBehavior="Forever"/>

 </Storyboard>

 </Path.Resources>

 </Path>

 <TextBlock x:Name="CycleCaption" Canvas.Left="65" Canvas.Top="10"

 Foreground="#FFB47C0D" FontSize="30" FontWeight="Bold"

 Text="Cycle" />

 <Rectangle Width="200" Height="70" RadiusX="30" RadiusY="30"

 Stroke="#FFB47C0D" StrokeThickness="4" Fill="Transparent"/>

</Canvas>

Element Path zawiera także element RotateTransform o nazwie CycleRotate. Na początku transformacja ta nie realizuje żadnego zadania, ponieważ kąt obrotu jest ustawiony na zero. Jednak element Path zawiera także element Storyboard o nazwie CycleLatched, który po aktywowaniu stale zwiększa kąt obrotu strzałki o małą wartość, co w efekcie powoduje obracanie się strzałki.

**Na ilustracji 3. przedstawiono definicję jednego z przykładowych wielościanów. W dolnej części kodu XAML widoczne są cztery elementy** Polygon, będące ścianami czworościanu foremnego. Elementy te znajdują się wewnątrz elementu Canvas o nazwie Model0 i są otoczone pierścieniem zbudowanym z początkowo niewidocznego elementu Ellipse o nazwie Ring0. Zdefiniowano dwa scenopisy (animacje) Storyboard — dla zdarzenia MouseEnter i MouseLeave. Animacja MouseEnter powoduje, że pierścień natychmiast staje się widoczny oraz w czasie 0,7 sekundy zwiększa rozmiar elementu o nazwie Model0, zmniejszając przy tym jego przezroczystość. Animacja MouseLeave jest odwróceniem animacji MouseEnter.

 Ilustracja 3. Przykład wielościanu

Kopiuj kod

<Canvas x:Name="Canvas0" Width="116.376" Height="116.376"

 Canvas.Left="671.812" Canvas.Top="341.812"

 MouseEnter="TriggerMouseEnter" MouseLeave="TriggerMouseLeave">

 <Canvas.Resources>

 <Storyboard x:Name="Canvas0MouseEnter">

 <DoubleAnimation Duration="00:00:00" Storyboard.TargetName="Ring0"

 Storyboard.TargetProperty="Opacity" From="0.0" To="1.0" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0"

 Storyboard.TargetProperty="Opacity" From="0.7" To="1.0" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0Scale"

 Storyboard.TargetProperty="ScaleX" From="0.7" To="1.0" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0Scale"

 Storyboard.TargetProperty="ScaleY" From="0.7" To="1.0" />

 </Storyboard>

 <Storyboard x:Name="Canvas0MouseLeave">

 <DoubleAnimation Duration="00:00:00" Storyboard.TargetName="Ring0"

 Storyboard.TargetProperty="Opacity" From="1.0" To="0.0" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0"

 Storyboard.TargetProperty="Opacity" From="1.0" To="0.7" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0Scale"

 Storyboard.TargetProperty="ScaleX" From="1.0" To="0.7" />

 <DoubleAnimation Duration="00:00:00.7"

 Storyboard.TargetName="Model0Scale"

 Storyboard.TargetProperty="ScaleY" From="1.0" To="0.7" />

 </Storyboard>

 </Canvas.Resources>

 <Canvas x:Name="Model0" Opacity="0.7"

 Width="116.376" Height="116.376" >

 <Canvas.RenderTransform>

 <ScaleTransform x:Name="Model0Scale" CenterX="58.188"

 CenterY="58.188" ScaleX="0.7" ScaleY="0.7"/>

 </Canvas.RenderTransform>

 <Polygon Canvas.ZIndex="-12545653" Fill="#FFCC0000"

 Stroke="#000000" StrokeThickness="1"

 Points="97.566,74.278 43.052,40.971 33.836,99.937 "/>

 <Polygon Canvas.ZIndex="-11948057" Fill="#FFCC0000"

 Stroke="#000000" StrokeThickness="1"

 Points="43.052,40.971 97.566,74.278 58.188,37.662 "/>

 <Polygon Canvas.ZIndex="-11309683" Fill="#FFCC0000"

 Stroke="#000000" StrokeThickness="1"

 Points="33.836,99.937 43.052,40.971 58.188,37.662 "/>

 <Polygon Canvas.ZIndex="-10481036" Fill="#FFCC0000"

 Stroke="#000000" StrokeThickness="1"

 Points="97.566,74.278 33.836,99.937 58.188,37.662 "/>

 </Canvas>

 <Ellipse x:Name="Ring0" Opacity="0" Stroke="#FFB47C0D"

 StrokeThickness="4" Width="116.376" Height="116.376" />

</Canvas>

Ponieważ scenopisy mogą pracować całkowicie niezależnie, wszystko działa tak, jak można się tego spodziewać, niezależnie od tego, jak szybko użytkownik porusza kursorem myszy. Zwykle animacja MouseLeave jednego przykładu działa jednocześnie z animacją MouseEnter nowo wybranego przykładu. Nie trzeba się martwić — Silverlight automatycznie obsługuje równoległą realizację wielu animacji.

Możecie zastanawiać się, w jaki sposób wyliczyłem współrzędne wierzchołków wielokątów tworzących czworościan, nie wspominając już o współrzędnych znacznie większej liczby wielokątów potrzebnych do budowy pozostałych brył. Należę do „leniwców”, którzy nigdy nie zabierają się do robienia czegoś, co komputer może zrobić szybciej. Wykorzystałem zmodyfikowaną wersję aplikacji Polyhedra. Wersja ta obliczała początkową fazę (złożona, zamknięta bryła) centralnej animacji dla wszystkich wielościanów. Wyniki obliczeń w postaci grup elementów Polygon umieściłem wewnątrz kilkunastu elementów Canvas, rozmieszczonych równomiernie po okręgu, i zapisałem całość w pliku do wykorzystania w głównym programie.

Równomierne rozmieszczenie elementów po okręgu wymaga przesuwania się o stały kąt, równy 2\*PI/LiczbaPrzykładów, i ustawiania środka każdej bryły we współrzędnych x= promień\*Cos(kąt), y=promień\*Sin(kąt). Ponieważ elementy Canvas pozycjonowane są z wykorzystaniem właściwości Left i Top, musiałem przesunąć punkt centralny o połowę szerokości i połowę wysokości pojedynczego obiektu.

# Wskazówki dotyczące pracy z XAML

Jak widać, XAML pozwala na uzyskanie dość bogatego interfejsu użytkownika prawie bez pisania dodatkowego kodu. Nawet w przypadku dużych plików, takich jak Page.xaml, inicjalizacja przebiega zaskakująco szybko. Zanim jednak przejdę do dalszej części opisu aplikacji, chciałbym przedstawić kilka wskazówek opartych na moich doświadczeniach z obecną implementacją Silverlight 2.0 (wersja beta z marca 2008).

Jak już wspomniałem, obecnie wyzwalacze mogą automatycznie uruchomić animację (za pośrednictwem metody Begin) tylko z poziomu zdarzenia Loaded. W przypadku innych zdarzeń potrzebna jest niewielka ilość kodu łączącego — trzeba napisać procedurę obsługi zdarzeń o następującej postaci:

Kopiuj kod

public void MouseEnterHandler(

 object o, EventArgs e) {

 this.MouseEnterStoryBoard.Begin();

}

Jeśli elementy Storyboard będą nazywane zgodnie z określoną konwencją nazewniczą, na przykład nazwa obiektu uzupełniona nazwą zdarzenia, można zmniejszyć liczbę potrzebnych procedur obsługi zdarzeń. Na przykład w aplikacji Polyhedra wszystkie przykładowe bryły mają jedną wspólną procedurę obsługi zdarzeń o następującym kodzie:

Kopiuj kod

public void MouseEnterHandler(

 object o, EventArgs e) {

 this.triggerStoryboard(o,"MouseEnter");

}

private bool triggerStoryboard(

 object o, string eventType) {

 Canvas el = o as Canvas;

 string name= el.GetValue(NameProperty) as String;

 Storyboard sb = el.FindName(name + eventType) as Storyboard;

 if (sb != null)

 sb.Begin();

 return (sb != null);

}

W Silverlight 2.0 Beta proces inicjalizacji (wywoływanie InitComponents) przeniesiono z procedury obsługi zdarzenia Loaded do konstruktora obiektu kodu schowanego. Podejście to jest bardziej eleganckie, jednak nie wszystkie operacje da się zrealizować z poziomu konstruktora. Nie jest możliwe na przykład wywoływanie metod Begin i Pause obiektu Storyboard — trzeba to zrobić z poziomu procedury obsługi zdarzenia.

Przeglądając kod przygotowanej przez Andyego Beaulieu przykładowej gry Silverlight pod tytułem Rocks!, polegającej na niszczeniu planetoid (gra i kod dostępne pod adresem [www.andybeaulieu.com/Home/tabid/67/EntryID/73/default.aspx](http://www.andybeaulieu.com/Home/tabid/67/EntryID/73/default.aspx)), odkryłem, że dobrym sposobem na uzyskanie animacji sterowanej z poziomu kodu aplikacji jest wykorzystanie elementu Storyboard z krótkim okresem animacji oraz procedurą obsługi zdarzenia Completed, w której zawarty jest kod realizujący jedną klatkę animacji i ponownie uruchamiający element Storyboard:

Kopiuj kod

public Page() { // konstruktory obiektu z kodem schowanym

 // niezbędne do inicjalizacji zmiennych

 InitializeComponent();

 this.animationTimer.Completed +=

 new EventHandler(animationTimer\_Completed);

}

void animationTimer\_Completed(object sender, EventArgs e) {

 [ Realizacja jednej klatki animacji ]

 this.animationTimer.Begin();

}

W Silverlight 2.0 Alpha z września 2007 zmieniono wymagania dotyczące obiektów Storyboard — każda animacja musi mieć teraz zdefiniowany cel, nawet jeśli nie jest on wykorzystywany:

Kopiuj kod

<Canvas.Resources>

 <Storyboard x:Name="animationTimer">

 <DoubleAnimation Duration="00:00:00.01"

 Storyboard.TargetName="bogusTimerTarget"

 Storyboard.TargetProperty="Width" />

 </Storyboard>

</Canvas.Resources>

<Canvas Name="bogusTimerTarget">

</Canvas>

Nie należy stosować wielu odrębnych kontrolek Silverlight na tej samej stronie HTML. Pierwsza implementacja aplikacji Polyhedra korzystała z odrębnej kontrolki dla każdej przykładowej bryły, co powodowało ogromne zapotrzebowanie na pamięć operacyjną. Ograniczenie liczby kontrolek może się wiązać z koniecznością przeniesienia części treści z kodu HTML do XAML.

Jedną z najważniejszych zalet XAML jest odciążenie programisty od rutynowych zadań, związanych z tworzeniem interfejsu użytkownika, co pozwala na skupienie się na kodzie dotyczącym domeny problemu. W naszym przykładzie domeną problemu jest sposób składania szablonów w celu uzyskania trójwymiarowych brył. Opisałem go w następnej sekcji.

# Jak poskładać wielościan

Podejrzewam, że wspomniana przeze mnie książka Charlesa Petzolda jest adaptacją znacznie starszej, pochodzącej jeszcze sprzed ery programowania obiektowego, książki Niklausa Wirtha pod tytułem „Algorithms+Data Structures=Programs” (wydanie polskie pod tytułem „Algorytmy + struktury danych = programy”). Chociaż upłynęło tak wiele czasu, książka ta jest dla mnie jedną z najważniejszych pozycji, jakie przeczytałem, i pomimo wszystkich zmian, jakie od tamtego czasu zaszły w językach i technikach programowania, jest nadal bardzo aktualna. W książce tej opisano technikę programowania polegającą na określeniu struktur danych pozwalających najlepiej zamodelować rozwiązywany problem, a następnie znalezieniu algorytmów przetwarzających i modyfikujących te struktury danych. Gdy zabieram się do pisania niestandardowego programu, bardzo często korzystam z tej właśnie techniki programowania.

Zanim dokonałem ostatecznego wyboru algorytmu dla aplikacji Polyhedra, rozważałem wiele różnych technik. Chciałem określić najmniejszą ilość informacji niezbędnych do realizacji animacji składania wielościanów. Okazuje się, że ze względu na to, że wszystkie krawędzie mają tę samą długość, w zasadzie wystarczy jedynie wiedza o wzajemnym usytuowaniu ścian w danym wielościanie. Logicznym rozwiązaniem jest więc zastosowanie struktury danych grafu. Aby uzyskać końcową postać XAML, zastosowałem kilka algorytmów przetwarzających graf w dwie różne drzewiaste struktury danych. Cały ten proces opiszę później.

Chociaż Windows® Presentation Foundation (WPF) obsługuje obiekty trójwymiarowe w XAML, Silverlight domyślnie obsługuje tylko obiekty dwuwymiarowe — zgodność z wieloma różnymi przeglądarkami jest znacznie łatwiej uzyskać, gdy nie trzeba martwić się o współpracę z procesorami graficznymi znajdującymi się w maszynach, na których uruchamiane są te przeglądarki. Oczywiście, jeśli bliżej przyjrzeć się problemowi, obraz trójwymiarowy na ekranie komputera jest jedynie złudzeniem. Niezależnie od wykonywanych operacji, obraz na powierzchni monitora jest jedynie zbiorem dwuwymiarowych wielokątów. Można samodzielnie napisać kod transformacji 3D, wyznaczający współrzędne wierzchołków wielokątów, i obejść się bez sprzętowych akceleratorów 3D. Jeżeli ograniczymy się do kilkudziesięciu wielokątów, wydajność będzie do zaakceptowania (tym bardziej, że między wersją alfa i beta podniesiono wydajność liczoną w klatkach na sekundę).

Po wybraniu bryły tworzony jest dwuwymiarowy, płaski szablon zwany siatką (po angielsku net, ale nie ma to nic wspólnego z .NET). Na proces ten składają się dwie fazy, przedstawione na **ilustracji 4**. Najpierw w pamięci operacyjnej tworzony jest graf, którego każdy węzeł (obiekt GraphNode) odpowiada jednej ścianie bryły (patrz plik Graph.cs). Graf ten budowany jest na podstawie zbioru informacji o krawędziach (które ściany łączą się z którymi), pobieranego z pliku zasobu .shp, osadzonego w podzespole aplikacji. Poniżej przedstawiono zawartość pliku cube.shp:



Ilustracja 4. **Budowa bryły — od grafu do XAML** (kliknij, aby wyświetlić ilustrację w większej rozdzielczości)

Kopiuj kod

1:3,4,5,2

2:1,5,6,3

3:2,6,4,1

4:3,6,5,1

5:1,4,6,2

6:2,5,4,3

Jako pierwszy węzeł, od którego zostanie rozpoczęte tworzenie siatki (szczegóły można znaleźć w pliku Net.cs), może zostać wybrany dowolny obiekt GraphNode. Dla każdego węzła tworzony jest dwuwymiarowy obiekt FlatFace o liczbie boków równej liczbie sąsiadów obiektu GraphNode. Następnie wybierany jest dowolny, sąsiadujący obiekt GraphNode i proces jest powtarzany, a kolejne obiekty FlatFace budowane są w taki sposób, by miały wspólną krawędź z obiektem bieżącym. Każdy obiekt GraphNode odwiedzany jest tylko raz, a proces kontynuowany jest do momentu, w którym odwiedzone zostaną wszystkie obiekty GraphNode. Wynikiem procesu jest drzewiasta struktura obiektów FlatFace.

W każdej klatce animacji 3D, w oparciu o siatkę uzyskaną w poprzednim etapie procesu budowany jest trójwymiarowy, częściowo poskładany wielościan (szczegóły tej operacji można znaleźć w pliku polyhedron.cs, omówionym w dalszej części tego artykułu). W tym celu na podstawie drzewa obiektów FlatFace tworzone jest drzewo obiektów Faces, w którym dwuwymiarowe współrzędne wierzchołków zastępowane są współrzędnymi trójwymiarowymi. Zastosowane przeze mnie układy współrzędnych przedstawiłem na **ilustracji 5**. Ponieważ chcę, by początkową pozycją wielościanu była płaszczyzna pozioma, wartość początkowa współrzędnej Y powinna wynosić zero. Dlatego przejście ze współrzędnych dwuwymiarowych na trójwymiarowe realizuję wykonując transformację: X=X, Y=0 i Z=Y.



Ilustracja 5. **Transformacja współrzędnych dwuwymiarowych na trójwymiarowe** (kliknij, aby wyświetlić ilustrację w większej rozdzielczości)

Następnie, korzystając z rekursji, dla każdej krawędzi w drzewie obiektów Face ustawiam kąt zgięcia. Kąt ten jest ułamkiem kąta (zwanego kątem dwuściennym), mierzonego między ścianami zamkniętej bryły (całkowicie złożonej). Teoretycznie możliwe jest bezpośrednie obliczenie kątów dwuściennych wyłącznie na podstawie informacji na temat sąsiadowania ścian, zawartych w pliku .shp.

Obliczenia te są dość proste dla przypadków szczególnych, takich jak trzy ściany schodzące się w jednym wierzchołku albo dowolna liczba ścian tego samego typu schodzących się w jednym wierzchołku. Niestety, obliczenia uogólnione (dla dowolnego przypadku) są bardziej złożone, dlatego zdecydowałem się na zapisanie tych kątów w osobnych plikach zasobów z rozszerzeniem .dihedrals. Oto fragment pliku cube.dihedrals:

Kopiuj kod

1,3:1.5707965056551

1,4:1.57079661280793

1,5:1.57079614793469

1,2:1.57079604078186

2,1:1.57079604078186

2,5:1.57079628466395

2,6:1.57079661280793

2,3:1.57079636892585

3,2:1.57079636892585

3,6:1.57079614793469

3,4:1.57079628466395

...

Wszystkie te wartości powinny w tym przypadku wynosić PI/2. Znikome różnice między nimi wynikają wyłącznie ze sposobu, w jaki wartości te zostały obliczone.

Ostatni etap procesu to seria transformacji, których efektem jest wyznaczenie rzutu wielościanu (w postaci zbioru wielokątów) na płaszczyznę ekranu. Jak można zauważyć, podczas animacji składania wielościan jest także obracany wokół osi pionowej. By uzyskać ten efekt, przesuwam punkt obrotu do początku układu współrzędnych, realizuję rotację i ponownie przesuwam go do położenia początkowego. Następnie wykonuję transformację perspektywy, dzięki czemu obiekty położone daleko (na osi Z) wydają się być mniejsze. Więcej informacji na temat wykonywania transformacji zamieściłem w następnym rozdziale.

Na koniec generowany jest zbiór wielokątów XAML reprezentujących rzut bryły 3D na płaszczyznę ekranu. Odpowiednia kolejność rysowania tych wielokątów przez Silverlight (według malejącej odległości od obserwatora) jest wymuszana poprzez ustawienie właściwości ZIndex każdego wielokąta w oparciu o współrzędną Z środka wielokąta 3D. Współrzędne są skalowane tak, by wartości typu float prawidłowo odwzorować za pomocą wartości typu int. Jest to dość uproszczony sposób obliczeń 3D, sprawdzający się tylko wtedy, gdy wielokąty są odpowiednio ułożone — na przykład nie przecinają się, co w tym przypadku można zagwarantować. Bardziej zaawansowane algorytmy 3D korzystają z mechanizmów takich jak bufory głębokości. Ponieważ do ich obsługi potrzebny jest dostęp do procesora graficznego, ich opis nie mieści się w zakresie tematycznym tego artykułu.

Została już tylko kosmetyka. Obiekty Polygon w języku XAML mogą być częściowo przezroczyste (efekt ten można uzyskać, ustawiając właściwość Opacity), co pozwala na uzyskanie artystycznego efektu częściowo przezroczystego wielościanu. I — jak się to mówi — to by było na tyle.

# Emulowanie obliczeń DirectX

Chciałbym poruszyć zagadnienie obliczeń stosowanych do uzyskania efektu animacji. Szczegółowe informacje na ten temat można znaleźć w wielu witrynach w Internecie, między innymi w wikipedia.com, euclideanspace.com, mathworld.wolfram.com, gamasutra.com czy gamedev.net.

Omawiając zamieszczone wcześniej przykłady XAML, wprowadziłem już niektóre transformacje 2D, takie jak RotationTransform czy ScaleTransform. W ich wypadku całą pracę wykonuje Silverlight. Gdy stosujemy XAML w środowisku WPF, możemy także korzystać z transformacji 3D, nie są one jednak dostępne w Silverlight. W WPF niskopoziomowe obliczenia wykonywane są w oparciu o DirectX — w bibliotece tej dostępny jest cały zbiór klas do wykonywania obliczeń na potrzeby transformacji 3D. By móc realizować takie same transformacje w Silverlight, utworzyłem biblioteki naśladujące obliczenia DirectX, pozwalające na stosowanie ich w środowisku Silverlight.

Osoby mające jakiekolwiek doświadczenie w stosowaniu Direct3D®, szybko rozpoznają znane implementacje klas Vector2, Vector3, Matrix itp. (patrz **ilustracja 6.**). Dla pozostałych zamieszczam pobieżny i niezbyt ścisły opis, uzupełniony kilkoma przykładami.



Ilustracja 6. **Klasy emulujące obliczenia DirectX** (kliknij, aby wyświetlić ilustrację w większej rozdzielczości)

Określenie lokalizacji punktu w przestrzeni dwuwymiarowej wymaga podania dwóch współrzędnych: X i Y. W przypadku przestrzeni trójwymiarowej potrzebne są trzy współrzędne: X, Y i Z. Pomimo że punkty nie są wektorami, punkty w przestrzeni dwuwymiarowej można przedstawiać jako obiekty Vector2, a w przestrzeni trójwymiarowej — jako obiekty Vector3. Wektory w rzeczywistości są obiektami o określonej wartości, kierunku i zwrocie, które można sobie wyobrazić jako strzałkę zaczynającą się w początku układu współrzędnych, a kończącą w określonym punkcie. W pewnym sensie można przyjąć, że wszystkie wektory rozpoczynają się w środku układu współrzędnych.

Trójwymiarowe bryły można przedstawiać jako zbiory ścian (wielokątów). Z kolei wielokąty mogą być reprezentowane przez zbiory punktów będących ich wierzchołkami. Praktycznie wszystkie działania, jakie możemy chcieć wykonać, polegają na transformacji tych punktów w inne zbiory punktów poprzez wykonywanie operacji takich jak rotacja, translacja (przesunięcie) czy skalowanie. W zasadzie każdą operację można przedstawić w postaci liczącej cztery wiersze i cztery kolumny tabeli nazywanej macierzą. Macierze w rzeczywistości mogą mieć dowolną liczbę kolumn i wierszy, ale nas interesuje tylko szczególny przypadek macierzy, stosowany do przekształcania punktów trójwymiarowych o tak zwanych współrzędnych homogenicznych. Operacje te nazywane są transformacjami liniowymi, ponieważ nie powodują zniekształceń obiektu, do którego są stosowane.

Istnieje bardzo użyteczny sposób łączenia dwóch macierzy w jedną, której zastosowanie daje taki sam efekt, jak operacje realizowane przez macierze składowe. Operacja ta nazywana jest mnożeniem macierzy i może być wielokrotnie powtarzana. Oznacza to, że cały szereg transformacji wykonywanych jedna po drugiej można przedstawić w postaci pojedynczej macierzy, której zastosowanie daje taki sam skutek. Użycie jednej macierzy pozwala na uzyskanie znacznie wyższej wydajności niż osobne realizowanie poszczególnych transformacji. Nie będę tu opisywał sposobu implementacji mnożenia macierzy ani ich stosowania — możemy po prostu używać ich jako narzędzia.

Aby dokonać transformacji punktu źródłowego na docelowy, także należy wykonać operację mnożenia, jednak tym razem mnożymy macierz przez wektor. W celu uniknięcia pomyłek z mnożeniem macierzy przez macierz, operację tę zaimplementowano w klasie Vector3 w metodzie TransformCoordinate.

Kilka przykładów powinno ułatwić zrozumienie tych działań. Najpierw przeanalizujmy fragment pliku polyhedron.cs, odpowiedzialny za zginanie ścian wzdłuż łączącej je krawędzi:

Kopiuj kod

Vector3 axis = axisTo - axisFrom;

Matrix foldTransform =

 Matrix.Translation(-axisFrom) \*

 Matrix.RotationAxis(axis, proportion \* \_dihedralAngle) \*

 Matrix.Translation(axisFrom);

Vector3[] p= Vector3.TransformCoordinate(

 face.Points,foldTransform);

Krawędź łącząca ściany zdefiniowana jest jako odcinek z punktu axisFrom do punktu axisTo. Operacja rotacji pozwala na obrócenie bryły wokół linii przebiegającej przez środek układu współrzędnych, dlatego przed wykonaniem rotacji musimy przesunąć jeden z punktów krawędzi do środka układu współrzędnych, a po wykonaniu rotacji przesunąć go z powrotem na swoje miejsce. Dlatego w celu otrzymania macierzy, której użyję do transformacji współrzędnych wierzchołków ściany, łączę trzy transformacje liniowe: przesunięcie do początku układu współrzędnych, rotację i przesunięcie w odwrotnym kierunku.

W kodzie tym zastosowałem mały trik — wektor reprezentujący oś obrotu także musi zaczynać się w środku układu współrzędnych, dlatego od wektora axisTo odjąłem wektor axisFrom, przesuwając tym samym wektor (axisFrom,axisTo) na miejsce (środek układu współrzędnych,axisTo-AxisFrom).

Jeśli nie znudziła się wam jeszcze matematyka, spróbujmy przeanalizować kolejny przykład, będący fragmentem pliku projector.cs. Kod ten służy do transformacji wielokątów tworzących wielościan na postać, jaką spodziewamy się zobaczyć na ekranie.

Kopiuj kod

Matrix projection =

 Matrix.Translation(-pivot) \*

 Matrix.RotationY(yaw) \*

 Matrix.RotationX(angle) \*

 Matrix.Translation(viewPoint) \*

 Geometry.Perspective(5);

Vector3[] p= Vector3.TransformCoordinate(face.Points,foldTransform);

Jak pamiętamy, wielościan podczas animacji składania jest obracany w płaszczyźnie poziomej. W związku z tym, punkt na osi obrotu jest przesuwany do środka układu współrzędnych, po czym następuje rotacja wokół osi pionowej (Y) o kąt odchylenia. Osoby, które pilotowały samolot (wszystko jedno, czy prawdziwy, czy w symulatorze), z pewnością wiedzą, że pozostałe dwa rodzaje rotacji nazywane są pochyleniem i przechyleniem.

Tym razem, zamiast przesunięcia bryły do pozycji wyjściowej, następuje przesunięcie do punktu obserwowania. Następnie wykonywana jest transformacja perspektywy, powodująca pomniejszenie obiektów znajdujących się daleko wzdłuż osi Z. Pomimo że wykonujemy tak wiele operacji jednocześnie, wszystkie te operacje można zapisać w pojedynczej macierzy 4x4, co — przynajmniej na mnie — robi spore wrażenie.

# Więcej do odkrycia

W bibliotece emulującej obliczenia DirectX zaimplementowałem także kwaterniony, pomimo że nie wykorzystałem ich w aplikacji Polyhedra. Kwaterniony są ciekawym sposobem łączenia zbioru rotacji 3D w pojedynczą rotację. Pozwalają także na płynne przechodzenie pomiędzy różnymi orientacjami przestrzennymi. Kwaterniony odkrył sir William Rowan Hamilton (z Dublina, mojego rodzinnego miasta), kiedy szedł sobie ulicą. By nie zapomnieć równania, natychmiast nagryzmolił je na ścianie mostu Broome Bridge (szczegóły dostępne pod adresem [www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quaternions.html](http://www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quaternions.html)). Chciałbym móc powiedzieć, że wszystkie graffiti są równie odkrywcze, ale chyba nikt mi nie uwierzy…

Kwaterniony pozwalają także uniknąć problemu określanego jako „gimbal lock”. Problem ten dotyczył między innymi nawigacji żyroskopowej w czasie misji księżycowych programu Apollo i wspomniano o nim nawet w filmie Apollo 13.

Nie opisałem tu wszystkich możliwości biblioteki — są tu jeszcze na przykład iloczyny skalarne i iloczyny wektorowe — mam jednak nadzieję, że to, co opisałem, dobrze pokazało możliwości zapewniane przez emulację obliczeń DirectX. Bibliotekę napisałem na potrzeby realizacji transformacji na hostowanym serwerze ASP.NET, na którym nie mogłem zainstalować DirectX. Z pewnością można znaleźć inne zastosowania tej biblioteki w środowiskach podobnego typu.

Mam nadzieję, że aplikacja Polyhedron jest dobrym przykładem na to, że Silverlight jest użytecznym i rozszerzalnym narzędziem do tworzenia interfejsu użytkownika aplikacji internetowych, umożliwiającym przykucie uwagi użytkownika i szybkie przekazanie informacji w sposób sprzyjający ich zapamiętaniu. Silverlight pozwala łatwo i kreatywnie wykorzystać dotychczasowe doświadczenia, zdobyte podczas tworzenia aplikacji .NET, i — w odróżnieniu od JavaScript — nie zmusza programisty do obsesyjnego dbania o minimalizację liczby linii kodu wykonywanego po stronie klienta.

**Declan Brennan** urodził się na tyle dawno, by pamiętać pierwszy mikroprocesor, ale nadal nie przyzwyczaił się do posiadania własnego dżina w butelce. Nie potrafi uwierzyć w swoje szczęście życia w świecie ograniczonym nie przez technologię, a jedynie przez wyobraźnię. Więcej o Declanie można przeczytać w witrynie [declan.brennan.name](http://declan.brennan.name).