Animace ve WPF

Bakalářská práce

Filip Gažák

Vedoucí bakalářské práce: Ing. Václav Novák, CSc.

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Rok 2010

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval/-a samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 23.4.2010

Anotace

Bakalářská práce „Animace ve Windows Presentation Foundation" se zabývá tvorbou grafických prvků a animací v modelu Windows Presentation Foundation. V teoretické části je popsána architektura a hlavní výhody modelu. V praktické části je pomocí příkladů a audiovizuální prezentace vysvětlena tvorba animací.

Abstract

The goal of the submitted thesis „Animation in Windows Presentation Foundation" is dealing with creation of graphics element and animation in model Windows Presentation Foundation. In the theoretic part is described architecture and the main advantages of model. In the practical part through examples and audiovisual presentation is explained creation of animation.

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Václavu Novákovi, CSc. za poskytnutí cenných rad a odborné vedení během mé práce.

Také bych rád poděkoval své rodině za pomoc a podporu se zpracováním práce.

Obsah

1 ÚVOD ... 8

2 WINDOWS PRESENTATION FOUNDATION ... 9

2.1 HISTORIE PLATFORMY .NET FRAMEWORK ... 9

2.1.1 .NET Framework 1.0 (2002) ... 9

2.1.2 .NET Framework 1.1 (2003) ... 9

2.1.3 .NET Framework 2.0 (2005) ... 9

2.1.4 .NET Framework 3.0 (2006) ... 10

2.1.5 .NET Framework 3.5 ... 10

2.1.6 .NET Framework 3.5 SP1 ... 11

2.2 MODEL WPF ... 12

2.2.1 Architektura ... 12

2.2.2 Třídy ... 13

2.3 XAML ... 15

3 ANIMACE VE WPF ... 16

3.1 TRANSFORMACE ... 17

3.1.1 Matematický zápis ... 18

3.1.2 RotateTransform ... 21

3.1.3 ScaleTransform ... 22

3.1.4 TranslateTransform ... 23

3.1.5 SkewTransform ... 24

3.1.6 Render a Layout Transform ... 25

3.1.7 TransformGroup ... 26

3.2 STYLY ... 27

3.2.1 Použití ... 27

3.2.2 Style ... 28

3.3 ŠABLONY ... 31

3.3.1 ControlTemplate ... 31

3.3.2 DataTemplate ... 32

3.4 ANIMACE ... 34

3.4.1 Systém.Windows.Media.Animation ... 35

3.4.2 Spouště ... 37

3.4.3 Storyboard ... 40

3.4.4 Základní animace *41*
3.4.5 Animace s klíčovými snímky *47*
3.4.6 Animace typu Path *53*
3.5 C# A XAML 54
3.5.1 Vazba C# a XAML *57*
3.6 NÁSTROJE 57

4 ZÁVĚR 59

5 PŘEHLED POUŽITÉ LITERATURY 60

6 SEZNAM OBRÁZKŮ 62

7 SEZNAM TABULEK 63

8 SEZNAM GRAFŮ 64

1 Úvod

Informační technologie a výpočetní technika jsou nedílnou součástí dnešního světa. Setkáváme se s nimi velice často a je velmi obtížné si představit, jak by svět vypadal bez počítačů nebo internetu. Toto odvětví prochází velmi rychlým vývojem, jak v oblasti hardwarové, tak i v softwarové.

V oblasti softwarového vývoje, zejména z hlediska uživatelského rozhraní, se vývojáři stále více snaží, aby jejich aplikace nabídly uživatelům něco nového a originálního. Grafická podoba aplikace se stala důležitým faktorem, který často ovlivňuje, jak bude aplikace úspěšná. Jednou z technologií, která má za cíl vytváření poutavého a efektivního uživatelského rozhraní je Windows Presentation Foundation (dále WPF). Tento nástupce technologie Windows Forms nabízí možnosti vektorové grafiky a animací, díky kterým aplikace nemusejí mít pouze strohý „formulářový“ vzhled.

Začlenění animací do technologie je bráno jako jedno ze stěžejních nástrojů pro tvorbu grafického rozhraní. Abych se mohl touto problematikou zabývat podrobněji, vybral jsem si jej jako téma své bakalářské práce.

Tato publikace má za cíl seznámit čtenáře s modelem WPF uvnitř .NET Framework 3.5 a popsat tvorbu animací pomocí názorných příkladů a vizuální prezentace.

2 Windows Presentation Foundation

Prezentační rozhraní WPF je koncipované jako jednotný framework pro tvorbu nové generace aplikací. Představení modelu prakticky odstartovalo nový pohled na vývoj aplikací. Stále více se začíná prosazovat koncept bohatšího uživatelského rozhraní a hlavně jeho grafická podoba.

Model WPF byl poprvé integrován do prostředí .NET Framework 3.0 a je standardně součástí Windows Vista, Windows 7 a Windows Server 2008. Po nainstalování je dostupný na Windows XP SP2 nebo Windows Server 2003.

2.1 Historie platformy .NET Framework

2.1.1 .NET Framework 1.0 (2002)

Verze 1.0 byla spojená s vývojovým prostředím Visual Studio .NET 2002, které bylo následovníkem populárního vývojového prostředí Visual Studio 6.0. Poprvé se objevují jazyky C# 1.0 a Visual Basic .NET. Na scénu přichází i platforma ASP.NET 1.0 pro vývoj webových aplikací. [6]

2.1.2 .NET Framework 1.1 (2003)

Verze spojená s vývojovým prostředím Visual Studio .NET 2003. Byly opraveny chyby z předchozí verze a doplněné API prostředí.

2.1.3 .NET Framework 2.0 (2005)

Verze spojená s vývojovým prostředím Visual Studio 2005. Tato platforma se v porovnání s předchozí rozrostla o 2372 nových tříd (počet 4482 se rozrostl na 6854), zároveň přibylo velké množství tříd pro SQL Server 2005. Rozrostla se i paleta dostupných ovládacích prvků pro WinForms a ASP.NET aplikace. S novou verzí frameworku se objevují nové programovací jazyky C# 2.0 a Visual Basic 2005. Pro vývoj webových aplikací se objevuje platforma ASP.NET ve verzi 2.0. [6]

2.1.4 .NET Framework 3.0 (2006)

Verze spojená s vývojovým prostředím Visual Studio 2008. Původní návrh operačního systému Windows Vista (v té době pod označením Longhorn) obsahoval integrovaný grafický systém Avalon. Po dalším vývoji došlo k oddělení tohoto bloku od operačního systému a byl začleněn do balíku s názvem WinFX. Poslední změnou bylo přejmenování balíku WinFX na .NET Framework 3.0. Jádrem nové verze tedy zůstává jádro z verze 2.0 (viz Obrázek č. 1).

.NET Framework 3.0 obsahuje nové technologie:

- **Windows Presentation Foundation** (WPF)
- **Windows Communication Foundation** (WCF) je technologie pro vytváření aplikací využívající architekturu orientovanou na služby. WCF obsahuje unifikovaný programovací model pro přenos zpráv, založený na skriptovacím jazyku WSDL (Web Service Definition Language). Na úrovni vrstvy WCF se vytvářejí informace o službách a operacích, které technologie vykonává. Aplikace a služby jsou propojené pomocí propojovacích bodů „Service Endpoint", které obsahují adresu klienta a informace o způsobu komunikace mezi ním a službou.
- **Windows Workflow Foundation** (WF) obsahuje programový model, engine a nástroje pro vytváření diagramů procesů (Workflow) a řízení procesů v aplikacích.
- **CardSpace** je technologie pro správu digitálních identit. Zjednodušuje a vylepšuje bezpečnost autentifikace v prostředí internetových aplikací. Identity představují pomyslné kartičky, které obsahují digitálně podepsaný XML dokument. [10]

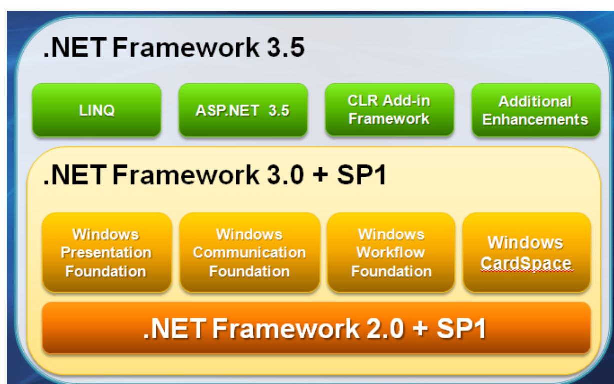
2.1.5 .NET Framework 3.5

Verze 3.5 navazuje na verzi 3.0. Opravuje bezpečnostní chyby a zároveň přináší nové funkce. Obsahuje rozšíření programovacích jazyků C# 3.0

a Visual Basic 9. Dále integrovanou podporu technologie ASP.NET, AJAX nebo databázový jazyk LINQ (Language Integrated Query). Rozšíření se týká také knihovny tříd a technologických vrstev WPF, WCF, WF a Windows CardSpace. [6]

2.1.6 .NET Framework 3.5 SP1

První Servis Pack pro framework verze 3.5 přinesl opět několik vylepšení, zejména slibuje vyšší výkon až o 40 % ve WPF aplikacích. Balíček obsahuje také novinky v oblasti datové platformy jako ADO.NET Entity Framework nebo ADO.NET Data Services. Zároveň byly přidány doplňková nástroje a komponenty pro jazyky C#, Visual Basic a Visual C++.



Obrázek 1: .NET Framework 3.5

2.2 Model WPF

WPF je zaměřené na uživatelsky „bohaté“ aplikace, takže je možné opustit grafickou podobu klasických formulářů a naplno využít možností vektorové grafiky. Vektorová grafika s využitím hardwarové akcelerace umožňuje bezztrátovou změnu velikosti elementů. Grafické operace jsou vykreslovány vrstvou DirectX a následně pomocí grafické karty, jestliže je tento hardware dostatečně výkonný. Tímto je plně využit potenciál moderních grafických karet a procesor není těmito operacemi zatěžován.

Oproti předchozí technologii přináší novinku v zobrazení aplikací vůči rozlišení. Zatímco klasické Win32 aplikace definují rozměry v jednotkách pixel, WPF aplikace používají jednotky rozměru 1/96 palce, které jsou určeny v „device independent units“ (*DIUs*). Po zvětšení rozlišení si objekty zachovají stejnou velikost. Již nedochází ke zmenšení okna aplikace a s tím spojené horší čitelnosti. [10]

Model přináší začlenění nástrojů pro práci s multimedii, zvukem a samozřejmě i s grafickými operacemi. Použití stylů nebo šablon dává designérům možnosti měnit vzhled ovládacích prvků a objektů, případně je možné z geometrických útvarů vytvářet prvky vlastní. Velmi přínosné jsou techniky, které dovolují objekty transformovat nebo animovat.

2.2.1 Architektura

Architektura WPF je založena třech základních modelech:

Aplikační model – pomocí deklarativního programování je možné jednoduše integrovat multimedia, pomocí databindingu je provázat s údaji a pomocí stylů nastavovat statický a dynamický vzhled aplikací.

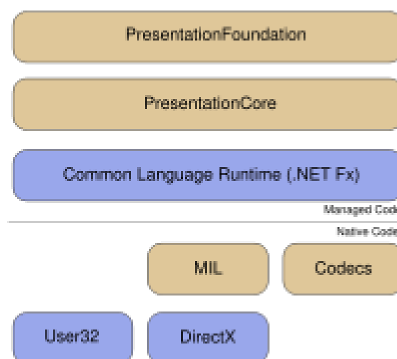
Grafický model – zahrnuje kromě vektorové grafiky i 2D a 3D grafiku.

Model dokumentů – rozšiřuje možnosti správy dokumentů v rámci aplikací. Textové informace je možné variabilně rozložit pomocí pokročilých

typografických metod. V tomto modelu můžeme v rámci aplikací řídit přístupová práva k dokumentům. [6]

K těmto modelům lze zařadit i blok základních služeb, tedy služeb na nejnižší úrovni. Obsahují modul jazyka XAML, komponenty pro vstup údajů a zpracování událostí, které v aplikacích nastanou a podsystém pro správu vlastností prvků.

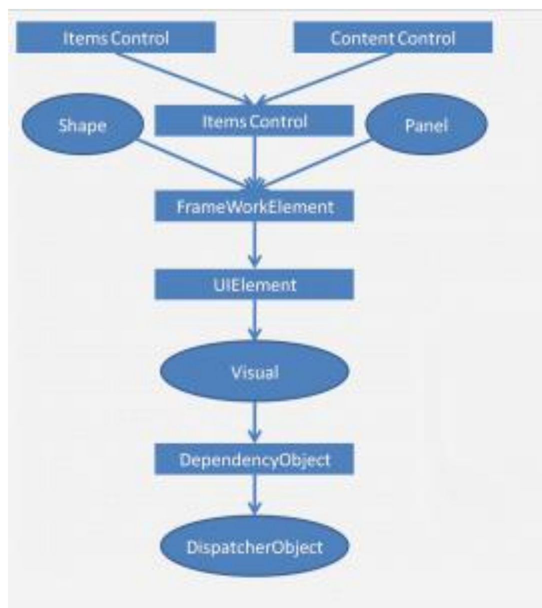
Hlavní strukturu WPF tvoří komponenty: *PresentationCore* – provádí základní služby WPF, *PresentationFramework* – zahrnuje typy jako kontrolky nebo styly, *WindowsBase* – obsahuje základní typy `DependencyObject`, *Milcore* – podporuje převod do DirectX, *WindowsCodecs* – umožňuje práci s grafikou. [10]



Obrázek 2: WPF architektura

2.2.2 Třídy

Knihovna tříd modelu WPF je velmi rozsáhlá. Následující diagram zobrazuje rozložení základních tříd, z nichž ostatní třídy vycházejí. Všechny třídy jsou součástí jmenného prostoru (*namespace*) System.Windows. [7]



Obrázek 3: Schéma tříd ve WPF

- `DispatcherObject` – abstraktní základní třída pro všechny objekty v rámci modelu. Je navržen jako STA model, který kontroluje správnou komunikaci mezi jednotlivými vlákny.
- `DependencyObject` – třída pro WPF objekty, které podporují vlastnost *dependency* (závislost). `DependencyObject` definuje metody *GetValue* a *SetValue*, které jsou důležitými prvky těchto vlastností.
- `Visual` – základní třída pro všechny objekty, které mají vlastní vizuální reprezentaci.
- `UIElement` – třída určená pro podporu událostí.
- `FrameworkElement` – třída, která přidává podporu pro styly, databinding, zdroje a základní mechanismy pro kontroly.
- `Control` – třída uchovávající kolekci kontrol jako *Button* nebo *ListBox*. Zároveň přidává vlastnosti z *FrameworkElement* – *Foreground*, *Background* nebo *Fontsize*.

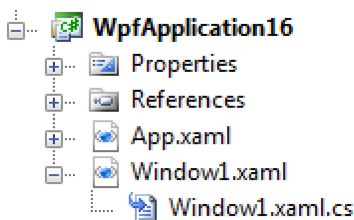
- `Shape` – základní třída, z níž se odvozují základní objekty jako obdélník, polygon, cesta a linka.
- `Panel` – z této třídy se odvozují všechny prvky sloužící pro rozmístění objektů. Patří sem *Canvas*, *Grid* a *StackPanel*. [7]

2.3 XAML

Jednou z novinek, kterou jsem ještě neuvedl, je naprosto nová technika tvorby aplikací. WPF přináší nový jazyk XAML (eXtensible Application Markup Language), neboli rozšiřitelný značkovací jazyk pro aplikace. Je založen na XML a přebírá jeho výhody jako např. přehlednost kódu.

Hlavním důvodem přidání jazyka XAML byla vidina oddělení programové části kódu od uživatelského rozhraní. Pomocí XAMLu se deklarativně nadefinuje vzhled aplikace a objektů, funkční logika se vytvoří standardně pomocí procedurálního jazyka. Tento postup dává nové možnosti programátorům, ale hlavně návrhářům aplikací. Již není nutné, aby se programátor zabýval vzhledem uživatelského rozhraní. Tato věc je přenechána grafikovi. Jazyk nemusí sloužit jen k definici grafického vzhledu, ale i k definici aplikace jako celku nebo k nastavení systémových zdrojů. Samozřejmě je i nadále možné vytvářet aplikace bez použití jazyka XAML.

WPF aplikace se skládá ze dvou souborů: první s příponou ".cs", ve kterém je programová část aplikace a druhý s příponou ".xaml", ve kterém je definováno uživatelské rozhraní.



Obrázek 4: Struktura aplikace

3 Animace ve WPF

V následující praktické části se čtenář dozví, jak se ve WPF pracuje s grafickými prvky a vytvářejí animací. Celá sekce je rozdělena do třech kapitol. První kapitola pojednává o transformacích, druhá kapitola o stylech a šablonách a třetí již o samotných animacích. Transformace a styly jsou pro animace velmi důležité, v některých případech dokonce nezbytné. Z tohoto důvodu jsem je zařadil do první a druhé kapitoly. V každé kapitole je problematika vysvětlena pomocí příkladů v jazyce XAML. Jednotlivé příklady jsou ještě okomentovány a zároveň jsou zdůrazněny důležité části kódu, které by měl čtenář znát. Zdrojové kódy příkladů, které byly v této publikaci použity se nacházejí na přiloženém CD.

První kapitola je věnována transformacím. Nejprve je čtenář seznámen s matematickým zápisem těchto operací a jejich zápisem pomocí jazyka XAML. Na názorných příkladech jsou dále probrány transformace *Translate*, *Rotate*, *Skew*, *Scale* a *TransformGroup*. Dále je vysvětlen rozdíl mezi vlastnostmi *Render* a *Layout*.

Ve druhé kapitole je probrána tématika stylů a šablon. V sekci stylů je popsán postup jak objekt resp. objekty „nastylovat" a jaká úskalí tyto operace představují. Sekce šablon je věnována tvorbě ovládacích prvků z různých objektů a dále je vysvětlen princip datové šablony.

Poslední třetí kapitola je již věnována tvorbě animací. V úvodu je vysvětlen pojem „animace" a jeho postavení v rámci WPF. Dále je probrána třída, do které animace v modelu patří. Zároveň jsou určeny datové typy vhodné pro animování. V kapitole je dále vysvětleno téma scénářů a spouští. Samotné téma animací je rozděleno na tři podkapitoly – základní, s klíčovými snímky a animace typu *path*. Každý druh je představen na několika příkladech s odlišnými datovými typy. Na závěr jsem uvedl dva příklady v jazyce XAML a C#.

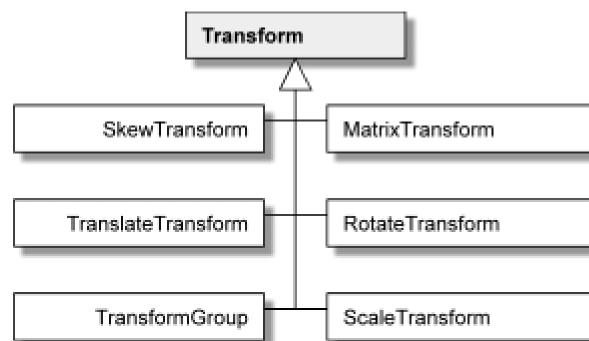
3.1 Transformace

Transformace jsou obzvlášť užitečné při animacích. Pokud chceme posunout nějaký objekt (např. `Rectangle`) z jednoho místa na druhé, bude jednodušší změnit pouze translační faktor, který nadále aplikujeme na celý obraz, než měnit všechny souřadnice jedním směrem. Ve WPF je možné transformovat každý objekt, jehož vlastnosti transformaci podporují jako např. šířka, výška, vzdálenost a úhel zkosení.

Transformaci lze aplikovat na libovolný objekt odvozený z `UIElement`. Tento objekt obsahuje vlastnost s názvem `RenderTransform`, do níž je potřeba vložit vlastnost typu `Transform`. Objekt `FrameworkElement` obsahuje svou vlastní definici vlastnosti typu `Transform`. Tato vlastnost se nazývá `LayoutTransoform`. Pokud chceme použít transformaci, je nutné vybrat právě jednu vlastnost, která požadovaný výsledek zajistí. Mezi vlastnostmi `RenderTransform` a `LayoutTransform` je znatelný rozdíl (viz kapitola *3.1.6*). [1]

V modelu WPF třída `Transform` obsahuje celkem šest typů transformace:

- Posunutí
- Otočení
- Zkosení
- Změna měřítka
- Maticová transformace
- Složená transformace



Obrázek 5: Transformace

3.1.1 Matematický zápis

Prakticky v každém grafickém programování se transformace a operace mohou vyjádřit pomocí matematického zápisu. Tímto způsobem si můžeme zjednodušit složité výpočty transformačních hodnot nebo konečných souřadnic objektů. Při matematickém zápisu transformací je běžné vyjadřovat je ve formě matic. Tento způsob má své zákonitosti a jasně daná pravidla, která je nutné dodržovat.

Nejjednodušší maticí transformace je matice 2×2, tedy matice obsahující dva sloupce a dva řádky. Je nutné zdůraznit, že tato matice nedefinuje posun. Tato matice obsahuje parametry a_x, a_y, které představují změnu velikosti. Parametry b_x a b_y představují rotaci.

$$\begin{pmatrix} \mathrm{a_x} & b_y \\ b_x & a_y \end{pmatrix}$$

Výpočet transformace lze vyjádřit jako násobení matic 1×2, kdy zadaný bod násobíme maticí transformace. Výsledkem musí opět být matice 1×2. V tomto násobení je předpoklad, že počet sloupců první matice musí být shodný s počtem řádků druhé matice. Hodnoty [x, y] představují původní bod před transformací. [1]

$$|x \quad y| \times \begin{pmatrix} a_x & b_y \\ b_x & a_y \end{pmatrix} = |x' \quad y'|$$

Ekvivalentní zápis pomocí rovnic:

$$x' = a_x.x + b_x.y$$
$$y' = b_y.x + a_y.y$$

Jak již bylo uvedeno, matice transformace o rozměrech 2×2 neobsahuje body posunu. Aby tato operace byla možná, je nutné rozšířit matice

o další řádek resp. sloupec. Z matice 1×2 vznikne 1×3 a z matice 2×2 vznikne 3×3. Aby rozšíření nemělo negativní vliv na funkčnost, musí být rovno číslu 1. Původní matice bude obsahovat v dolním řádku parametry d_x a d_y. Matice obsahuje hodnoty a_x, a_y, b_x, b_y, d_x, d_y, které definují konkrétní transformaci. [1]

$$\begin{pmatrix} a_x & b_y & 0 \\ b_x & a_y & 0 \\ d_x & d_y & 1 \end{pmatrix}$$

Výpočet transformace pomocí maticového násobení a rovnic vypadá následovně:

$$|x \quad y \quad 1| \times \begin{pmatrix} a_x & b_y & 0 \\ b_x & a_y & 0 \\ d_x & d_y & 1 \end{pmatrix} = |x' \quad y' \quad 1|$$

$$x' = a_x.x + d_x.y + d_x$$
$$y' = b_y.x + a_y.y + d_y$$

Matice, která obsahuje hodnoty 1 pouze na hlavní diagonále, se nazývá jednotková. Tato matice nemá žádný vliv na transformaci, jelikož nemá určené žádné hodnoty.

Transformační matice se uchovává ve struktuře s názvem *Matrix*. Tato struktura obsahuje šest vlastností, které odpovídají dostupné transformaci. Parametry M11 a M22 jsou standardně nastaveny na hodnotu 1, zbývající na hodnotu 0. Hodnoty ve třetím sloupci nelze měnit, jsou tedy vždy nastaveny na 0, 0, 1.

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

Následující příklady ukazují maticový zápis s použitím zadaných hodnot.

- Posun – posouváme o 50px ve směru osy X a o 20px směru osy Y. Zápis v maticovém provedení má hodnoty 50 a 20 na pozicích *OffsetX* a *OffsetY*, jelikož se jedná jen o posun. Ostatní hodnoty v matici zůstávají původní.

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 50 & 20 & 1 \end{pmatrix}$$

- Změna měřítka – změna šířky objektu 2x a výšky 3x. Hodnoty jsou zadány pouze na souřadnicích *M11* a *M22*. Ostatní (pokud nenastavujeme ještě posun) zůstavají standardní.

$$M = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Rotace - pokud otáčíme jen o úhel, matice osahuje 4 parametry pro *sin(α)* a *cos(β)*. Při rotaci a zároveň posunutí jsou vyplněny i hodnoty na pozicích *OffsetX* a *OffsetY*.

$$M = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Pokud rotujeme o 60 stupňů, nejprve zjistíme jednotlivé hodnoty, které poté zapíšeme do matice. Sinus 60 stupňů je přibližně 0,866 a kosinus je 0,5.

$$M = \begin{pmatrix} 0{,}5 & 0{,}866 & 0 \\ -0{,}866 & 0{,}5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

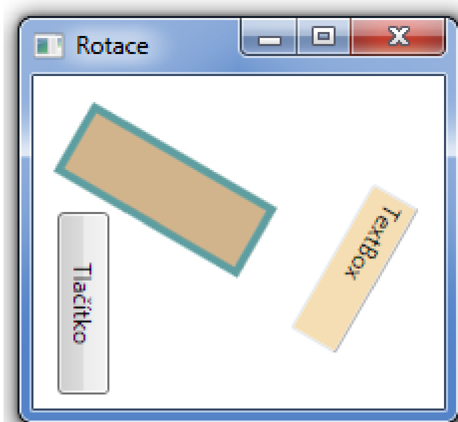
3.1.2 RotateTransform

Transformace `RotateTransform` je velmi užitečný nástroj jak otáčet nejrůznější objekty. Zvláště se tato vlastnost hojně využívá u animací. Hlavní atribut je `Angle`, který určuje úhel rotace. Atributy `CenterX` a `CenterY` určují posun otáčeného objektu.

```
<RotateTransform Angle="" CenterX="" CenterY="">
</RotateTransform>
```

Následující příklad ukazuje použití transformace rotace na dva objekty – obdélník a textové pole. `TextBox` rotuje o 120 stupňů a zároveň se posouvá o 80px vlevo a 10px dolů.

```
<!--Rotace textového boxu-->
<TextBoxText="TextBox">
  <TextBox.RenderTransform>
    <RotateTransform Angle="120" CenterX="80" CenterY="10">
    </RotateTransform>
  </TextBox.RenderTransform>
</TextBox>
<!--Rotace obdélníka-->
<Rectangle>
  <Rectangle.RenderTransform>
    <RotateTransform Angle="30"></RotateTransform>
  </Rectangle.RenderTransform></Rectangle>
```



Obrázek 6: RotateTransform

Objekt se standardně otáčí kolem svého levého horního rohu. Pokud potřebujeme, aby se otáčel kolem svého středu, máme na výběr ze dvou možností.

1/ nastavit hodnoty otáčení přímo ve vlastnosti `CenterX` a `CenterY`, kde zadáme hodnoty jako polovinu šířky respektive výšky. Např. obdélník z přecházejícího příkladu:

```
<RotateTransform Angle="30" CenterX="50"CenterY="20">
</RotateTransform>
```

2/ nastavit v objektu vlastnost `RenderTransformOrigin`.

```
<Rectangle RenderTransformOrigin="0.5 0.5">
</Rectangle>
```

Druhá možnost je samozřejmě jednodušší, protože nemusíme počítat hodnoty potřebné pro správné otočení. Což může být při velkém počtu objektů celkem náročné. Zároveň je nutné si uvědomit co hodnoty u `RenderTransformOrigin` znamenají: hodnota [0 0] otočí standardně kolem horního levého roku, hodnota [1 0] otočí kolem pravého horního rohu, hodnota [1 1] otočí kolem pravého dolního rohu a hodnota [0 1] otočí kolem levého dolního rohu.

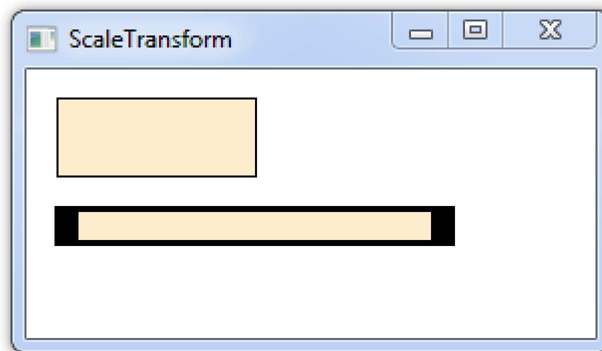
3.1.3 ScaleTransform

Touto transformací je možné zmenšovat či zvětšovat velikosti objektů. Má čtyři atributy, `ScaleX`, `ScaleY` (jenž mají výchozí hodnotu 1), `CenterX` a `CenterY` (mají výchozí hodnotu 0).

```
<ScaleTransform ScaleX="2" ScaleY="0.5" CenterX="" CenterY="">
</ScaleTransform>
```

Následující příklad demonstruje použití `ScaleTransform` na obdélník, konkrétně zvětšení šířky 2x a zmenšení výšky 0,5x.

```
<Rectangle Height="40" Width="100" Fill="BlanchedAlmond">
    <Rectangle.RenderTransform>
       <ScaleTransform ScaleX="2" ScaleY="0.5">
       </ScaleTransform>
    </Rectangle.RenderTransform>
</Rectangle>
```



Obrázek 7: ScaleTransform

I u této vlastnosti mění objekt *Rectangle* svou velikost podle levého horního rohu původního objektu. Pro změnu okolo středu slouží opět vlastnosti *CenterX*, *CenterY* nebo *RenderTransformOrigin*.

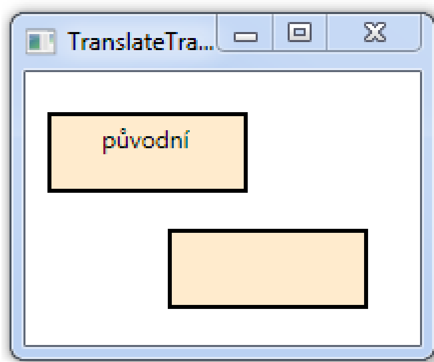
3.1.4 TranslateTransform

Třída `TranslateTransform` je určená pro posun objektů v panelech, které nemají absolutní pozicování. Obsahuje dvě vlastnosti *X* a *Y*. Atribut *X* určuje posunutí ve směru horizontální osy, atribut *Y* určuje posunutí ve směru osy vertikální.

```
<TranslateTransform X="" Y="">
</TranslateTransform>
```

V příkladu se posouvá objekt o 60px ve směru obou os.

```
<Rectangle Height="40" Width="100" Fill="BlanchedAlmond">
  <Rectangle.RenderTransform>
    <TranslateTransform X="60" Y="60">
    </TranslateTransform>
  </Rectangle.RenderTransform>
</Rectangle>
```



Obrázek 8: TranslateTransform

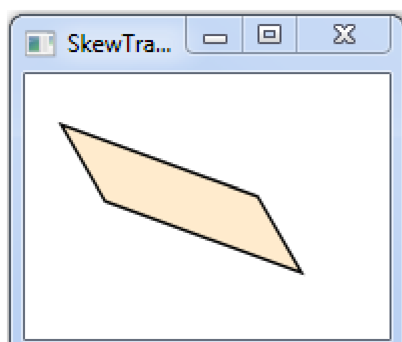
3.1.5 SkewTransform

Tato transformace má čtyři vlastnosti, *AngleX*, *AngleY*, *CenterX* a *CenterY,* jejichž výchozí hodnota se rovná 0. Jak již název napovídá, vlastnosti *AngleX* a *AngleY* jsou úhly, které v intervalu -90 až 90 stupňů dávají konečné výsledky. Úhly mimo tento interval pouze duplikují výsledky vzešlé z hodnot uvnitř intervalu. [1]

```
<SkewTransform AngleX="" AngleY="" CenterX="" CenterY="">
</SkewTransform>
```

V následujícím příkladě je ukázáno jak lze pomocí `ScaleTransform` jednoduše zkosit daný objekt. V tomto případě je zkosení ve směru osy X o 30 stupňů a ve směru osy Y o 20 stupňů.

```
<Rectangle Height="40" Width="100" Fill="BlanchedAlmond">
   <Rectangle.RenderTransform>
       <SkewTransform AngleX="30" AngleY="20"
                      CenterX="50" CenterY="20">
       </SkewTransform>
   </Rectangle.RenderTransform>
</Rectangle>
```



Obrázek 9: SkewTransform

3.1.6 Render a Layout Transform

Systém grafiky a vzhledu ve WPF zpracovává transformace `RenderTransform` a `LayoutTransform` výrazně jinak. Pro transformaci `RenderTransform` systém přebírá obrázek vykreslený metodou `OnRender`, která aplikuje transformaci a umístí obrázek na obrazovku. Jestliže obrázek náhodně překrývá některý další prvek v programu, nebo je pod něčím skryt, nehraje roli. Element je oříznut podle hranice aplikačního okna, ale jinak může být kdekoliv. [1]

```
<!-- RenderTransform -->
<UniformGrid Rows="3" Columns="3">
  <Button Content="Tlačítko1" />
  <Button Content="Tlačítko3">
     <Button.RenderTransform>
        <RotateTransform Angle="30" />
     </Button.RenderTransform>
  </Button>
  <Button Content="Tlačítko2"></Button>
</UniformGrid>

<!-- LayoutTransform -->
<UniformGrid Rows="3" Columns="3">
  <Button Content="Tlačítko1"></Button>
  <Button Content="Tlačítko3">
     <Button.LayoutTransform>
        <RotateTransform Angle="30" />
     </Button.LayoutTransform>
  </Button><Button Content="Tlačítko2"></Button>
</UniformGrid>
```

Obrázek 11: RenderTransform Obrázek 10: LayoutTransform

3.1.7 TransformGroup

Hlavní funkcí elementu `TransformGroup` je možnost přiřadit k objektu více než jednu transformaci. `TransformGroup` neobsahuje žádné atributy. Jednotlivé transformace se jen nadefinují uvnitř elementu a model je následně aplikuje. Zároveň zde záleží na pořadí, model aplikuje transformaci od první po poslední. Při použití rotace a poté zkosení je výsledek jiný, než když je aplikujeme v opačném pořadí.

```
<Button.LayoutTransform>
    <TransformGroup>
        <SkewTransform AngleY="35" />
        <ScaleTransform ScaleX="1.5" ScaleY="2" />
        <RotateTransform Angle="33" />
        <TranslateTransform X="23" Y="40" />
    </TransformGroup
</Button.LayoutTransform>
```

3.2 Styly

Jednou z mnoha výhod modelu Windows Presentation Foundation je i jednoduchá možnost, jak obohatit naše aplikace pomocí grafiky a grafických objektů. Grafická stránka aplikace je dnes velmi důležitá a často rozhoduje o její úspěšnosti.

Pojem „stylování“ je velmi běžný, ve skutečnosti jej používáme prakticky denně. Například, když píšeme dopis v textové aplikaci, používáme mnoho vlastností nastavení textu, jako je druh písma, barva nebo velikost. Všechny tyto vlastnosti můžeme označit jako styly.

Ovšem nejvíce se s použitím stylů hovoří v souvislosti s tvorbou webových stránek. Než se naplno ujala technologie CSS, způsob zobrazení webových stránek se zapisoval společně s kódem, který definoval i obsah stránky. Kaskádové styly toto pojetí zcela změnily, vzhled se zapisuje zvlášť a již není nutné definovat vlastnosti pro každý objekt samostatně. Ve WPF se styly používají velmi podobně.

3.2.1 Použití

Používat styly je velmi výhodné pokud máme v aplikaci více objektů se stejným vzhledem. Následující příklad ukazuje XAML zápis tlačítka bez použití stylů.

```
<Button Width="94" Height="29" Content="Tlačítko">
  <Button.Effect>
      <DropShadowEffect BlurRadius="7" Color="#FF53B62A"
               RenderingBias="Quality" ShadowDepth="7"/>
  </Button.Effect>
  <Button.Background>
    <LinearGradientBrush EndPoint="0,1" StartPoint="0,0">
             <GradientStop Color="#FFF3F3F3" Offset="0"/>
           <GradientStop Color="#FFEBEBEB" Offset="0.5"/>
           <GradientStop Color="#FFDDDDDD" Offset="0.5"/>
             <GradientStop Color="#FF54EE11" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Z příkladu je patrné, kolik vlastností se prvku musí nastavit, aby vypadal jako na obrázku. Vše bude v pořádku, pokud použijeme v aplikaci takové tlačítko jen jedno. Problém nastane, jakmile budeme chtít použít těchto prvků více a následně je budeme editovat. V tomto případě se bude kód několikrát zbytečně opakovat, čemuž se samozřejmě snažíme vyhnout. Právě takové situace řeší použití stylů. Pro množství objektů, které mají stejné vlastnosti, vytvoříme styl, který následně na každý objekt aplikujeme.

3.2.2 Style

Základní XAML zápis elementu `Style` má tuto podobu:

```
<Style x:Name="styl" TargetType="">
    <Setter Property="" Value="" />
</Style>
```

Atribut `x:Name` obsahuje jméno stylu, atribut `TargetType` určuje, pro který objekt je styl určen. Subelement `Setter` obsahuje parametr `Property`, který definuje vlastnost objektu a parametr `Value` nastavuje její hodnotu.

Styly se nejčastěji definují v sekci zdrojů `Resources`, aby bylo možné následné sdílení ve více elementech a prvcích. Podobně jako u CSS se na styly odkazujeme pomocí klíčů. Přiřazení stylu je možné i pomocí atributu `TargetType`.

Následující příklad demonstruje použití přiřazení stylu pomocí klíčového slova `TargetType` a klíče. Styl bude aplikován na příklad z kapitoly 3.2.1.

```
<Canvas>
  <Canvas.Resources>
     <Style TargetType="Button">
        <Setter Property="FontSize" Value="14" />
        <Setter Property="Height" Value="24"/>
        <Setter Property="Width" Value="94"/>
        <Setter Property="Effect">
          <Setter.Value>
            <DropShadowEffect BlurRadius="7" Color="#FF53B62A"
                 RenderingBias="Quality" ShadowDepth="7"/>
          </Setter.Value>
        </Setter>
```

```
        <Setter Property="Background">
          <Setter.Value>
        <LinearGradientBrush EndPoint="0,1"StartPoint="0,0">
                  <GradientStop Color="#FFF3F3F3"
                                Offset="0"/>
                  <GradientStop Color="#FFEBEBEB"
                                Offset="0.5"/>
                  <GradientStop Color="#FFDDDDDD"
                                Offset="0.5"/>
                  <GradientStop Color="#FF54EE11"
                                Offset="1"/>
                  </LinearGradientBrush>
          </Setter.Value>
        </Setter>
      </Style>
    </Canvas.Resources>
  <Button Content="Tlačítko" />
  </Canvas>
```

Element `Setter` definuje vlastnosti: velikost písma, výšku a šířku objektu. Pokud použijeme na objektu efekty nebo složitější popis barvy pozadí (např. s odstínem) v elementu `Setter` se jako vlastnost zvolí `Effect`, respektive `Background` a dále se rozšíří o element `Setter.Value`. V tomto příkladě je styl automaticky přiřazen ke všem objektům `Button`, neboť jsme použili jen atribut `TargetType`. Toto řešení je dostačující do doby, než budeme chtít více těchto objektů, ale s jiným vzhledem.

Řešení je možné, pokud místo atributu `TargetType` použijeme klíč `x:Key`. Tento klíč je podřízen elementu `Style` a nastavuje jeho jméno.

```
<Canvas>
   <Canvas.Resources>
       <Style x:Key="mujStyl">
            <Setter Property="" Value="" />
       </Style>
   </Canvas.Resources>
  <Button Style="{StaticResource mujStyl}"/>
</Canvas>
```

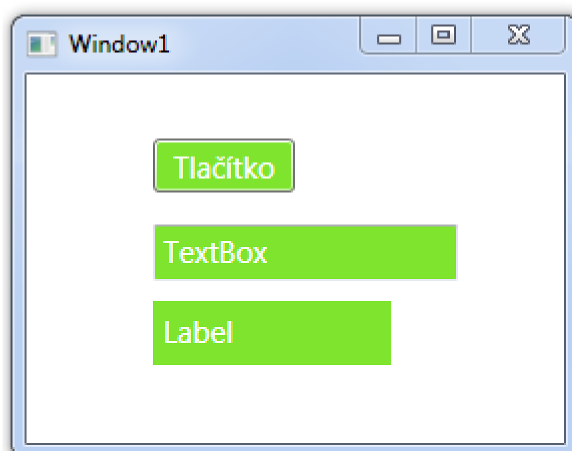
Pomocí klíčového slova `x:Key` jsme styl pojmenovali „*mujStyl*“, pomocí atributu `StaticResource` je následně přiřazen prvku `Button`. Pojmenování stylu nám dává větší možnosti jeho využití. Nejenom, že je ho možné nastavit u stejných, ale i u zcela odlišných prvků.

Následující příklad ukazuje použití jedno stylu s názvem *„mujStyl"* na tři objekty – `Button`, `TextBox` a `Label`.

```
<Canvas>
  <Canvas.Resources>
    <Style x:Key="mujStyl">
     <Setter Property="Control.FontSize" Value="15" />
     <Setter Property="Control.Foreground" Value="White" />
     <Setter Property="Control.Background" Value="#FF7EE42D" />
    </Style>
   </Canvas.Resources>
<Button Style="{StaticResource mujStyl}" Height="27"
Width="71" Canvas.Left="63">Tlačítko</Button>

<TextBox Style="{StaticResource mujStyl}" Height="28
Width="152" Canvas.Left="63" Canvas.Top="75" />

<Label Style="{StaticResource mujStyl}" Height="32"
Width="119">Label</Label>
</Canvas>
```



Obrázek 12: Vícenásobný styl

3.3 Šablony

Prakticky každá kontrola ve WPF má svou šablonu, která definuje její základní zobrazení. Je možné si představit, že každý prvek se zobrazuje tak, jak ho známe právě proto, že je na něm použita šablona. Tvorba vlastních šablon nám dává zcela nové možnosti grafického designu aplikací.

Může se zdát, že použití stylů a šablon je velmi podobné, ovšem tyto techniky jsou zcela protichůdné. Styly se používají pouze pro nastavení vlastností nebo spouští elementů, zatímco s pomocí šablon můžeme vytvořit ovládací prvky, které se budou chovat stejně jako základní. Ovšem jejich koncepci si již nadefinujeme sami. Předností je možnost složení ovládacího prvku z více elementů.

K dispozici máme několik druhů šablon, nejpoužívanější jsou typy dva. Typ `ControlTemplate` nastavuje celkový vzhled prvku či elementu a `DataTemplate` definuje způsob zobrazení obsahu prvků.

3.3.1 ControlTemplate

Jak jsem již uvedl, s pomocí objektu typu `ControlTemplate` je možné nastavit celkový vzhled ovládacího prvku. Samozřejmostí je sestavení prvku z několika různých objektů, např.: `Rectangle`, `Ellipse` nebo `TextBox`.

Příklad demonstruje vytvoření šablony pro prvek `Button`. Ten se skládá z jednoho obdélníka a objektu `ContentPresenter`, který zobrazuje obsah tlačítka. [3]

```
<Style x:Key="mujStyl" TargetType="{x:Type Button}">
   <Setter Property="Template">
       <Setter.Value>
          <ControlTemplate TargetType="{x:Type Button}">
             <Grid>
                <Rectangle x:Name="rectangle" Fill="#FF65D2C2"
                  Stroke="Black" StrokeThickness="2"
                  RadiusX="7.5" RadiusY="7.5"
                  RenderTransformOrigin="0.5,0.5">
```

```
            </Rectangle>
              <ContentPresenter Content="Tlačítko"
                VerticalAlignment="Center
                HorizontalAlignment="Center"/>
          </Grid>
```

V tomto případě jsme použili typ šablony, která se aplikuje na všechny objekty `Button` na scéně. Pokud chceme mít šablon více a aplikovat je na samostatné prvky, definujeme je jako zdroje. Toto použití je prakticky stejné jako u stylů. Opět použijeme atribut `x:Key` pro pojmenování šablony a následně prvku `Button` pomocí `StaticResource` tuto šablonu přiřadíme.

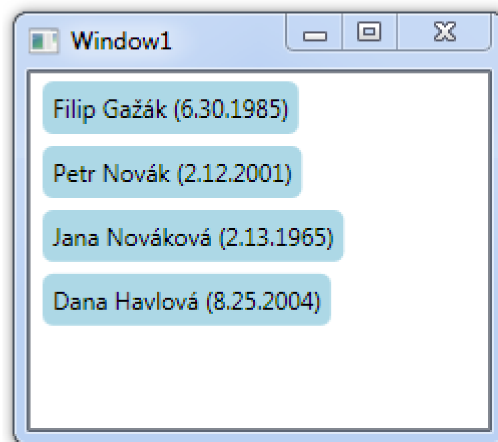
```
<Canvas>
   <Canvas.Resources>
       <ControlTemplate x:Key="moje" TargetType="Button">
        <!--Vlastnosti -->
        </ControlTemplate>
    </Canvas.Resources>
  <Button Template="{StaticResource moje}"></Button>
</Canvas>
```

3.3.2 DataTemplate

Šablony typu `DataTemplate` se využívají především pro úplnou kontrolu způsobu zobrazení dat. Je celkem běžné, že pro zobrazování dat se nepoužívá výchozí nastavení, které je příliš unifikované a strohé – tím pádem uživatelsky nepřívětivé a může vést k horšímu pochopení nebo reprezentaci dat. [1]

Třída `System.Windows.DataTemplate` je jedna z forem šablon používaná v modelu WPF. Pokud se snažíme zobrazit nějaký objekt s daty, který nemá předdeklarovaný způsob vykreslení jeho obsahu, umístíme ho do datové šablony, kde tyto vlastnosti následně naprogramujeme. V souvislosti s `DataTemplate` se využívá techniky `DataBindingu`. Pomocí této techniky propojujeme data a uživatelské rozhraní. Prakticky jde o svázání vlastností mezi datovým zdrojem a cílem. Jakmile se změní hodnoty ve zdroji, dojde ke

změně i v provázané vlastnosti. Tímto způsobem lze provést vazbu na jakýkoliv objekt nebo vlastnost objektu. [2]



Obrázek 13: DataTemplate

3.4 Animace

V počítačové grafice mají animace svou nezastupitelnou roli. Setkat se s nimi není nijak obtížné. Například počítačové hry nebo filmy se dají považovat za jednu velkou animaci. Na internetu se vyskytují v podobě reklamních panelů, upoutávek, nebo například na různých interaktivních webových stránkách. Jako animaci můžeme považovat i změnu vlastnosti tlačítka po najetí myší.

Animace je tedy iluze, která je tvořena za sebou jdoucí sérií obrázků, kde následující snímek je mírně odlišný od předcházejícího. Lidský mozek vnímá skupinu obrázků jako samostatnou měnící se scénu. Ve filmu je tato iluze vytvořena pomocí kamery, která zaznamenává mnoho fotografií každou sekundu. Pro lidské oko je zásadní hranice kolem 24 snímků za sekundu, kdy již dokáže rozpoznat spojitý obraz.

V dnešní době, kdy jsou uživatelé počítačů stále náročnější, je nutné přistupovat k tvorbě aplikací odlišně, než například před deseti lety. Dnešní uživatelé chtějí aplikace poněkud zábavnější, než jen strohé šedivé formuláře. Při tvorbě aplikací se tedy klade důraz mimo funkčnosti také na vzhled a uživatelskou přívětivost. Při použití animací se uživatelské rozhraní aplikace stane zajímavější a v některých případech i použitelnější. Samozřejmě je nutné k tomu přistupovat s rozvahou, aby nebylo docíleno spíše opačného účinku.

Windows Presentation Foundation obsahuje sadu grafických a projektových nástrojů, které umožňují vytvoření atraktivního uživatelského prostředí a dokumentů. K úspěchu přispívá několik vlastností WPF. Ucelený grafický systém je vytvořen tak, aby objekty měnily svou velikost nebo umístění bez blikání nebo zbytečného překreslování. Zároveň tomu napomáhá, že všechny grafické operace jsou ve WPF prováděny na grafické kartě

.

Pro úspěšné animování vlastností objektů je nutné dodržet tyto požadavky:

- vlastnost musí být typu *dependency property*
- je nutné použít správný datový typ
- vlastnost musí patřit do třídy, která dědí animovatelné vlastnosti

3.4.1 Systém.Windows.Media.Animation

Pro práci s animacemi nám Windows Presentation Foundation umožňuje použít kolekci tříd. Celá tato kolekce spadá pod jmenný prostor `Systém.Windows.Media.Animation` a obsahuje bezmála 200 tříd. Tyto třídy se dělí do skupin podle toho, pro jaký datový typ jsou určeny. Při tvorbě je nutné, uvědomit si, jakou vlastnost budeme animovat. Podle toho také zvolit správný vhodný datový typ. Asi nejběžnější je použití typu `double`, který umožňuje animovat vlastnosti jako je změna velikosti písma `FontSize`, změna výšky `Height` a šířky `Width`, změna pozice, ale také například změna velikosti úhlů.

Celkem je možné animovat 22 datových typů: `Boolean, Byte, Char, Color, Decimal, Double, Int16, Int32, Int64, Matrix, Object, Point, Point3D, Quaternion, Rect, Rotation3D, Single, Size, String, Thickness, Vector a Vector3D`.[1]

<u>Existují tři druhy animací:</u>

- základní – hodnoty přecházejí lineárně od počáteční ke koncové
- animace s klíčovými snímky (*Frame* animace) – hodnoty jsou definovány v několika po sobě jdoucích snímcích
    - *Discrete, Linear, Spline*

- animace s použitím cesty (*Path* animace) – hodnoty jsou definovány na základě zadané cesty nebo trajektorie

V následující tabulce je uveden seznam možných datových typů, zároveň je uvedeno, jaké druhy animací podporují. Všechny typy podporují animaci *Discrete* s klíčovými snímky, naopak *Path* animací podporují jen typy `Double`, `Matrix` a `Point`.[1]

		AnimationUsingKeyFrames			Animation
	Animation	**Discrete**	**Linear**	**Spline**	**UsingPath**
Boolean		•			
Byte	•	•	•	•	
Char		•			
Color	•	•	•	•	
Decimal	•	•	•	•	
Double	•	•	•	•	•
Int16	•	•	•	•	
Int32	•	•	•	•	
Int64	•	•	•	•	
Matrix		•			•
Object		•			
Point	•	•	•	•	•
Point3D	•	•	•	•	
Quaternion	•	•	•	•	
Rect	•	•	•	•	
Rotation3D	•	•	•	•	
Single	•	•	•	•	
Size	•	•	•	•	
String		•			
Thickness	•	•	•	•	
Vector	•	•	•	•	
Vectror3D	•	•	•	•	

Tabulka 1: Datové typy ve WPF

3.4.2 Spouště

Při tvorbě animací je nutné si uvědomit, jak tyto události fungují. Většina aplikací je tvořena tak, že daná animace nebo jen změna vlastnosti objektu se uskuteční po nějaké události. Tato událost může být různá: načtení formuláře, přejetí myší přes objekt, kliknutí na objekt nebo třeba časová změna.

Pro tyto účely Windows Presentation Foundation nabízí vlastnosti typu `Trigger` neboli spouště. Tyto vlastnosti jsou nastaveny tak, že hlídají předem nadefinované vlastnosti. Pokud se nějaká změní, tak reagují. Spouště se v modelu WPF používají velmi často, zejména jsou spojeny s animacemi, ale použít je můžeme prakticky kdekoliv, kde je nutné, aby se po uživatelském vstupu něco stalo.

WPF obsahuje třídu `TriggerBase`, která dále obsahuje pět různých druhů spouští: `Trigger`, `MultiTrigger`, `EventTrigger`, `DataTrigger` a `MultiDataTrigger`.

3.4.2.1 Trigger

`Trigger` je nejčastější vlastností ze třídy `TriggerBase`. Stanovuje jak má prvek či element reagovat na změnu konkrétní vlastnosti. Reaguje tak, že změní cílovou vlastnost pomocí vlastnosti `Setter`.

Následující příklad demonstruje použití vlastnosti `Trigger`. V projektu jsou dvě tlačítka. První tlačítko reaguje na kliknutí a přejezd myší. Pokud se na něj klikne, zvětší se velikost písma, při přejezdu myší se změní barva pozadí tlačítka. Druhé tlačítko změní barvu textu pokud na něm uděláme jakoukoliv operaci myší.

```
<Canvas>
   <Canvas.Resources>
      <Style x:Key="styl_1" TargetType="Button">
         <Style.Triggers>
            <Trigger Property="IsPressed" Value="true">
              <Setter Property="FontSize" Value="15" />
            </Trigger>
            <Trigger Property="IsMouseOver" Value="true">
               <Setter Property="Background" Value="green" />
```

```
   </Trigger>
      </Style.Triggers>
        </Style>
          <Style x:Key="styl_2" TargetType="Button">
            <Style.Triggers>
               <Trigger Property="IsFocused" Value="true">
                   <Setter Property="Foreground" Value="red" />
               </Trigger>
            </Style.Triggers>
         </Style>
     </Canvas.Resources>
 <Button Style="{StaticResource styl_1}"/>
 <Button Style="{StaticResource styl_2}"/>
</Canvas>
```

3.4.2.2 MultiTrigger

Oproti jednodušší verzi spouště je prvek `MultiTrigger` rozdílný v použití podmínek. Pro spuštění nějaké akce je nutné splnit minimálně dvě takové podmínky. Tuto spoušť je tedy výhodné použít, jestliže existuje více událostí spouštějící akci nebo je komplikované nastavení jejich pořadí.

V předchozím příkladu se měnila barva pozadí a velikost písma u prvku `Button`. Každá akce nastala po splnění vlastnosti spouště a mohla vždy proběhnout jen jedna. Následující příklad je pouze modifikací předchozího a ukazuje, jak s použitím vícenásobné spouště spustit více akcí najednou. Je ovšem nutné, aby byly splněny všechny podmínky (*Conditions*). Z příkladu je patrné použití vlastností podmínek místo definování elementu `Trigger`. Prvek `Button` tedy změní velikost písma a jeho barvu, pouze v případě, že se na prvek klikne a myš se nachází nad ním.

```
<Style x:Key="styl_1" TargetType="Button">
  <Style.Triggers>
     <MultiTrigger>
         <MultiTrigger.Conditions>
              <Condition Property="IsMouseOver" Value="True" />
              <Condition Property="IsPressed" Value="True" />
         </MultiTrigger.Conditions>
      <Setter Property="FontSize" Value="15" />
      <Setter Property="Foreground" Value="green" /
     </MultiTrigger>
   </Style.Triggers>
</Style>
```

3.4.2.3 EventTrigger

Spoušť `EventTrigger` je velmi specifická a odlišná od základní spouště `Trigger`. Zde se již nedefinuje vlastnost, která se bude měnit v rámci nějaké uskutečněné akce, tato spoušť se aktivuje, pokud nastanou jiné specifické události. Zejména se jedná o události, které zasahují do chování jednotlivých objektů. Je tedy patrné, že tyto události se již nezapisují do samotné sekce `Trigger`, ale pro jejich průběh je nutné použít scénář `Storyboard` (téma scénářů bude probráno v následující kapitole). Jelikož tento typ neumožňuje definovat vlastnosti objektů, nachází uplatnění právě u spouštění animací nebo přehrávání zvuků.

Následující jednoduchý příklad ukazuje základní XAML zápis objektu `EventTrigger`. Vlastnost `SourceName` odkazuje na atribut `Name` elementu s událostí, která spouští animaci. `RoutedEvent` je konkrétní spouštěcí událost. Vlastnost `Actions` určuje co se stane, když dojde k dané události.

```
<EventTrigger SourceName="myBtn" RoutedEvent="Button.Click">
     <EventTrigger.Actions>
          <BeginStoryboard>
               <Storyboard>

               </Storyboard>
          </BeginStoryboard>
     </EventTrigger.Actions>
</EventTrigger>
```

3.4.2.4 DataTrigger

Objekt `DataTrigger` je typ spouště, která umožňuje pomocí vazby nastavit vlastnost elementu, na který tato vazba směřuje. Výhodné použití je v projektu s více elementy, kdy změnou vlastnosti jednoho změníme vlastnost druhého elementu. [9]

3.4.2.5 MultiDataTrigger

`MultiDataTrigger` je podobný jako `MultiTrigger`. Je nutné zde opět použít podmínky a vazby mezi objekty. Podmínka se zapisuje do části

Condition a její hodnota do vlastnosti Value. Pro úspěšné vykonání spouště je nutné, aby byly splněny všechny zadané podmínky.

3.4.3 Storyboard

Element Storyboard neboli scénář, je jeden z hlavních prvků, který použijeme při sestavování animace. Jsou to objekty rozšiřující časovou osu o parametry a směrování animace. Má také schopnost sjednotit více časových os dohromady.

Každý scénář musí být součástí spouště typu EvenTrigger, při jejíž aktivaci se aktivuje i její podřízený scénář.

XAML zápis vypadá následovně. Samotný scénář musí být podřízen elementu BeginStoryboard, jenž ovládá jeho průběh. Atribut TargetName určuje jméno animovaného objektu a TargetProperty animovanou vlastnost.

```
<EventTrigger>
  <BeginStoryboard>
    <Storyboard Storyboard.TargetName=""
           Storyboard.TargetProperty=""/>
  </BeginStoryboard>
</EventTrigger>
```

Element BeginStoryboard spouští podřízený scénář. Ve stejné třídě ovšem existují další elementy, kterými lze průběh animace ovládat.

- StopStoryboard – zastavení probíhajícího scénáře
- PauseStoryboard – pozastavení probíhajícího scénáře
- ResumeStoryboard – spuštění pozastaveného scénáře
- RemoveStoryboard – odstranění scénáře
- SetStoryboardSpeedRatio – změna rychlost scénáře
- SeekStoryboard – skok na určitý čas ve scénáři
- SkipStoryboardToFill – skok na konec scénáře

3.4.4 Základní animace

Základní animace představují nejjednodušší formu animací ve WPF. Definujeme zde pouze délku trvání, počáteční a koncové hodnoty. Hodnoty animace se poté vykreslí postupně od počáteční po koncovou, neboli lineárně. Tyto animace využijeme v případě, kdy není nutné, aby se chovala různě v jednotlivých časových jednotkách. Příkladem může být animace, kdy chceme změnit velikost nějakého objektu z počátečních 20px na 120px během pěti vteřin. Tento proces proběhne tak, že po každé vteřině se velikost objektu zvětší o 20px.

Animace typu Double

Následující příklady ukazují využití základní animace typu `Double`, kde pracujeme pouze s parametry, které mají hodnoty reálných čísel.

V příkladu jsou celkem tři scénáře obsahující jednotlivé animace. Každý scénář je vnořen do objektu `EvenTrigger`, který reaguje na události a spouští animaci.

V prvním případě se scénář spustí po kliknutí na tlačítko s názvem „*btn*“. Měnit se bude hodnota definovaná elementem `TargetProperty.` V tomto případě posunem na plátně z původních 17px na 50px v celkovém čase tří vteřin. Objekt, u kterého tato změna nastane je určen elementem `TargetName`, tedy obdélník.

Druhý scénář se spustí po kliknutí na obdélník s názvem „*rec*“, v tomto případě se animuje samotný obdélník, respektive zmenší se jeho šířka na třetinu původní během pěti vteřin. Element `RepeatBehaviour` zajišťuje, že animace proběhne celkem 3x.

Třetí scénář se aktivuje, pokud se zaškrtne objekt `CheckBox`. Animace je pomocí `TargetName` směřována na tlačítko a jeho velikost písma. Ta se během deseti vteřin změní z původní hodnoty na hodnotu 19 bodů.

(Opakující kód jsem vypustil)

```
<EventTrigger SourceName="bnt" RoutedEvent="Button.Click">
  <BeginStoryboard>
     <Storyboard>
        <DoubleAnimation Storyboard.TargetName="rec"
                  Storyboard.TargetProperty="(Canvas.Left)"
                  From="17" To="50" Duration="0:0:3">
        </DoubleAnimation>
     </Storyboard>
  </BeginStoryboard>
</EventTrigger>

<EventTrigger SourceName="rect"
              RoutedEvent="Rectangle.MouseDown">
    <BeginStoryboard>
       <Storyboard>
          <DoubleAnimation Storyboard.TargetName="rec"
                        TargetProperty="Width"
                        From="150" To="50" Duration="0:0:5"
                        RepeatBehavior="3x">
          </DoubleAnimation>

  <EventTrigger SourceName="checkBox"
                RoutedEvent="CheckBox.Checked">
            <BeginStoryboard>
               <Storyboard>
                  <DoubleAnimation Storyboard.TargetName="btn"
                    Storyboard.TargetProperty="FontSize"
                    To="19" Duration="0:0:10">
                  </DoubleAnimation>
```

Objekt Animation obsahuje různé atributy, které ovládají způsob průběhu animace a její časový průběh. [1]

Duration: atribut definuje po jakou dobu bude animace probíhat, zapisujeme ho ve tvaru *hodiny:minuty:sekundy*.

```
Duration="0:1:10"
```

From / To: první atribut určuje počáteční hodnotu vlastnosti, která se bude animovat, druhý atribut cílovou hodnotu vlastnosti.

```
From="23" To="100"
```

From / *By*: při použití atributu *By* bude koncová hodnota součtem právě těchto dvou atributů, například hodnoty budou 12 a 15, tak cílová hodnota, do které animace proběhne se rovná 27.

```
From="12" To="15"
```

RepeatBehaviour: atribut, který zajistí opakované spuštění animace. V příkladu je použita hodnota 3x, časová osa se tedy zopakuje třikrát po sobě. U atributu

je možné nastavit hodnotu `Forever`, animace pak bude probíhat neustále dokola.

```
RepeatBehavior="Forever"
```

Autoreverse: pokud je hodnota tohoto atributu nastavena na `True`, tak se animace po dokončení zopakuje, ovšem v opačném směru.

```
AutoReverse="True"
```

IsAditive: u tohoto atributu je možné opět nastavit jeho hodnotu na `True`, to zajistí, že se hodnoty v atributech `From` a `To` přičtou k počáteční a cílové hodnotě.

```
IsAdditive="True"
```

Pokud chceme pracovat s transformacemi, využijeme právě animace typu `Double`. Lze použít všechny druhy, které WPF nabízí, tedy posun, rotaci, zkosení, změnu velikosti a maticovou transformaci.

V následujícm příkladu je použito textové pole uvitř objektu `Border`. Textové pole má nastavenou animaci posunu ve směru osy X o 20px během jedné vteřiny a opakuje se neustále dokola. U objektu `Border` je použita animace rotace kolem svého středu, otáčí se o 30 stupňů během čtyř vteřin. Samotná transformace musí být uvnitř elementu `RenderTrasform` a pomocí atributu `x:Name` pojmenována. Na toto jméno se dále odkazujeme přímo v elementu `DoubleAnimation`, v příkladu „*posun*“ a „*rotace*“.

```
<Border RenderTransformOrigin="0.5 0.5">
  <TextBox Text="Animace" >
     <TextBox.RenderTransform>
       <TranslateTransform x:Name="posun" />
     </TextBox.RenderTransform>
  </TextBox>
    <Border.RenderTransform>
      <RotateTransform x:Name="rotace" />
    </Border.RenderTransform>
</Border>

<DoubleAnimation Storyboard.TargetName="rotace"
              Storyboard.TargetProperty="Angle"
              To="30" Duration="0:0:4"
              AutoReverse="True"
```

```
                RepeatBehavior="Forever">
</DoubleAnimation>
<DoubleAnimation Storyboard.TargetName="skew"
                 Storyboard.TargetProperty="X"
                 To="20" Duration="0:0:1"
                 AutoReverse="True"
                 RepeatBehavior="Forever">
</DoubleAnimation>
```

Animace typu Color

Tento typ animací se využívá pro změny vlastností objektů, které přímo souvisejí s barvou, jako jsou barevné přechody a výplně nebo například i barva písma.

V následujícím příkladu jsou použita tři tlačítka a jedna elipsa s názvem „*ellipse*", přičemž každé tlačítko spouští jednu animaci.

První tlačítko s názvem „*btn1*" aktivuje animaci samo na sobě, konkrétně přechod jeho barvy písma ze zelené na modrou během čtyř vteřin, poté dojde k obrácenému přechodu.

Druhé tlačítku spustí animaci změnou barvy celého plátna. Jelikož je použita vlastnost `RepeatBehaviour` s hodnotou 4x, dojde k opakujícímu se problikávání plochy.

Barevnou výplň elipsy změní aktivace třetího tlačítka. U objektu je výplň definovaná parametrem `GradientBrush`, tedy barevným přechodem (mezi žlutou a modrou barvou). Animace proběhne během tři a půl sekundy a změní původní přechod odlišnými barvami.

```
<!-- Animce pozadí tlačítka -->
<Button Name="btn1" Background="Blue">Tlačítko1</Button>
    <EventTrigger SourceName="btn1" RoutedEvent="Button.Click">
       <BeginStoryboard>
         <Storyboard>
           <ColorAnimation Storyboard.TargetName="btn1"
            Storyboard.TargetProperty="Background.Color"
            To="Green" Duration="0:0:4" AutoReverse="True">
           </ColorAnimation>
         </Storyboard>
        </BeginStoryboard>
     </EventTrigger>
```

```
<!-- Animace barvy plátna -->
<Button Name="btn2" Width="75">Tlačítko 2</Button>
   <EventTrigger SourceName="btn2" RoutedEvent="Button.Click">
   <BeginStoryboard>
     <Storyboard>
         <ColorAnimation Storyboard.TargetName="cnv"
          Storyboard.TargetProperty="Background.Color"
          To="#FF4AE221" Duration="0:0:4"
          RepeatBehavior="4x">
         </ColorAnimation>
     </Storyboard>
   </BeginStoryboard></EventTrigger>

<!-- Animace výplně elipsy -->
<Button Name="btn3" Width="75">Tlačítko3</Button>
    <EventTrigger SourceName="btn3" RoutedEvent="Button.Click">
       <BeginStoryboard>
          <Storyboard>
            <ColorAnimation Storyboard.TargetName="vypln"
             Storyboard.TargetProperty="GradientStops[1].Color"
            To="green" Duration="0:0:3.5">
            </ColorAnimation>
           </Storyboard>
        </BeginStoryboard>
     </EventTrigger>
```

Animace typu Thickness

Animace toho typu pracují s vlastností `thickness` objektů. Ve většině případů jde o jejich velikosti ohraničení, neboli `BorderThickness`.

Následující příklad demonstruje použití animace na dvou objektech – textové pole, které obsahuje popis „*ThicknessAnimation*“ a je uvnitř objektu `Border`. Akce se spouští po načtení scény pomocí atributu `Canvas.Loaded`. Objekt `Border` změní během dvou vteřin své ohraničení na hodnoty 15, 15 ,25 ,25 („*levý, pravý, horní, dolní*“), textové pole z výchozí hodnoty 1 na hodnotu 5.

```
<EventTrigger RoutedEvent="Canvas.Loaded">
  <BeginStoryboard>
    <Storyboard>
<!--animace ohraničení objektu Border-->
      <ThicknessAnimation Storyboard.TargetName="border1"
       Storyboard.TargetProperty="BorderThickness"
       From="1,1,1,1" To="15,15,25,25" Duration="0:0:2"
       RepeatBehavior="7x" AutoReverse="True">
      </ThicknessAnimation>
```

```
<!--animace ohraničení textového pole-->
      <ThicknessAnimation Storyboard.TargetName="textBox1"
       Storyboard.TargetProperty="BorderThickness"
       From="1" To="5" Duration="0:0:2"
       RepeatBehavior="3x"></ThicknessAnimation>
        </Storyboard
     </BeginStoryboard>
  </EventTrigger>
<Border CanvasName="border1" BorderBrush="#FF91C8EE"
 BorderThickness="1">
<TextBox Name="textBox1"
BorderBrush="Chartreuse">ThicknessAnimation</TextBox>
```

Animace typu Point

Point animace je další možnost, jak animovat pohyb objektů po dráze určené pomocí bodů (*points*). Jako atributy animace určujeme počáteční a koncové body pohybu. Ovšem tento typ můžeme využít například i při přechodech dvou barev pomocí bodů.

V příkladu je názorně předvedena animace pohybu elipsy, která se pohybuje směrem dolů. Poté vytváří dojem odrazu a vrací se zpět na své původní místo. Elipsa je definovaná pomocí elementu `EllipseGeometry`, ten obsahuje atribut `Center`, který určuje její střed. Pomocí animace se střed objektu posune na pozici [40, 220] a stejnou cestou se vrací zpět.

```
<!--Deklarace elipsy pomocí elementu Geometry-->
<Path Fill="#FF54E3E8" Height="262" Width="85">
   <Path.Data>
      <EllipseGeometry x:Name="pointAnimace"
       Center="40,50" RadiusX="34" RadiusY="27" />
    </Path.Data>

<!--Pohyb elipsy -->
 <Path.Triggers>
    <EventTrigger RoutedEvent="Canvas.Loaded">
         <BeginStoryboard Name="myStoryboard">
           <Storyboard>
             <PointAnimation
               Storyboard.TargetProperty="Center"
               Storyboard.TargetName="pointAnimace"
               Duration="0:0:3.5" To="40,220"
               RepeatBehavior="10x"
               AutoReverse="True" />
           </Storyboard>
```

```
            </BeginStoryboard>
        </EventTrigger>
    </Path.Triggers>
</Path><Rectangle Height="3" Name="rectangle1" Stroke="Black"
Width="102" Fill="Black" />
```

3.4.5 Animace s klíčovými snímky

Základní animace dovolují pouze změnu jedné hodnoty na druhou, což znemožňuje vytvoření složitějších přechodů, zejména definovat více atributů a jejich hodnotu v určitém čase. Zároveň není možné pohybovat objekty po jiné trase, než je přímka, aniž by bylo nutné tvořit pro každou změnu směru novou animaci. Nejen díky tomu WPF podporuje animace s klíčovými snímky, neboli *frame* animace.

U *frame* animací je tedy možné pracovat s více hodnotami. Každá hodnota představuje snímek a celá jejich množina na časové ose tvoří kompletní animaci, přičemž snímky musí být zadávány v pořadí, ve kterém přicházejí na řadu. Podle způsobu přechodu mezi jednotlivými snímky dělíme tyto animace na tři druhy.

- Klíčové snímky s lineárním přechodem - princip je zde podobný jako u základních animací, změna hodnot zde probíhá lineárně. Tento typ je určen pro 17 datových typů.
- Klíčové snímky s diskrétním přechodem - u tohoto typu nadefinujeme hodnoty a animace pouze mezi nimi přeskakuje. Diskrétní animace je určena pro všechny datové typy.
- Klíčové snímky typu *Spline* - hodnoty jsou zde generovány nelineárním způsobem, ale pomocí křivky. Stejně jako první typ je určen pro 17 datových typů.

XAML zápis elementu pro animace se snímky je doplněn slovy `UsingKeyFrames`, pro návratovou hodnotu `Double` bude vypadat následovně: `DoubleAnimationUsingKeyFrames`. Uvnitř toho elementu se

dále nachází objekt, který přímo určuje, o jaký typ přechodu se jedná:

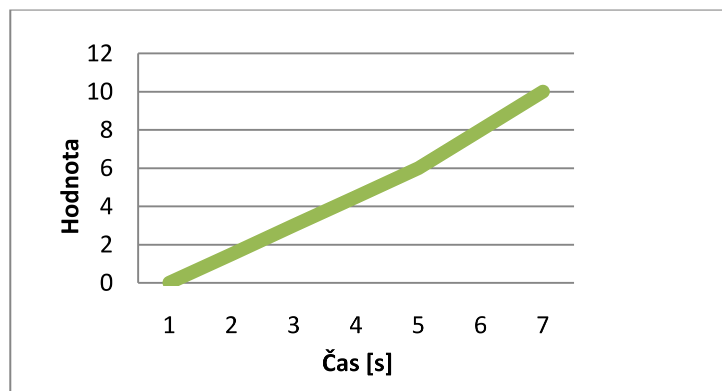
- `LinearDoubleKeyFrame`
- `DiscreteDoubleKeyFrame`
- `SplineDoubleKeyFrame`

Lineární přechody

Tento typ je prakticky stejný jako základní animace. Změna hodnot probíhá lineárně. Tabulka ukazuje případ, jak jsou dopočítány hodnoty v závislosti na čase. Tučně jsou označeny hodnoty, které jsme nastavili, zbytek je dopočítán automaticky.

Tabulka 2: Lineární přechod

Čas [s]	0	**1**	2	**3**	4	5	**6**
Hodnota	0	**1,5**	3	**4,5**	6	8	**10**



Graf 1: Lineární přechod

Následující příklad ukazuje pohyb elipsy pomocí *frame* animace s lineárním přechodem hodnot. Samotný pohyb je transformace, která obsahuje

dva atributy – `x:Name` značí jméno a atribut `Y` určuje osu, podél které se objekt bude pohybovat.

```
<!--Transformace posunu po ose Y-->
<Ellipse.RenderTransform>
    <TranslateTransform x:Name="Posun" Y="0" />
</Ellipse.RenderTransform></Ellipse>

<!--LinearKeyFrame Animace-->
<Canvas.Triggers>
   <EventTrigger RoutedEvent="Canvas.Loaded">
      <BeginStoryboard>

        <Storyboard>
         <DoubleAnimationUsingKeyFrames
            Storyboard.TargetName="Posun"
            Storyboard.TargetProperty="Y"
            Duration="0:0:12">
          <LinearDoubleKeyFrame Value="20" KeyTime="0:0:0.5" />
          <LinearDoubleKeyFrame Value="140" KeyTime="0:0:2" />
          <LinearDoubleKeyFrame Value="10" KeyTime="0:0:3.5" />
          <LinearDoubleKeyFrame Value="120" KeyTime="0:0:6" />
          <LinearDoubleKeyFrame Value="0" KeyTime="0:0:12" />
          </DoubleAnimationUsingKeyFrames>
  </Storyboard></BeginStoryboard></EventTrigger>
```

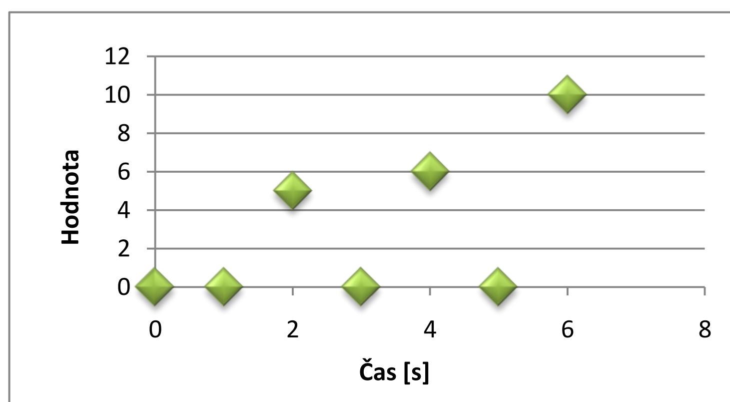
Jak je vidět, celá definice je podobná předchozím animacím, pouze snímky jsou subelementy elementu `DoubleAnimationUsingKeyFrames`. Každý snímek obsahuje atributy `KeyTime` a `Value`, které označují hodnotu animované vlastnosti v konkrétním čase. V tomto konkrétním příkladu se elipsa během půl vteřiny dostane na pozici 20px, poté postupně na 140px. Celá animce trvá 12 vteřin, v tomto čase se již elipsa nachází na svém původním místě.

Diskrétní přechody

U diskrétního přechodu přechází animace pouze z „místa na místo“, tedy přeskakuje na hodnoty, které jsme nastavily (viz následující tabulka). [8]

Tabulka 3: Diskrétní přechod

Čas [s]	0	1	2	3	4	5	6
Hodnota	0	0	5	0	6	0	10



Graf 2: Diskrétní přechod

Jako příklad k tomuto přechodu jsem vybral animaci typu `String`, protože tento datový typ podporuje pouze animaci s diskrétními hodnotami. Jak ukazuje následující ukázka, hodí se například pro animování obsahu v objektech. Animace trvá deset vteřin, během této doby dojde k postupnému najetí jednotlivých písmen slov „*String Animation*“. Je zde celkem deset snímků, každý trvá jednu vteřinu.

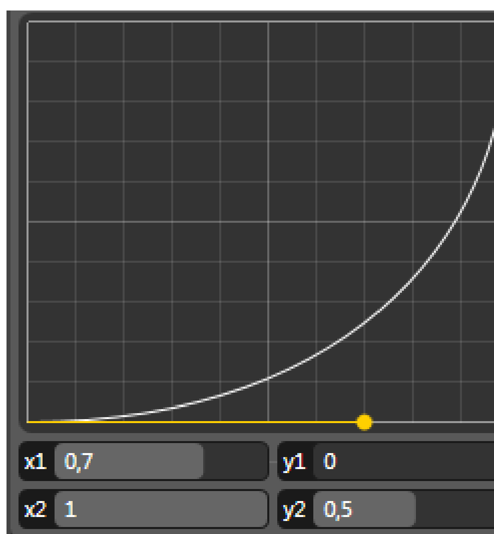
```
<EventTrigger SourceName="button1"
              RoutedEvent="Button.Click">
   <BeginStoryboard>
      <Storyboard>
        <StringAnimationUsingKeyFrames
         Storyboard.TargetName="label1"
         Storyboard.TargetProperty="(Label.Content)"
         Duration="0:0:10">
    <DiscreteStringKeyFrame KeyTime="0:0:1" Value="S" />
    <DiscreteStringKeyFrame KeyTime="0:0:2" Value="St"/>
    <DiscreteStringKeyFrame KeyTime="0:0:3" Value="Str" />
    <DiscreteStringKeyFrame KeyTime="0:0:4" Value="Stri" />
    <DiscreteStringKeyFrame KeyTime="0:0:5" Value="Strin" />
    <DiscreteStringKeyFrame KeyTime="0:0:6" Value="String" />
    <DiscreteStringKeyFrame KeyTime="0:0:7" Value="String An"/>
    <DiscreteStringKeyFrame KeyTime="0:0:8"
                            Value="String Anim" />
    <DiscreteStringKeyFrame KeyTime="0:0:9"
                            Value="String Anima" />
```

```
    <DiscreteStringKeyFrame KeyTime="0:0:10"
                            Value="String Animace" />
        </StringAnimationUsingKeyFrames>
</Storyboard></BeginStoryboard>
</EventTrigger>
```

Spline přechody

Třetí možnost, jak přecházet mezi snímky je pomocí křivky. Element `Spline` obsahuje navíc atribut `KeySpline`, který určuje řídící body křivky. Tato křivka začíná v bodě [0, 0] a končí v bodě [1, 1], tyto souřadnice nesmí být menší než 0 a větší než 1. Tímto způsobem lze vytvořit objekty, které zrychlují a zpomalují.

Obrázek ukazuje křivku, která má řídící body [0,7, 0] a [1, 0,5]. Podle dráhy je patrné, že zde bude postupné zrychlování objektu.



Obrázek 14: Bézierova křivka

Následující příklad demonstruje, jak je možné rozpohybovat elipsu a její pohyb řídit pomocí křivky. Celá animace trvá 12 vteřin a je rozdělena do dvou částí. Během prvních osmi vteřin se elipsa pohybuje po vodorovné ose, dochází

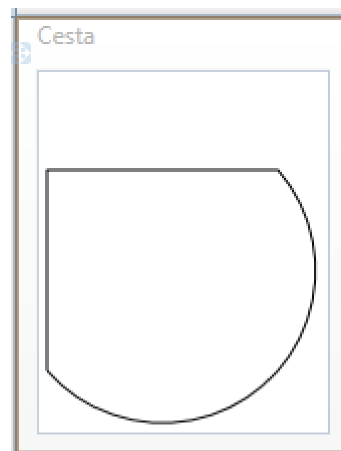
zde k postupnému zrychlování a zpomalování. Během dalších čtyř vteřin je pohyb uskutečněn po svislé ose. Každý atribut `KeySpline` určuje výslednou podobu Bézierovy křivky, v pořadí x_1,y_1,x_2,y_2.

```
<!--definice pohybu po vodorovné ose-->
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
                      Storyboard.TargetName="posun"
                     Storyboard.TargetProperty="X">
    <SplineDoubleKeyFrame KeyTime="00:00:04"
       Value="472" KeySpline="0.7,0,1,0.5"/>
    <SplineDoubleKeyFrame KeyTime="00:00:08"
       Value="0" KeySpline="0,0.5,1,0.5"/>
    <SplineDoubleKeyFrame KeyTime="00:00:11"
       Value="208" KeySpline="0,0.8,0.2,1"/>
    <SplineDoubleKeyFrame KeyTime="00:00:12"
       Value="0" KeySpline="0.88,0,1,0.3"/>
</DoubleAnimationUsingKeyFrames>
<!--definice pohybu po svislé ose-->
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
                      Storyboard.TargetName="posun"
                     Storyboard.TargetProperty="Y">
     <SplineDoubleKeyFrame KeyTime="00:00:08"
         Value="0"/>
     <SplineDoubleKeyFrame KeyTime="00:00:11"
         Value="-176" KeySpline="0,0.8,0.2,1"/>
     <SplineDoubleKeyFrame KeyTime="00:00:12"
         Value="0" KeySpline="0.88,0,1,0.3"/>
     </DoubleAnimationUsingKeyFrames>
<Ellipse x:Name="ellipse">
 <Ellipse.RenderTransform>
     <TranslateTransform x:Name="posun" X="0" Y="0"/>
</Ellipse.RenderTransform></Ellipse>
```

Z této ukázky je patrné, že s použitím *Spline* snímků dokážeme vytvářet velmi efektivní animace s dojmem realističtějšího pohybu. Při vytváření příkladu se mi velmi osvědčil XAML editor Microsoft Expression Blend 3. Zejména pro jeho kvalitní nástroj pro definování křivky pohybu, kde jen zadáme řídící body a hned vidíme, jak křivka vypadá. Dále je možné ji upravovat i v grafickém návrhu. Animace se zde vytvářejí velmi snadno, ovšem za cenu velmi nepřehledného kódu.

3.4.6 Animace typu *Path*

Pro pohyb objektů podél různé cesty využíváme animace typu *Path*. Cesta může být složena z různých útvarů, např. úsečka, oblouk nebo Bézierova křivka. Dráhu pohybu definujeme pomocí elementu `PathGeometry` a atributem `Figures` přímo její podobu.



Příklad sestavení dráhy:

```
Figures="M 5 150 V 120 50 H 5 120
         A 10 10 45 1 1 5 150"
```

Parametr `M` značí počáteční bod [5, 150]. Cesta je složena ze tří útvarů – první je vertikální úsečka `V`, která začíná v počátečním bodě a končí v bodě [120, 50]. `H` značí vodorovnou úsečku, která opět začíná v počátečním bodě a končí v bodě [5, 120], `A` značí oblouk, který se otáčí o 45 stupňů a spojuje obě úsečky.

V příkladu je použita předchozí cesta, po které se pohybuje obdélník. Celá animace je definována v elementu `DoubleAnimationUsingPath`, který obsahuje důležitý atribut `PathGeometry`, jenž se pomocí klíčového slova `StaticResource` odkazuje na dříve definovanou cestu. Průběh animanice je rozdělen do dvou částí, první pro horizontální a druhá pro vertikální pohyb.

```
<!---Definice cesty-->
<PathGeometry x:Key="cesta"
    Figures="M 5 150 V 120 50 H 5 120
             A 10 10 45 1 1 5 150" />
   <Storyboard>
      <DoubleAnimationUsingPath
        Storyboard.TargetName="obdelnik"
        Storyboard.TargetProperty="(Canvas.Left)"
        Duration="0:0:10"
        PathGeometry="{StaticResource cesta}"
        Source="X" />
```

```
<DoubleAnimationUsingPath
   Storyboard.TargetName="obdelnik"
   Storyboard.TargetProperty="(Canvas.Top)"
   Duration="0:0:10"
   PathGeometry="{StaticResource cesta}" Source="Y" />
```

3.5 C# a XAML

V kapitole *2.3* jsem uvedl, že použití jazyka XAML pro tvorbu aplikací v modelu WPF, není bezpodmínečně nutné. Záleží na programátorovi, zda XAML použije nebo aplikaci naprogramuje jedním z .NET jazyků. Ovšem tímto se připraví o několik výhod, které tento deklarativní jazyk přináší. Oddělení programové části od uživatelského prostředí urychluje případnou editaci v kódu, jelikož výsledný kód je přehlednější a v některých případech i kratší. XAML je koncipován jako jazyk pro rychlé, stručné a intuitivní psaní. Zároveň je jeho syntaxe poměrně lehce pochopitelná.

Příklady k jednotlivým kapitolám jsem vytvářel pouze v XAMLu. Na tomto jazyku je z části model WPF postaven a jeho zařazení bylo dle mého názoru velmi správným krokem, hlavně z hlediska rychlosti psaní a jednoduchosti.

Následující dva příklady ukazují naprogramování stejných animací v jazyce XAML a pomocí jazyka C#. Z ukázek je jisté, že jazyk XAML je pro tvorbu animací o poznání vhodnější. Zápis složitějších animací bude, pomocí procedurálního jazyka výrazně delší a výsledný kód bude složitější.

Příklad č. 1 v C#:

```
DoubleAnimation doubleAnimace = new DoubleAnimation();
doubleAnimace.From = 0;
doubleAnimace.To = 100;
doubleAnimace.Duration = new
Duration(TimeSpan.FromSeconds(6));
doubleAnimace.RepeatBehavior = RepeatBehavior.Forever;
doubleAnimace.AutoReverse = true;

TranslateTransform translateTransform = new
TranslateTransform();
rectangle1.RenderTransform = translateTransform;
translateTransform.BeginAnimation
(TranslateTransform.XProperty, doubleAnimace); }
```

Příklad č. 1 v XAMLu:

```
<Rectangle x:Name="rectangle1" />
<Rectangle.RenderTransform>
<TranslateTransform x:Name="posun" X="0" />
</Rectangle.RenderTransform>

<DoubleAnimation Storyboard.TargetName="posun"
                 Storyboard.TargetProperty="X"
                 From="0" To="100" Duration="0:0:6"
        AutoReverse="True" RepeatBehavior="Forever">
</DoubleAnimation>
```

Příklad č. 2 v C#:

```
DoubleAnimationUsingKeyFrames dAnimace = new
DoubleAnimationUsingKeyFrames();

dAnimace.Duration = TimeSpan.FromSeconds(10);

dAnimace.KeyFrames.Add(new LinearDoubleKeyFrame(350,
KeyTime.FromTimeSpan(TimeSpan.FromSeconds(2)))
);
dAnimace.KeyFrames.Add(new LinearDoubleKeyFrame(120,
KeyTime.FromTimeSpan(TimeSpan.FromSeconds(5)))
);
dAnimace.KeyFrames.Add(new DiscreteDoubleKeyFrame(400,
KeyTime.FromTimeSpan(TimeSpan.FromSeconds(7)))
);
dAnimace.KeyFrames.Add(
new LinearDoubleKeyFrame(0,
KeyTime.FromTimeSpan(TimeSpan.FromSeconds(10)))
);

TranslateTransform translateTransform = new
TranslateTransform();
rectangle1.RenderTransform = translateTransform;
translateTransform.BeginAnimation
(TranslateTransform.YProperty, dAnimace);
```

Příklad č. 2 v XAMLu:

```
<Rectangle Name="rectangle1" Stroke="Black"
  HorizontalAlignment="Left" Width="100" Height="130" >
    <Rectangle.RenderTransform>
      <TranslateTransform x:Name="posun" Y="0" />
     </Rectangle.RenderTransform>
 </Rectangle>
<DoubleAnimationUsingKeyFrames
    Storyboard.TargetName="posun"
    Storyboard.TargetProperty="Y"Duration="0:0:10">
   <LinearDoubleKeyFrame KeyTime="0:0:2"
                         Value="350" />
   <LinearDoubleKeyFrame KeyTime="0:0:5"
                         Value="120" />
   <DiscreteDoubleKeyFrame KeyTime="0:0:7"
                           Value="400" />
   <LinearDoubleKeyFrame KeyTime="0:0:10"
                         Value="0" />
</DoubleAnimationUsingKeyFrames>
```

3.5.1 Vazba C# a XAML

Jazyk XAML zahrnuje funkce, které umožňují spojení jeho kódu s kódem „v pozadí“. Tyto metody umožňují volat objekty nebo události pomocí jazyka C# nebo jiný z NET jazyků. V souvislosti s animacemi tuto možnost využijeme k ovládání jejího startu nebo celkového průběhu. Pro propojení `Storyboardu` s metodami použijeme atribut `x:Name`, který v metodě následně voláme.

```
<Storyboard x:Name="storyboardRotace">
```

Na takto vytvořený a pojmenovaný element se dále odkazujeme pomocí vytvořených metod v kódu.

```
private void buttonStart_Click(object sender, RoutedEventArgs e)
        {
            storyboardRotace.Begin();
        }
private void buttonStop_Click(object sender, RoutedEventArgs e)
        {
            storyboardRotace.Stop();
        }
private void buttonResume_Click(object sender, RoutedEventArgs e)
        {
            storyboardRotace.Resume();
        }
```

3.6 Nástroje

Všechny uvedené příklady jsem vytvářel pomocí .NET Framework 3.5 ve dvou vývojových prostředí od společnosti Microsoft: Microsoft Visual Studio 2008 a Microsoft Expression Blend 3.

1. **Microsoft Visual Studio 2008**

   Visual Studio nabízí efektivní nástroje pro tvorbu WPF aplikací. Tato verze již standardně obsahuje XAML editor, který podporuje např. drag-and-drop a funkci doplňování kódu *IntelliSence*. Ovšem neobsahuje žádné nástroje pro grafický návrh WPF aplikace, proto je nutné použít nástroje jiné např. již zmiňovaný Expression Blend 3.

2. **Microsoft Expression Blend 3**

Tento nástroj je součástí balíku Expression Studio a je určen pro tvorbu grafického prostředí WPF a Silverlight aplikací. Grafický editor prostředí obsahuje nástroje pro efektivní tvorbu animací – časové osy, transformační editor a `Storyboard` panel. Expression Blend je prakticky propojen s Visual Studiem. Pokud chceme naprogramovat funkční logiku aplikace, automaticky se spustí Visual Studio, které je na to jistě vhodnější.

Vývojové prostředí generuje XAML kód automaticky, ovšem tímto vznikají určité nepříjemnosti. Oproti ručnímu psaní je v některých případech kód velmi nepřehledný a generátor přidává elementy, které jsou zbytečné (např. při transformaci objektu jsou do kódu vloženy i takové transformace, které se vůbec nepoužívají). Ovšem i přes tyto nedostatky je Expression Blend výborný XAML editor, jehož nástroje pro tvorbu animací jsou na velmi vysoké úrovni.

4 Závěr

Úkolem mé bakalářské práce bylo seznámit se s modelem Windows Presentation Foundation uvnitř .NET Framework 3.5. Model jsem popsal v teoretické části práce. Zejména se zabývám jeho architekturou a novinkami, které tato technologie nabízí. Zároveň jsem v této části stručně popsal historii platformy .NET Framework.

Dalším cílem bylo seznámit se s tvorbou a návrhem animací. Tuto praktickou část jsem koncipoval ve formě výkladových kapitol, kde čtenáře seznamuji s jednotlivými aspekty jazyka XAML. K jednotlivým částem jsem vytvořil ukázkové příklady a jejich obsah doplnil výkladem.

Při tvorbě výukových kapitol jsem se soustředil spíše na začínající tvůrce, proto jsem volil koncepci jasně srozumitelných příkladů, které budou pro čtenáře přínosné a poslouží pro lepší pochopení problematiky. Dále jsem vytvořil vizuální prezentaci, kde názorně předvádím tvorbu několika druhů animací.

V některých případech jsem nemohl problematiku probrat více do hloubky, hlavně z důvodu omezených zdrojů, které k modelu WPF existují. Jelikož se počet vývojářů této technologie stále zvyšuje, dá se očekávat, že i zdroje a výuková literatura se budou postupně rozšiřovat.

Původně jsem chtěl do práce zařadit i oblast 3D grafiky a animací, ovšem toto téma je v modelu velmi rozsáhlé, proto jsem od něj upustil. I přesto se mi podařilo podat čtenáři dobrý základ, jak se v modelu pracuje s grafikou a animacemi. Proto se domnívám, že tato práce splňuje mé vytýčené cíle.

5 Přehled použité literatury

[1] PETZOLD, Charels. *Mistrovství ve Windows Presentation Foundation*. Vydání první. Brno : Computer Press, a. s., 2008. 928 s. ISBN 978-80-251-2141-2.

[2] MSDN Library - *Customize Data Display with Data Binding and WPF* [online]. [cit. 2010-03-20]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/magazine/cc700358.aspx>.

[3] ItemsControl: *Dr. WPF* [online]. 2009 [cit. 2010-03-25]. Dostupné z WWW:<http://drwpf.com/blog/2008/01/03/itemscontrol-d-is-for-datatemplate/>.

[4] *Kirupa.com.* Easing in Silverlight and WPF [online]. 2009 [cit. 2010-04-02]. Dostupné z WWW:<http://www.kirupa.com/blend_silverlight/easing_sl_wpf_pg2.htm>

[5] NAGEL, Christian, et al. *C# 2008 Programujeme profesionálně*. 1. vydání. Brno : Computer Press, 2009. 1904 s. ISBN 978-80-251-2401-7.

[6] LACKO, Ľuboslav. *Visual Studio 2008 a .NET Framework 3.5* [online]. Bratislava : Microsoft, 2009 [cit. 2010-02-10]. Dostupné z WWW: <http://www.microsoft.sk>.

[7] SHEKHAR, Shashi. *Shashi Shekhar´s blog* [online]. 2008 [cit. 2010-04-20]. Architecture of WPF. Dostupné z WWW: <http://hashishshaker.com/references/wpf/?p=6>.

[8] BLÁHA, Michal. *Vyvojar.cz* [online]. 2007 [cit. 2010-01-16]. WPF - Animace. Dostupné z WWW: <http://www.vyvojar.cz/Articles/456-4-wpf-animace.aspx>.

[9] *MSDN Library* [online]. 2009 [cit. 2010-04-10]. Animation Overview. Dostupné z WWW:<http://msdn.microsoft.com/en-us/library/ms752312.aspx>.

[10] *MSDN Library* [online]. 2008 [cit. 2009-011-27]. Windows Presentation Foundation. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

6 Seznam obrázků

Obrázek 1: .NET Framework 3.5 ... 11
Obrázek 2: WPF architektura ... 13
Obrázek 3: Schéma tříd ve WPF ... 14
Obrázek 4: Struktura aplikace ... 15
Obrázek 5: Transformace ... 17
Obrázek 6: RotateTransform ... 21
Obrázek 7: ScaleTransform ... 23
Obrázek 8: TranslateTransform ... 24
Obrázek 9: SkewTransform ... 25
Obrázek 10: LayoutTransform ... 26
Obrázek 11: RenderTransform ... 26
Obrázek 12: Vícenásobný styl ... 30
Obrázek 13: DataTemplate ... 33
Obrázek 14: Bézierova křivka ... 51

7 Seznam tabulek

TABULKA 1: DATOVÉ TYPY VE WPF 36

TABULKA 2: LINEÁRNÍ PŘECHOD.. 48

TABULKA 3: DISKRÉTNÍ PŘECHOD 50

8 Seznam grafů

GRAF 1: LINEÁRNÍ PŘECHOD 48
GRAF 2: DISKRÉTNÍ PŘECHOD 50

Obsah přiloženého CD

Přiložené CD obsahuje elektronickou verzi této práce, zdrojové kódy použitých příkladů a vizuální prezentaci s postupem vytvoření jednoduchých animací.