Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Rozšíření programu pro grafický návrh podoby rekonstruovaných domů

Plzeň 2013 Pavel Lorenz

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 27. června 2013

Pavel Lorenz

Poděkování

Rád bych zde poděkoval vedoucí bakalářské práce Ing. Janě Hájkové Ph.D. za její rady a čas, který do mne investovala při řešení problematiky kolem této práce. Dále svému nadřízenému Ing. Kryštofu Pešlovi ve firmě Inter-Informatics s r.o. a druhému nadřízenému Petru Maškovi ve firmě Skeleton.cz s r.o. za schovívavost a pochopení během dokončovacích prací.

V neposlední řadě bych také rád poděkoval své rodině a přítelkyni za pevné nervy a trpělivost.

Abstract

In today's globalization world we can see, more than anytime, that the all human activities have a negative impact on our environment. We have started to take care about our limited source of energy. We can watch, in the long term, increasing of energy prices. It necessarily leads to be thrifty not only from ecologic but also economic term. The communism era has left us many ugly prefabricated buildings, because of construction of these buildings there are lots heat losses and the lodgers must pay increasingly higher bills for every year.

The revitalization of prefabricated buildings tries to stop these heat losses. It is a complex set of construct modifications that reduce heating costs, increase the noise characteristics, extends overall life and make the house much nicer. It's not easy to find agreement with the customer about new graphic design. At this time there is no exist a simple tool that would be available to ordinary people and which could easily present their idea about graphic design of their house.

This bachelor thesis is a continuation of the work of Bc. Pavel Kraft, who develop the original of application. In this work I have to try improving it and the main goals are focus to 3D projection, better control in terms of user control and especially extensibility. I would like to continue with this program in my next studies.

Obsah

1 Úvod

V dnešním globalizovaném světě se čím dál více ukazuje, jak veškerá lidská činnost má široký dopad na naše okolí. Začíná se více hledět na to, abychom zbytečně neplýtvali omezenými zdroji energie. Dalším neméně důležitým aspektem je pak neustále zvyšování cen energií, a tak se šetří nejen z ekologického ale i ekonomického hlediska. Z komunistické éry nám v České Republice, a nejen zde, zůstalo mnoho ohyzdných panelových domů, kde díky jejich konstrukci dochází k velkým únikům tepla, a tak všem nájemníkům s každoročním zvedáním cen energií narůstají i režijní náklady. Tyto úniky lze zastavit pomocí tzv. revitalizace panelových domů. Jde o komplexní soubor stavebních úprav, které sníží náklady za vytápění, zvýší protihlukové vlastnosti domu, prodlouží jeho celkovou životnost a v neposlední řadě zkrášlí samotný dům. Domluvit se se zákazníkem na přesné podobně grafického vzhledu bývá nelehký úkol a na trhu v současné době neexistuje jednoduchý nástroj, který by byl dostupný běžnému uživateli a kde by snadno mohl prezentovat svoji představu.

Cílem této práce je rozšířit program pro grafický návrh podoby rekonstruovaných domů. Tato práce volně navazuje na bakalářskou práci z roku 2011 vytvořenou kolegou Pavlem Kraftem, který souhlasil s možností rozšíření.

Cílem projektu je zanalyzovat původní aplikaci, vytipovat kritická místa a vylepšit je. Prozkoumat architekturu, případně ji přepracovat tak, aby aplikace byla snadno rozšiřitelná i do budoucna, například v rámci diplomové práce. Dalším záměrem je usnadnit uživateli práci s návrhem panelového domu a umožnit mu přenést svou představu do aplikace. Dále pak prozkoumat možnosti 3D projekce v .net frameworku a aplikovat tyto poznatky na projekt. Rozšířit aplikaci o prostorové zobrazení a přiblížit tak grafický návrh běznému publiku.

2 Analýza původní aplikace

Původní aplikace byla vytvořena kolegou Bc. Pavlem Kraftem v rámci bakalářské práce. Pro samotnou implementaci byl zvolen .NET framework s technologií WPF (3.2) a technikou „Rychlého vývoje" (viz kapitola 4.1), což má negativní dopad na rošiřitelnost aplikace.

Jeho práce je rozdělena do dvou částí. V první části se navrhuje půdorys a umist'ují se jednotlivé panely na zdi. V druhé části se pak změní projekce z „ptačí perspektivy" do pohledu „ze předu" na zdi. Uživateli je zde umožněno kreslit obrazce na zdi a prohlížet si je ze všech světových stran. Během projekce je uživatel upozorněn, pokud jsou rozměry na protějších stranám rozdílné a je pouze na něm, jak si s tímto poradí. Dále je umožněno ukládání rozpracovaného návrhu do specifického binárního formátu.

2.1 Kritická místa

Cílem následující sekce je mimo jiné vytipovat kritická místa původní aplikace.

2.1.1 Návrh půdorysu

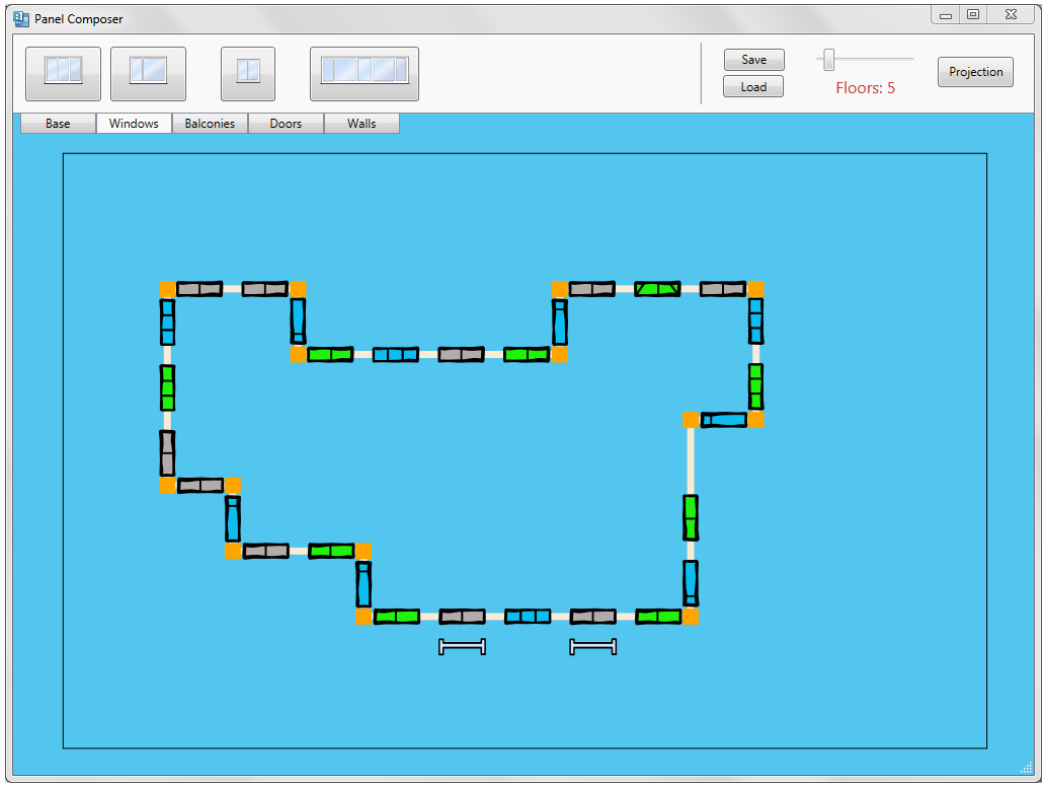
Základním tvarem půdorysu je obdélník. Pomocí zmenšování/zvětšování a rozdělování stěn lze tento tvar libovolně měnit. Jednotlivé délky stěn se odvíjejí od fixní délky jednoho panelu a navíc nelze vytvořit zeď šikmou. Uživatelské ovládání je místy nelogické a nebo minimálně „přes ruku".

- pevná délka stěny
- nemožnost tvorby šikmých stěn
- uživatelské ovládání

2.1.2 Umisťování panelů

Na předpřipravený půdorys lze v této části přidávat různé typy panelů, například okna, balkóny, dveře či čisté stěny. Jak lze vidět na obrázku 2.1, tato část je udělána vhodně a jediné, co by se tomu dalo vytknout je, že jednotlivé panely jsou umisťovány na stejně dlouhou plochu ač je velikost panelů různá.

- různá velikost panelu



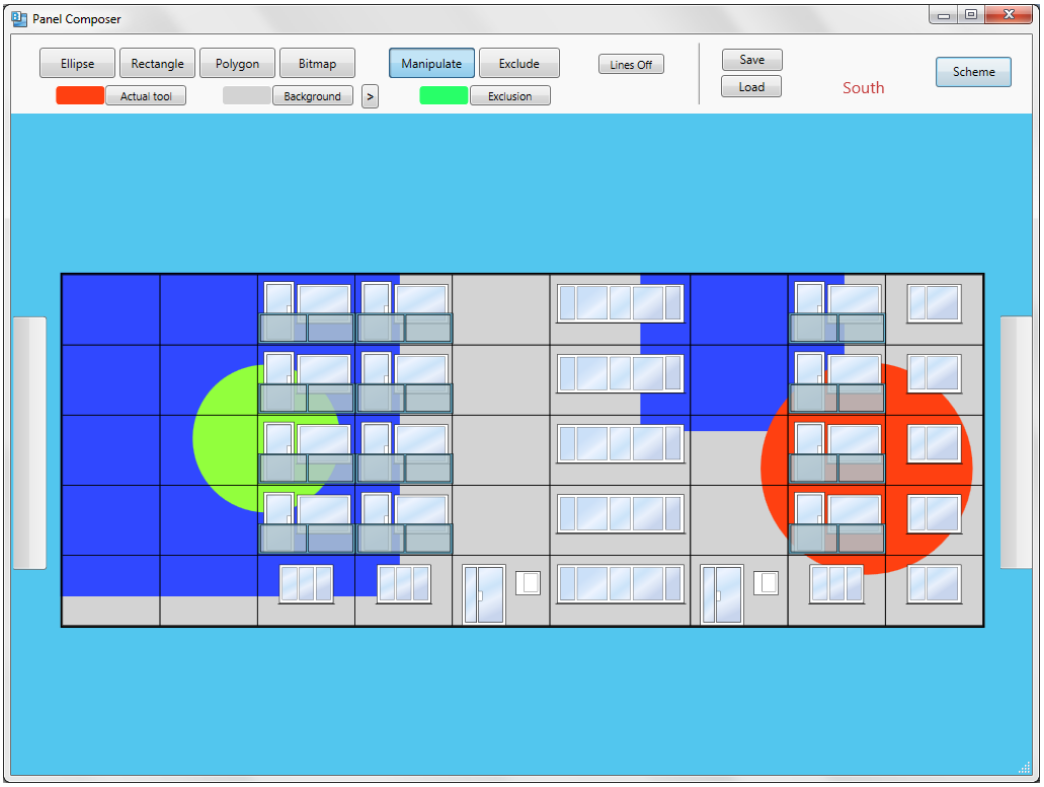
Obrázek 2.1: Návrh půdorysu domu

2.1.3 Kreslení na stěny

Po přepnutí do projekce umožňuje aplikace prohlížet panelový dům ze všech světových stran. Při této projekci lze zároveň kreslit na jednotlivé plochy domu, jak lze vidět na obrázku 2.2.

Ovládání není příliš uživatelsky přívětivé. Velkou nevýhodou pak je nemožnost vyexportovat dílčí stěny a ponechat kreslení na některém z jiných nástrojů.

- nemožnost exportu do obrazového formátu
- uživatelské ovládání



Obrázek 2.2: Zobrazení domu a kreslení

2.1.4 Struktura aplikace

Celá aplikace je postavena na jediném okně a layout je stále stejný. Rozpracovaná data lze načíst a uložit, ale chybí tomu jistá přidaná hodnota. Pokud si dá uživatel tu práci a namodeluje si panelový dům dle svých představ, tak nedostane nic, jen primitivní nástroj na kreslení na předpřipravené stěny. Tyto pohledy navíc nijak nezohledňují zalomení stěn.

Výstupní formát je binární, takže dekódovat takto připravený soubor z aplikace třetí strany je velice obtížné.

Této jednoduché struktuře odpovídá i architektura samotné aplikace. Většina částí je vygenerována pomocí nástrojů *Microsoft Visual Studio* nebo *Microsoft Expression Blend*.

2.1.5 Shrnutí

Může se zdát, že původní aplikace má jen samá negativa. Chtěl bych zde zdůraznit, že tomu tak není. Autor původní bakalářské práce vytyčil směr, jakým by se měl vývoj takovéto aplikace odvíjet a svojí prací ukázal způsob, jakým by se měla ubírat.

Do této práce se pokusím přidat své poznatky z vývoje desktopových aplikací, navázat na myšlenku P. Krafta a posunout tuto aplikaci dále.

2.2 Nápady na inovace

Cílem této práce je výše uvedené nedostatky pokud možno odstranit, aplikaci vylepšit a rozšířit. Samotný koncept návrhu panelového domu od půdorysu, přes doplnění specifických panelů až po samotné kreslení se nikterak měnit nebude.

Návrh půdorysu bude rozšířen o možnost kreslení obecného polygonu a velikost stěn nebude omezena na velikost panelu. Při umisťování panelů bude brána v potaz velikost panelů. Kreslení na stěny bude vylepšeno o možnost exportu do obecného obrazového formátu.

Zásadní částí nové práce pak bude promítnutí výsledku do trojrozměrné podoby, což by mělo přiblížit aplikaci široké veřejnosti. Dále také připravit takovou architekturu aplikace, aby byla do budoucna snáze rozšiřitelná.

3 Volba technologie pro tvorbu desktopových aplikací

V této kapitole budou popsány různé přístupy tvorby desktopových aplikací ve frameworku .NET. Původní aplikace je psána v technologii Windows Presentation Foundation (zkrázeně WPF) od společnosti Microsoft. Desktopové aplikace lze psát i pomocí starší technologie Windows Forms.

3.1 Windows Forms

Windows Forms technologie je ekvivalentem technologie Abstract Window Toolkit (zkráceně AWT) v Java API. Tato technologie se objevila již u prvních verzí .NET frameworku a umožňuje snadné tvoření okenních aplikací.

WinForms mají strmou křivku učení a pro programátory, kteří přechází z Javy o mnoho jednodušší. Aplikace založené na této technologii jsou událostně řízeny, tedy každá akce „aktivuje" sadu událostí a je jen na programátorovi, zda se na dané události zaregistruje a bude na ně reagovat či ne.[wik(2013a)] Mezi nesporné výhody také patří to, že tato technologie je léty prověřená a mnoho programátorů s ní má zkušenosti.

Technologie je také přenositelná i na linuxovou platformu pomocí projektu Mono [mon(2013)]. Společnost Microsoft technologii *Windows Forms* považuje za dostatečně vyspělou a jejímu vývoji již nadále nevěnuje pozornost.

3.2 WPF

Naproti tomu technologie WPF přišla s .net frameworkem 3.0 a snaží se řešit neduhy technologie Windows Forms. WPF je technologie, která přináší mnoho změn a naučit se ji, znamená pochopit zcela jiný přístup k programovaní desktopových aplikací.

WPF technologie je plně hardwarově akcelerovaná a vykreslována pomocí Direct3D. Veškerá grafika je tvořena vektorově, což znamená, že výsledná

aplikace je nezávislá na rozlišení. Závislost na vykreslovací vrstvě Direct3D s sebou přináší i jednu nevýhodu, závislost na operačním systému Windows. WPF technologie dále plně odděluje design od výkonného kódu a to tak, že zavádí XAML, jazyk, založený na XML sloužící k definici uživatelského rozhraní (zkráceně UI). Designéři tak mohou paralelně pracovat s programátory, každý nezávisle na jiné vrstvě [wik(2013b)].

Díky přepracovanému objektovému modelu lze také snadněji měnit vzhled grafických komponent či dokonce celého okna pomocí stylů či šablon. Mezi nejdůležitější změny je nutné zahrnout také binding, tedy možnost provázat uživatelské rozhraní s výkonným kódem tak, aby obě vrstvy byly na sobě nezávislé. [Chris Sells(2007)]

Tato technologie je propagována společností Microsoft. Delší křivku učení kompenzuje znovupoužitelnost nabytých znalostí i pro další technologie jako je například Silverlight, Windows Phone a aplikace na systém Windows RT.

3.2.1 XAML

XAML je značkovací jazyk. Jedná se o XML derivát, podobně jako např. HTML jazyk. Slouží k popisu uživatelského rozhraní aplikace. Cokoliv zapsané v jazyce XAML je možné zapsat i v jazyce C#. Zkompilované XAML a C# soubory obsahující totožné prvky uživatelského rozhraní se nijak neliší. Vše zapsané pomocí jazyka XAML má svůj ekvivalent v jazyce C#. Důvodem k jeho využití je jednak striktní oddělení aplikační logiky od jejího rozhraní (vzhled je snadno modifikovatelný a přístupný bez vlivu na logiku), především však je to XML struktura jazyka XAML, díky které může být obsah libovolně zpracováván v komplexních vývojových prostředích (Microsoft Expression Studio). S využitím knihoven WPF, pro které byl ostatně jazyk XAML navržen, lze vytvářet jakýkoliv vektorový grafický obsah a dále jej využít v aplikaci. Pomocí jazyka C# lze (přirozeně) též přistupovat k WPF knihovnám, děje se tak ale pouze pokud je třeba obsah dynamicky měnit za běhu aplikace, definice rozsáhlých statických struktur je možná, ale velmi nepohodlná. [Kraft(2011)]

4 Technika programování

Před počátkem implementace jakékoliv aplikace je vhodné se pozastavit a promyslet, k čemu bude aplikace sloužit. Zda bude do budoucna rozvíjena, či jednorázově použita, například pro import dat. Na mnoha prezentacích lze pozorovat, jak lze velice snadno a rychle vyvinout aplikaci pomocí té či oné technologie, ale pokud začneme stavět reálnou aplikaci, mohly brzy by vyvstat potíže.

V této kapitole se pokusím rozebrat oba dva přístupy a popsat, v čem je jejich síla.

4.1 Rychlý vývoj

Tento přístup lze aplikovat v případě potřeby aplikace pro jednorázovou akci či potřeby rychlého vývoje prototypu. Pro tento způsob lze bez obtíží použít obě výše zmíněné technologie.

Výhody

- rychlost
- není nutná hlubší znalost technologie
- vhodné pro jednorázové aplikace

Nevýhody

- GUI je silně provázané s výkonným kódem
- nemožnost paralelního programování
- nepřehlednost
- problematické rozvíjení aplikace
- nemožnost testování

Pomocí designeru lze navrhnout grafické uživatelské rozhraní (GUI) a implementovat handlery, v *code behind* si vytáhnout potřebná data, která následně lze rychle zobrazit přiřazením do některého z grafických prvků na GUI.

4.2 Návrhové vzory

Alternativou k rychlému vývoji je pak využití již známých a léty prověřených návrhových vzorů. Jedním z nejznámějších je Model-View-Controller (zkráceně MVC).

Aplikace je rozdělena do tří oddělených bloků, které jsou na sobě nezávislé. Jednotlivé vrstvy lze modifikovat bez dopadu změn na vrstvy ostatní či vrstvy plně nahradit jinými.

Model reprezentuje bussiness logiku aplikace a samotná data, která se mají prezentovat. *View* tvoří prezentační vrstvu, která zobrazuje uživateli poskytnutá data a *Controller* je prostředník, který má na starosti veškeré toky událostí v aplikaci [Bernard(2013)].

Vzhledem k tomu, že WPF podporuje tzv. Databinding, lze tento model „ušít na míru" právě této technologii.

4.2.1 Model-View-ViewModel

Model-View-ViewModel (zkráceně MVVM) vychází z principů vzoru MVC a byl navrhnut architektem WPF/Silveright Johnem Gossmanem.

- Představuje určitého prostředníka, který adaptuje Model pro potřeby View.
- Je zodpovědný za stav View, avšak bez nutnosti přímého zjišťování hodnot jednotlivých prvků ve View.
- Volá model nebo služby přes jejich interface.
- Vystavuje veřejné vlastnosti, které jsou v XAML bindovány. Aby mohlo docházet ke komunikaci mezi View a ViewModelem, je vhodné aby

ViewModelem tyto veřejné vlastnosti byly publikovány jako notifikační. To tedy znamená, že:

- jednoduché typy jsou vystaveny pomocí vlastností typu DependencyProperty, případně je na nich implementován interface INotifyPropertyChanged
- kolekce prvků jsou vystaveny pomocí ObservableCollection<T> nebo jejích potomků
- pro příkazy, které vedou na volání metod modelu/služeb, jsou zpřístupněny commandy přes implementované rozhraní ICommand

Výsledek je takový, že *View* se přidělí *ViewModel* jako DataContext a např. tlačítku se přiřadí, že po stisku má vyvolat příkaz **BackCommand**. At' už je příkaz definován či ne, na kompilaci *View* nemá vliv. Tlačítko pouze ví, že má volat příkaz s názvem **BackCommand** a o více se nezajímá.

Analogicky např. pro GridView nastavíme, kde má hledat seznam položek, které má zobrazit a kam má uložit referenci právě aktivního řádku a o víc se GridView nezajímá. Jakmile bude z ViewModelu informován o změně seznamu položek, tak se sám obnoví.

Vzhledem k této nezávislosti lze prostě odmazat prvek GridView a zaměnit jej za ComboBox bez jakékoliv změny na ostatních vrstvách [Jirava(2010)].

4.2.2 Inversion of Control

Technika Inversion of Control (IoC) je založena na myšlence vytvářet jednotlivé třídy jako na sobě nezávislé komponenty, které jsou programovány oproti rozhraní a komunikují mezi sebou jen a pouze přes ně.

Samotné propojení jednotlivých komponent je pak ponecháno na tzv. IoC kontejneru. Tento kontejner je třída, která ví, jakou má vrátit instanci pro dané rozhraní a tento úkol je pouze v její režii. Důsledkem tohoto stylu programování je absence klíčového slova *new* v kódu a veškerá inicializace je ponechána na kontejneru.

Výhodou tohoto programování pak je snadná výměna komponent jedné za druhou, například za třídu sloužící k testování.[Augustýn(2010)]

Výhody

- vytváření instancí na základě konfigurace
- nastavování životnosti objektů
- tvoření tříd jako samostatné komponenty

Nevýhody

- nutnost konfigurace kontajneru

4.2.3 Dependecies Injection

Dependecies Injection (zkraceně DI) je návrhový vzor, který definuje způsob získávání objektů potřebných k jeho činnosti. V praxi se nejčastěji používají dva typy injekce. Prvním je *Constructor Injection*, kde k vytvoření instance objektu je potřeba mít již vytvořené instance parametrů konstruktoru. Druhým typem je *Setter Injection*, kde se vytvoří instance objektu a naplní se všechny její metody označené jako *setter*. [Augustýn(2010)]

Techniky *IoC* a *DI* se často využívají pospolu a ve výsledku znamenají, že parametry konstruktorů budou rozhraní a kterou instancí budou doplněny, tak se ponechá na konfiguraci IoC kontajneru.

Caliburn.Micro je odlehčený, ale robustní M-V-VM framework, a **Managed Extensibility Framework** je implementací technik IoC/DI. Největší silou obou těchto nástrojů je jejich snadné propojení a vzájemná kooperace.

4.3 Frameworky

Následující kapitola popisuje frameworky postavené na výčtu návrhových vzorů z kapitoly 4.2

4.3.1 Caliburn.Micro

Aplikace jakéhokoliv návrhového vzoru na aplikaci bude znamenat jistou režii navíc. Tento investovaný čas na počátku se nám brzy vrátí při budoucích úpravách aplikace, kdy při rychlém vývoji by každý zásah znamenal mnoho programátorských hodin. Tato režie je ale víceméně pořád stejná a např. provazbit View s ViewModelem znamená zkopírovat 5 řádek kódu a pozměnit název cílového ViewModelu.

Pro usnadnění práce byl vytvořen framework Caliburn a jeho mladší a odlehčený „bráška" Caliburn.Micro. Tento framework vznikl v rukách fenomenálního programátora Roba Eisenberga, který postavil celý projekt na myšlence „Convention-over-configuration".

Pokud budou dodržována jistá pravidla, tak framework bude tyto konfigurace řešit sám. Například výše zmíněné provazbení *View* s *ViewModel*em lze zajistit tak, že namespacu *ViewModel*u odmažeme slova **Model** a dostaneme cestu k *View*.

Ukázka úpravy namespace:

```
ViewModel: Zcu.PanelComposer.ViewModels.Pages.HomeViewModel.cs
View: Zcu.PanelComposer.Views.Pages.HomeView.cs
```

Tato pravidla jsou samozřejmě konfigurovatelná. Dále je zde připraveno mnoho utilit jako je například *Event Aggregator* sloužící k asynchronnímu předávání zpráv např. mezi jednotlivými okny či stránkami. Framework podporuje asynchronní tasky a automaticky provazbuje prvky na *View* s vlastnostmi *ViewModel*u.

Caliburn.Micro si také rozumí s Managed Extensibility Frameworkem (MEF) a ve spojení s technikou Inversion of Control (IoC) a Dependency Injection (DI) se z tohoto frameworku stává robustní nástroj, který programátory víceméně nutí programovat oproti interfacům.

Výhodou tohoto frameworku je také to, že je použitelný pro WPF 4.0/4.5, Silverlight 4.0/5.0, Windows Phone 7.1/8.0 a WinRT (Metro). [Eisenberg(2013)]

4.3.2 Managed Extensibility Framework

Původně skupina nadšenců vytvářela framework, který je postaven na technice IoC/DI, ale nebude fungovat tak, že bude pracovat v rámci jedné aplikace, ale umí prohledávat jednotlivé assembly. Byl vymyšlen mechanismus, který pomocí reflexe prohledá celou assembly a vyhledá pouze třídy, které obsahují specifické atributy.

Všechny třídy, které reprezentují dané rozhraní, je nejprve nutné implementovat a následně jim přidat atribut **Export** s typem rozhraní. Analogicky je nutné takto doplnit všechny settery, případně konstruktory, které je třeba plnit při inicializaci objektu a ty poté rozšířit o atribut **Import**.

IoC kontejner pak při spuštění aplikace prohledá všechny dostupné assembly, případně složku, záleží na nastavení, a vytvoří si strom závislostí. Není tedy nutné žádným způsobem kontejner konfigurovat, protože to za nás obstará mechanismus postavený na reflexi.

Tento framework se samotnému Microsoftu natolik zalíbil, že jej zahrnul do .net frameworku od verze 4.0. Pomocí tohoto nástroje lze tak stavět aplikace, které podporují pluginy. [mef(2012)]

5 Komunikace s DB

Jednou z klíčových věcí každé aplikace je ukládání rozpracovaných dokumentů. Tuto funkcionalitu lze zajistit několika možnými způsoby, záleží na způsobu použití těchto dat. Osobně preferuji ukládání dat do lokálních databází, které za nás řeší problémy konzistence a umožňují programátorům soustředit se pouze na klíčové problémy.

Pokud má program spolupracovat s jinými aplikacemi, je vhodné vytvořit funkce pro export či import. Ideální je nepoužívat binární ukládání, protože od tohoto způsobu se v dnešní době upouští. Příkladem může být balík programů Microsoft Windows Office od verze 2007 a vyšší. Při exportu a importu se preferují buď zavedené formáty, nebo využití značkovacího jazyka XML.

Pro desktopové aplikace, u kterých je naplánován provoz bez nutnosti internetového připojení, je vhodné využít některé z lokálních databází. Příkladem může být produkt SQLLite či odlehčená verze Microsoft SQL databáze, často označovaná jako SQL Compact Edition.

5.1 Datasety

Jedním z nejstarších přístupů, avšak dodnes využívaných, jsou datasety. Pomocí množství podpůrných objetků, jako jsou například SqlCommandy, SqlDataReader a mnoho dalších, se dostává programátorům do rukou mocný nástroj pro relativně přímý přístup do databáze. Relativně proto, že vše jde přes perzistentní vrstvu v paměti, která urychluje čtení, umožňuje provést mnoho změn najednou a promítnout vše do databáze.

Práce s datasety znamená velikou režii navíc, ale výhodou pak je, že má programátor veškerý kód pod svojí kontrolou, což se u použití ORM mapperu říci nedá.

Bohužel datasety ve spolupráci s MS SQL Compact Edition trpí jistými nedostatky:

- problémy s návratovými hodnotami (MS SQL CE neumí skládání dotazů)

- nemožnost lazy loadingu
- vyšší režie

Výhodou je pak nejvyšší výkon ze všech dostupných technologií.

5.2 Linq2SQL

S vydáním .net frameworku 3.5 přichází Microsoft s novým integrovaným dotazovacím jazykem **Linq**. Tento jazyk je psán dostatečně obecně a lze je aplikovat na kolekce, slovníky, xml i databáze. Právě poslední zmíněná komunikace s databází byla zpočátku omezena pouze na produkty od Microsoftu, tedy klasickou serverovou databázi a již zmiňovanou kompaktní verzi. Přímo v kódu lze pak skládat dotazy s komfortem samotného Visual Studia či jiného programovacího prostředí (zkráceně IDE) a samotný SQL dotaz je pak složen na pozadí.

Výhody

- komfort IDE
- lazy loading
- možnost urychlení pomocí kompilace dotazů
- odstraňuje problémy se skládání dotazů

Nevýhody

- omezenost na produkty od Microsoftu
- prográmator nemá „pod kontrolu" samotné SQL dotazy
- vazby N:M (není to klasický ORM Mapper)

Linq2SQL znamená odlehčení práce pro programátory, ale tato výhoda s sebou nese samozřejmě jistou režii, a tak je na tom výkonově tato technologie v porovnání s datasety hůře. Pokles výkonu je ale velice nepatrný.

Protože *Linq2SQL* není klasickým ORM mapperem, jeho největší nevýhodou je, že si neumí poradit s vazbami N:M a tak tato práce zůstává na samotném programátorovi.

5.3 Entity framework

Microsoft si byl velice vědom nevýhod technologie *Linq2SQL* a tak hned v následujícím service packu 1 vydal nový produkt, Entity framework (zkráceně EF), který je plnohodnotným ORM mapperem. Tato technologie odstraňuje problém s omezením na MS SQL databáze a vazbami M:N.

Entity framework se stal nástupcem *Linq2SQL* a Microsoft se již nadále vývojem této technologie (*Linq2SQL*) nezabývá. Přístup do databáze je rovněž poskytován za pomoci dotazovacího jazyku **LINQ**.

Výhody

- komfort IDE
- lazy loading
- možnost urychlení pomocí kompilace dotazů
- odstraňuje problémy se skládání dotazů
- vazby N:M
- plnohodnotný ORM mapper
- použitelný pro většinu běžných databází

Nevýhody

- komfort si vybírá daň na výkonu
- prográmator nemá „pod kontrolu" samotné SQL dotazy

Mezi programátory využívající ORM mappery a ostatními, využívající klasický přístup přes datasety či přímý přístup přes Stored procedury existuje odvěká rivalita a obě dvě strany se nemohou shodnout, který způsob je ten jediný správný. ORM mappery usnadňují práci s databázemi a odstíňují programátora od přímé znalosti databáze či T-SQL jazyka. Programátor tak nemusí znát více jazyků a může se soustředit na jeden hlavní. Druhá strana věci je však to, že programátor nemá vliv na výsledný SQL dotaz, což u některých typů aplikací může být velice důležité.

Osobně si myslím, že oba přístupy mají své místo a záleží ryze na typu aplikace.

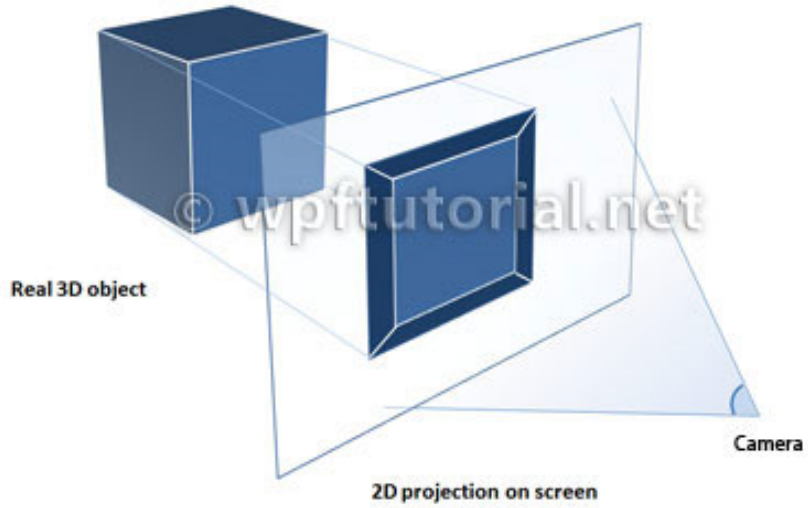
6 WPF a 3D

Celá sekce **WPF a 3D** byla vypracována v rámci předmětu Projektu 5 a bylo čerpáno zejména z knihy „Programming WPF. 2nd ed.". [Chris Sells(2007)]

Tato technologie je úzce vázaná na Windows a DirectX. Zatímco Windows Forms trpěly tím, že byly napřímo rasterizovány, WPF technologie jde cestou opačnou, tedy vše vykreslit vektorově a zobrazení ponechat na grafické kartě. Grafika je proto hardwarově akcelerovaná, ale to neznamená, že by WPF byla tou správnou volbou pro tvorbu náročných 3D aplikací či her. 3D se zde využívá pro vytvoření zajímavých efektů či zobrazení jednoduchých modelů.

Udává se, že by modely neměly přesahovat více než 20 000 bodů a 60 000 trojúhelníků. Pro náročnější aplikace je vhodné zvolit technologii jinou.

Všechny prvky ve WPF lze zapsat deklarativně pomocí *XAMLu* či procedurálně v kódu. Oba způsoby lze také kombinovat pomocí bindingu. [Microsoft(2013)]



Obrázek 6.1: Projekce [Mosers(2013)]

6.1 Základy 3D

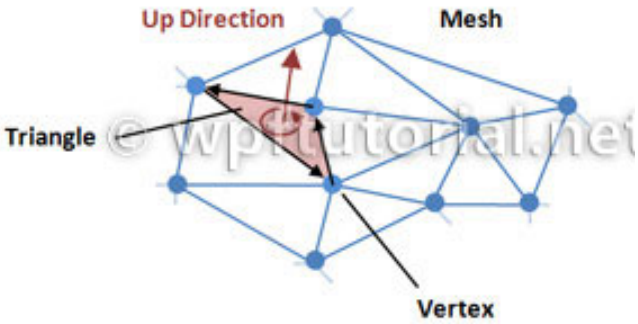
Základní stavební kámen 3D grafiky je počítačový model jakožto **objekt**. Protože naše obrazovky jsou ale dvourozměrné, je nutné tyto objekty nějakým způsobem převést na 2D obraz (viz obr. 6.1). Tento převod se nazývá **rendering**. Pro rendering je nutné definovat, odkud se na model díváme, tedy určit pozici **kamery**. Abychom vůbec něco mohli pozorovat, je nutné také definovat zdroj **světla**.

A protože svět kolem nás je plný barev, je také vhodné nadefinovat **materiál** daného objektu. Např. ho můžeme obarvit jednou barvou nebo zobrazit tak, aby vypadal jako dřevo. Protože dřevo není jednobarevné, použijeme například obrázek dřeva, tedy **texturu**.

Svět v počítačové grafice je složen z trojúhelníků a dnešní hardware je uzpůsoben právě k tomu, aby uměl s trojúhelníky pracovat rychle. [Mosers(2013)]

Vlastnosti trojúhelníků

- je vždy konvexní
- definuje plochu/rovinu
- každý polygon lze definovat jako sadu trojúhelníků (princip triangulace)



Obrázek 6.2: Popis meshe [Mosers(2013)]

Jakýkoliv objekt lze (s jistou odchylkou) rozložit na síť trojúhelníků. Tato síť se nazývá **mesh** (viz obr. 6.2). Síť je definován výčtem bodů ve

3D prostoru. Tyto body se nazývají vrcholy (angl. vertex, plur. **vertices**). Jednotlivé vrcholy se spojují do tzv. trojúhelníků (angl. **triangle(s)**). Každý trojúhelník má dvě strany, přední a zadní. Přední strana je definována pomocí takzvaného pravidla pravé ruky, tedy záleží na pořadí, v jakém jsou trojúhelníky skládány.

6.2 3D Kontejner

Základní prvek WPF pro zobrazení trojrozměrného prostoru je **Viewport3D**. Tento prvek definuje prostor v obrazovce, pomocí něhož se pohlíží na scénu do světa 3D.

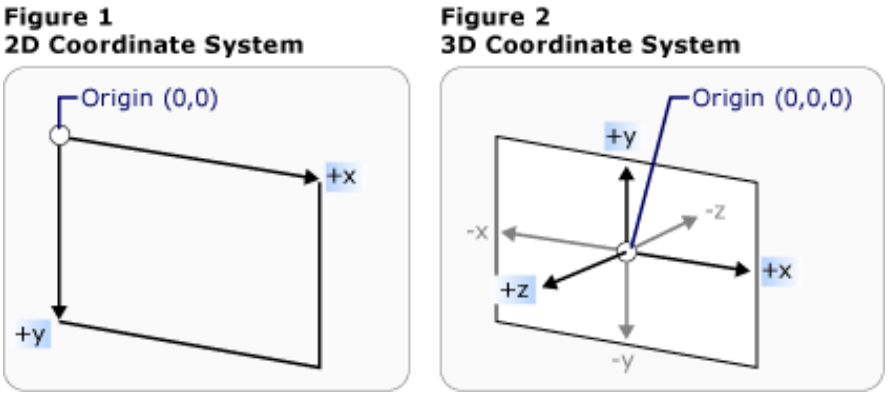
Viewport3D v sobě zapouzdřuje všechna data o nastavení kamery, světel a všech 3D modelech popř. informace o jejich transformaci. K tomu, abychom zobrazili trojrozměrný prostor, musíme nadefinovat tři věci. Kameru, alespoň jeden zdroj světla (jinak bychom nic neviděli) a model.

6.3 Souřadnice v prostoru

Vykreslování ve dvourozměrném prostoru je zajištěno z levého horního rohu renderovací oblasti (typicky pak obrazovky). X-ová osa je kladná ve směru z levé strany do pravé strany, Y-ová osa pak shora dolů, jak je naznačeno na obr. 6.3. Toto využijeme zejména u texturování.

V trojrozměrném systému je tomu jinak. Počáteční bod je uprostřed renderovaní části, x-ová osa je shodná s vykreslováním v 2D prostoru, ale Y-ová osa je obráceně, tedy kladná směrem nahoru a záporná směrem dolů. Z-ová osa je pak kladná směrem k pozorovateli.

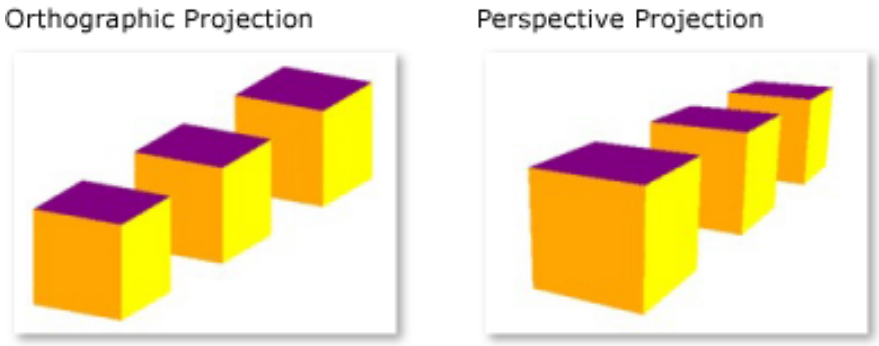
Toto využijeme zejména u sestavování 3D modelu.



Obrázek 6.3: Souřadnice v prostoru [Microsoft(2013)]

6.4 Kamery

Viewport3D v současné době umožňuje nastavení tří typů kamer: PerspectiveCamera, OrthographicCamera a MatrixCamera (viz obr. 6.4). Pomocí těchto kamer můžeme měnit pohled na trojrozměrný svět.



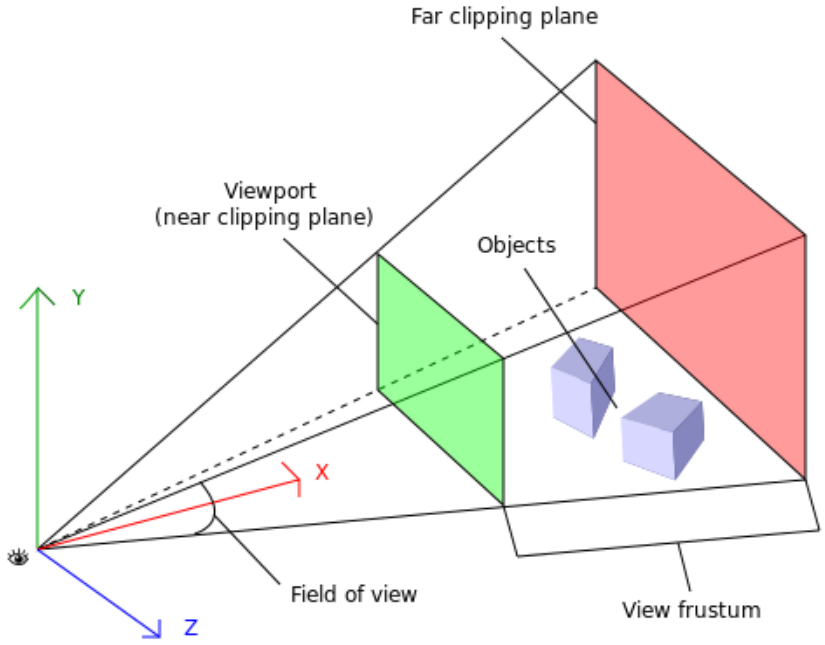
Obrázek 6.4: Rozdíl mezi projekcemi [Microsoft(2013)]

Lidskému oku nejpřirozenější je kamera perspektivní, která vzdálené objekty zobrazuje menší. Tento pohled se nejvíce přibližuje reálnému světu.

Oproti tomu kamera ortografická je jednoduchá, zobrazuje objekty tak, jak si je nadefinujeme, a nedochází k žádné deformaci. Tento pohled se využívá u CAD systému, kde nám nejde o reálný pohled, ale o skutečné rozměry, dle kterých budeme řídit konstrukce.

6.4.1 Perspektivní kamera

Základní nastavení této kamery je její umístění - *Position* a směr - *LookDirection*, kterým budeme hledět na trojrozměrný prostor. Perspektivní kameru si lze představit jako jehlan, který má vrchol v bodu, který jsme nastavili jako pozici kamery, směřuje na 3D-scénu. U tohoto pohledového jehlanu můžeme navíc nastavit úhel – *FieldOfView*, tedy záběr kamery. S kamerou lze také otáčet pomocí parametru *UpDirection*, který nastavuje směr vzhůru do kamery. Defaulftně je tento parametr nastaven v kladném směru osy Y.



Obrázek 6.5: Jehlan perspektivní kamery [libGDX(2012)]

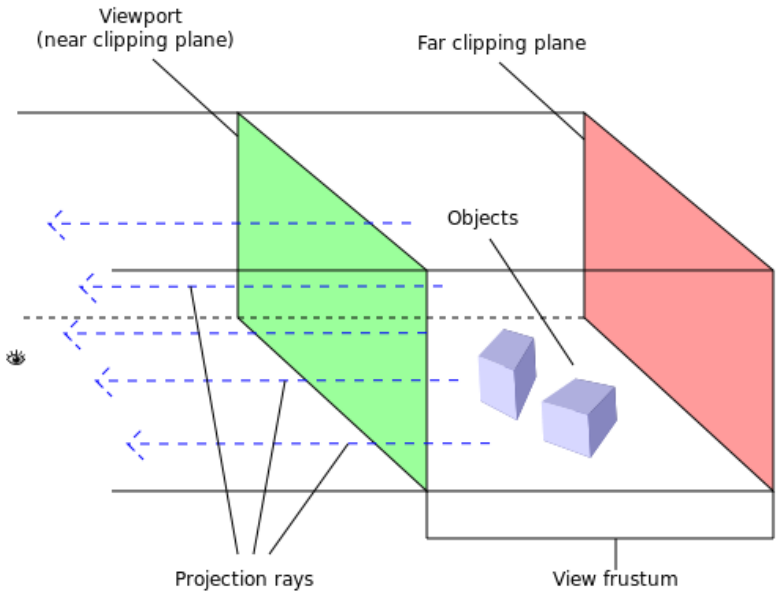
Závěrem je třeba nastavit ořezávání jehlanu pomocí dvou ploch. První, ta bližší, udává plochu, před kterou nebude žádný objekt vykreslen, zadní naopak schová vše za ní (*NearPlaneDistance* a *FarPlaneDistance*).

Listing 6.1: Ukázka nastavení perspektivní kamery

```
<Viewport3D>
  <Viewport3D.Camera>
    <PerspectiveCamera UpDirection="0,1,0" Position="0,0,5"
       LookDirection="0,0,-1" FieldOfView="45"
       NearPlaneDistance="3" FarPlaneDistance="50" />
  </Viewport3D.Camera>
</Viewport3D>
```

6.4.2 Ortografická kamera

Zásadní rozdíl mezi perspektivní a ortografickou kamerou je způsob ořezávání prostoru. Zatímco perspektivní kamera využívá jehlan, ortografická si vystačí s kvádrem.



Obrázek 6.6: Kvádr ortografické kamery [libGDX(2012)]

Způsob nastavení kamery je tedy velice podobný s tím rozdílem, že nenastavujeme úhel, ale pouze šířku - *Width*, kterou chceme zobrazit.

Listing 6.2: Ukázka nastavení ortografické kamery

```
<Viewport3D>
  <Viewport3D.Camera>
    <OrthographicCamera UpDirection="0,1,0" Position="0,0,5"
       LookDirection="0,0,-1" Width="10" NearPlaneDistance="3
       " FarPlaneDistance="50" />
  </Viewport3D.Camera>
</Viewport3D>
```

6.4.3 Maticová kamera

Posledním typem kamery je kamera maticová. Tato kamera je zde spíše kvůli přenosu kamery mezi jednotlivými 3D formáty, které právě tímto způsobem uchovávají nastavení.

Maticová kamera se nastavuje pomocí dvou matic typu 4x4. Prvním parametrem je *ViewMatrix*, tento parametr nastavuje pozici a orientaci kamery. Svým způsobem nahrazuje parametry *Position*, *UpDirection* a *LookDirection* z předchozích typů kamer. Druhým parametrem je pak *ProjectionMatrix*, kterým nastavujeme, jak bude 3D prostor transformován do 2D obrazu.

6.5 Modely

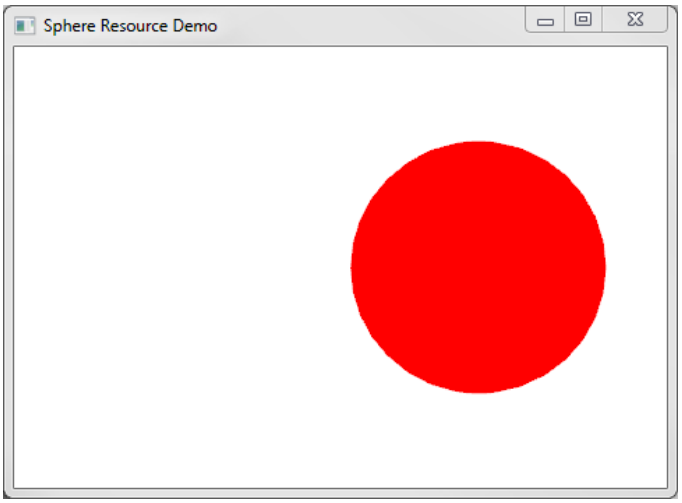
Model je reprezentace 3D modelu pomocí trojúhelníků. Viewport3D smí obsahovat pouze jednu kameru, ale „neomezené" množství modelů. Tyto modely jsou zapouzdřeny do objektu ModelVisual3D. Obsahem tohoto prvku mohou být světla nebo geometrická reprezentace, tedy popis povrchu (**meshe**) pomocí trojúhelníků. Speciálním prvkem je pak Model3DGroup, který umožňuje tyto prvky kombinovat. Např. lze vytvořit geometrickou reprezentaci koule, která bude zároveň svítit. Pomocí transformace celého ModelVisual3D prvku lze docílit pohybu po obloze.

6.6 Světla

Ve WPF jsou k dispozici celkem 4 typy světel. Světla budou popisovány od výkonově nejméně náročné až po nejnáročnější.

6.6.1 Ambientní světlo

Jde o základní a nejjednodušší světlo. Osvětlí všechny objekty stejně bez ohledu na pozici či orientaci. Vlivem tohoto světla nejsou vidět vůbec stíny, a tak červená koule vypadá jako vyplněná rudá kružnice. Výhodou je samozřejmě výpočetní náročnost. Jediným parametrem tohoto světla je barva.



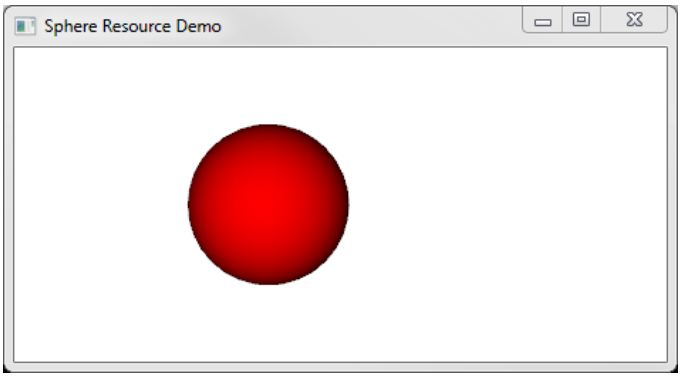
Obrázek 6.7: Koule pod ambientním světlem

Listing 6.3: Nastavení ambientního světla

```
<AmbientLight Color="White" />
```

6.6.2 Směrové světlo

Jak už sám název napovídá, jedná se o směrové světlo a lze si ho představit jako zářící slunce. Neudává se tedy pozice, ale pouze směr záření. Po osvětlení modelu se již přibližujeme realitě. Objekty jsou stínované, ale nevrhají stíny mezi sebou.



Obrázek 6.8: Koule pod směrovým světlem

Listing 6.4: Nastavení směrového světla

```
<DirectionalLight Color="White" Direction="0,0,-1" />
```

6.6.3 Bodové světlo

Bodová světla patří do lokálního osvětlovacího modelu, tedy lze pomocí něj simulovat např. světlo pouličních lamp či lustr. Základní nastavení bodového světla je jeho pozice a barva. Tento typ patří mezi ty výkonově náročnější, ale také umožnuje komplexnější nastavení.

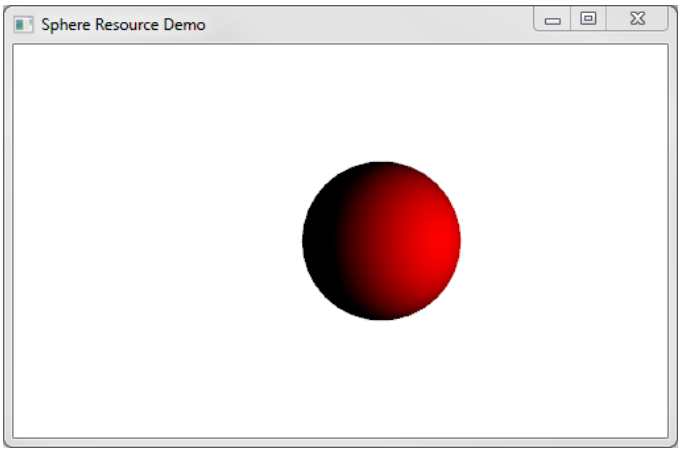
Existují zde tři koeficienty *ConstantAttenuation (CA)*, *LinearAttenuation (LA)* a *QuadraticAttenuation (QA)*. Výsledné zeslabení světla se pak počítá podle vzorce 6.1, kde *distance* je zdálenost od zdroje světla.

$$light = CA + LA * distance + QA * distance^2 \tag{6.1}$$

Dle nastavení těchto parametrů lze ovlivnit svítivost.

Listing 6.5: Nastavení bodového světla

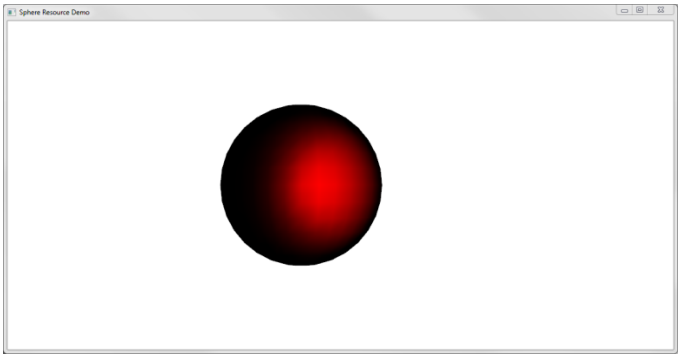
```
<PointLight Color="White" Position="0,0,0" Range="30"
   ConstantAttenuation="0.5" LinearAttenuation="0.05"
   QuadraticAttenuation="0.005" />
```



Obrázek 6.9: Koule pod bodovým světlem

6.6.4 Reflektorové světlo

Reflektorové světlo se v zásadě moc neliší od bodového světla. Rozdíl je v šíření světla. Zatímco bodově světlo je všesměrové, tak reflektorové světlo svítí pouze směrem jedním.



Obrázek 6.10: Koule pod reflektorovým světlem

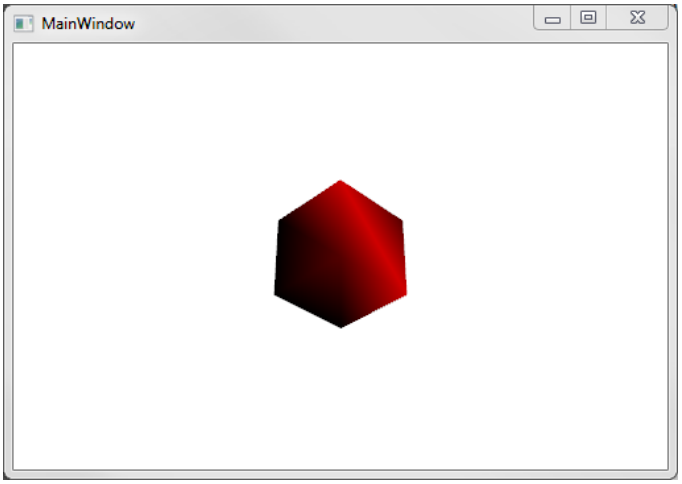
Zásadním parametrem je tedy směr světla – *direction*, osvětlení lze dále ovlivnit pomocí dvou úhlů. *InnerConeAngle* specifikuje oblast, ve které bude světlo v plné intenzitě. V oblasti mezi *InnerConeAngle* a *OuterConeAngle* se bude intenzita postupně snižovat.

Listing 6.6: Nastavení reflektorového světla

```
<SpotLight Color="White" Position="0,0,0" Direction="0,0,-1"
   InnerConeAngle="45" OuterConeAngle="90" />
```

6.7 Geometrická reprezentace

K popisu samotného modelu v 3D prostoru slouží prvek *MeshGeometry3D*. V současné době neexistuje jiná možnost, jak popsat model v technologii WPF. Tento prvek popisuje povrch modelu pouze pomocí sady vrcholů, jednotlivých trojúhelníků, normál a koordinátů pro texturování.

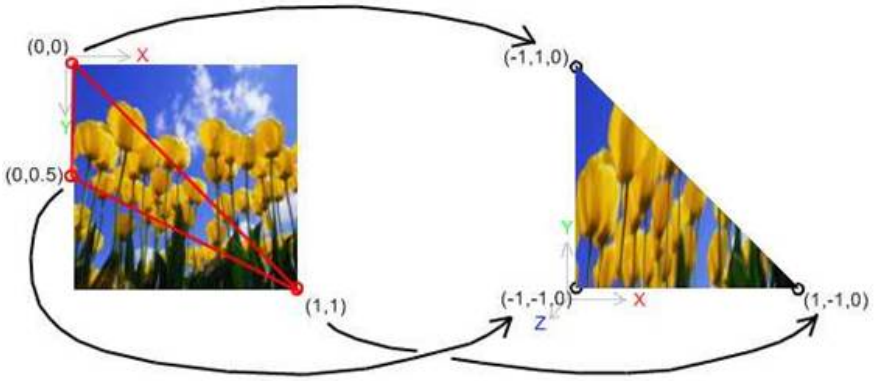


Obrázek 6.11: Krychle

Zatímco např. technologie OpenGL umožňuje i složitější konstrukce než trojúhelníky (strips, fans, polygons, quads aj.), tak WPF technologie podporuje pouze trojúhelníky. Pokud je třeba tyto primitiva načíst, tak je nutné je manuálně převést. U sestavování trojúhelníků je nutné si dát pozor na „pravidlo pravé ruky". Při nastavování koordinátů pro textury je nutné si uvědomit, že 2D má jiný souřadný systém než 3D viz obrázek 6.12.

Listing 6.7: Geometrická reprezentace krychle

```
<GeometryModel3D.Geometry>
  <MeshGeometry3D Positions="-1,1,1  -1,-1,1  1,-1,1  1,1,1
     -1,1,-1  -1,-1,-1  1,-1,-1  1,1,-1"
    TriangleIndices="0,1,2  0,2,3
      3,2,6  3,6,7
      4,5,1  4,1,0
      7,6,5  7,5,4
      1,5,6  1,6,2
      4,0,3  4,3,7" />
</GeometryModel3D.Geometry>
```



Obrázek 6.12: Mapování textur [Sonnino(2007)]

6.8 Materiál

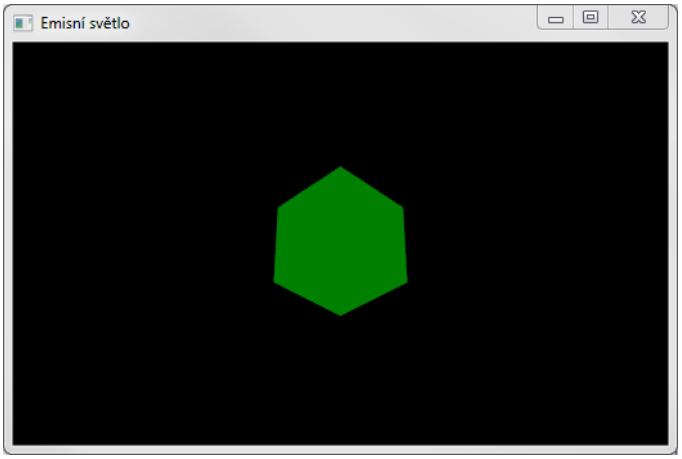
Nastavením kamer, světel a vytvořením grafické reprezentace, např. krychle, ještě není trojrozměrná reprezentace viditelná. Důvodem je to, že krychle nemá žádný materiál, tedy není co vykreslovat.

Materiálem může být 2D kresba, barva, a nebo část uživatelského prostředí např. tlačítko. Jednotlivé materiály lze mezi sebou kombinovat a to tak, že je zanoříme do prvku *MaterialGroup*. Základem každého materiálu je parametr *Brush*.

Typy materiálů jsou seřezeny od výkonově méně náročných až po ty složitější.

6.8.1 Emisní materiál

Zatímco ostatní typy materiálu potřebují světlo k tomu, aby byly vidět, tak emisní materiál sám o sobě „svítí". Ne že by osvětloval okolní objekty, ale lze jej pozorovat bez zdroje světla.



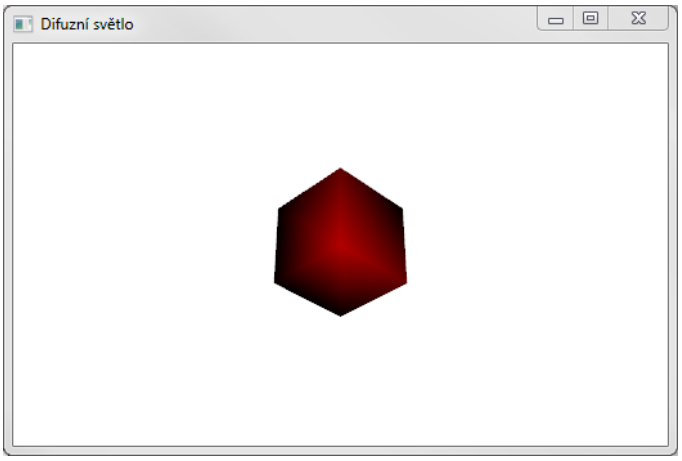
Obrázek 6.13: Nastavení emisního materiálu

Listing 6.8: Nastavení emisního materiálu

```
<EmissiveMaterial Brush="Green" />
```

6.8.2 Difuzní materiál

Difuzní materiál popisuje tu oblast modelu, kam dopadá světlo. Tato část bude vykreslená matným materiálem, který lze nadefinovat pomocí parametru *Brush*. Oblast, kam nedopadá světlo, bude tmavá (stín). K tomu, abychom takto potažený objekt viděli, musí existovat světlo.



Obrázek 6.14: Krychle potažená difuzním materiálem

Listing 6.9: Nastavení difuzního materiálu

```
<DiffuseMaterial Brush="Red"/>
```

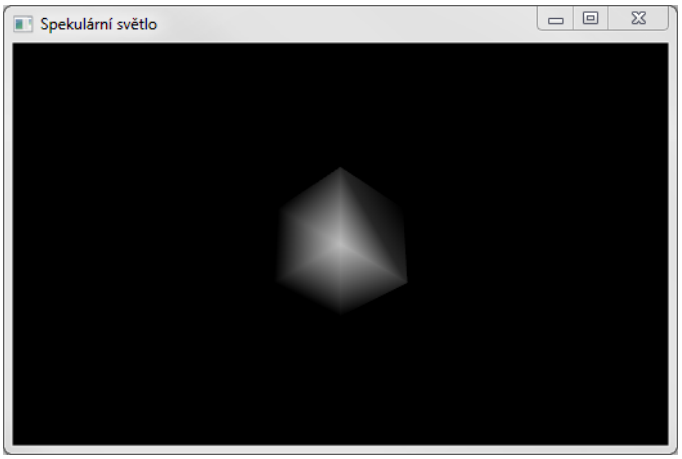
6.8.3 Spekulární materiál

Tento typ materiálu definuje lesklý povrch. Zobrazuje odlesky světla a materiál pak více připomíná reálný objekt. Tento typ materiálu se obvykle používá v kombinaci s difuzním materiálem.

Spekulární materiál lze pozorovat pouze pod bodovým nebo reflektorovým světlem. Tento materiál obsahuje ještě jeden parametr navíc. Nazývá se *SpecularPower* a určuje sílu odrazu světla. Čím nižší číslo je, tím je odraz silnější.

Listing 6.10: Nastavení spekulárního materiálu

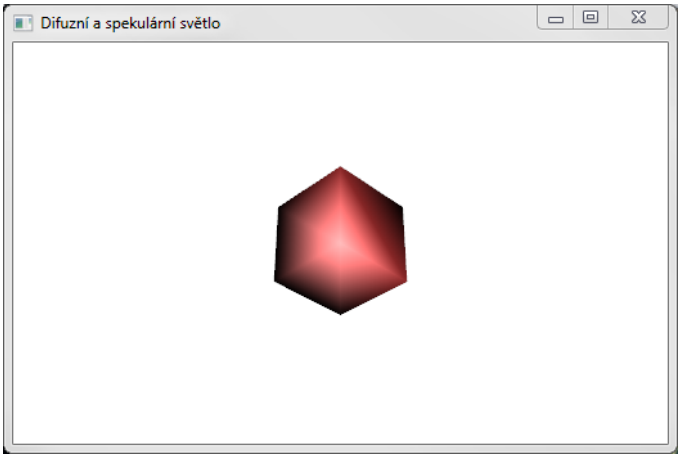
```
<SpecularMaterial Brush="White" SpecularPower="2" />
```



Obrázek 6.15: Krychle potažená spekulárním materiálem

6.8.4 Kombinované materiály

Jednotlivé materiály lze mezi sebou kombinovat pomocí prvku *MaterialGroup*. Výsledným efektem je pak reálnější zobrazení objektu.



Obrázek 6.16: Kombinace difuzní a spekulárního materiálu

Listing 6.11: Nastavení MaterialGroup prvku

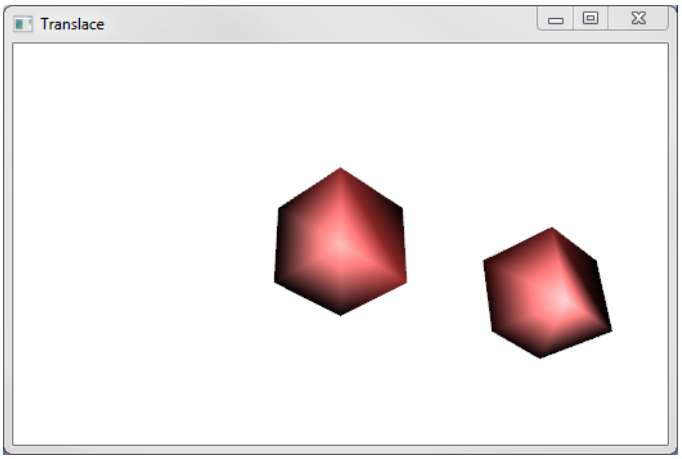
```
<GeometryModel3D.Material>
  <MaterialGroup>
    <DiffuseMaterial Brush="Red"/>
    <SpecularMaterial Brush="White" SpecularPower="2" />
  </MaterialGroup>
</GeometryModel3D.Material>
```

6.9 Transformace

Jednotlivé modely či celou scénu lze transformovat. Tedy pokud je krychle definována v počátku souřadné soustavy a je potřeba ji posunout, otočit či zmenšit/zvětšit, není nutné redefinovat souřadnice, ale lze použít již stávající a pouze je transformovat.

6.9.1 Translace

Translace neboli posuv. Touto transformací lze pohybovat modelem po jednotlivých osách X, Y nebo Z. Jednotlivé složky pohybu se nastavují pomocí offsetu. Pohyb je zadán relativně. Při pozorování translace je nutné mít správně nastavenou kameru.



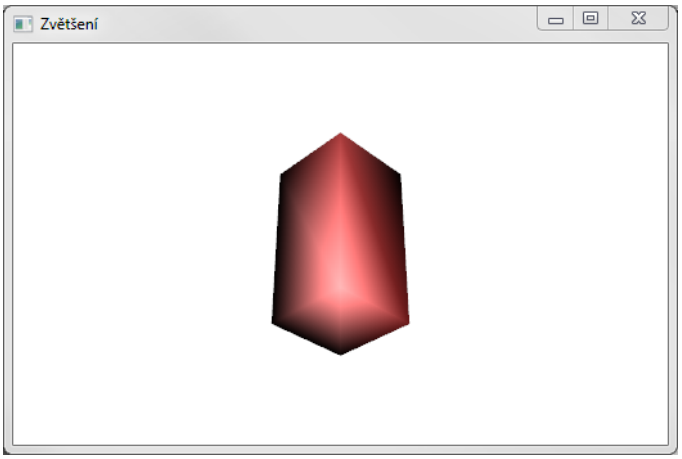
Obrázek 6.17: Ukázka translace

Listing 6.12: Nastavení translace po ose X a Z

```
<TranslateTransform3D  OffsetX="5" OffsetZ="2"/>
```

6.9.2 Změna velikosti

Scaling neboli změna rozměrů. Tato transformace obsahuje celkem 3 parametry začínající slovem *Scale*. Následuje písmeno označující osu, dle které bude objekt zmenšen či zvětšen. Snadno tak lze z krychle vytvořit kvádr.



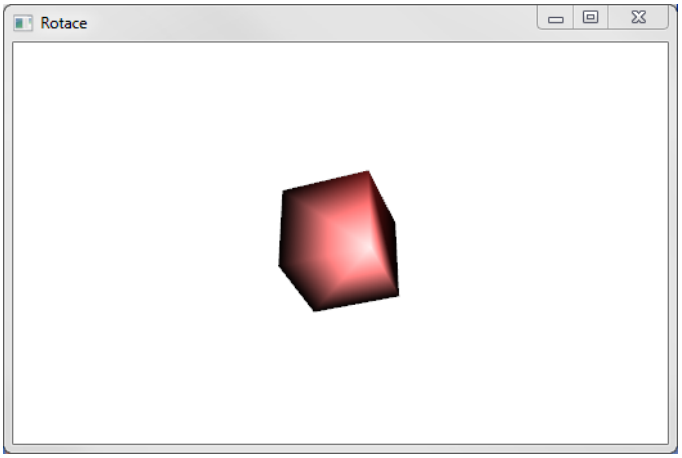
Obrázek 6.18: Dvojnásobné zvětšení po ose Y

Listing 6.13: Nastavení zvětšení

```
<ScaleTransform3D ScaleY="2" />
```

6.9.3 Rotace

Poslední transformací je rotace. Nejprve je nutné zadat osu, dle které bude objekt rotován a pak úhel. Úhel se zadává ve stupních.



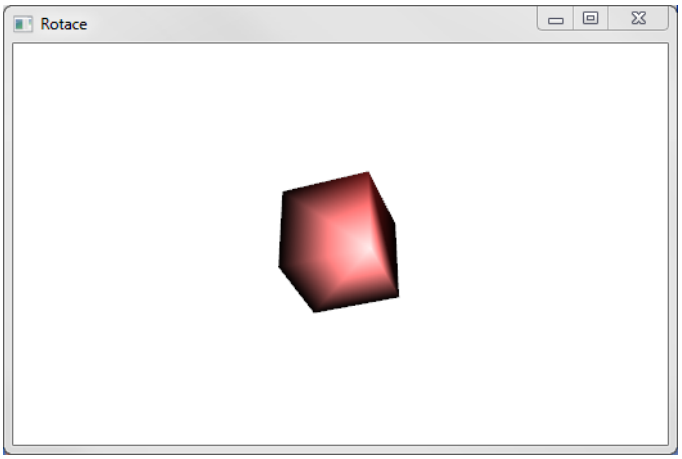
Obrázek 6.19: Rotace krychle o 25 stupňů v ose X

Listing 6.14: Nastavení rotace

```
<RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D Axis="0,1,0" Angle="25" />
  </RotateTransform3D.Rotation>
</RotateTransform3D>
```

6.9.4 Skládání transformací

Jednotlivé transformace se také dají skládat pomocí prvku Transform3DGroup, ale z hlediska výkonnostního se to moc nedoporučuje. Jako alternativa se pak nabízí transformace maticová.



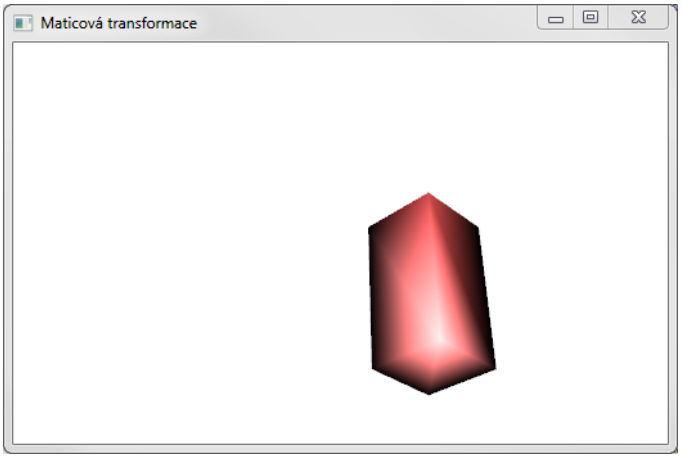
Obrázek 6.20: Kombinace transformací

Listing 6.15: Nastavení skládání transformací

```
<Transform3DGroup>
  <ScaleTransform3D ScaleY="2" />
  <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D Axis="0,1,0" Angle="25" />
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</Transform3DGroup>
```

6.9.5 Maticová transformace

Maticová transformace umožňuje veškeré akce jako předchozí transformace a co víc, není třeba využívat Transform3DGroup, což umožňuje vyšší výkon.



Obrázek 6.21: Maticová tranformace

Jediným parametrem této transformace je *Matrix*, tedy 4x4 matice. Prvními třemi čísli na diagonále lze nastavit *scaling* a poslední řádka (tedy první 3 čísla) nastavuje posuv.

Listing 6.16: Nastavení dvojnásobného zvětšení po Y a posuv o 3 v ose X

```
<MatrixTransform3D Matrix="
1,0,0,0,
0,2,0,0,
0,0,1,0,
3,0,0,1" />
```

7 Implementace

V následující kapitole budu popisovat implementovanou architekturu aplikace. U složitějších algoritmů se pozastavím a blíže popíši danou problematiku.

7.1 Výběr nástrojů

Aplikace je vyvinuta na platformě Microsoft .NET verze 4.0 s využitím technologie **WPF** (viz kapitola 3.2). Jako programovací jazyk byl zvolen C#. Technologie **WPF** byla zvolena proto, že umožňuje snadno oddělovat aplikační vrstvu s vrstvou prezenční. Toto oddělení bylo navíc podpořeno doporučeným návrhovým vzorem M-V-VM (viz kapitola 4.2.1). Pro snazší implementaci byl použit jednoduchý framework **Caliburn.Micro** (viz kapitola 4.3.1) a pro snazší dekompozici je podpořen dalším frameworkem **MEF** (viz kapitola 4.3.2).

Výhodou všech těchto technik, technologií a frameworků je to, že umí spolu úzce spolupracovat a po jejich pochopení usnadňují vývojáři mnoho práce a také udržují kód přehledný, čímž je splněn požadavek na rozšiřitelnost a znuvupoužitelnost aplikace.

Pro samotný vývoj byly využity tyto nástroje:

- Microsoft Visual Studio 2010
- Microsoft SQL Server Management Studio (v. 10.50.1617.0)
- .NET framework 4.0

Dále byly využity tyto knihovny třetích stran:

- Helix 3D Toolkit - snazší manipulace s 3D objekty [objo(2013)]
- log4net - pomocná knihovna pro logování [Apache.org(2013)]

Nároky samotné aplikace nejsou vyšší než nároky na operační systém Windows s nainstalovaným .NET frameworkem 4.0.

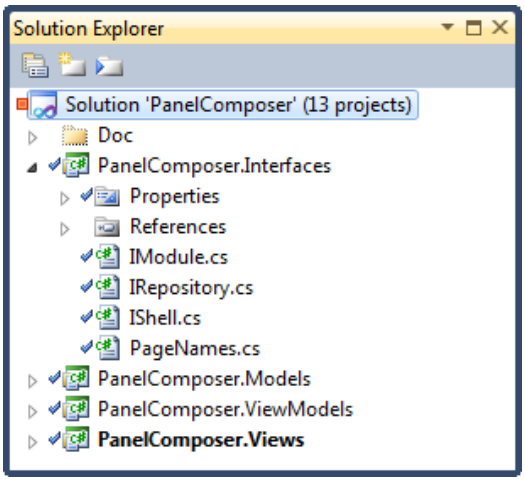
7.2 Struktura aplikace

Aplikace je rozdělena do 4 samostatných částí. Každá je vytvořena v úplně oddělené sestavě (angl. assembly).

- PanelComposer.Interfaces
- PanelComposer.Models
- PanelComposer.ViewModels
- PanelComposer.Views

7.2.1 Interfaces

Knihovna *Interfaces* obsahuje sdílené rozhraní a výčet obrazovek (viz obr. 7.1). Tato knihovna je sdílena mezi zbylými třemi knihovnami.

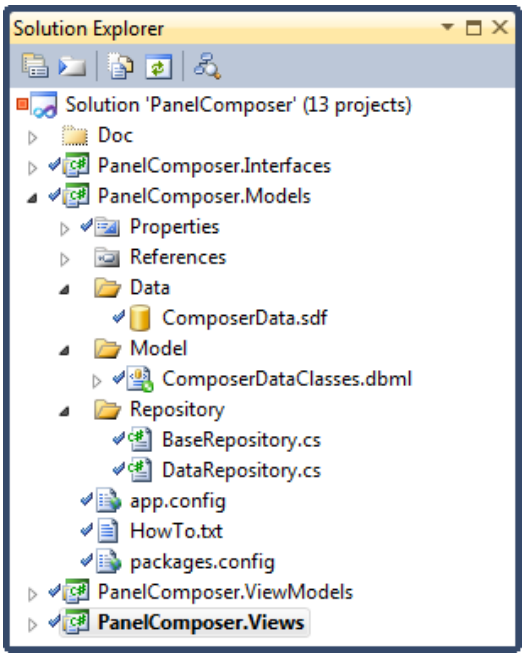


Obrázek 7.1: Obsah knihovny Interfaces

7.2.2 Models

Knihovna *Models* (viz obr. 7.2) umožňuje komunikaci s lokální databází a obsahuje implementaci rozhraní **IRepository**. **Repository** je návrhový vzor, který sjednocuje přístup k různým entitám v databázi. Tato implementace je generická, lze ji tedy používat pro všechny entity stejně. Rozhraní **IRepository** obsahuje CRUD operace nad databází.

Samotná databáze se nachází ve složce data. Jedná se o lokální databázi MS SQL Compact Edition, kterou lze používat jako snadné uložiště dat. K samotné databázi pak aplikace přistupuje pomocí **LINQ2SQL** (viz kapitola 5.2). Protože Visual Studio 2010 neumí vytvořit datový model, tzv. *DataClasses*, je nutné tento model generovat pomocí utility *sqlmetal*. Bližší popis, jak na to, je rozepsán v textovém souboru **HowTo**.



Obrázek 7.2: Obsah knihovny Models

7.2.3 ViewModels

Poslední knihovna *ViewModels* (viz obr. 7.3) je vrstvou aplikační. Úzce spolupracuje s *Model*em a zpracovává data z databáze. Zaroveň s těmito daty manipuluje a poskytuje je vrstvě *View*. Každá třída, která komunikuje s prezenční vrstvou aplikace končí v názvu postfixem **ViewModel**, dále je třeba, aby tyto třídy implementovaly rozhraní **INotifyPropertyChanged**. Tím docílíme toho, že jakákoliv změna vlastností na *ViewModelu* bude dopropagována na *View*. *ViewModel* obsahuje 4 složky.

Shell

Ve složce *Shell* se nachází implementace ViewModelu pro okna aplikace. Toto okno je zodpovědné za navigaci po ostatních stránkách a ovládá samotné menu. Každá stránka je oddděna od třídy **NavigateScreen** a obsahuje název z výčtu **PageNames**. Tímto lze snadno identifikovat každou stránku a požádat hlavní okno (shell) o její zobrazení. Taková událost (jako je změna aktivní stránky) může být zajímavá i pro ostatní stránky. Samotná změna je vyvolána využitím tzv. *EventAggregatoru*, který zajistí to, že všechny stránky, které jsou v něm registrovány a implementují rozhraní **IHandle<T>** daného typu událostí, tak budou při změně informováni.

Pages

Jak již bylo zmíněno v předchozích odstavcích, všechny stránky jsou uloženy ve složce *Pages*. Každý ViewModel reprezentuje aplikační vrstvu pro jednu stránku. Zde je zejména nutné dodržet jmennou konvenci, tedy název odpovídá názvu **UserControl**u na View akorát s postfixem **ViewModel**.

Events

Pro veškeré události, které lze vyvolat přes *EventAggregator* je nutné vytvořit objekt. Tento objekt nemusí obsahovat nic, ale pokud potřebujeme předat nějakou informaci, tak se nám toto velice hodí. Události jsou využity vždy, pokud je potřeba informovat více nezávislých zdrojů najednou.

Core

Tato složka obsahuje důležité třídy, které je vhodné oddělit od ViewModelu, například, protože stejnou funkci využívá více tříd (např. práce se soubory na disku). Třída **ClippingEar** obsahuje implementaci triangulaci obecného polygonu, konkrétně jde o implementaci algoritmu ořezávání „uší".

Tento algoritmus pracuje tak, že nejprve seřadí jednotlivé vrcholy v protisměru hodinových ručiček. Další postup je, že vezme 3 vrcholy a kontroluje, jestli prostřední z nich je natočen na pravou či levou stranu. Pokud je na straně levé, lze tento vrchol prohlásit za „ucho" a to uříznout jej, viz ukázka kódu (**??**).

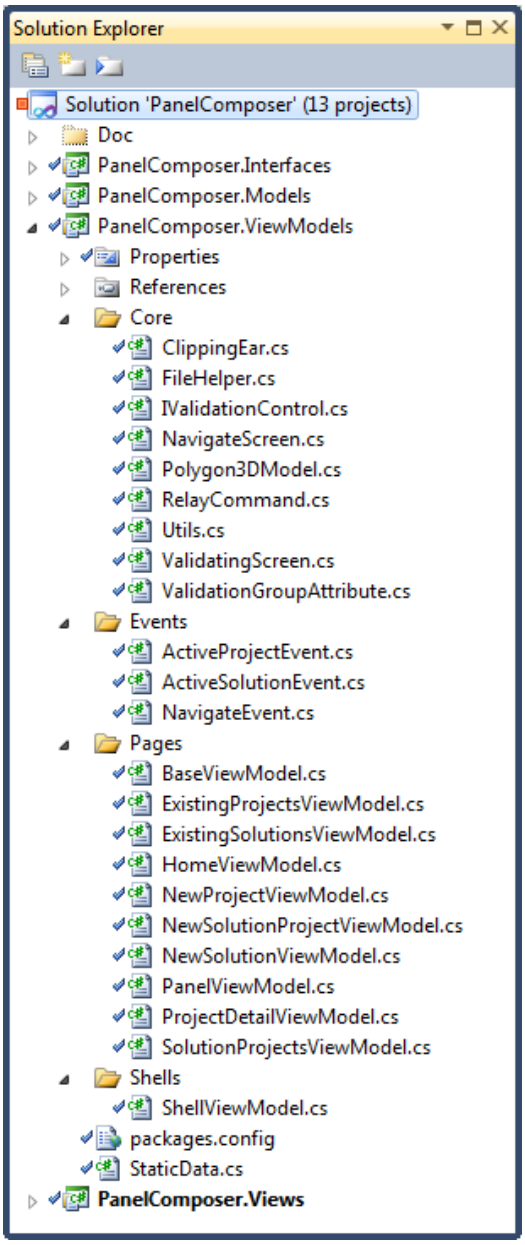
Listing 7.1: Ukázka algoritmu ořezávání uší

```
/// <summary>
/// Triangulates the specified points.
/// </summary>
/// <param name="points">The points.</param>
public static IList<IList<Point>> Triangulate(IList<Point>
   points)
{
    // zdrojovy seznam bodu
    IList<Point> source = null;
    // seznam seznamu bodu
    IList<IList<Point>> result = new List<IList<Point>>();
    source = MakeClockOtherwise(points);
    int x = 0;
    // odrezavame usi, dokud nam nezbyde jeden trojuhelnik
    while (source.Count > 3)
    {
        // vytazeni bodu
        Point p0 = source[(x - 1 + source.Count) % source.
           Count];
        Point p1 = source[(x) % source.Count];
        Point p2 = source[(x + 1) % source.Count];
        // zkontrolujeme, zdali P1 je nalevo nebo napravo
        if (IsLeft(p0, p2, p1) < 0)
        {
            // pokud je vlevo, je treba zkontroloval zdali
               trojuhelnik neprochazi zkrz existujici hrany
            bool isOk = true;
            // projedeme vsechny body a budeme kontrolovat,
               zdali se nachazi uvnitr noveho trojuhelniku
            foreach (Point p in source)
            {
                // pouzite body automaticky preskocime
```

```
                if (p == p0 || p == p1 || p == p2) continue;
                // pokud je uvnitr, nastavime flaf isOk na
                    false
                if (IsInsideTriangle(p0, p1, p2, p)) isOk =
                    false;
            }
            // pokud je vse v poradku, tak vygenerujeme
                trojuhelnik
            if (isOk)
            {
                result.Add(new List<Point>() { p0, p1, p2 });
                source.Remove(p1);
            }
        }
        x++;
    }
    result.Add(source);
    return result;
}
```

FileHelper usnadňuje práci s ukládáním obrázků do složky *Drawing* v kořenovém adresáři aplikace, vše ukládá pod kódovými názvy a vytváří záznam v databázi. **NavigateScreen** a **ValidatingScreen** jsou dvě rozšíření třídy *Screen* z frameworku Caliburn.Micro. Tyto dvě třídy implementují vše potřebné pro každou stránku a umožňují snadné volání navigace a **ValidatingScreen** navíc obsahuje validace jednotlivých vlastností třídy, což je hojně využíváno u zakládacích či editačních obrazovek. Třída **Utils** obsahuje pomocné metody pro převod plátna na obrázek PNG.

Nejdůležitější třídou jádra je **Polygon3DModel**, který na základě předaných bodů polygonu, výšky a cesty k obrázku vygeneruje kompletní 3D model panelového domu.



Obrázek 7.3: Obsah knihovny ViewModels

7.2.4 Views

Samotná aplikace je spouštěna z assembly *Views* (viz obr. 7.4). Ta je tedy vstupním bodem aplikace a zároveň prezenční vrstvou. Nejdůležitější třídou **View** je jednoznačně **AppBootstrapper**, který je vyvolán hned po startu aplikace a za běhu propojuje jednotlivé *assembly*, dále prohledává všechny třídy, které obsahují atribut *Export* a všechny je načte do slovníku. Tento průzkum provede ještě jednou a pokusí se naopak vyhledat všechny objekty, které obsahují atribut *Import*. Vnitřně si vybuduje strom závislostí, a jakmile si vyžádáme instanci některého z těchto zaregistrovaných objektů, tak automaticky inicializuje vše potřebné za nás. Toto je práce frameworku **MEF** (viz kapitola 4.3.2).

Na druhou stranu, bootstrapper je objektem druhého frameworku **Caliburn.Micro** (viz kapitola 4.3.1), který naopak využije vlastností předchozího frameworku a vyžádá si vytvoření instance rozhraní **IShell**, který reprezentuje okno aplikace. O samotné propojení *View* s *ViewModelem* se postará úplně sám.

Neméně důležitý je také soubor **app.config**, který obsahuje konfiguraci logovací knihovny **Log4NET** a nastavení spojení s databází.

Shells

Tato složka obsahuje obdobně jako u *ViewModelu* reprezentaci hlavního okna. Tentokrát ale nejde o aplikační vrstvu, ale o vrstvu prezenční. Třída **ShellView** tedy obsahuje design hlavní stránky.

Pages

Analogicky jako u předchozího odstavce, složka *Pages* obsahuje design jednotlivých stránek.

Themes

Tato složka obsahuje tzv. **ResourcesDictionary**. Tento soubor obsahuje obecné styly, které jsou používány v celé prezenční vrstvě.

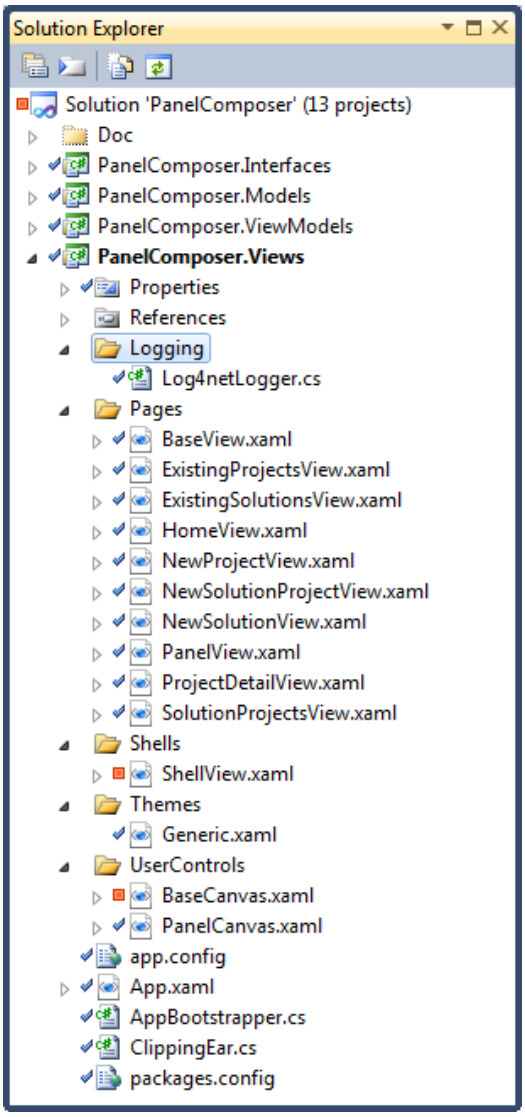
UserControls

Tato složka obsahuje specifické uživatelské kontrolky, které jsou výrazně pozměněny oproti jejím standardním implementacím. Zde jsou konkrétně umístěny dvě kontrolky. **BaseCanvas** je kreslící plátno speciálně upravené tak, aby vyhovovalo kreslení půdorysu panelových domů. Tedy umožňuje kreslení polygonu.

PanelCanvas je opět upravené kreslící plátno, které ale tentokrát na základě vstupních dat generuje výslednou texturu 3D modelu.

Logging

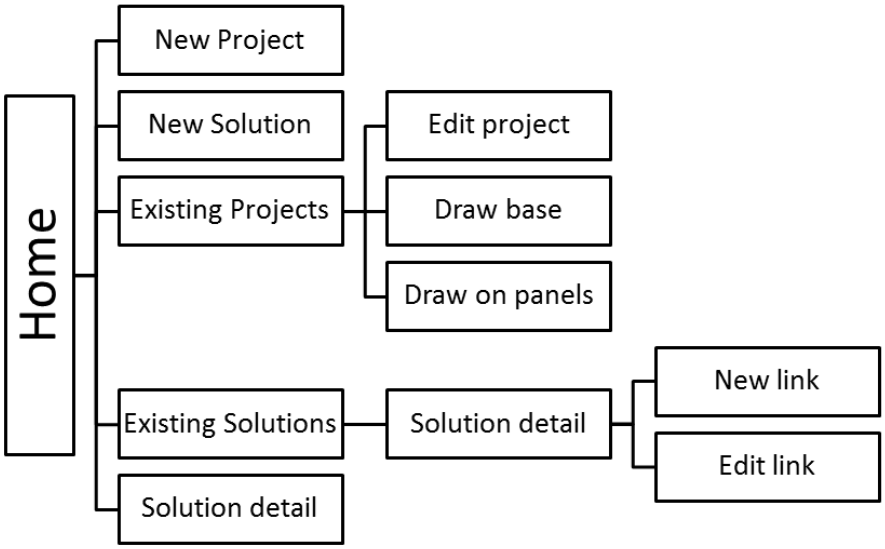
Tato složka obsahuje implementaci rozhraní **ILog** od frameworku **CM** a propojuje funkcionalitu tohoto frameworku s logovací knihovnou **Log4NET**.



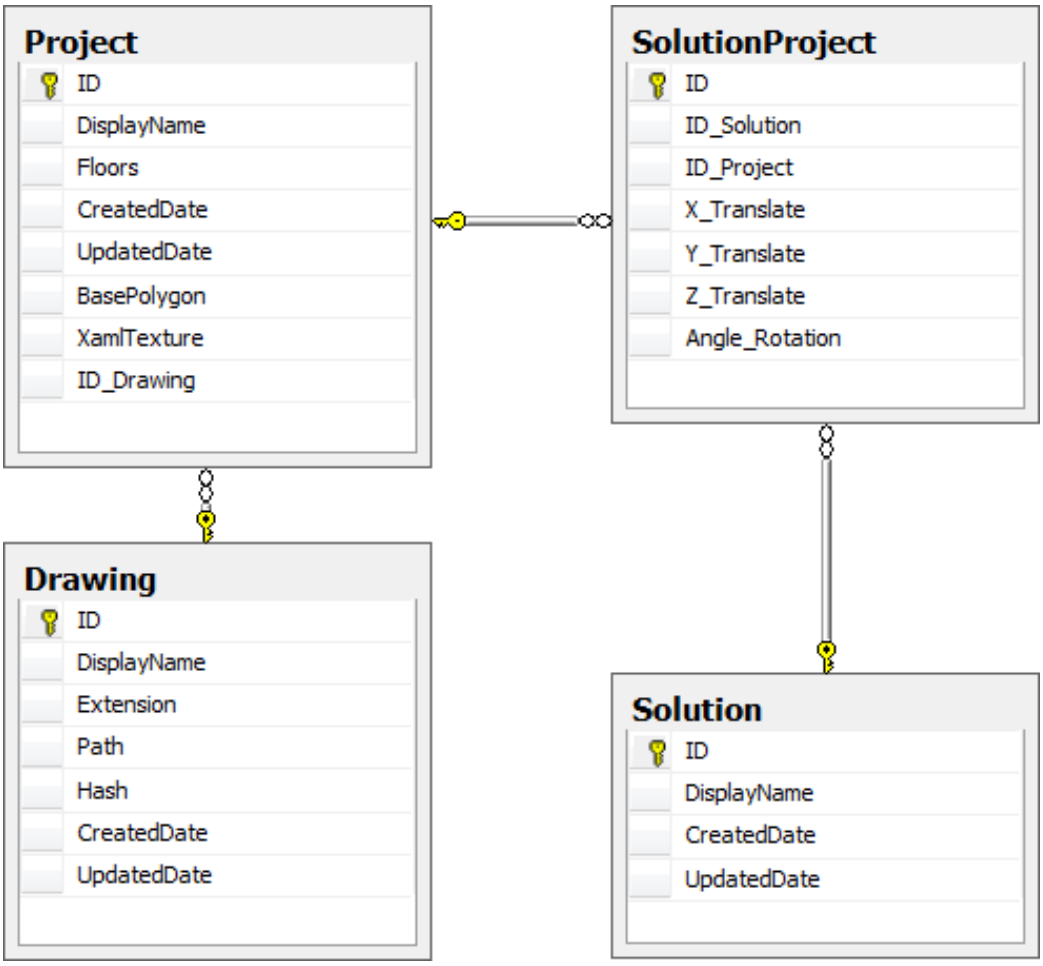
Obrázek 7.4: Obsah knihovny Views

7.3 Navigace a databáze

Následující obrázek 7.5 popisuje možný průchod aplikací. Tedy odkud kam se lze dostat z dané stránky. Tato navigace se obecně nazývá **Screen Flow**. Databázové schéma 7.6 popisuje jednotlivé entity v databázi a vztahy mezi nimi. Lze tedy snadno vypozorovat, že každý *Solution* může obsahovat 0 až N projektů a s každým tímto projektem lze samostatně manipulovat. Toto popisuje vazební tabulka *SolutionProject*.



Obrázek 7.5: Obsah knihovny Views



Obrázek 7.6: Databázové schéma

8 Práce s programem

Celý proces v nové aplikaci je založen na tom, že každý jednotlivý panelový dům je jedním projektem. Tyto projekty lze následně shlukovat v tzv. **Solutionu** (česky řešení). Solution je soubor dílčích projektů a lze jej využít pro zobrazení více panelových domu vedle sebe, např. sídliště.

Názvy **Solution** a **Project** byly převzaty z terminologie nejznámějšího *IDE* - Visual Studio. Celá aplikace je lokalizována do angličtiny, protože tento jazyk je standardem v IT oblasti. Česká lokalizace může být příkladem dalšího rozšíření.

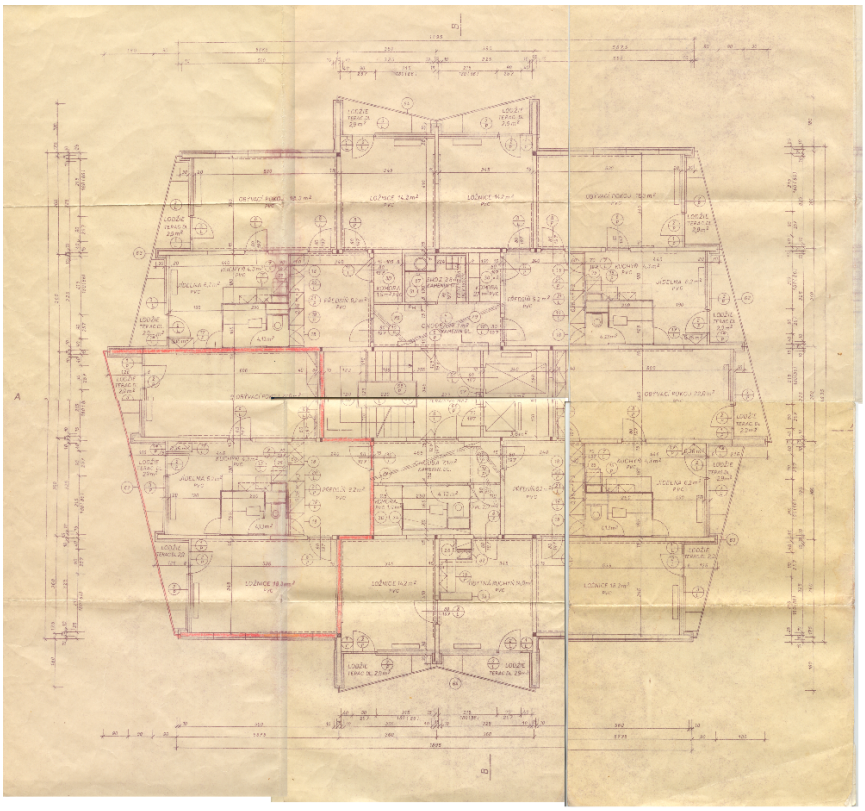
Uživatelská dokumentace je přiložena v elektronické podobě na CD.

8.1 Tvorba projektu

Tento projekt lze založit vyplněním jeho jména pro snadné rozpoznání z pohledu uživatele a vyplněním počtu pater. Základ každého domu je jeho půdorys a tak dalším nezbytným krokem je právě nákres samotného půdorysu. Toto základní schéma, jak již bylo řečeno, může být obecný polygon. Většina panelových domů v České Republice má obdélníkový tvar, ale existuje mnoho výjimek. Například panelové domy v Plzni na Borech (viz obr. 8.1) mají tvar šestihranu a tyto domy nebylo možné rozkreslit v původní aplikaci.

8.1.1 Kreslení půdorysu

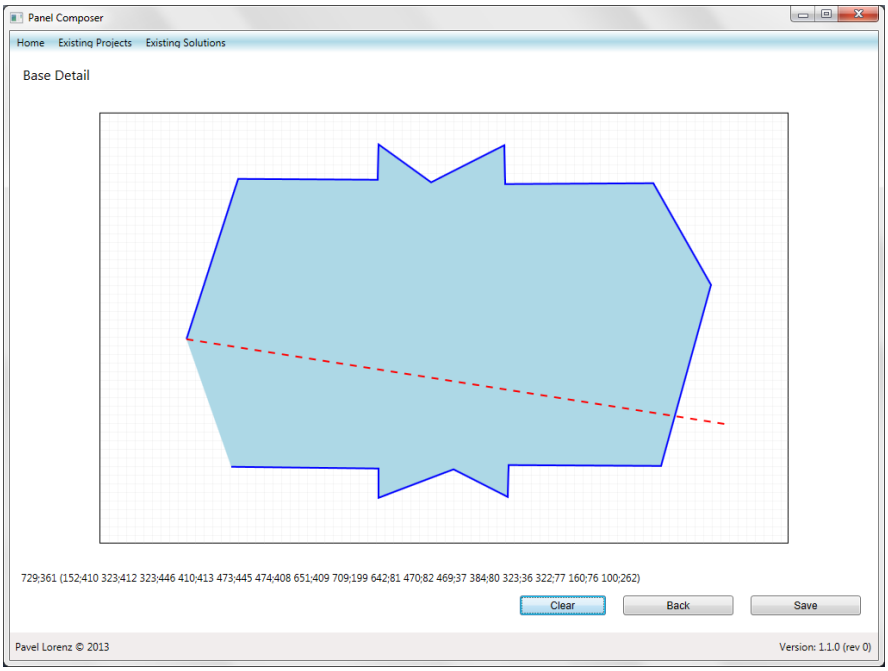
Návrh půdorysu je velice jednoduchá záležitost. Levým tlačítkem myši ukotvíme první bod polygonu a dalším stiskem postupně vytváříme polygon. Jakmile dojde k překřížení dvou stěn, nová stěna změní barvu na červenou, styl na přerušovanou čáru a kurzor myši se změní na zákaz (viz obr. 8.2). Tímto se aplikace brání tvorbě „špatných" polygonů. Samotný polygon pak ukončíme pomocí stisku pravého tlačítka myši.



Obrázek 8.1: Panelový dům v Plzni na Borech

8.1.2 Umist'ování panelů na stěny

Jakmile je návrh základů panelového domu hotov, povolí se na detailu projektu nová položka „Draw on panels". Při prvním vstupu na tuto stránku bude vygenerována textura odpovídající rozměrům půdorysu a výšky nastavené dle počtu pater. Nyní lze výběrem specifického panelu snadno umist'ovat jednotlivé panely na texturu domu. Zde opět funguje automatická kontrola, která hlídá, aby uživatel nemohl překládat panely přes sebe (viz obr. 8.3). Speciálním typem panelu jsou pak dveře, které je nutné umístit na panel stejné velikosti tedy typu „A" (typ se zobrazí po najení na tlačítko panelu). Uživatel je o této skutečnosti informován pomocí změny kurzoru.



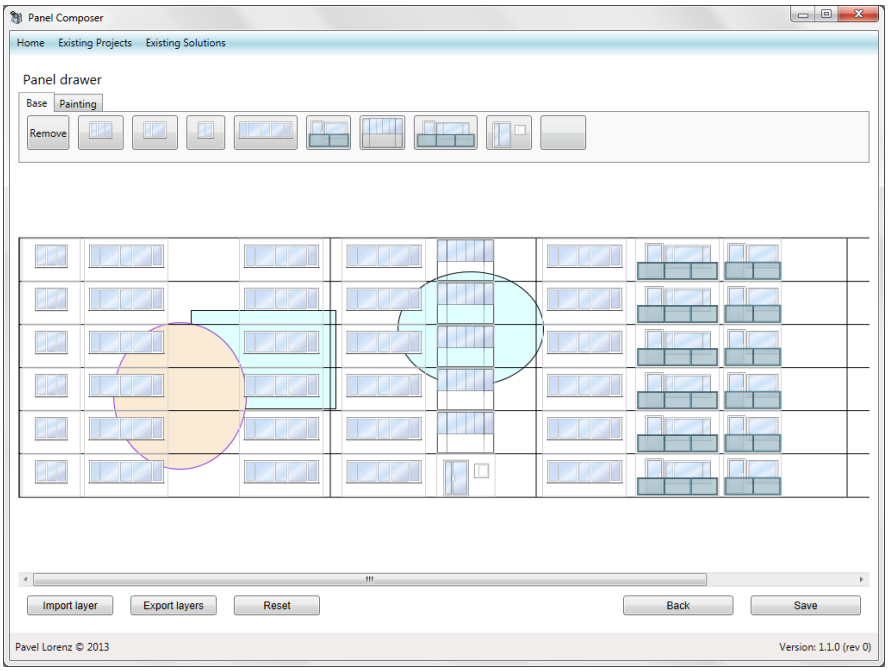
Obrázek 8.2: Kreslení půdorysu

8.1.3 Kreslení na stěny

Jsou-li panely rozmístěny na stěny, následuje poslední část procesu tvorby panelového domu, kreslení. Aplikace umožňuje pouze kreslení základních obrazců jako je obdélník, elipsa a čára (viz obr. 8.4). Pro složitější kresby je doporučeno použít funkci pro export obrázku a následně využít některý z komerčních kreslících programů např. PaintBrush, Photoshop, Gimp, Paint.NET aj,. Ideální volbou pak je software, který umí kreslení ve vrstvách, protože **Export** vytvoří jeden obrázek pro každou vrstvu. Pomocí funkce **Import** lze naopak obrázek zpět uložit do aplikace pro tvorbu panelových domů.

8.1.4 Detail projektu

Po dokončení celého procesu tvorby projektu lze na detailu pozorovat 3D projekci právě navrženého panelového domu. 3D model lze otáčet pomocí stisku pravého tlačítka a následným pohybem myši. Analogicky lze použít

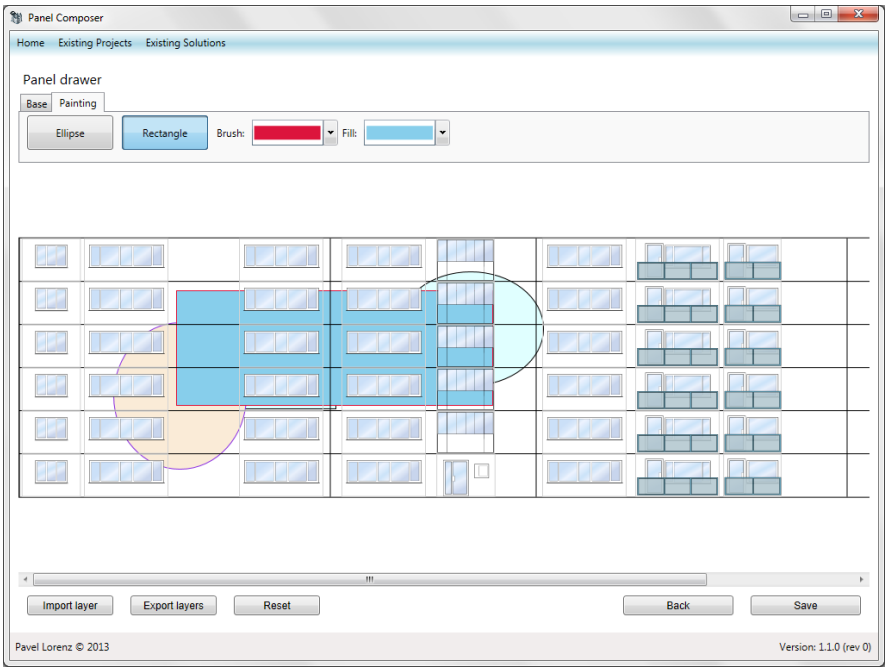


Obrázek 8.3: Umisťování panelů na stěny

malou projekční *kostku* v pravém dolním rohu a kliknutím levého tlačítka myši na danou stěnu. Celý prostor lze také posunout pomocí stisku kolečka myši a tažením.

8.2 Tvorba solutionu

Máme-li vyhotovený větší počet projektů a chtěli bychom vidět, jak by domy vypadaly vedle sebe, lze využít **solutionu**. Pro založení **solutionu** stačí pouze zadat název, to jen pro snazší orientaci uživatele v přehledu již vytvořených řešení. Detail založeného řešení obsahuje přehled všech již přiložených projektů ve spodní části obrazovky. Na levé straně se nachází přehled všech akcí, které lze provádět nad **solutionem** či nad právě aktivním **projektem**.



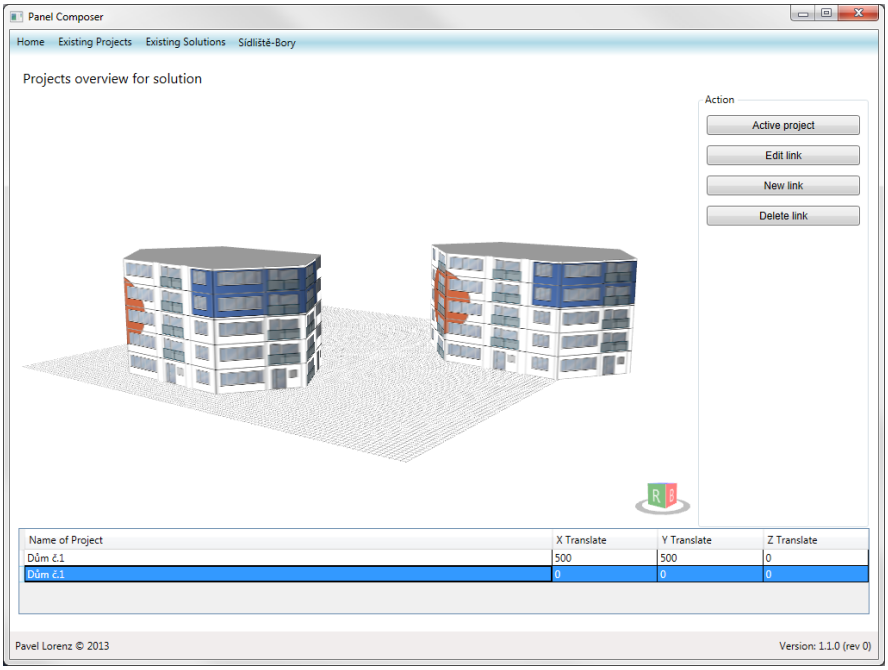
Obrázek 8.4: Kreslení na stěny

8.2.1 Tvorba linku

Propojení projektu se solutionem lze docílit pomocí tzv. **linku** (česky spoje). V každém **linku** se definuje, který projekt chceme připojit a jak daný projekt (dům) posunout či otočit ve výsledném řešení.

8.3 Shrnutí

Pomocí těchto dvou procesů, *tvorby projektu(ů)* a *tvorby řešení* se nám do ruky dostává mocný nástroj k tvorbě trojrozměrné projekce jak pro jednotlivé domy, tak celá sídliště (viz obr. 8.5).



Obrázek 8.5: Sídliště

9 Závěr

Cílem bakalářské práce bylo, seznámit se s předchozím řešením aplikace pro grafický návrh podoby rekonstrukce panelových domů a zhodnotit možnosti dalšího rozšíření, čemuž se věnuje celá kapitola „Analýza původní aplikace" (viz kap. č.2).

Dále byla prakticky vyvinuta nová aplikace, která se v té původní pouze inspiruje. Byla kompletně přepracována architektura aplikace, jejímu návrhu byla věnována velká část práce. Dále bylo užito mnoho doporučených návrhových vzorů, čímž byl splněn požadavek na další rozšiřitelnost aplikace.

Největší prostor byl věnován problematice 3D zobrazení a celý výzkum je podrobně popsán v kapitole „WPF a 3D" (viz kap. č. 6), která byla zpracována v předmětu Projekt 5. Veškeré poznatky byly použity při tvorbě dynamických 3D modelů, které jsou generovány na základě nakresleného půdorysu, což může být jakýkoliv polygon, jehož strany se neprotínají.

Vzhledem k tomu, že i kdyby byl věnován celý prostor bakalářské práce kreslení na stěny, tak bychom nikdy nevytvořili tak komfortní uživatelské prostředí jako nabízí mnoho komerčních aplikací. Bylo proto od této myšlenky upuštěno a naopak byl umožněn export a import textury domů do obrazového formátu PNG. Uživatel tak může snadno používat svůj oblíbený obrazový editor a nemusí být limitován touto aplikací.

Veškerá práce je nově ukládána do lokální databáze *MS SQL Compact Edition* a tak má uživatel vždy přehled o všech projektech realizovaných v této aplikaci.

Jako jedno z možných rozšíření se tedy nabízí import a export celého projektu či solutionu. Dále se nabízí vylepšení grafického vzhledu aplikace např. využitím *Ribbon* knihovny. Vytvoření propracovanější manipulace s 3D objekty či 3D prezentací, např. s balkóny ve formě 3D a nikoliv textury. Samotnému kreslení na stěny bych se spíše vyhnul, avšak jako jisté rozšíření se nabízí vytvoření pluginu do některého z komerčních programů pro kreslení a usnadnit tak jejich kooperaci.

Přehled zkratek

WPF	Windows Presentation Foundation
AWT	Abstract Window Toolkit
XAML	Extensible Application Markup Language
UI	Uživatelské rozhraní
GUI	Grafické uživatelské rozhraní
MVC	Model View Controller
MVVM	Model View ViewModel
IoC	Inversion of Control
DI	Dependecies Injection
MEF	Managed Extensibility Framework
DB	Databáze
SQL	Structured Query Language
ORM	Object-relational mapping
LINQ	Language Integrated Query
IDE	Integrated Development Environment
CM	framework Caliburn.Micro

Literatura

[mef(2012)] *Managed Extensibility Framework* [online]. 2012. [cit. 25.04.2013]. Dostupné z: `http://mef.codeplex.com/`.

[mon(2013)] *Mono* [online]. 2013. [cit. 25.04.2013]. Dostupné z: `http://www.mono-project.com/Main_Page/`.

[wik(2013a)] *Windows Forms* [online]. 2013a. [cit. 25.04.2013]. Dostupné z: `http://en.wikipedia.org/wiki/Windows_Forms/`.

[wik(2013b)] *Windows Presentation Foundation* [online]. 2013b. [cit. 25.04.2013]. Dostupné z: `http://en.wikipedia.org/wiki/Windows_Presentation_Foundation/`.

[Apache.org(2013)] APACHE.ORG. *Logging services* [online]. 2013. [cit. 27.06.2013]. Dostupné z: `http://logging.apache.org/log4net/`.

[Augustýn(2010)] AUGUSTýN, M. *Augiho web* [online]. 2010. [cit. 25.04.2013]. Dostupné z: `http://www.augi.cz/programovani/iocdi-v-net/`.

[Bernard(2013)] BERNARD, B. *Úvod do architektury MVC* [online]. 2013. [cit. 25.04.2013]. Dostupné z: `http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/`.

[Chris Sells(2007)] CHRIS SELLS, I. G. *Programming WPF. 2nd ed.* Sebastopol, Calif. : O'Reilly, 2007. ISBN 978-0-596-51037-4.

[Eisenberg(2013)] EISENBERG, R. *Caliburn.Micro* [online]. 2013. [cit. 25.04.2013]. Dostupné z: `http://caliburnmicro.codeplex.com/`.

[Jirava(2010)] JIRAVA, J. *Používáme Model-View-ViewModel – úvod* [online]. 2010. [cit. 25.04.2013]. Dostupné z: `http://xaml.cz/wpf/pouzivame-model-view-viewmodel-uvod/`.

[Kraft(2011)] KRAFT, P. Program pro grafický návrh podoby rekonstruovaných domů, 2011.

[libGDX(2012)] LIBGDX. *Android/iOS/HTML5/desktop game deveploment framework* [online]. 2012. [cit. 25.04.2013]. Dostupné z: `https://code.google.com/p/libgdx/`.

[Microsoft(2013)] MICROSOFT. *3-D Graphics Overview* [online]. 2013. [cit. 25.04.2013]. Dostupné z: `http://msdn.microsoft.com/en-us/library/ms747437.aspx/`.

[Mosers(2013)] MOSERS, C. *WPF Tutorial.net* [online]. 2013. [cit. 25.04.2013]. Dostupné z: `http://www.wpftutorial.net/`.

[objo(2013)] OBJO. *Helix 3D Toolkit* [online]. 2013. [cit. 27.06.2013]. Dostupné z: `http://helixtoolkit.codeplex.com/`.

[Sonnino(2007)] SONNINO, R. *Creating a 3D book-shaped application with speech and ink using WPF 3.5* [online]. 2007. [cit. 25.04.2013]. Dostupné z: `http://www.codeproject.com/Articles/22352/Creating-a-3D-book-shaped-application-with-speech/`.