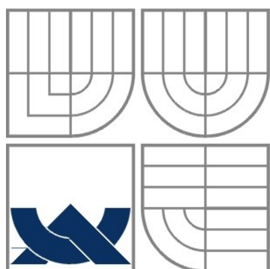
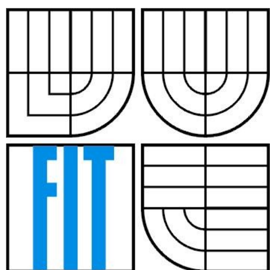


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NÁVRH A IMPLEMENTACE FUNKČNÍCH CELKŮ APLIKACE PRO ZPRACOVÁNÍ OBRAZU

DESIGN AND IMPLEMENTATION OF FUNCTIONAL UNITS OF AN APPLICATION FOR DEMONSTRATION OF IMAGE PROCESSING METHODS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Pavel Fadrhonc
AUTHOR

VEDOUCÍ PRÁCE — Ing. Filip Orság, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá implementací aplikace IMPRODEMO, demonstrující metody zpracování obrazu. Autor použije existující knihovny pro práci s obrazem a implementuje komplexnější aplikaci s moderním uživatelským rozhraním. Při vývoji použije framework WPF, metodu vývoje pomocí testů nazvanou Test Driven Development, dále návrhový vzor MVVM a prostředek pro jednoduché připojování rozšíření MEF. Popisem těchto použitých technologií a metodologií se zabývá druhá kapitola. Třetí kapitola popisuje návrh aplikace a ukázuje diagramy tříd. Popisuje zároveň komunikaci mezi jednotlivými vrstvami MVVM. Čtvrtá kapitola popisuje vývoj pomocí testů a konkrétní úskalí objevená při aplikování této metodologie. Pátá kapitola prezentuje způsob, jakým lze zakomponovat kód napsaný v jazyce C++ do prostředí frameworku .NET a jazyka C#. Závěrem je shrnuta veškerá práce, jsou definovány výsledky a navrhnuty možnosti pokračování a rozšíření práce.

Abstract

The thesis is dealing with implementation of IMPRODEMO image processing application. Author has used existing image processing libraries and has implemented complex application with modern user interface. In order to develop the application, he used framework WPF, method for developing application using tests called Test Driven Development, design pattern MVVM and MEF tool for easy integrating of extensions. Chapter two deals with these technologies and methodologies and describes them. Third chapter describes design of application and presents class diagrams. Fourth chapter describes test driven development and particular pitfalls that emerged from using this methodology. Fifth chapter presents the mean of integrating code written in C++ language into .NET framework and into C# language. At the end, whole work is summarized, resuls are defined and possibilities of resuming and extending the work are proposed.

Klíčová slova

WPF, MVVM, MEF, NUnit, TDD, .NET, návrhový vzor, návrh aplikace, zpracování obrazu, OOD, C++/CLI, managed, unmanaged, nativní kód

Keywords

WPF, MVVM, MEF, NUnit, TDD, .NET, design pattern, application design, image processing, OOD, managed, unmanaged, native code

Citace

Bc. Pavel Fadrhonc: Návrh a implementace funkčních celků aplikace pro demonstrování metod zpracování obrazu, diplomová práce, Brno, FIT VUT v Brně, 2011

Návrh a implementace funkčních celků aplikace pro demonstrování metod zpracování obrazu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením
Ing. Filipa Orsága, Ph.D .
Další informace mi poskytl Ing. Jan Čepela.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Fadrhonc
24.5. 2011

Poděkování

Chci poděkovat především externímu konzultantovi a hlavnímu programátoru KORSYSu Ing. Janu Čepelovi za poskytnutí důležitých myšlenek při řešení návrhu aplikace a materiálů k jeho zpracování. Chci také poděkovat Ing. Filipu Orságovi, Ph.D za dohlížení nad prací a pomoc s formální stránkou projektu. Také chci poděkovat mému spolupracovníkovi Ing. Martinu Hasmandovi, který stojí za implementaci funkcionality marker trackingu a kalibrace kamerového systému a poskytl mi cenné informace týkající se navrhu tříd, které tyto funkcionality implementují. V neposlední řadě chci poděkovat mojí přítelkyni Katce za vytrvalou podporu během vypracovávání práce.

©Bc. Pavel Fadrhonc, 2011
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Obsah 1
1 Úvod 3
1.1 Členění práce 3
2 Popis použitých technologií a metod 5
2.1 WPF – Windows Presentation Foundation 5
2.2 MVVM – Model View ViewModel 7
2.3 MEF – Managed Extensibility Framework 8
2.4 TDD - Test Driven Development 9
2.4.1 csUnit a NUnit 11
2.5 Další použité technologie 12
2.5.1 Event Aggregator 12
2.5.2 Commanding 13
3 Objektový návrh IMPRODEMO 14
3.1 View 16
3.2 ViewModel 20
3.3 Model 25
3.4 Komunikace mezi vrstvami MVVM 29
4 Funkční testy 31
4.1 Technika testování 31
4.2 Testování ToolsPanelViewModel 32
4.2.1 Testování bez nutnosti okna 32
4.2.2 Testování interakce mezi ViewModely 33
4.3 Testování na úrovní vrstvy Model 33
4.4 Shrnutí testování 37
5 Interoperabilní vrstva pro práci s obrazem a externími zařízeními 38
5.1 Použitý jazyk – C++/CLI 38
5.2 Wrapper pro třídu Matrix32 38
5.2.1 Vytvoření a zánik Matrix32 39
5.2.2 Další komponenty Matrix32 40
5.3 Wrapper pro práci s kamerou 41
6 Závěr 44
6.1 Dosažené výsledky 44
6.2 Přínos práce 44
6.3 Možností využití a dalšího rozvoje 44

Seznam použitých zdrojů **45**

Seznam příloh **46**

Příloha A **49**

Příloha B **58**

1 Úvod

Moderní aplikace zpracování obrazu vyžadují rychlé zpracování obrazu v reálném čase na různých prostředích. Takové aplikace se rozšiřují i do mobilních telefonů a do běžných komerčních systémů, kde není předpokládán zvlášť velký výkon použitého stroje. Zároveň je kromě výkonu požadována atraktivnost a modernost uživatelského prostředí a pružné reagování na požadavky zákazníka.
V současné době je na trhu vývoje softwaru velké množství kvalitně pracujících a zavedených firem a proto je každá možnost konkurenční výhody vysoce ceněná. Tou může být i rychlost vývoje aplikace, rozšiřitelnost a znovupoužitelnost proprietárních komponent firmy. Rozšiřují se tedy metody agilního vývoje, je velká snaha o recyklování již vyvinutých knihoven a zároveň se hledají způsoby k rutinnímu řešení podobných problémů v návrhu architektury a sestavování aplikací.

Tato práce se zabývá vývojem aplikace demonstrující metody zpracování obrazu, konkrétně metody korelace obrazu a nalezení a sledování objektu v obraze. Aplikace používá tyto metody v existujících knihovnách napsaných v jazyce C++ a poskytuje jim moderní prezentaci a architekturu danou návrhovým vzorem MVVM (Model View ViewModel) [4]. Ten je podporován frameworkem WPF [2], který byl pro účely vývoje této nové aplikace nazvané IMPRODEMO (IMage PROcessing DEMOnstration) vybrán jako moderní, multifunkční nástroj obsahující přímou podporu zobrazení objektů grafickou kartou (pomocí DirectX [20]) a poskytující jednoduchý způsob nastavení a změnu atraktivního designu.

Aplikace IMPRODEMO používá existující knihovny pro metody zpracování obrazu vyvinuté v rámci práce na aplikaci KORSYS (KORelační SYSém). Ta měla podobné použití jako nově vyvíjená aplikace IMPRODEMO s tím rozdílem, že obsahovala mnohem komplexnější uživatelské rozhraní, umožňovala nastavené velké množství parametrů, což bylo pro neznalého uživatele matoucí a nutilo ho používat uživatelskou příručku. Zároveň, jak se do projektu přidávala další a další funkcionalita, vzhledem k tomu, že na projektu pracovalo více programátorů a neexistovala žádná jasná pravidla pro vývoj aplikace a psaní kódu, začínala aplikace trpět antipatternem "Big Ball of Mud" [1], který se vyznačuje tím, že projekt nemá žádnou jasně určenou či viditelnou architekturu a úprava či změna kódu je velmi náročná, vzhledem k nelogickým závislostem na různých místech v kódu.

1.1 Členění práce

Z důvodu najasných pravidel při vývoji a nejasné představě o vyvíjených třídách bylo při vývoji nové aplikace IMPRODEMO rozhodnuto o použití agilní vývojové metody Test Driven Development (TDD) [21]), která by měla usměrnit a sjednotit snahy při implementaci funkcionality. Zároveň byl grafický toolkit wxWidgets [27], používaný při vývoji KORSYSu vyměněn za framework WPF. Důvodem pro tuto změnu a vlastnostmi WPF se zabývá podkapitola 2.1. Kromě toho bylo v rámci snadné rozšiřitelnosti a zachování dobré struktury programu zavedeno několik technik: frameworkem WPF podporovaný návrhový vzor MVVM (podkapitola 2.2), framework MEF [22] (podkapitola 2.3) pro snadnou rozšiřitelnost programu a nahrávání rozšíření za běhu, koncept testování Test Driven Development (podkapitola 2.4) a nástroj NUnit (podkapitola 2.4.1), který tento koncept implementuje.

V kapitole 3 je předveden návrh celé aplikace podle návrhového vzoru MVVM, jsou zde uvedeny jednotlivé diagramy tříd, přičemž podkapitoly sledují jednotlivé vrstvy použitého návrhového vzoru od prezentační směrem k doménové. Kapitola 3.1 popisuje prezentační vrstvu

View, jakým způsobem bude navrženo samotné uživatelské rozhraní a jakým způsobem se budou data prezentovat uživateli. Kapitola 3.2 popisuje vrstvu ViewModel. Ta uchovává stav objektů prezentujících se ve vrstvě View a zajišťuje konverzi dat mezi View a vrstvou Model, která je popsána v kapitole 3.3. Poskytuje přístup k samotným datům aplikace a zároveň výpočetním nástrojům, algoritmům zpracování obrazu a třídám pro práci s externími zařízeními.

Kapitola 4 ukazuje, jakým způsobem byl při vývoji aplikace IMPRODEMO použit koncept Test Driven Development. Vysvětlí způsob vývoje aplikace pomocí testů, ukáže konkrétní vývojový cyklus a popíše úskalí nastávající při vývoji pomocí testů.

V Kapitole 5 je rozbrán způsob znovupoužití kódu, napsaného v jazyce C++ a převáděného do strojového kódu, v prostředí jazyka C# spravovaného runtime prostředím od Microsoftu. Jedná se o kód napsaný v jazyce C++/CLI, který zabaluje existující třídy do nativního kódu a poskytuje tak třídy použitelné v prostředí jazyka C#. Opět jsou zde popsána všechna úskalí tohoto přístupu včetně propagování událostí a uvedeny řešení těchto netriviálních problémů.

Poslední kapitola číslo 6 je již závěrem celé práce a přehledně shrnuje dosažené výsledky a ukazuje možnosti rozšíření a vylepšení aplikace a vývojového přístupu.

2 Popis použitých technologií a metod

Tato kapitola obsahuje popis technologií a metod použitých při návrhu a vývoje aplikace IMPRODEMO. Kromě stručného popisu nejdůležitějších vlastností obsahuje také odůvodnění zahrnutí dané technologie či metody do vývoje aplikace.

2.1 WPF – Windows Presentation Foundation

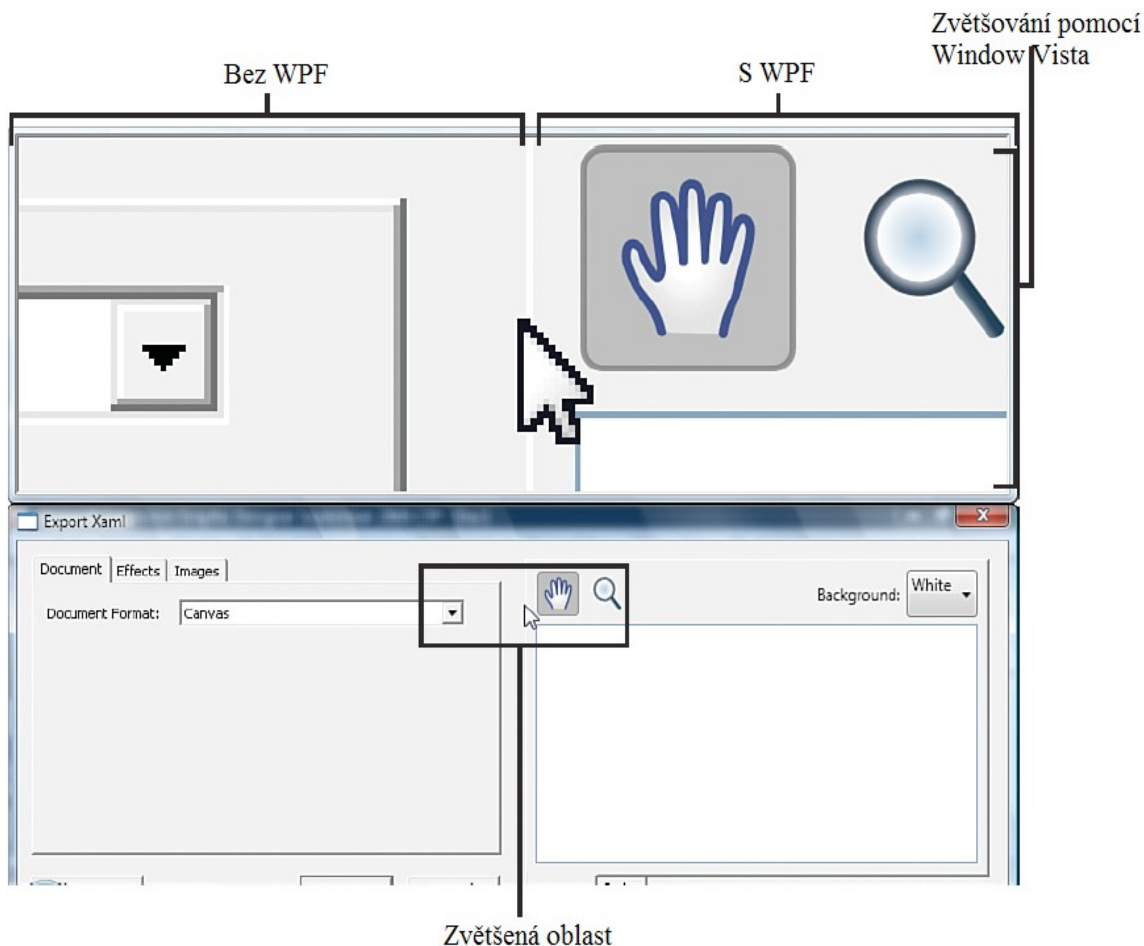
WPF je grafický prezentační framework od firmy Microsoft, který je neoficiálním následníkem WinForms a oproti němu obsahuje několik zásadních vylepšení. Mezi ně patří:

1. **Široká integrace médií**
   Pokud chtěl uživatel použít 2D operace, využívat 3D funkce, zakomponovat podporu videa, řeči a dokumentů do své aplikce před WPF, musel použít několik různých technologií s celkově nekonzistentním rozhraním a vymyslet, jak je všechny sloučit dohromady. WPF přináší jednotné rozhraní pro všechny typy médií a techniky používané při manipulaci s jedním typem medií se dají aplikovat i pro ostatní typy. Pro aplikaci IMPRODEMO tato výhoda znamená, že při práci s kamerami je možno využít vestavěnou podporu videa, při zobrazování korelovaného výstupu je možno použít podporu 2D obrazu a při rekonstrukci 3D obrazu zase podporu pomocí DirectX.

2. **Nezávislost na rozlišení**
   WPF podporuje zobrazování veškerého svého obsahu vektorovou cestou, což umožňuje jednoduché přibližování po celé pracovní ploše vyvíjené aplikace bez hran, které kazí vizuální dojem z aplikace. Vzhledem k tomu, že IMPRODEMO má být použito k výukovým účelům, je pravděpodobná demostrace metod na velké ploše, kde by hrany mohly být vidět a zkazit tak výsledný dojem z jinak dobře navrženého grafického uživatelského rozhraní.

3. **Hardwarová akcelerace**
   V dnešní době je síla a výkon grafických karet 10x až 100x vyšší než výkon procesoru (v GFLOPS) [1]. Microsoft se rozhodl tuto sílu využít a tak jedna z hlavních předností WPF je renderování veškerého grafického obsahu pomocí DirectX a tedy urychlení grafické karty. Odlišuje se tak od předcházejících technologií, které používaly GDI, případně vylepšenou verzi GDI+. Ačkoliv WPF obsahuje softwarovou renderovací jednotku, výkoný grafický hardware se v případě náročnějších aplikací projeví tím, že uleví procesoru a ten se může věnovat ostatním činnostem. Použítí hardwarové akcelerace obstarává WPF automaticky
   a z pohledu uživatele i programátora je tak naprosto transparentní. Z pohledu IMPRODEMO je tato vlastnost velmi důležitá, v programu se totiž používá vykreslování pořízených snímků z kamery spolu s graficky prezentovanými informacemi o korelaci, či sledování markerů. To vše probíhá v reálném čase a je tudíž zásadní, aby toto zobrazovánií bylo co nejrychlejší. WPF poskytuje jednotné API k vykreslování všech objektů a zároveň plátno (*canvas*) s velmi intuitivním způsobem práce. Implementovat tak funkcionalitu obsaženou v KORSYSu, kde

se pro vykreslování používala hardwarová akcelerace pomocí OpenGL, je tak mnohem snažší, a to zároveň pomocí jazyka vyšší úrovně C#.



Obrázek 2.1: Efekt nezávislosti na rozlišení.

4. XAML
WPF zavádí podporu přirozeného oddělení prezentační části aplikace od její *bussiness* logiky. Je toho dosaženo pomocí souborů XAML, což je vlastně soubor ve formátu xml, kde strom prvků vyjadřuje zobrazované komponenty WPF. Speciální vlastností a výhodou, kterou XAML přináší do WPF je fakt, že XAML je deklarativní jazyk a tudíž umožňuje programátorům a potažmo designérům navrhnout grafické uživatelské rozhraní bez použití procedurálního programování. XAML tak lépe umožňuje kooperaci více osob bez speciálních předchozích znalostí. Pro IMPRODEMO je tato vlastnost důležitá především z pohledu architektury. Protože prezentační část aplikace je vynuceně napsaná v XAMLu, je velmi těžké do této vrstvy zakomponovat implementaci částí, které mají místo v jiné úrovni softwarové architektury. Příspívá tak nejen k čistotě a srozumitelnosti kódu, ale také zároveň k jeho rozšiřitelnosti.

5. Rozšiřitelnost a možnost přizpůsobení
WPF nabízí široké možnosti vlastního přizpůsobení jakéhokoliv grafického prvku pomocí mocného nástroje šablon. S pomocí minimálního množství kódu lze prvek upravit téměř jakýmkoliv možným způsobem (například prvek *CheckBox* obsahující animaci a tabulku, ačkoliv tato kombinace může znít nepoužitelně, ukazuje velkou sílu šablon ve WPF). Tato

vlastnost je v IMPRODEMU využívaná hlavně pro svoji jednoduchost a pro vytváření chybějících prvků.

2.2 MVVM – Model View ViewModel

Model-View-ViewModel [4] je navrhový vzor, který byl navržen a vytvořen firmou Microsoft. Vzniknul jako specializace návrhového vzoru Presentation Model navrženého Martinem Fowlerem a návrhového vzoru Model-View-Controller (MVC) [31]. Vlastnost, kterou mají všechny tyto vzory společnou je ta, že oddělují prezentační část od doménové. MVC přidává navíc přímou podporu pro WPF vlastnosti jako je *databinding* (vysvětleno níže) a XAML.

MVVM obsahuje následující základní architekturální vrstvy:

Model znamená objektový model, který reprezentuje skutečný stav dat nebo vrstvu přístupu k objektům, které reprezentují skutečný stav. Obsahuje tedy buď skutečné třídy, nebo jen poskytuje rozhraní, jak k nim přistupovat. Z pohledu IMPRODEMO se uplatní druhá část. Model bude sada tříd přenesená z KORSYSu, původně napsaná v jazyce C++, nyní zabalená v C# *wrapperu* tak, aby se dala použít ve WPF.

View je vrstva, která obsahuje vše, co se zobrazuje v uživatelském rozhraní, např. tlačítka, okna, kreslící canvas apod.

ViewModel je prostředník mezi daty zobrazovanými, tak jak je vidí uživatel (View) a daty uloženými a zpracovávanými v programu (Model). Funguje tak jako jakýsi konvertor, který převádí data z View do Modelu a naopak, zároveň může také přímo posílat příkazy, nebo je příjímat a definovat na základě jednoho příkazu nějakou složitější funkcionalitu.

WPF obsahuje navíc jednu vlastnost, která souvisí s MVVM, tudíž jsem její popis nechal až do fáze, kdy je jasné, co je MVVM. Tou vlastností je *databinding*.

Databinding je jednoduchý a konzistentní způsob, jak aplikace prezentují a interagují s daty. Prvky uživatelského rozhraní ve vrstvě View mohou být svázány se zdroji dat ve vrstvě ViewModel nebo Model. Ve WPF je navíc nativní podpora pro filtrování, řazení a konvertování stran účinkujících v datovém svazku.

Z technické stránky *databinding* poskytuje pohodlnou manipulaci se zdroji dat pro uživatelské prvky hlavně z toho důvodu, že ve chvíli, kdy jsou data správně nastavena, prvek svázan s těmito daty automaticky registruje změnu ve zdrojových datech a příslušně obnoví svůj stav. Takže zatímco například ve wxWidgets je nutno psát explicitní kód na ovladač události každého prvku grafického rozhraní a určit kdy a za jakých podmínek se má proměnná reprezentující stav objektu obnovit, ve WPF a MVVM stačí jen zmínit v XAML dokumentu název proměnné a svázání a vzájemná reakce na změny je hotova. Pokročilejší a méně obvyklý databinding se samozřejmě píše explicitně, nicméně mechanismus, který používá umožňuje přirozeně oddělit zpracování dat a příspívá k upravitelnosti programu.

Proč MVVM a WPF?

MVVM je přirozený návrhový vzor pro WPF, což je důvodem pro to, že ho vývojáři WPF používali na vývoj nástroje pro WPF Microsoft Blend. Mnoho vlastností a aspektů WPF používají stejný model oddělení stavu a chování, který je podporovaný MVVM.

Hlavním důvodem pro MVVM je *databinding*, který je popsán výše, jsou tu však další dvě vlastnosti, které činí tento návrhový vzor ideálním pro WPF. Těmi jsou datové šablony a systém

zdrojů. Datové šablony aplikují View na objekty ViewModelu, které jsou vidět v rozhraní. Je možno deklarovat šablony v XAML a nechat systém zdrojů, aby automaticky našel a aplikoval tyto šablony za programátora, a to vše za běhu programu.

Dalším důvodem, proč je MVVM ideálním návrhovým vzorem pro WPF je, že třídy ViewModelu jsou jednoduché na testování. Pokud se aplikační logika vyskytuje pouze ve třídách ViewModelu, je možno jednoduše napsat kód, který je otestuje. Unit testy a View jsou potom jen dva druhy zákazníků, které používají ViewModel. Unit Testování je pro IMPRODEMO další důležitou vlastností popsanou v podkapitole 2.4.

Testovatelnost tříd ViewModelu může také pomoci k řádnému navržení uživatelského rozhraní. Když se navrhuje aplikace, může fakt, zda jde test pro danou třídu napsat bez instantizace jakéhokoliv UI prvku, rozhodnout, zda bude třída umístěna do View nebo do ViewModelu.

Poslední výhodou je fakt, že protože View je defacto jen uživatel obecného rozhraní ViewModelu, je mnohem snažší jeden View odstranit a nahradit ho jiným a poskytnout tak aplikaci zbrusu nový vzhled a dokonce i uživatelské rozhraní. Je tak poměrně přímočaré rozdělit vývojářský tým na programátory funkčnosti, kteří zabezpečují funkcionalitu aplikace vývojem ViewModelu a designéry aplikace, kteří navrhují uživatelské rozhraní pomocí XAML. Spojení jejich práce pak jen obnáší vyvořit správně *databinding* z View do ViewModelu. Ve třetí kapitole bude popsáno, jaké třídy se budou v aplikaci vyskytovat a jak budou umístěny v rámci MVVM.

2.3 MEF – Managed Extensibility Framework

MEF je technologie k vytváření modulárních rozšiřitelných aplikací. Knihovna v podstatě implemetuje návrhové vzory Dependecy Injection a Inversion of Control [5]. Umožňuje programátorovi najít a použít rozšíření bez jakéhokoliv konfiguračního souboru. Zároveň umožňuje vývojáři jednoduše zapouzdřit kód bez mnoha napevno vytvořených závislostí. MEF umožňuje nejen vytvořit rozšíření, která se dají použít napříč celou aplikací, ale také taková, která se dají použít ve více aplikací bez nutnosti přepisovat rozhraní. Framework totiž definuje jednoduché rozhraní a ulehčuje tak programátorovi práci v tom smyslu, že nemusí definovat vlastní rozhraní (což nejen zabere čas a úsilí, ale také nemusí zohledňovat použití v budoucích aplikacích, na druhou stranu může být konkrétnější a tím pádem vhodnější na danou aplikaci).

Pokud chceme vyvíjet aplikaci, která má být do budoucna přístupná k rozšířením, máme několik základních možností.

1. Nejjednodušší přístup spočívá v tom, že zdrojový kód každého rozšíření aplikace vložíme přímo do programu. Tento přístup má očividně nevýhodu v tom, že při přidání jakéhokoliv dalšího rozšíření musíme upravit zdrojový kód naší aplikace. Navíc, pokud jsou rozšíření vyvíjeny a dodávány třetími stranami, nemusíme mít vůbec jejich zdrojový kód k dispozici a tudíž nemůžeme definovat jejich spolupráci s našimi komponenty.
2. Poněkud sofistikovanější přístup spočívá v tom, že definujeme rozhraní, které komponenty implementují a API pro komunikaci s hlavním programem. Toto vyřeší problém s přístupem ke zdrojovému kódu, ale stále to má svoje nevýhody:
    - Neexistuje zde žádná schopnost hledání rozšíření (komponent). Ty potom musí být zapsány v nějakém konfiguračním souboru, který musí být někým spravován a jedná o další položku, která musí být na paměti vývojáře.

- Komponenty spolu nemohou komunikovat, pokud nemají definovaný komunikační kanál přes samotnou aplikaci. Pokud aplikace komunikaci neočekává, je téměř nemožné komunikaci zajistit.
- Poslední nevýhodou je, že vývojář komponenty musí pracovat s pevně danou závislostí na tom, která *assembly* [14] obsahuje rozhraní, které daná komponenta implementuje. Díky tomu je obtížné, aby komponenta byla použita ve více než jedné aplikaci. Také to může přinést problém, pokud chceme vytvořit testovací *framework* pro komponenty.

Místo toho, aby byly komponenty definovány explicitině, MEF poskytuje možnost je najít implicitně přes mechanismus skládání. Komponenta frameworku MEF specifikuje závislosti (nazývané *import*) a schopnosti (nazývané *export*), které poskytuje. Ve chvíli, kdy je taková komponenta vytvořená, MEF systém doplní její závislosti podle toho, co je dostupné z ostatních komponent.

Tento přístup řeší problém popisovaný v přecházející části. Protože části popisují svoje schopnosti deklarativně, je možno je nalézt za běhu programu, což znamená, že aplikace může využívat tyto komponenty bez napevno nastavených referencí nebo konfiguračnich souborů. MEF umožňuje komponenty prozkoumat pomocí jejích metadat, bez toho aby se instantizovaly nebo nahrály jejich *assembly*. To ve výsledku znamená, že není potřeba určovat kde a jak by měly být rozšíření nahrány.

Důvod, proč byl MEF vybrán při vývoji aplikace IMPRODEMO, je jednoduchý. Jako každá aplikace se i pro aplikaci IMPRODEMO předpokládá, že budou přibývat další funkce. Je tedy dobré, pokud je zvolen mechanismus, který přidávání dalších rozšíření co nejvíce usnadní a zprůhlední. MEF je nejefektivnější mechanismus pro tento účel a zároveň je nativně podporován v .NET frameworku 4. To ho činí ideálním kandidátem na tuto pozici.

Dalším důvodem je výborná možnost testování. Protože k tomu, aby třída fungovala je potřeba jen rozhraní kompnent, které používá, je možno testovat každou vrstvu MVVM naprosto odděleně. Přispívá to k modulárnosti programu a ztenčuje to vazby, což umožňuje různým programátorům pracovat na různých částech bez větších konfliktů či závislostí.

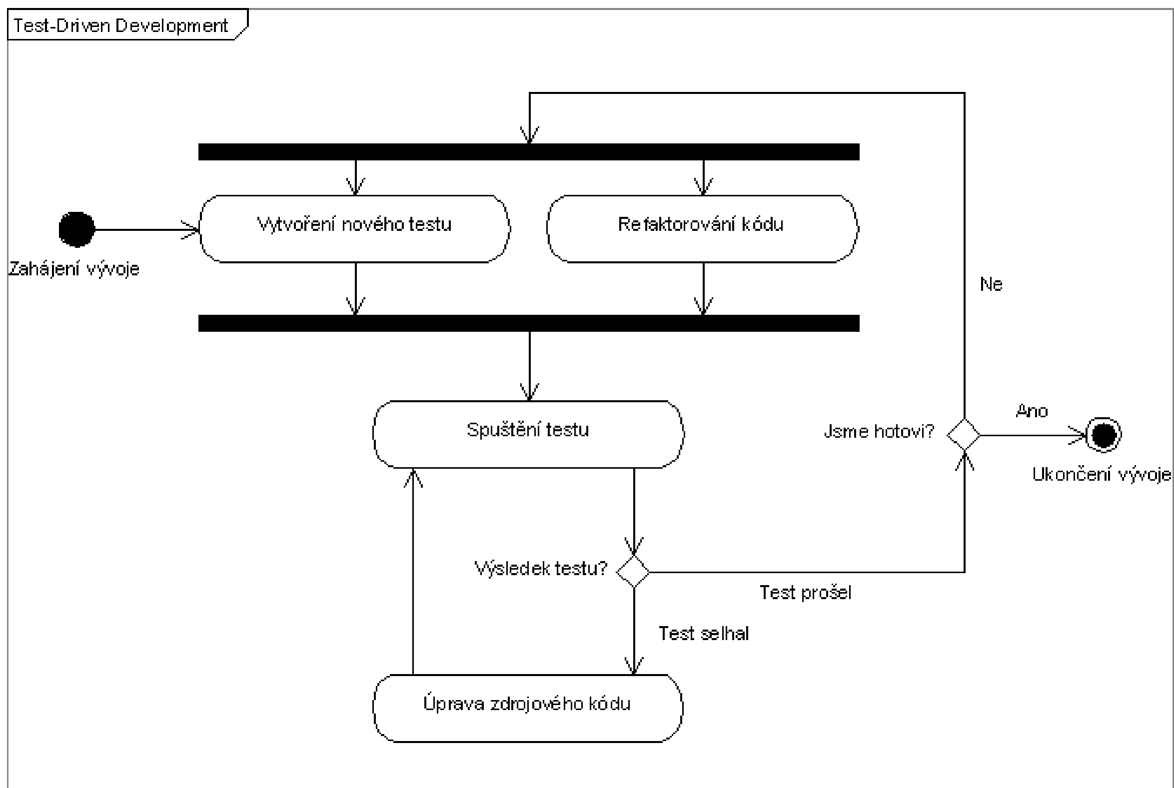
2.4 TDD - Test Driven Development

Od počátku vývoje softwaru existuje snaha vytvořit techniky a postupy, které přivedou do celého procesu pořádek a učiní ho lineárním a předvídatelným. Posledních několik let se stávají velmi populární tzv. agilní metody vývoje softwaru [28]. Jednou z nejznámějších je extrémní programování a jednou z nejdůležitějších složek extrémního programování je technika vývoje řízeného testy, anglicky Test Driven Development (dále již jen TDD). Co přesně TDD je, co obnáší a proč bylo rozhodnuto o jeho použítí ve vývoji aplikace IMPRODEMO vysvětlí tato podkapitola.

TDD je technika založená na psaní automatických testů pro danou třídu. Je to způsob vývoje softwaru, který opakuje jeden základní cyklus:

- Vývojář napíše kód testu pro funkci/třídu, kterou se chystá zrovna naimplementovat. Takový test je napsán, aby okamžitě selhal, neboť prozatím třída či funkce ještě není napsaná.
- Vývojář napíše kód funkce či třídy tak, aby vyhovoval napsanému testu. V tuto chvíli nezáleží na kvalitě kódu. Ta totiž bude upravovaná až v dalším kroku, kde

- vývojář kód refaktorizuje. To znamená, že upravuje a zmenšuje množství efektivního kódu tak dlouho, dokud je kód minimální a co nejčistčí. Během refaktorizace využívá již napsaného testu a průběžně ho spouští, aby zajistil, že upravovaný kód neztrácí na své funkcionalitě.



Obrázek 2.2: Kroky TDD procesu.

Refaktorizace je velmi důležitou součástí TDD. TDD bez refaktorizace je jen TFD (Test First Development, což znamená, že se jen aplikuje napsání testů ale refaktorizace již neprobíhá). Obrázek 2.2 názorně ukazuje jednotlivé kroky TDD procesu. Nyní si je podrobněji popíšeme jejich úskalí a vlastnosti.

- **Vytvoření nového testu:** V TDD musí programování každé funkcionality začínat napsáním testu. Tento test musí nevyhnutelně selhat, protože daná funkcionalita ještě nebyla napsána. Pokud neselže, nově psaná funkcionalita již existuje, nebo je test špatně napsán. Aby mohl vývojář napsat test, musí detailně porozumět, co a jak má přesně nová funkcionalita obnášet a provádět. V tom se TDD liší od psaní testu až po tom, co je kód dokončen a nutí vývojáře premýšlet nad požadavky před psaním kódu. Je to malý, ale důležitý rozdíl.
- **Spuštění testu:** První spuštění testu je důležité, protože vývojář je pak ujištěn, že kód je napsán „dobře", tedy že selže při prvním spuštění. Vyloučí se tak chybná možnost projití testu i bez napsání jakéhokoliv funkčního nového kódu.
- **Úprava/napsání zdrojového kódu:** V této fázi jde o to, aby vývojář napsal nějaký kód, který by měl způsobit, že test projde v pořádku. Nejde o to, aby kód byl

optimalizovaný či elegantní, je dokonce důležité, aby v tuhle chvíli nepředvídal nějakou budoucí funkcionalitu či možnost chyby.

- **Spuštění testu 2:** V tuto chvíli by již měl test projít v pořádku. Pokud neprojde, je nutné opakovat tento a předchozí krok tak dlouho, dokud se tak nestane. Až se dostane k tomuto bodu, je možno pokračovat k závěrečné fázi, kterou je
- **Refaktorizace kódu:** V tuto chvíli je možno začít optimalizovat a uhlazovat napsaný kód. Protože vývojář má funkční test a správně napsanou funkcionalitu, může se vždy ujistit, že jeho úprava nepoškozuje požadovanou funkcionalitu. Samotná refaktorizace již není cílem tohoto výkladu, pro přehled však uvedu několik praktik, které se v rámci refaktorizace používají a které přispívají ke srozumitelnosti a celkové kvalitě kódu:
    - odstranění magických proměnných
    - lepší a více deskriptivní pojmenování proměnných
    - nahradit podmínky polymorfismem
    - opakující se kód vložit do zvláštní funkce
    - přístup k proměnným přes metody (*getter*/*setter*) místo přímého přístupu

Po dokončení všech těchto kroků je vytvořen test stejně tak jako nová funkcionalita. Jakákoliv budoucí změna ve funkcionalitě by se měla projevit nejdříve jako změna v požadavcích, potom v testu a až potom v samotné funkcionalitě, a to teprv v hrubé podobě a až potom, co je ověřena nová funkčnost, je možné kód upravit jeho refaktorizací.

Další vlastnosti a výhody pro projekt

Údaje založenné na studii [8] vypovídají, že programátoři, kteří používali TDD, museli napsat více testů a zároveň, programátoři, kteří napsali více testů, byli více produktivní. Ti, co začali psát testy na nových projektech se nechali slyšet, že mnohem méně častěji mají potřebu používat debugger. Pokud test neočekávaně selhlal a řešení nebylo jasně vidět, bylo účinnější vrátit se k předchozí verzi (v případě, že projekt pracoval s nějakou implementací verzovacího systému).

TDD nabízí více než jen ověření správnosti funkčnosti, ale zároveň pomáhá ujasnit si návrh programu a funkčnosti právě psaných tříd. Když se píšou testy, musí si vývojář představit, jak tyto třídy budou používány klientem (v prvním případě to jsou testy).

Nikde se také neříká, že celá nová třída musí být pokryta jedním testem. Většinou existuje více menších testů pro danou třídu, které se lépe spravují a mají společné téma – funkcionalitu, nebo třeba fakt, že testují výjimky, které může daná třída v případě selhání generovat.

TDD často vede k více modularizovanému, flexibilnímu a rozšiřitelnému kódu. To je proto, že metoda nutí programátora přemýšlet v rámci několika malých jednotek, které mohou být napsány a testovány nezávisle a spojeny dohromady později. To vede k menším, více konkrétněji zaměřeným třídám, které jsou volněji vázané, a čistším rozhraním.

Vzhledem k tomu, že KORSYS trpěl velmi komplexními a monolitickými třídami a nejasnými, složitými a rozsáhlými rozhraními, byl TDD zvolen jako metoda agilního vývoje. Všechny výše uvedéné výhody zmíněné v [7] adresují problémy KORSYSu a mají za cíl tyto problémy při vývoji IMPRODEMO odstranit.

2.4.1 csUnit a NUnit

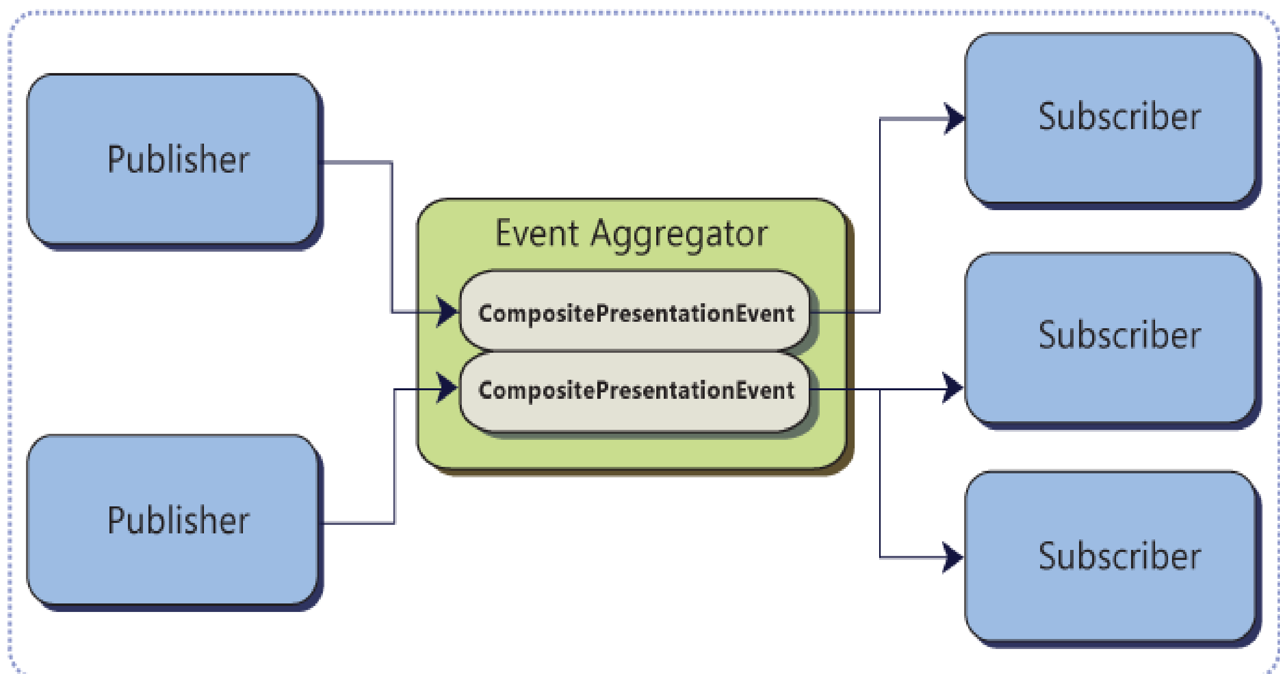
Pro účely testování je nutno zvolit konkrétní testovací nástroj. V zadání práce byl původně zvolen nástroj csUnit [23], což je open source testovací nástroj pro .NET framework. Je těsně spjat s TDD refaktoringem a ostatními technikami z agilního vývoje softwaru jako je například extrémní programování [24]. Bohužel během vývoje bylo zjištěno, že tento nástroj je již několik posledních let nerozvíjen a tudíž není kompatibilní s vývojovým prostředím Visual Studio 2010. Byl tedy vybrán jiný nástroj a to NUnit [8]. Přechod z csUnit na NUnit byl velmi bezešvý, neboť oba tyto nástroje používají stejné atributy pro definování testů a i stejné třídy s velmi podobnými signaturami metod.

NUnit je volně šiřitelný testovací nástroj pro .NET Framework. Původně byl portován z Junit [25] a nyní je napsán kompletně v jazyce C#. NUnit patří mezi rodinu xUnit [26] nástrojů, které jsou založeny na návrhu podle Kenta Becka. Dokáže pracovat s jakýmkoliv .NET jazykem, což je pro IMPRODEMO užitečné, neboť třídy obstarávající matematické výpočty se budou přesouvat a zabalovat do wrapperů, které budou napsány v C++/CLI [18].

2.5 Další použité technologie

Kromě výše uvedených technologií se v průběhu návrhu a vývoje ukázalo, že je užitečné do práce na IMPRODEMO zahrnout i další pomocné technologie, implementující například některé návrhové vzory. V drtivé většině jsou tyto již implementovány a použity vložením existujících knihoven pro jazyk C# a prostředí .NET.

2.5.1 Event Aggregator



Obrázek 2.3: Schéma EventAggregatoru.

Event Aggregator je prostředek pro realizováni volných vazeb v aplikaci. Je to komponenta vytvořená v rámci projektu PRISM [13], který poskytuje nejlepší postupy v aplikacích vyvíjených s vědomím budoucího rozšíření a s důrazem na snadnou udržovatelnost. Jeho jednoduchý princip spočívá v tom, že umožňuje definovat událost a tu potom posílat všem zájemcům (Subscriber). Zároveň lze tuto událost posílat z několika míst pomocí různých zasílatelů (Publishers). To všechno bez reference na konkrétní třídy, kterým chceme událost poslat. Vše, co potřebujeme, je Event Aggregator a typ události, který chceme posílat, nebo odebírat.

Další výhoda Event Aggregatoru spočívá ve faktu, že umožňuje posílat události i napříč assembly [14]. Není tedy nutné znát, z jakých assemblies se naše aplikace skládá a ani je v současné assembly referencovat.

2.5.2 Commanding

Systém příkazů, nazývaných *Commanding* je inspirován návrhovým vzorem Command [29] a je součástí frameworku WPF. Primárním účelem je reprezentace příkazu, čím se však liší od obvyklého zpracování příkazů pomocí ovladače událostí je zapouzdření všech potřebných informací k vykonání příkazu, abstrahování od vyvolávajícího objektu a vytvoření samostatné sémantické jednotky. Toto umožňuje několika nezávislým zdrojům vyvolat stejný příkaz a zároveň různým cílům umožňuje rozdílné zpracování stejného příkazu typu *Command*. Dalším smyslem tohoto systému je možnost určit, kdy je příkaz možno vyvolat. Obecně může být *Commanding* model přítomný ve WPF rozdělen na čtyři hlavní koncepty. Jsou jím: příkaz, zdroj příkazu, cíl příkazu a svázání příkazu.

Všechny příkazy usilující o to být součástí *Commanding* systému musí být třídy implementující rozhraní *ICommand*. Tedy musí podporovat jednak metodu *Execute*, která je určená k vykonání příkazu, dále metodu *CanExecute*, vracející boolovskou hodnotu informující, zda vyvolaný příkaz lze v daném stavu na daném cíli vykonat a jako poslední je nutno implementovat ovladač událostí *CanExecuteChanged*, který definuje logiku určující, zda-li daný příkaz lze vykonat. Manažer spravující příkazy tuto událost vyvolá v případě, že dojde k nějaké změně relevantní ke zdroji, vázaného na příslušný *Command*.

Zdroj příkazu je obecně jakýkoliv prvek, který příkaz může vyvolat. Jsou to objekty implementující rozhraní *ICommandSource*. Můžou jimi být například prvky uživatelského rozhraní jako tlačítka či položky menu. Rozhraní definuje tři položky. Jsou jimi: *Command*, která obsahuje samotnou třídu příkazu, *CommandTarget*, definující objekt nad kterém se má příkaz vykonat a nakonec *CommandParameter*, položka definující informace předané ovladačí událostí zpracovávající příkaz.

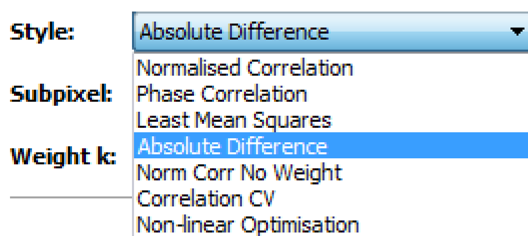
Cíl příkazu je prvek na kterém se příkaz vykonává. Pokud není explicitně nastaven, bere se cíl jako prvek s objektem vybraným pomocí klávesnice.

Systém *Commandingu* je užitečná součást WPF přispívající k modularitě programu a umožnující samostatné sémantické vyvíjení a testování příkazové funkcionality. Ve vývoji aplikace IMPRODEMO je tento systém hojně používán právě pro výše uvedené vlastnosti a nativní podporu WPF.

3 Objektový návrh aplikace IMPRODEMO

V této kapitole je popsán objektý návrh aplikace IMPRODEMO. Původcem všem funkcí pro IMPRODEMO je aplikace KORSYS, která poskytuje uživateli několik funkcí pro zpracování obrazu. Hlavní funkcní KORSYSu je 2D korelace obrazu, což je metoda, jak nalézt vzorek v obrazu, nebo-li, jak daný obraz „koreluje“ (souhlasí) s jiným. Digitální korelace obrazu je optická metoda běžně používaná v průmyslu na měření deformací, posunutí a pnutí v materiálu.

Existuje několik matematických metoda korelace a část z nich KORSYS poskytuje. Kromě obecné a základní metody hrubé síly [9] poskytuje další pokročilé metody, některé rychlejší a méně přesné, nekteré pomalejší a více přesné. U každé metody je možno nastavit několik různých parametrů. Obrázek 3.1 ukazuje metody dostupné v KORSYSu a Obrázek 3.2 druhy parametrů, co se dají u korelace nastavit.

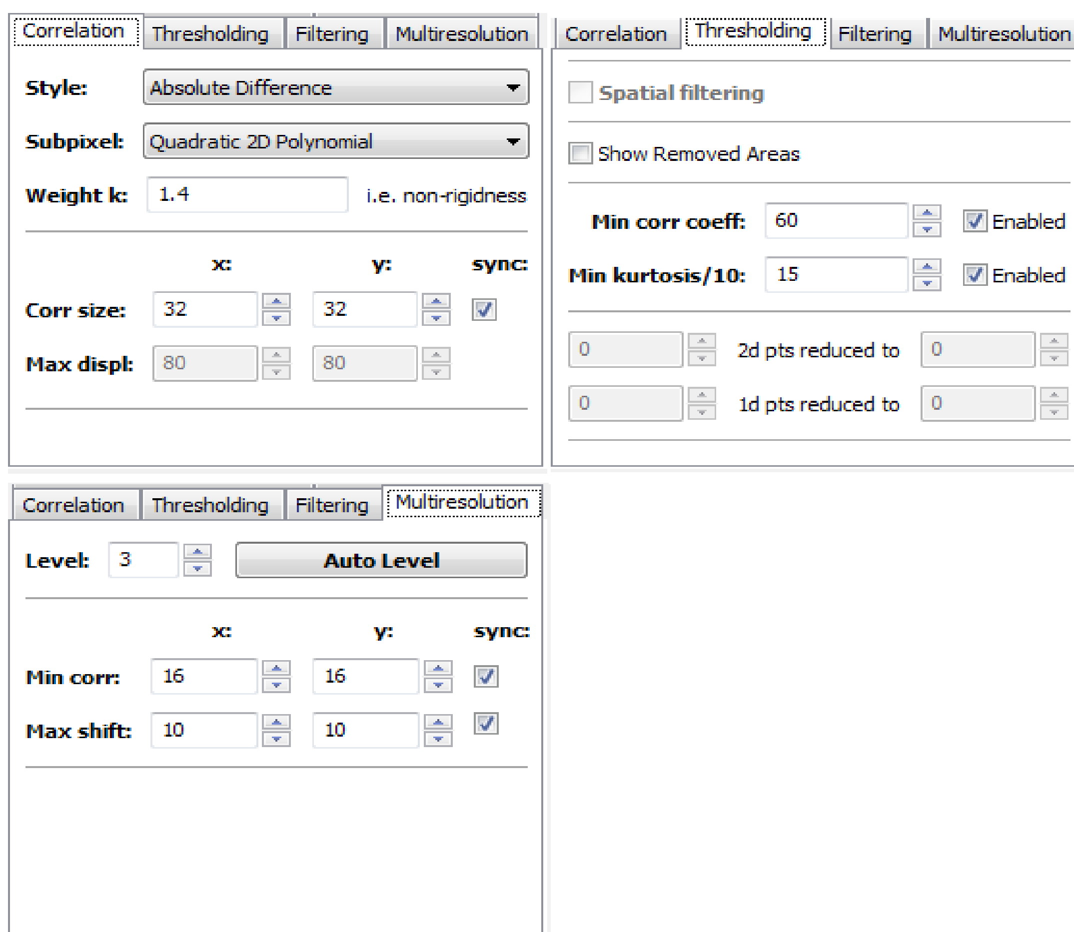


Obrázek 3.1: Druhy korelace v KORSYSu.

Další z funkcí, které KORSYS obsahuje je sledování markerů. Pomocí speciální značky (markeru), která se umístí na objekt snímaný kamerou je možno v obraze tuto značku rozpoznat a polohu značky potom vynést například do grafu, nebo jí spárovat s jinou značkou a do grafu vynášet jejich vzdálenost. Obecně lze softwarově polohu značky jakkoliv zpracovat a do grafu lze například vynášet derivaci polohy značky, tedy její rychlost.

KORSYS je software pro obecné použití, znamená to tedy, že korelaci a sledování markerů lze provést na jakémkoliv vstupu, který mu je předložen, ať už na kameře, nebo na samostaných obrázcích. Časem přišel požadavek na software, který je určen pro konkrétnější aplikaci a který má v balíčku s hardwarovým vybavením sloužit jako učební pomůcka do škol. Požadavky na tento software nebyly zprvu úplně jasné a zjišťovaly se iterativně. Základní požadavky však byly tyto:

1. Software musí umožňovat pokus typu mechanicka zkouška. Tento pokus zahrnuje natahování a deformování dodaného vzorku na trhacím hardwarovém zařízení, což na straně softwaru znamená korelaci obrazu.
2. Software má podporovat pokusy demonstrující fyzikální mechaniku objektů jako je kmitání či zrychlení. Tento typ pokusu vyžaduje na straně softwaru implementaci metod sledování markerů či nacházení objektů různých tvarů v obraze.

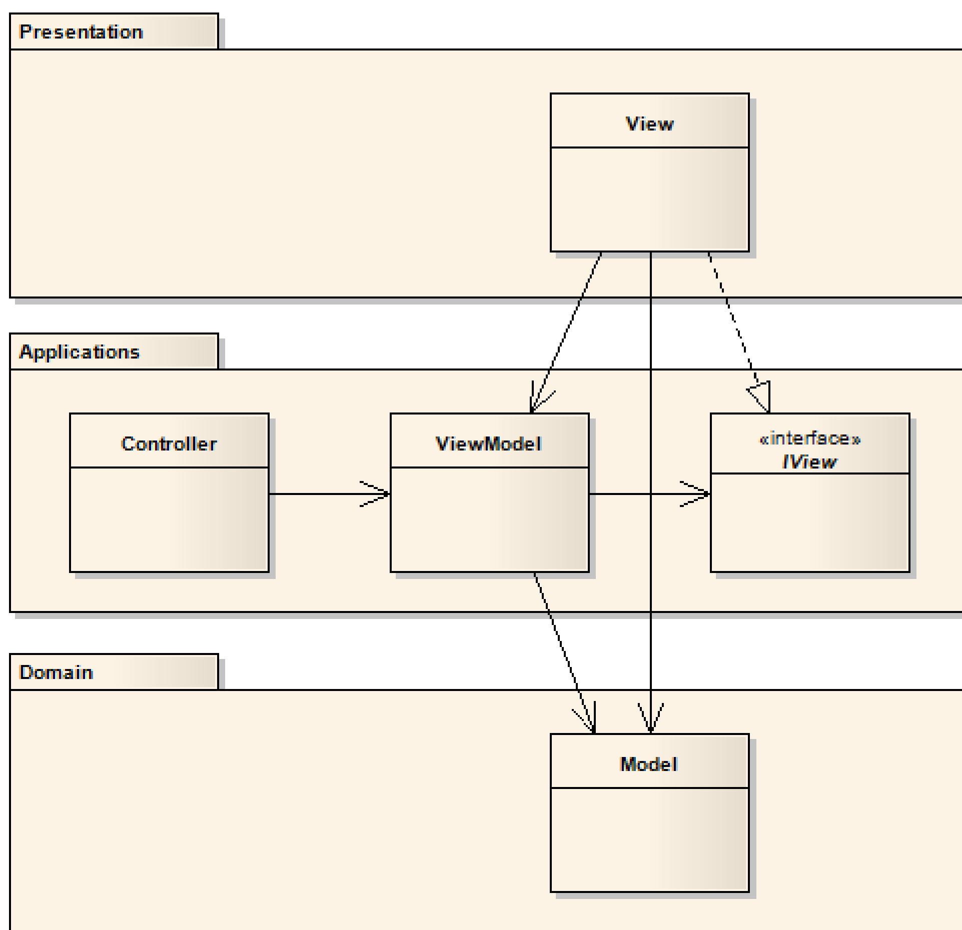


Obrázek 3.2: Parametry korelace v KORSYSu.

Dalším požadavkem na funkčnost je možnost kalibrace. Kalibrace v kontextu digitální korelace znamená možnost zobrazit uživateli délkové údaje ve skutečných fyzikálních jednotkách. Za předpokladu, že do obrazu vložíme značky a programu předáme jejich vzdálenost ve fyzikálních jednotkách, je možno tyto značky v obrazu najít a potom polohu či ostatní délkové údaje zobrazovat ve skutečných fyzikálních jednotkách a nikoliv v počítačových (jako jsou pixely), což je pro uživatele často nepoužitelné. Tato funkcionalita již byla naimplementována v KORSYSu, proto se do IMPRODEMO přenese ve formě třídy zabalené do příhodného *wrapperu* napsaného v jazyce C++/CLI.

Protože software je určen pro školy, předpokládá se základní schopnost práce s počítačem a tudíž byl vznesen požadavek na vysokou ergonomii uživatelského rozhraní zahrnující především jednoduchost a intuitivnost ovládání. Zároveň, v rámci zaujmutí a zvýšení atraktivnosti softwaru při nabízení potenciálním zákazníkům, přišel požadavek na estetickou stránku GUI, tedy moderní a čistý vzhled aplikace. Ten by měl obstarat použití WPF, který poskytuje velmi jednoduchou možnost skinování a použití grafických témat.

Kapitola 3.2 o MVVM podala čtenáři náhled a přehled o tom, co to MVVM je a jak funguje. Nyní se vytvoří třídy a umístí se do kontextu tohoto návrhového vzoru. Na obrázku 3.3 je diagram znázorňující architekturu MVVM. Tato konkrétní forma je přítomná ve WPF Application Frameworku (WAF) [10], který bude v IMPRODEMO použit a který usnadňuje použití MVVM ve WPF.



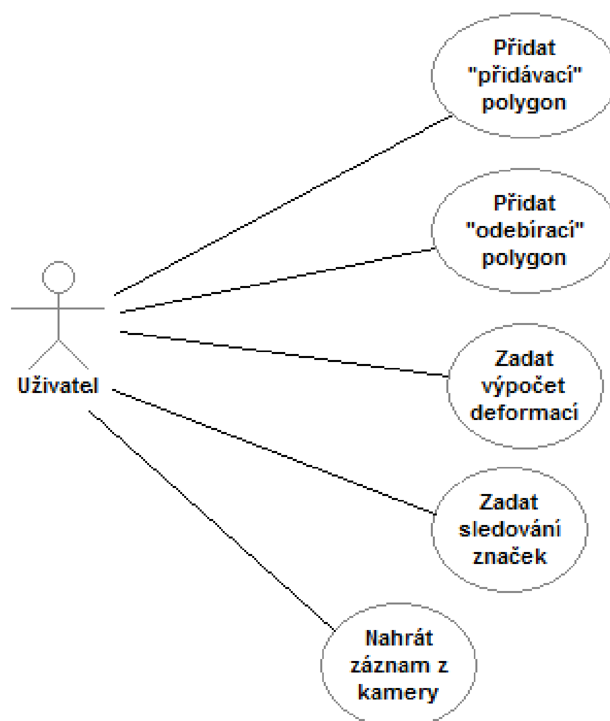
Obrázek 3.3: Architektura MVVM podle [10].

Nyní v každé podkapitole předvedu třídy, které v dané vrstvě budou přítomné, ukážu jejich diagram tříd, tak jak jsem ho navrhnul a popíši několik případů použítí. Budu sestupovat postupně od nejvyšší třídy View, protože grafickým návrhem je v našem případě určená funkcionalita celé aplikace. Tento grafický návrh je možno si prohlédnout v příloze A.

U všech tříd se vyskytuje operace, která má název shodný s názvem třídy. Tuto operací nazýváme konstruktor a slouží ke konstrukci daného objektu, nastavení počátečních vlastností a uvedení do výchozího stavu. Pokud tato operace nemá žádnou speciální funkcionalitu, není detailně popisována.

3.1 View

Podle návrhu grafického uživatelského rozhraní (dál jen GUI) v příloze A byl v rámci vrstvy MVVM modelu View vytvořen jednoduchý případ užití zobrazený na Obrázku 3.4.



Obrázek 3.4: Případ užití pro View.

Podle případu užití byly navrhnuty tyto třídy:

- *AdvancedOptionsWindow* reprezentující okno s názvem „Advanced Options", kde se dá nastavovat pokročilé nastavení aplikace,
- *GraphWindow* reprezentující okno s grafy, zobrazující polohy nalezených objektů v obraze
- *MainWindow*, která reprezentuje třídu hlavního okna, kam se budou vykreslovat všechny výsledky operací stejně tak jako zobrazované snímky a
- *ToolsPanel* reprezentující okno s panely nástrojů, pomocí kterého se bude ovládat celá aplikace.

Na obrázku 3.5 vidíme diagram tříd pro tyto třídy. Všechny třídy budou dědit ze standartní WPF třídy *Window*. Nyní bude popsán smysl třídních metod a proměnných uvedených na obrázku 3.5. Všechny jsou silně svázány s grafickým návrhem uvedeným v příloze A. Pro porozumnění návrhu je tedy nutné konfrontovat informace uvedné zde s informacemi uvedenými v tomto návrhu.

GraphWindow

GraphWindow je třída reprezentující okno s grafy. Vzhledem k tomu, že je náleží do části View, měly by všechny její metody pracovat pouze se zobrazovanými daty a měly by poskytovat metody, odrážející toto chování, nižším vrstvám. Tato třída a její metody by něměly vědět nic o tom, odkud se data, co přicházejí, berou nebo co znamenají. Třída obsahuje metodu *AddHistoryChannel*, která přidá do grafové komponenty kanál, reprezentující pozici markeru vybraného v prvku ComboBox, obsahující nalezené markery. S touto metodou dále souvisi metoda *AddHistoryData*. Ta umožní přidat hodnotu do specifikovaného kanálu, přidaného pomocí předchozí metody.

Kanály je samozřejmě možné i smazat. K tomu slouží metoda *DeleteLastChannel*, která přímo odráží tlačítko se stejným popiskem, jak je vidět v grafickém návrhu v příloze A. Není tedy

možné smazat jakýkoliv kanál a to z důvodu zmíněné jednoduchosti ovládání aplikace. Třída však obsahuje možnost smazat kanál s předaným indexem. Metoda je však soukromá a není tedy určená pro veřejné použití. Tuto metodu používá právě metoda *DeleteLastChannel* v rámci zobecnění mazání grafu.

Grafové okno dále umožňuje nastavit celé okno do výchozí pozice pomocí *ResetHistoryGraph*. To obnáší smazání všech kanálů a nastavení obsahu prvků *ComboBox* na jejich výchozí hodnoty. Veřejná metoda *ShowHide* implementuje *toggle* funkcionalitu zobrazení či skrytí grafového okna.

Okno obsahuje dále možnost zapnout automatickou detekci a přidání markerů a automatické mazaní grafů, které jsou již neaktivní. Oproti standartním GUI aplikacím, které by obsahovaly ovladač událostí zpracovávající zaškrtnutí prvku *CheckBox,* je v IMPRODEMO využit WPF koncept *databindingu*. Proto třída o grafických prvcích popsaných v XAML souboru neví, pokud se neoznačí atributem *x:Name*.

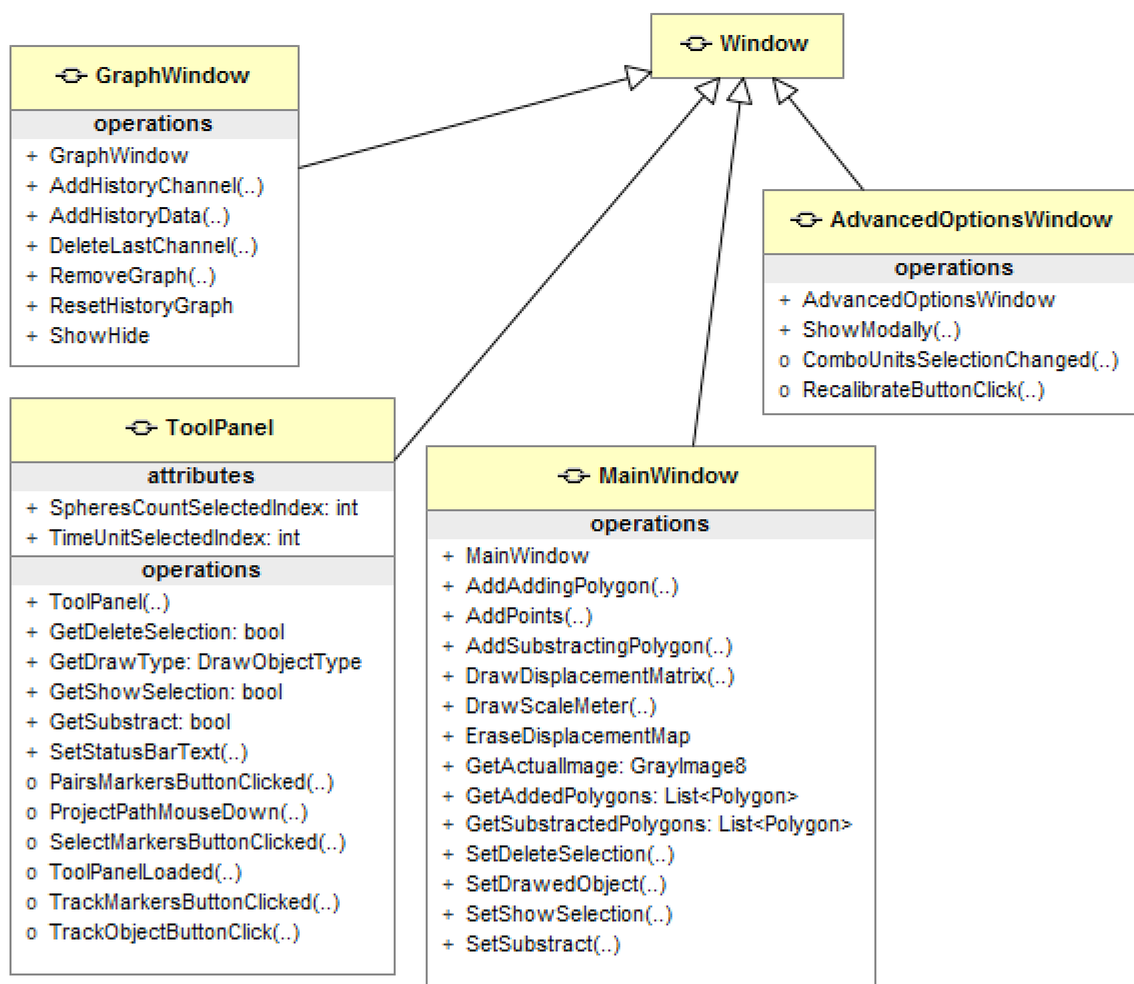
AdvancedOptionsWindow

Tato třída obstarává práci s oknem grafů. Poskytuje několik málo metod, kromě konstuktoru je zde přítomna metoda *ShowModally,* přikazující oknu vykreslení v modálním režimu. Dále ovladač události *ComboUnitsSelectionChanged,* umožňující zpracovat reakci na klik na prvek *ComboBox* ukazující jednotky. Konkrétní funkcionalita se bude obsluhovat nastavením jednotek ve sdílené třídě reprezentující nastavení aplikace. Poslední ovladač události nazvaný *RecalibrateButtonClick* bude přistupovat ke službě kalibrace, která bude obsažená v doménové části (nebo-li vrstva *Model*) aplikace. Výsledek rekalibrace bude poté určovat koeficient, kterým se budou násobit obrazové jednotky (pixely) tak, aby se rozměry a polohy v obraze zobrazovaly podle zvolené fyzikální jednotky.

MainWindow

Třída hlavního okna bude poskytovat více funkcionality, než jen pouhé zobrazení aktuálního snímku, který přichází z kamery nebo ze záznamu. Bude možné na něm interaktivně vybírat korelovanou oblast myší, přičemž klikem se budou umisťovat vrcholy polygonu. Bude možné s vybraným polygonem posouvat a mazat ho. Všechno toto bude implementováno v soukromých ovladačích událostí okna. Pro okolní svět a především *Controllery*, které zobrazování v oknech ovládají, jsou připraveny metody jako *AddAddingPolygon* a *AddSubstractingPolygon*, což jsou metody, které přijímají soubor bodů, konvertují je v typ zobrazitelný v canvasu okna a zobrazí ho na danou polohu. Lze přidat jednotlivé body do aktuálně aktivního a kresleného polygonu pomocí *AddPoints*.

Pokud zapneme funkci korelace obrazu, musíme mít možnost vypočtená data zobrazit v hlavním zobrazovacím okně (*MainWindow*). K tomu slouží metoda *DrawDisplacementMatrix*, která matici čísel konvertovanou na barevnou bitmapu zobrazí na zadané souřadnice v okně. Aby bylo jasné, jak velké posunutí je každou barvou reprezentováno, umožňuje *DrawScaleMeter* převzít maximální a minimální hodnotu matice posunutí a vygenerovat grafický prvek informující uživatele o posunutí.



Obrázek 3.5: Diagram tříd pro View.

Protože je při výpočtu, který je volaný z *ToolsPanelu,* potřeba snímek zobrazený aktuálně v hlavním okně a informace o vyznačené oblasti zájmu, poskytuje *MainWindow* metody jako *GetAddedPolygons* a *GetActualImage*.

Metody *SetDeleteSelection* a *SetSubstract* slouží k nastavení módu práce se zájmovou oblastí. Pomocí *SetDrawedObject* lze nastavit typ kresleného objektu. V grafickém návrhu v příloze A byl navržen kromě polygonu také čára a bod. Ty mají umožňovat vybrat konkrétní oblast hodnot ze zobrazené barevné mapy, která se má vynést do grafu.

ToolsPanel

Třída *ToolsPanel* je poslední v části *View* a obsahuje několik metod pro práci s celým ovládacím panelem. Metody *GetDeleteSelection, GetDrawType, GetShowSelection a GetSubstract* poskytují informace o stavu *toggle* tlačítek umístěných v *ToolsPanelu*. Metoda *SetStatusBarText* umožňuje předaný řetězec nastavit v dolní oblasti aplikace jako status a pomoci tak uživateli v orientaci v probíhajících procesech.

Metody *PairsMarkersButtonClicked, SelectMarkersButtonClicked, TrackMarkersButtonClicked a TrackObjectButtonClicked* jsou ovladače událostí reagující na zamáčknutí tlačítek aktivujících různé metody *marker trackingu*. Tyto metody předávají řízení

do nižších vrstev aplikace, kde se konkrétní služby snaží v obraze najít markery. Metoda *ProjectPathMouseDown* předává řízení otevření cesty pro projekt.

Jak je vidět z Obrázku 3.3, může View komunikovat se všem vrstvami ležící pod ním. Má přímý přístup ke všem vrstvám a tudíž je může přímo volat. Toto bude v aplikaci konkrétněji využito. Zde navrhovaný diagram je návrh, který se s konkrétní implementací změní. Bude doplněno množství metod souvisejících s detaily aplikace, které nejsou pro základní myšlenku a filozofii aplikace důležité a nijak ji neovlivňují.

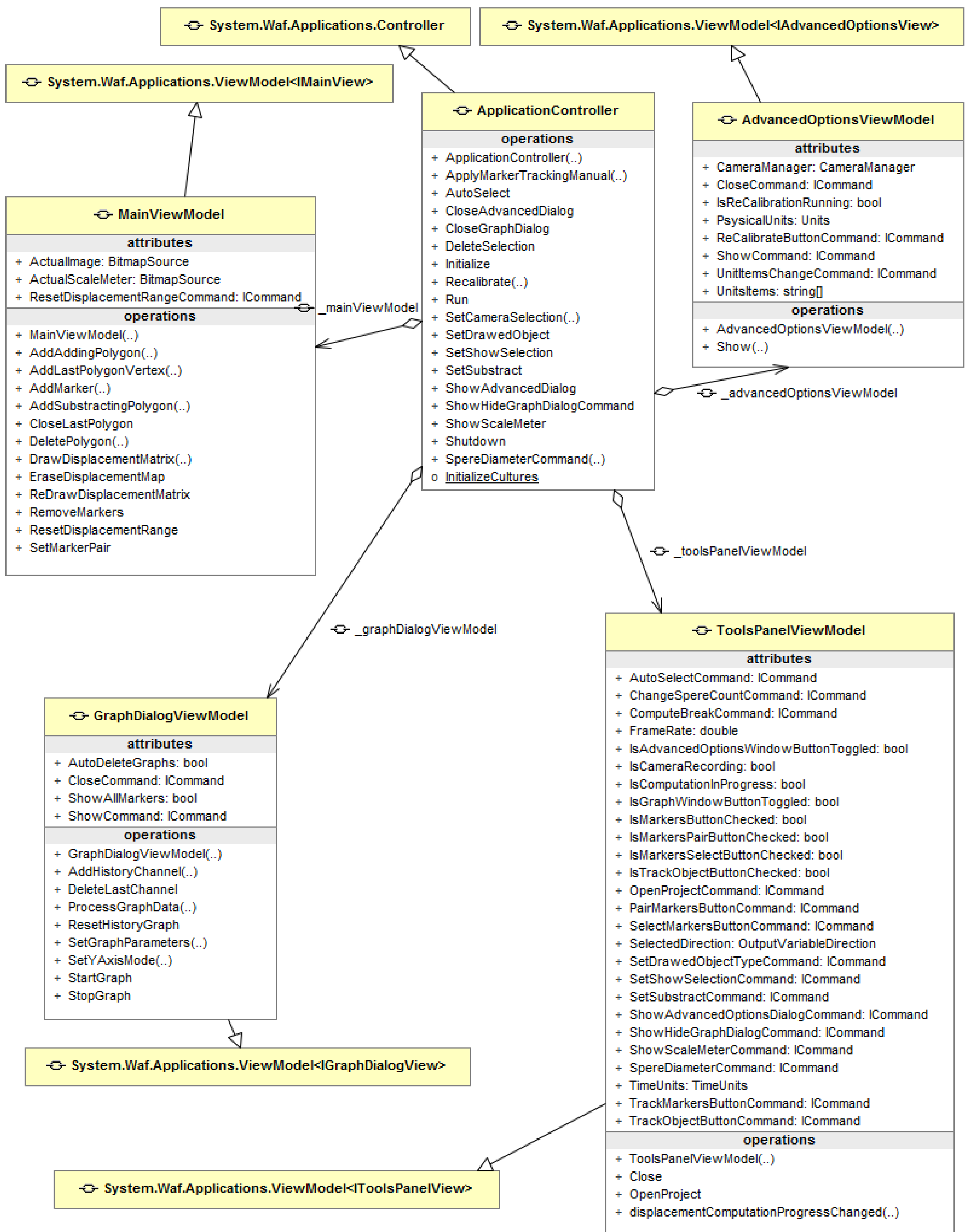
3.2 ViewModel

Jak víme z kapitoly 2.2, ViewModel je vrstva, která zpracovává data z uživatelského rozhraní a předává je ke skutečným výpočtům do vrstvy Model. Vrácená data poté konvertuje do podoby vhodné k zobrazení a poskytuje je zpět vrstvě View. Na obrázku 3.6 vidíme diagram tříd pro tuto vrstvu. Nyní si třídy ve vrstvě ViewModel blíže popíšeme, včetně navrhované funkcionality v jejich metodách.

Třídy typu ViewModel

Mezi tyto třídy patří *GraphDialogViewModel*, *MainViewModel*, *AdvancedOptionsViewModel*, *ToolPanelViewModel*. Tyto třídy fungují jako příjemce zpráv ze třídy nacházející se ve *View – GraphDialogView*. Tyto zprávy jsou třídy, které implementují rozhraní ICommand (kapitola 2.5.2) a jejich zasílání se neobjevuje v popisu tříd z *View*. Nachází se totiž v XAML souboru těchto tříd a tak nejsou součástí třídního návrhu.

Všechny operace tříd v této vrstvě jsou dány příslušnými rozhraními z *frameworku* WAF. Toto rozhraní, jak uvidíme později, slouží k přístupu k *ViewModelům* z části *Model*. Přestože podle architektury MVVM na Obrázku 3.3 nemá *Model* k *ViewModel* části přístup (komunikace je jen jednostranná), je možno tuto komunikaci zabezpečit pomocí MEF (podkapitola 2.3).



Obrázek 3.6: Diagram tříd pro ViewModel.

GraphDialogViewModel

Atributy třídy *GraphDialogViewModel* reflektují některé prvky uživatelského rozhraní, které jsou s těmito vlastnosmi (v jazyce C# se pro ně používá výraz Properties [12]) svázány pomocí mechanismu *databindingu*. Jsou jimi například *ShowAllMarkers* a *AutoDeleteGraph*. Dalšími atributy jsou třídy typu *ICommand*, které slouží k oddělenému zpracování akcí, které mohou být vyvolány interakcí s uživatelským rozhraním. V případě této třídy jimi jsou *CloseCommand* a *ShowCommand*, které se starají o správné zavření a schování okna s grafy.

Co se týče metod poskytovaných třídou *GraphDialogViewModel*, jsou tu metody přijímající data v „hrubé" podobě z *Modelu* a konvertující a předávající zkonvertovaná data do příslušného View. Mezi tyto metody patří *AddHistoryChannel* a *DeleteLastChannel.* Zbytek metod využívá *ApplicationController* řídící workflow celé aplikace k vykonání požadovaných akcí. Přepokládá se, že tyto metody jako mezičlánek mají v části *ViewModel* smysl, neboť mohou provádět ještě nějaké konverzní či čístící operace. Je však možné, že v konkrétní implementaci bude *ApplicationController* přistupovat přímo k View přes jeho rozhraní.

MainViewModel

Třída *MainViewModel* poskytuje přístup k zobrazenému obrázku pomocí *ActualImage*, k zobrazenému prvku *ScaleMeter* pomoc *ActualScaleMeter,* a vzhledem k tomu, že ScaleMeter je součástí jenom hlavního okna, bylo vybráno, aby obslužné tlačítko resetující jeho rozsah spouštělo příkaz, umístěný právě v *MainViewModel,* a to *ResetDisplacementRangeCommand*.

Dále jsou v *MainViewModel* umístěny metody umožnující přidat polygon do zájmové oblasti v hlavním okně pomocí metod *AddAddingPolygon*, *AddSubstractingPolygon* a *AddLastPolygonVertex.* Metody přejímají body a konvertují je na příslušnou třídu zobrazitelnou ve *View.* Manipulaci s polygony doplňují metody *DeletePolygon* a *CloseLastPolygon*. Poslední zmiňovaná metoda určí, že na hlavním okně se kliknutím přidávají vrcholy k novému polygonu a starý je pro přidávání vrcholů již uzavřen. Stav okna *MainWindow* je tedy reflektován a držen v jeho *ViewModelu*.

Další skupina metod souvisí s výpočtem korelací a zobrazení příslušné barevné mapy, tak jak je vidět v příloze A na záložce *iBreak*. *DrawDisplacementMatrix* přejímá matici z výpočetní jádra a konvertuje ji na barevnou bitmapu zobrazitelnou v hlavním okně. Tu poté nechá vykreslit. *EraseDisplacementMap* ji smaže a nakonec *ReDrawDisplacementMatrix* ji vykreslí s aktuálními nastaveními minimálních a maximálních hodnot rozsahů. Tyto hodnoty se s každou další vypočtenou mapou můžou měnit a součásti soukromé sekce třídy *MainViewModel* je aktuální stav těchto hodnot a rozsahů. V případě, že je kliknuto na „*Reset Range*" (a spuštěna metoda *ResetDisplacementRange*), tlačítko prvku *ScaleMeter* tyto rozsahy upraví podle aktuální matice a zmíněná metoda umožní překreslit stejnou matici s jinými rozsahy.

Poslední skupina metod souvisi s markery, které jsou v taktéž v hlavním okně zobrazovány a to v případě že jsou nalezeny v obraze. Je možné marker přidat do zobrazení v hlavním okně. Vrstva Model však neví, jakým způsobem se marker zobrazuje, takže jen předá typ zobrazovaného markeru. To umožňuje metoda *AddMarker. SetMarkerPair* přijímá dva markery a způsobí vykreslení spojné úsečky v hlavním okně, indikující nastavený pár markerů. Vzdálenost markerů se potom může vynést do grafu. Příkaz *RemoveMarkers* jednoduše odstraní všechny zobrazené markery.

AdvancedOptionsViewModel

ViewModel určený k uchovávání stavu okna *AdvancedOptionsWindow* neposkytuje žádné metody ke konverzi. Neprobíhá zde žádná složitější logika a tak jsou zde prakticky jen property, reflektující stav okna a *databindingem* svázané se zobrazovanými ovládacími prvky, a *ICommand* třídy vykonávající akce spuštěné po interakci s UI. *CameraManager* je třída poskytující prostředky pro správu kamer, jejich výběr, nastavení či zjištění posledního snímku. Zde má svoje místo, neboť v okně *AdvancedOptions* je možnost přepnutí na jinou kameru. *CloseCommand*, *ReCalibrateButtonCommand* a *UnitsItemChangeCommand* jsou příkazy implementující funkcionalitu, která se spouští po zavření okna, stlačením tlačíka „*Calibrate*“ či výběru jednotky v prvku *ComboBoxu*. Příkazy připraví potřebná data a poté předají řízení *ApplicationControlleru*, který distribuuje potřebné příkazy ostatním zůčastněným třídám.

ToolsPanelViewModel

Ve *ViewModelu* pro *ToolsPanel* se nachází velké množství příkazů reprezentovaných třídou implementující rozhraní *ICommand*. Všechny slouží jako reakce na interakci s uživatelským rozhraním a jejich funkce je v zásadě taková, že připraví potřebné zdroje pro vykonání akce a ty potom předají *ApplicationControlleru*, který řídí workflow celé aplikace.

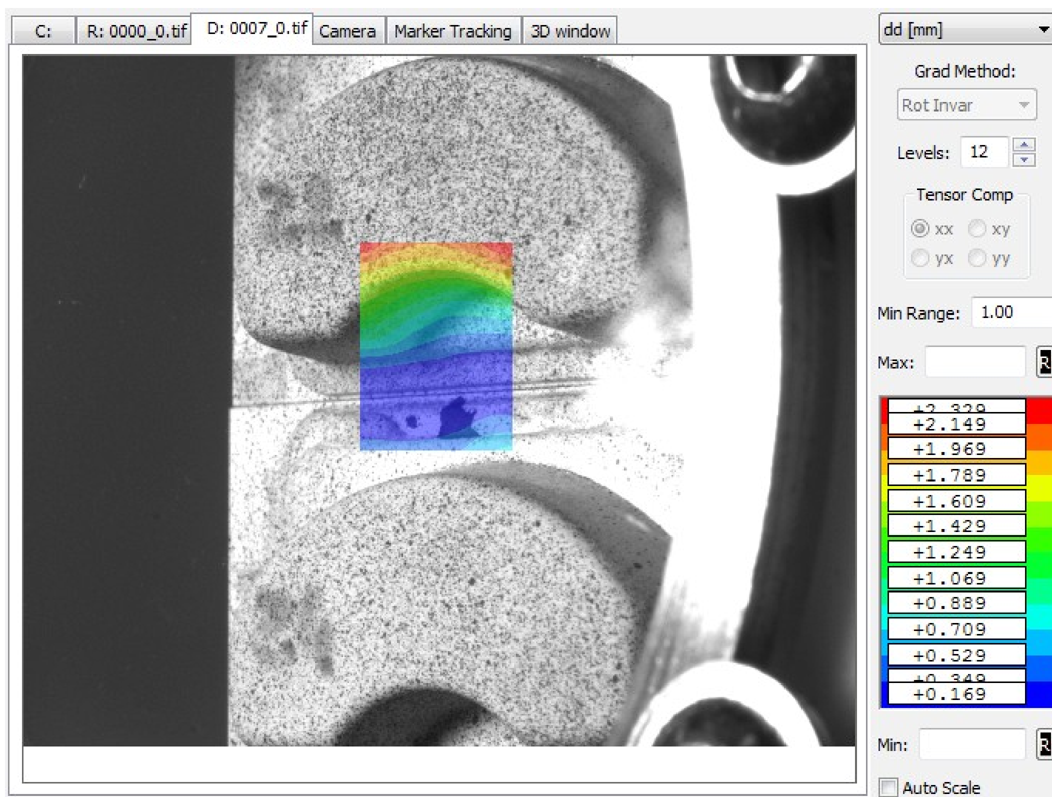
Dalšími skupinou atributů *ToolsPanelViewModelu* jsou properties svázané pomocí *databindingu* s vlastnostmi komponent v XAML souboru *ToolsPanelView*. Zde uvádím příklad svázání property *FrameRate* s komponentou *TextBox* umístěnou v ToolsPanelu.

Definice prvku UI v XAML souboru:

```
<TextBox x:Name="frameRateTxtBox" IsReadOnly="False" Margin="5">
    <TextBox.Text>
        <Binding
            Converter="{StaticResource frameRateUnitsConverter}"
            Mode="TwoWay"
            Path="FrameRate" >
        </Binding>
    </TextBox.Text>
</TextBox>
```

Zde vlastnosti tagu *Binding* určují několik důležitých informací. Zaprvé je jím cíl *bindingu* v atributu *Path*. Identifikátor vložený do závorek určuje název property umístěné ve *ViewModelu*. Zadruhé je jím mód *databindingu*. Řetezec „*TwoWay*“ určuje standartní oboustranou komunikaci. Znamená to, že úprava property se promítne do vlastnost *TextBox.Text*, stejně tak jako úprava této vlastnosti se promítne do obsahu property. Způsob, jakým se promítne je buď přímá kopie, v případě, že jsou property a vlastnost stejného typu, nebo nějaká konverze. WPF již obsahuje některé přirozené konverzní mechanismy (celé číslo na desetinné), zde však probíhá konverze komplikovanější, neboť hodnota zobrazená v UI se mění podle jednotek vybraných v ComboBoxu v horní částí ToolsPanelu. Naproti tomu hodnota obsažená v property *FrameRate* je vždy ve snímcích za sekundu. Použije se proto instance dodatečné konverzní třídy nazvané *frameRateUnitsConverter*, která tuto konverzi zajistí.

Všechny atributy, vyskytující se na diagramu tříd na obrázku 3.6 v části *ToolsPanelViewModel*, které neimplementují rozhraní *ICommand,* jsou property svázané s příslušným prvkem v UI.



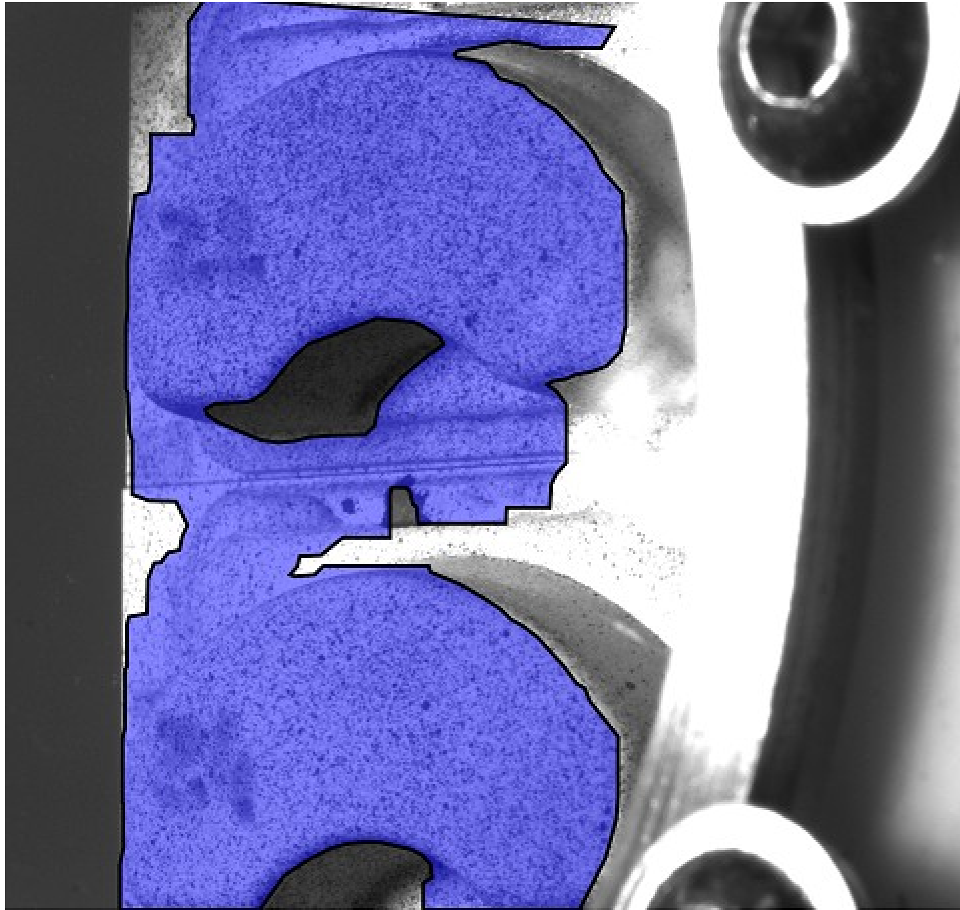
Obrázek 3.7: Zobrazený posunutý snímek.

ApplicationController

Tato třída je řídícím centrem celého programu. Obsahuje logiku příkazů vyvolaných z UI a má reference na všechny důležité součásti programu. Metoda *ApplyMarkerTrackingManual* nastaví odebírání snímků ze zdroju snímků a jeho zpracování *marker trackerem*. Výsledek se poté předává *MainViewModelu*, který markery příslušně zobrazuje.

AutoSelect je metoda automatického nalezení oblasti v obrazu, která je oblastí zájmu pro korelaci, nebo-li počítání deformací. Jejím vstupem je aktuálně zobrazený snímek a výstupem množina polygonů identifikující zájmovou oblast. Je to oblast materiálu, který byl pokryt speciálním postříkem, tak, jak je vidět na obrázku 3.8. Na tomto obrázku vidíme snímek průmyslového výrobku nanesený speciálním postřikem tak, aby bylo možné jednotlivé částice v procesu korelace rozpoznat a určit tak deformaci materiálu. Po zapnutí korelace byla automaticky detekování určitá část nanesená postřikem. Vidíme dokonce přidání tzv. substracted polygonu, tedy polygon, který říká, že jeho oblast se má z výpočtu vynechat.

Proces rekalibrace, zde reprezentován metodou *Recalibrate,* se sestává, podobně jako AutoSelekce, ze získání aktuálně zobrazeného snímku a poté zavolání příslušné služby z části *Model*. Ta v tomto případě ale hledá v obraze několik značek, které jsou v předem nastavené vzdálenosti. Podle toho, kde je v obraze kalibrační proces najde (a jak velké budou) dokáže nastavit správné rozměry a zobrazovat potom všechny polohy a vzdálenosti ve skutečných fyzikálních jednotkách.



Obrázek 3.8: Automaticky zvolená oblast zájmu.

Další metody, jako *SetCameraSelection, SetDrawedObject, SetShowSelection* a *SetSubstract* prakticky jen předávají stav z jednoho ViewModelu do druhého. Například *SetSubstract* předává zapnutí kreslení substract polygonu z *ToolsPanelViewModel* do *MainViewModel*, neboť právě zde se populují události kliknutí na hlavní okno a přidání vrcholu do zvoleného typu polygonu.

ApplicationController defacto implementuje část *Controller* tak, jak je zobrazená v diagramu MVVM podle WAF (Obrázek 3.3). Je tedy jakýmsi rozšířením standartního MVVM a skloubením tohoto návrhového vzoru a návrhového vzoru MVC [31]. Je to jedno z vylepšení, které je v programu typu IMPRODEMO velice výhodné, neboť program obsahuje pokročilejší řídící logiku a netriviální kooperaci zůčastněnných komponent. Část *Controller* ve formě třídy *ApplicationController* je tedy velmi výhodná zejména z hlediska separování logiky od prezentace a místa uschování dat.

3.3 Model

Jak již bylo řečeno v kapitole 2.2, vrstva Model z obecného hlediska reprezentuje *bussiness* vrstvu aplikace a poskytuje přístup ke službám a datům používaným v aplikci. Z pohledu IMPRODEMO *Model* poskytuje výpočty a zajišťuje funkční část celé aplikace. Zajišťuje několik nezbytných funkcionalit a těmi jsou: vstup z kamery, kalibrace, práce se snímky, selekce, korelace a sledování markerů.

Na obrázku 3.9 vidíme diagram tříd pro vrstvu *Model*. Tento diagram obsahuje pouze třídy pro práci s kamerou a obecně zdroji snímků, třídy pro počítání korelace a třídu reprezentující selekce.

Třídy pro práci s markery a kalibrací jsou uvedeny až na obrázku 3.10. Následuje popis hlavní funkcionality a navrhované komunikace těchto tříd.

CameraManager

Tato třída slouží ke spravování kamer připojených k počítači. Umožňuje je jednoduše vytvářet, nastavovat hromadně několik vlastností (např. *FrameRate*), umožňuje získat aktuálně běžíci kameru (*GetActiveCamera*) a nastavit jestli má přijímat snímky či ne (property *IsActiveCameraRunning*). V class diagramu můžeme vidět, že *CameraManager* používá 1 až n *CameraSource* objektů.

CameraSource

Třída, reprezentující objekt kamery. Dědí z objektu *CameraWrapper,* což je *wrapper* napsaný v jazyce C++/CLI poskytující nastavení a práci s připojenou kamerou. Detaily o wrapperu jsou v kapitole 5, zde si jen popíšeme tuto třídu a co za funkcionalitu navíc by měla v prostředí C# a IMPRODEMA přinášet. Jedná se o příhodnou možnost získat poslední získaný snímek pomocí přístupu k property *LastImage*. Ten je možno označit jako zpracovaný pomocí *SetFrameAsProcessed.*

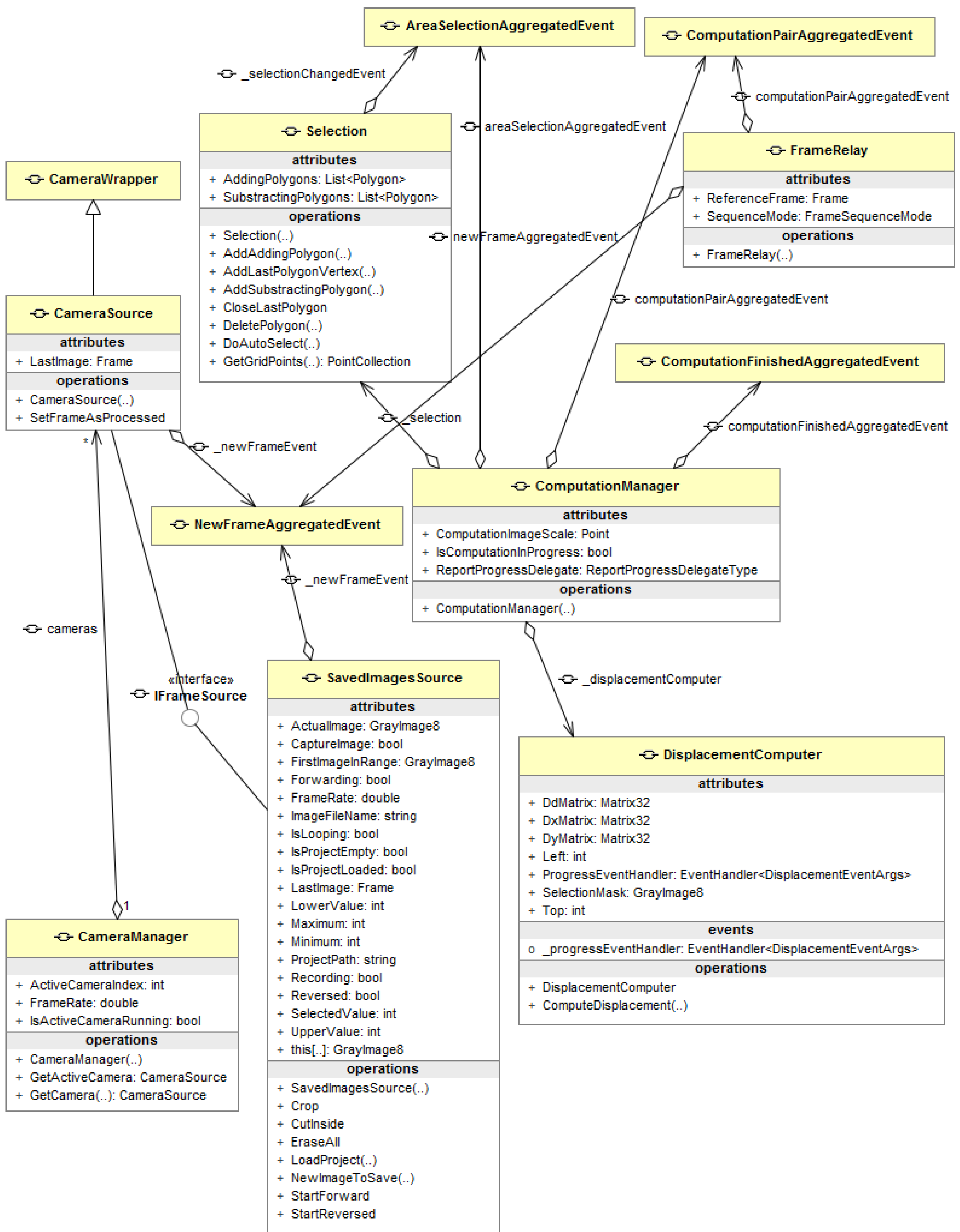
Důležitý mechanismus používaný k šíření nových snímků spočívá v implementaci rozhraní *IFrameSource*. To vynucuje publikování snímků pomocí *EventAggregatoru* (podkapitola 2.5.1). Příspívá tak k volnějším vazbám v aplikaci. Objekt *CameraSource* bez toho, aby věděl, kdo všechno bude snímek odebírat, jednoduše ve chvíli, kdy zaregistruje nové příchozí snímek z kameru, vyvolá událost *NewFrameAggregatedEvent* pomocí *EventAggregatoru* a všem subjektům, které se zaregistrovali k jeho odebírání je ve specifikovaných ovladačech událostí daná událost předána.

SavedImageSource

Stejně jako *CameraSource* poskytuje *SavedImageSource* nově příchozí snímky z kamery díky tomu, že implementuje rozhrani *IFrameSource*, které vynucuje používání *NewFrameAggregatedEvent*. Kromě toho má tato třída ještě funkcionalitu nahrávání záznamu z disku podle specifikované cesty. Používá k tomu třídu *XMLOutput* (Obrázek 3.10). Dále poskytuje rozhraní pro manipulaci se záznamenanými snímky. Je možné nastavit index aktuálně zobrazeného snímku pomocí *SelectedValue*. Zároveň je možné nastavit indexy rozsahů, ve kterých se index snímku může pohybovat pomocí *LowerValue* a *UpperValue*. Pro lepší představivost je vhodné podívat se na návrh uživatelského rozhraní v příloze A na stranu obsahující návrh *ToolsPanel*. V části *sequence operation* se nachází prvek *Frame Seeker*, jehož horní posuvník obsahuje dva ukazatele udávající rozsah definovaný právě pomocí *LowerValue* a *UpperValue*.

Vzhledem k tomu, že *SavedImageSource* se používá pro manipulaci se zaznamenanými snímky, je možné specifikovat cestu k projektu s uloženými snímky pomocí *ProjectPath*. Projekt v nastavené cestě lze poté otevřít pomocí přikazu *LoadProject*.

Při přehrávání snímků uložených na disku lze zvolit rychlost jejich přehrávání (*FrameRate*), zda se po přechodu z posledního snímku bude opět pokračovat na prvním (*IsLooping*), či zda se bude přehrávat v opačném směru (*Reversed*).



Obrázek 3.9: Diagram tříd pro vrstvu Model část 1.

Na záznamu lze provádět operace *Crop* (vyříznutí všech smímků mimo rozsah nastavený pomocí *UpperValue* a *LowerValue*), *CutInside* (opak předcházející operace, vyřízne snímky ve stanoveném rozsahu) a *EraseAll* (smaže všechny snímky).

Selection

Třída reprezentuje oblast zájmu pro výpočet korelací (modrá oblast v části Selekce v příloze A). Poskytuje možnost přidat polygon zájmu pomocí *AddAddingPolygon* nebo také polygon, který zájmovou oblast ruší pomocí *AddSubstractingPolygon*. Tento je vidět na obrázku 3.8 v šedivé barvě. Je možné přidat vrchol pouze k posledně přidanému polygonu, tato třída tak přímo odráží práci s polygony v hlavním okně.

Událost *AreaSelectionAggregatedEvent* slouží k notifikování všech zájemců, že selekce byla změněna. Obsahuje v sobě referenci na aktuální selekci a tím pádem aktuální seznam polygonů.

Poslední metodou, kterou *Selection* poskytuje je *GetGridPoints*. Ta poskytuje body pro výpočet korelace z obsažených polygonů. Detaily o výpočtu korelace jsou v kapitole 5.

FrameRelay

Při výpočtu korelací musí mít výpočetní algoritmus referenční a posunutý snímek. Při příchodu snímku z kamery je nutno určit, jestli je právě příchozí snímek referenční nebo posunutý. O to se stará právě tato třída. Obsahuje několik módu členění snímků, přičemž základní je takový, že při zapnutí počítání se první snímek označí jako referenční a každý další je posunutý. Tento mód se používá při meření deformací v materiálech. V případě pokusu typu „tok částic“ nás však zajímá průběžný pohyb částic, proto se referenční snímek pohybuje spolu s posunutým. Tento pokus však současná verze aplikace IMPRODEMO neobsahuje.

FrameRelay přijímá snímky pomocí zaregistrování se k *NewFrameAggregatedEvent* a publikuje pár snímků k počítání pomocí *ComputationPairAggregatedEvent*.

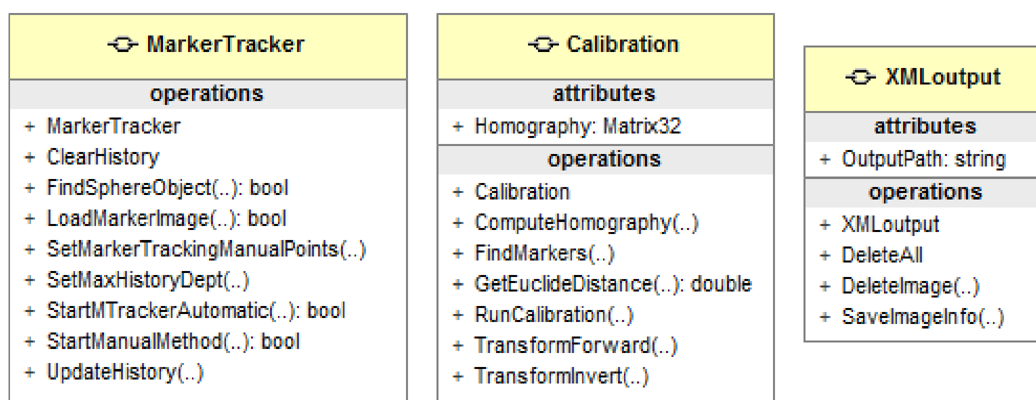
ComputationManager

ComputationManager je třída zajišťující logiku výpočtu a publikování výsledků. Obsahuje ovladač událostí reagující na změnu *Selection*, dále ovladač událostí přijímající pár snímků určených k výpočtu a nakonec používá službu *DisplacementComputer*, aby zavolal výpočet korelační matice nad předanými snímky a vybranou oblastí zájmu a výsledek publikoval do vyšších vrstev (*ViewModel* a *View*) pomocí *ComputationFinishedAggregatedEvent*.

Obsahuje property *ReportProgressDelegate*, která umožňuje přirazení metody typu *ReportProgressDelegateType* a následné informování o průběhu výpočtu korelační matice pomocí této metody.

DisplacementComputer

Tato třída patří mezi tzv. *services* nebo-li služby. To znamená, že třída poskytuje jen funkcionalitu, která nepotřebuje žádnou interakci s okolními komponentami a z pohledu uživatele této třídy se práce se service omezuje na tři kroky: předání vstupu, spuštění služby, odebrání výstupu. Třída *DisplacementComputer* slouží k výpočtu korelační matice nad předaným referenčním snímkem a zájmovou oblastí. Výsledek výpočtu jsou posunutí ve třech maticich. Matice *DxMatrix* poskytuje posunutí ve směru osy x, *DyMatrix* ve směru osy y a *DdMatrix* obsahuje euklidovskou vzdálenost posunutí. Na Obrázku 3.7 je vidět aplikace KORSYS a zobrazená matice posunutí ve formě barevné mapy. Vpravo je nástroj *ScaleMeter*, který ukazuje kvantitativní informaci o posunu a přiřazuje ji informaci barevné.



Obrázek 3.10: Diagram tříd pro vrstvu Model část 2.

MarkerTracker

Třída poskytuje metody ke zpracování obrazu a nalezení značek (markerů) v obraze. Umožnuje nahrát libovolný tvar pomocí *LoadMarkerImage*, zadat vlastni sledované body v obraze a potom sledovat tyto body pomocí nalezení okolí a pomocí metod *SetMarkerTrackingManualPoints* a *StartManualMethod*. Dále umožňuje marker načtený ze souboru automaticky vyhledávat v obraze pomocí *StartMTrackerAutomatic*. Vzhledem k tomu, že markery se mohou ztrácet díky např. proměnným světelným podmínkám, je zaveden koncept historie, který znamená, že *MarkerTracker* se pokouší nově nalezený marker přiřadit existujícímu v uživatelsky nastavené historii. Tato historie se nastavuje pomocí *SetMaxHistoryDepth* a pomocí *UpdateHistory* se dá aktualizovat.

Calibration

Třída poskytuje přepočet do fyzikálních jednotek. Před spuštěním samotné kalibrace pomocí *RunCalibration* je nutné nastavit matici homografie pomocí *ComputeHomography*. Ta zajišťuje správné koeficienty při přepočtu obrazu do fyzikálních jednotek. Nicméně úplně prvním krokem je spuštění metody *FindMarkers* na obraze, který by měl obsahovat příslušné kalibrační markery. Po úspěšném nastavení homografie lze volat metodu *TransformForward*, která předané souřadnice kovertuje do fyzikálních jednotek. Metoda *TransformInvert* zase takto zkonvertovanou hodnotu převede podle kalibračních parametrů zpět do obrazových jednotek.

3.4 Komunikace mezi vrstvami MVVM

V předchozích třech podkapitolách byly popsány tři základní vrstvy IMPRODEMO – *Model*, *View, ViewModel* a třídy, které se v nich vyskytují a které implementují požadovanou funkcionalitu. Není však zatím jasné, jakým způsobem bude komunikace mezi jednotlivými vrstvami probíhat. V komunikaci jsou totiž zakomponované nejen samotné vrstvy, ale také mechanismy WPF a MEF. Proto zde prezentuji příklad jednoduché komunikace, situaci, kdy je požadováno, aby se po stisku tlačítka „Options“ v třídě *ToolsPanel* zobrazilo toto dialogové okno. Je to netriviální úkol, neboť je zde nutná komunikace mezi jednotlivými *ViewModely*, kterou zajišťuje *ApplicationController.*

Zde jsou jednotlivé kroky komunikace:

1) V ToolsPanel.xaml se nachází deklarativně určený příkaz, který pomocí *databindigu* po stisku tlačítka přejde rovnou k jeho zpracování. Není zde tedy žádný ovladač událostí, jak je v GUI aplikacích typické. Toto je část *View.*

2) V *ToolsPanelViewModel.cs* se nachází vlastnost (property) třídy *ToolsPanelViewModel* nazývající se *ShowAdvancedOptionsDialogCommand*. Tato property je typu *ICommand*, implementuje tedy přímo *databinding*. Ve třídě *ApplicationController*, která patří do části *Controller* v architektuře MVVM tak, jak je popsaná na obrázku 3.3, se do tohoto příkazu přiřadí konkrétní implementace příkazu. Je to funkce nacházející se v třídě *ApplicationController* s názvem *ShowAdvancedDialog*.

3) V této funkci se využije mechanismus MEF. Ten umožňuje získat přístup k *ViewModel* části *AdvancedOptionsDialog* i bez toho, aby měl na ní přímou referenci. Engin MEF tak za chodu projde kontejner zaregistrovaných komponent a pokud požadovanou komponentu nalezne, tak ji instantizuje a předá zpět volané třídě. Tak se v třídě *ApplicationController* získá reference na *ViewModel AdvancedOptions* dialogu. Vyvolá se metoda *Show* tohoto *ViewModelu*. Protože ale *ViewModel* nemá žádnou referenci na *View* a nemůže komunikovat s *View* (pouze naopak, viz Obrázek 3.3), je zde použit další mechanismus, jak informovat prezentační vrstvu, aby zobrazila dialog. Tento mechanismus je implementace návrhového vzoru *Separated Interface*[11], poskytovaná frameworkem WAF. Umožňuje přístup k View prostřednictvím rozhraní, které *View* implementuje. Tak se vyvolá metoda *ShowModally* třídy *AdvancedOptionsView* a dialogové okno se otevře.

Jak je tedy vidět, prostředky a metodologie použité v IMPRODEMU mohou být dvousečné. Jejich složitost se projeví ve snaze realizovat jednoduchý příkaz jako otevření okno negativně. Nicméně v naprosté většině komplikovanějších úkonů přináší použité nástroje pořádek, přehlednost a jasně odlišitelnou architekturu a je potom snažší projekt v pokročilejších fázích vývoje a za účasti více programátorů udržovat a rozvíjet.

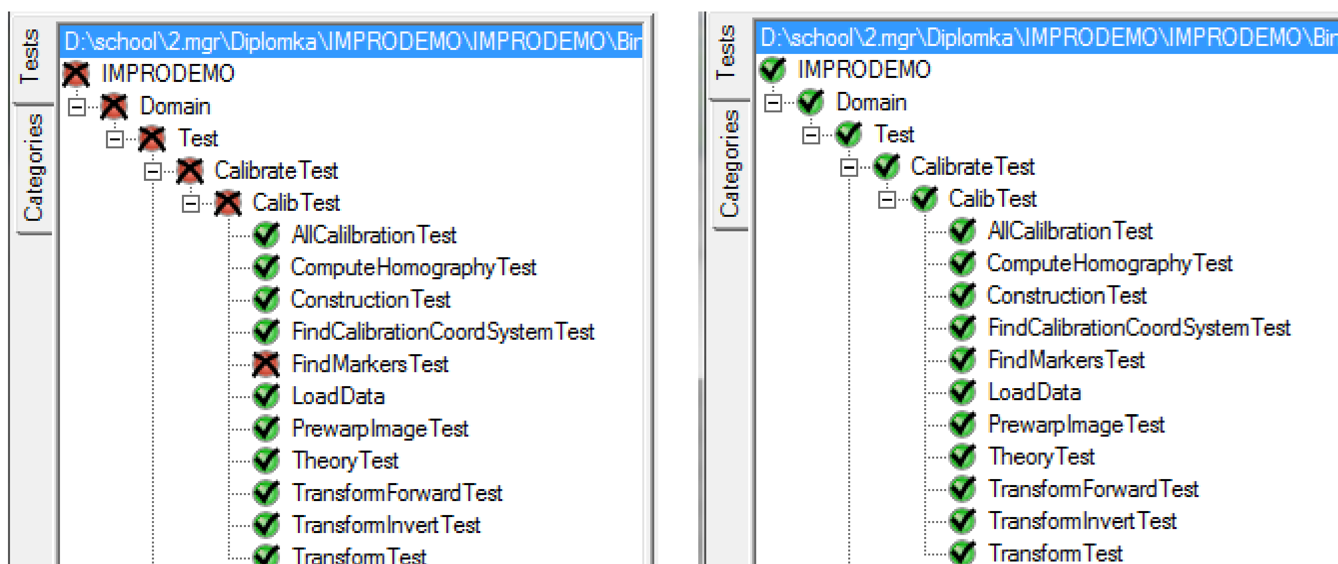
4 Funkční testy

4.1 Technika testování

Při implementaci IMPRODEMO je následována filozofie TDD (Test-Driven Development) tak, jak je popsaná v kapitole 2.4. Ta říká, že než začne proces implementace třídy podle návrhu, napíše se sada tzv. testů. V případě námi použitého nástroje na testování NUnit (podkapitola 2.4.1) sadu testů reprezentuje třída označená tagem *[TextFixture]*. Konkrétní nezávislé testy jsou reprezentovány metodami v takto označené třídě. Aby metoda byla nástrojem NUnit zaregistrována jako spustitelný test, musí být označena tagem *[Test]*. Každý test, po jeho spuštění, může buď selhat, nebo projít. Tuto klíčovou funkcionalitu testovacího nástroje poskytuje statická třída *Assert* obsažená ve jmeném prostoru nástroje NUnit, která nabízí velké množství způsobů, jak vyhodnotit platnost předaného výrazu. Zde si uvedeme několik příkladů signatur metod této třídy.

1. `public static void Contains<T>(T expected, ICollection<T> toBeSearched);`
2. `public static void DoesNotContain(string searchString, string toBeSearched);`
3. `public static void Equals(int expected, int actual);`
4. `public static void False(bool expression);`
5. `public static void StartsWith(string expected, string actual);`

První metoda umožní prohledat předanou kolekci dat a vyhodnotit nalezeni objektu typu T, druhá dělá logický doplněk předchozí akce, pouze s tím rozdílem, že vyhledává v řetězci. Třetí metoda porovnává dvě celé čísla, čtvrtá vyhodnocuje logický výraz a pátá určuje, zda řetězec *actual* začíná řetězcem *excpected*. Metody poskytují možnost manipulovat s více druhy dat a všechny poskytují možnost v posledním, dodatečném parametru specifikovat řetězec popisující účel daného ověření. Detailní seznam metod, které poskytuje třída Assert lze nalézt v [15].



Obrázek 4.1: Ukázka testování v NUnit.

V momentě, kdy je napsaný test, je možno ho spustit. Na Obrázku 4.1 vlevo lze vidět, jak jeden z testů na kalibraci neprojde. Ovlivní to tak celou testovou jednotku IMPRODEMO. Po správné úpravě kalibračního algoritmu, je již vše v pořádku a test projde (Obrázek 4.1 vpravo).

4.2 Testování ToolsPanelViewModel

ToolsPanel reprezentuje panel s veškerými nástroji, které IMPRODEMO poskytuje. V částí *View* existuje deklarativní popis uživatelského rozhraní a *code-behind* sekce. Vše je v rovině prezentační vrstvy. V části *ViewModel* existuje třída reprezentující stav UI a delegující příkazy, které z UI přicházejí. Zároveň slouží *ToolsPanelViewModel* jako *Controller*, tzn. řídí logiku aplikace. V této kapitole popíši několik testů, pokrývajících vzorové akce vyskytující se v IMPRODEMO a týkající se interakcí v *ToolsPanel*.

4.2.1 Testování bez nutnosti okna

První test demonstruje testování zobrazení a skrytí *ToolsPanel*. K tomuto účelu je použito tzv. *mock* objektů. Jsou to objekty implementující stejné rozhraní jako okna aplikace v prezentační vrstvě s tím rozdílem, že obsahem metod této třídy není žádný funkční kód. Třída je pojatá jako naprosté mininum k tomu, aby zastoupila roli plnohodnotného okna z pohledu třídy s ní interagující. Např.: pokud má *MainView* metodu *AddPolygon*, která přidá do hlavního okna objekt třídy *System.Windows.Shapes.Polygon*, potom třída *MainViewMock*, implementující stejné rozhraní (*ImainView*) obsahuje metodu *AddPolygon*, která zvýší index třídní proměnné indikující počet polygonů o 1.

Konkrétní test zobrazení tedy vypadá takto:

```
[Test]
public void ShowWindowTest()
{
    ToolsPanelViewMock toolsView =
       (ToolsPanelViewMock)_compContainer.GetExportedValue<IToolsPanelView>();

    Assert.False(toolsView.IsVisible, "ToolsPanel should be initially
closed.");
    _toolsPanelViewModel.Show();
    Assert.True(toolsView.IsVisible, "ToolsPanel should now be shown.");
}
```

Prvním příkazem se z kontejneru MEF (kapitola 2.3) získá třída exportovaná jako *IToolsPanelView*. Pokud tato ještě neexistuje, MEF ji vytvoří. V našem případě je to třída *ToolsPanelMock*. Nejdříve se pomocí třídy *Assert* zjistí, zda je okno ToolsPanelu vidět hned po vytvoření. Toto by nemělo nastat, proto je použita metoda *False*. Po aktivování okna příkazem *Show* se ověří komplementární metodou zda okno naopak vidět je. V obou případech, pokud metoda *Assert* vyhodnotí výraz nesprávným způsobem, test selže.

4.2.2 Testování interakce mezi ViewModely

Předchozí příklad testu ukazoval jednoduchou interakci v rámci jednoho okna. Okno se po otevření buď otevře, nebo neotevře. Mnohem častěji vyskytující jsou však případy, kdy probíhá nějaká forma

složitější komunikace. V této podkapitole demonstrujeme situaci, ve které se vyvolá akce „*AutoSelect*“ prostřednictvím kliknutí na příslušné tlačítko v UI *ToolsPanelu*. Tato akce obsahuje velké množství akcí, v kontextu ToolsPanelViewModel však testujeme jen některé z nich. Takto vypadá test:

```
[Test]
public void AutoSelectTest()
{
    MainViewMock mainView =
        (MainViewMock)_compContainer.GetExportedValue<IMainView>();

    List<Polygon> addedPolygons = mainView.GetAddedPolygons();
    List<Polygon> substractedPolygons = mainView.GetSubstractedPolygons();

    Assert.AreEqual(0, addedPolygons.Count);
    Assert.AreEqual(0, substractedPolygons.Count);

    GrayImage8 image = new GrayImage8();
    image.LoadImg("../../TestImages/0000_0.tif");
    mainView.SetActualImage(image);

    _toolsPanelViewModel.AutoSelectCommand.Execute(null);

    addedPolygons = mainView.GetAddedPolygons();
    substractedPolygons = mainView.GetSubstractedPolygons();

    Assert.AreNotEqual(0, addedPolygons.Count);
    Assert.AreNotEqual(0, substractedPolygons.Count);
}
```

Nejdříve si získáme mock verzi okna *MainView*. Do té se zobrazují automaticky nalezené polygony, a také se z ní získává obraz, nad kterým se automatická selekce zjišťuje. Dále se přesvědčíme, že okno zatím žádné polygony neobsahuje. V dalším kroku se načte obrázek, nad kterým se zaručeně vyskytují oba druhy polygonů – ty co zájmovou oblast rozšiřují, i ty, co jí redukují. Nastavíme tento obraz jako aktuálně zobrazený a nyní vykonáme *AutoSelectCommand*, který simuluje kliknutí na tlačítko *AutoSelect* v UI. Zda se klik správně propaguje do *ViewModelu* není předmětem testování této vrstvy, stejně tak, zda autoselekce proběhla správným způsobem, tedy, zda obsahuje správný počet a druh polygonů (to se testuje až ve vrstvě *Model*). Zda byl příslušný příkaz, reprezentovaný vyvoláním metody *Execute* nad třídou *AutoSelectCommand,* vykonán, zjistíme tak, že si ověříme, že přibyly polygony v hlavním okně. To zajišťují poslední dva řádky používající třídu *Assert*.

4.3 Testování na úrovní vrstvy Model

Vrstva *Model*, někdy také nazývaná doménová vrstva, nebo-li vrstva najišťujíc tzv. *bussiness-logic* je na testování nejvíce přímočará. Drtivá většina jejich tříd jsou služby, tj. z pohledu ostatních tříd je to samostatný celek, který přebere svoje vstupy a po nějaké době je přetransformuje na výstupy. V ideálním případě název metody či třídy je dostatečně deskriptivní na to, aby bylo jasné, co ona služba dělá. V aplikaci IMPRODEMO se vyskytuje značné množství tříd zajišťujících přístup a tranformaci surových dat. Zde si popíšeme některé druhy testů implementovaných v aplikaci a ukážeme si, jak se takový test vytváří a jak probíhá vývoj podle TDD. To vše na třídě *Selection*, která reprezentuje zájmovou oblast pro výpočet korelace.

Vývoj této třídy podle TDD bude probíhat podle algoritmu na obrázku 2.2. Vytvářím třídu, která má reprezentovat obecnou oblast zájmu pro výpočet korelace. K těmto účelům zavedu dvě třídní pole, *AddingPolygons* (dále bude používán výraz *add-polygony*), které budou mít za funkci definovat zájmovou oblast nad dodaným snímkem a *SubstractingPolygons* (dále jen *sub-polygony*), které budou mít za funkci zájmovou oblast odebrat. Polygony budou reprezentovány třídou *Polygon*, která je obsažena ve frameworku WPF. Chci aby funkce *sub-polygonů* byla taková, že pokud vytvořím oblast *sub-polygonů* a umístím je na stejnou pozici, kde se nachází stejně velká oblast *add-polygonů*, výsledné zájmová oblast bude nulová. Test pro takovou funkcionalitu vypadá takto:

```
[STAThread]
[Test]
public void BasicSubstractionTest()
{
    var selection = new Selection();

    Polygon addPolygon = new Polygon();
    addPolygon.Points.Add(new Point(0, 0));
    addPolygon.Points.Add(new Point(10, 0));
    addPolygon.Points.Add(new Point(10, 10));
    addPolygon.Points.Add(new Point(0, 10));

    Polygon subPolygon = new Polygon();
    subPolygon.Points.Add(new Point(0, 0));
    subPolygon.Points.Add(new Point(10, 0));
    subPolygon.Points.Add(new Point(10, 10));
    subPolygon.Points.Add(new Point(0, 10));

    selection.AddingPolygons.Add(addPolygon);
    selection.SubstractingPolygons.Add(subPolygon);

    PointCollection gridPoints = selection.GetGridPoints(0, 0, 5, 5, 1.0, 1.0);
    Assert.True(gridPoints.Count == 0);
}
```

Atribut *[STAThread]* je přítomen z toho důvodu, že testovací nástroj Nunit běží v MTAThread (MultiThreadAppartment). Oproti tomu elementy, které jsou součásti WPF UI a které dědí z *UIElement* vyžadují, aby aplikace běžela v STA (*Single Thread Appartment*). Detaily o STA a MTA v [16].

Test začne vytvořením nové selekce a nadefinováním jednoho *add-polygonu* s určitými vrcholy. Poté je vytvořen naprosto identický *sub-polygon*. Tyto polygony jsou vloženy do selekce, do kontejnerů, které obsahují příslušné druhy polygonů. Poté je vyžádána kolekce bodů, nad kterými se bude provádět výpočet korelace. Jsou stanoveny parametry, tedy nulový offset v obou osách, krok 5 pixelů a měřítko 1:1. Na konci je ověřeno, že žádné body v takovéto selekci nejsou.

Takto napsaný test nepůjde zkompilovat z toho důvodu, že třída *Selection* neposkytuje používané metody. Proto, podle vývojového diagramu na Obr. 2.2, nyní proběhne úprava kódu takovým způsobem, že se doplní minimální množství kódu, aby test uspěl. To znamená, vytvoření třídních proměnných *AddingPolygons* a *SubstractingPolygons* a metody *GetGridPoints*. Ta může v tuto chvíli vypadat takto:

```
public PointCollection GetGridPoints(int Offsetx, int Offsety, int Stepx,
    int Stepy, double scaleX, double scaleY)
{
    return new PointCollection();
```

```
}
```

Tím, že metoda vrátí prázdnou kolekci bodů splní podmínku, že její velikost je nulová. Z pohledu smyslu selekce je toto řešení samozřejmě nesprávné, ale pointa TDD je vyvíjet postupně kód požadované třídy a přidávat vždy jen to nejnutnější na splnění testu.

V dalším kroku se přidá další test definující chování selekce. Bude říkat, že při vytvoření jednoho *add-polygonu* s rozměry 10x10 pixelů a při kroku 1 v obou osách očekávám 100 bodů sloužících k vypočtení selekce. Tento test vypadá stejně jako test nazvaný *BasicSubstractionTest* s tím rozdílem, že se přidává jen jeden *add-polygon* a v metodě *GetGridPoints* předávám jako krok *Stepx* a *Stepy* hodnoty 1. Také závěrečné ověření pomocí metody *Assert.True* je vůči hodnotě 100 místo 0.

Nyní nezbývá již nic jiného, než do metody *GetGridPoints* napsat funkční kód. K tomu se využije třída *GridComputer*, která je přenesená z KORSYSu a implementovaná v jazyce C++/CLI. Ta umožňuje získat přímo požadované body pomocí metody *Compute*. Tato metoda naplní instanci *GridComputer* těmito body a tu potom stačí jen vrátit. Za předpokladu, že *GridComputer* je správně naimplementován (detaily v kapitole 5), splní podmínky jakéhokoliv testu na selekce co se týče jejich rozměrů, pozic, nebo překrytí.

Dalším z testů selekce bude test na přítomnost polygonů. Nyní je cílem, aby nebylo možné zadat výpočet na prázdné selekci. K tomuto účelu se vytvoří výjimka *SelectionException* a test, který jí bude zachytávat. Vypadá následovně:

```
[Test]
public void NoPolygonsTest()
{
    var selection = new Selection(new EventAggregator());
    try
    {
        selection.GetGridPoints(0, 0, 0, 0, 1.0, 1.0);
    }
    catch(SelectionException)
    {
        Assert.True(true);
        return;
    }

    Assert.True(false);
}
```

Získávání výpočtových bodů pomocí metody *GetGridPoints* je nyní vloženo do klauzule try-catch. Vzhledem k tomu, že je očekáváno vyvolání výjimky, je obsahem catch bloku metoda *Assert.True* s argumentem *true*, která se vyvolá jen v případě vyvolání výjimky. V opačném případě se program dostane k jistému neúspěchu testu. Abych tento test prošel, je nutno opět upravit metodu *GetGridPoints*. Ta nyní vyvolá výjimku v situaci, kdy zjistí, že nemá k dispozici žádné polygony.

Další druh testů se provádí u kamerového objektu. Testujeme, zda správným způsobem funguje *wrapper* napsaný nad kamerovým objektem přeneseným z KORSYSu. Zde je ale situace jiná v tom smyslu, že pracujeme s externím zařízením. Musíme proto zavést pomocné mechanismy. Jedním z nich je třída *CallbackHelper*, která slouží pro uchovávání počtu přijatých snímků. Pro příjem snímků z kamerového objektu třídy *CameraWrapper* je totiž potřeba zaregistrovat tzv. event handler nebo-li ovladač události. Ten třída *CallbackHelper* obsahuje a kromě toho také uchovává počet zpracovaných snímků. Třída vypadá takto:

```
public class CallbackHelper
{
```

```
        public int FramesAcquired { get; set; }

        public void Reset() { FramesAcquired = 0; }

        public void OnNewFrame(object o, CameraEventArgs camArgs)
        {
            Assert.True(camArgs.Frame != null);

            if (camArgs.Frame != null)
                FramesAcquired++;
        }
    }
```

Metoda Reset se volá v rámci metody označené atributem *[SetUp]*. Před každým testem je totiž potřeba vrátit ovladač událostí do původního stavu. Třída testů obsahuje různé testy inicializace kamerového objektu, já zde uvedu jen ty, které stojí za zmínku a nějakým způsobem obsahují neobvyklý způsob zpracování testu.

Test funkčnosti nastavení rychlosti dodávání snímků (FPS neboli *Frames per second*) funguje na principu nastavení FPS a poté odebrání určitého počtu snímků a změření doby tohoto odběru. Je to vidět na následujícím kódu:

```
    [Test]
    public void FrameRateTest()
    {
        int camCnt = CameraWrapper.CameraCount;
        Assert.NotNull(camCnt);

        var cam = new CameraWrapper(0);

        cam.OnNewFrame += _counter.OnNewFrame;
        cam.FrameRate = 20;

        cam.StartDma();
        while (_counter.FramesAcquired < 1)
        { }
        cam.StopDma();

        _counter.Reset();
        cam.StartDma();
        DateTime startTime = DateTime.Now;
        while (_counter.FramesAcquired < 10)
        { }
        DateTime endTime = DateTime.Now;
        cam.StopDma();
        System.TimeSpan timeSpan = endTime.Subtract(startTime);

        Assert.Less(Math.Abs(timeSpan.TotalMilliseconds - 500), 100);
    }
```

Na začátku testu se ověří, že existuje alespoň jedna kamera. V případě že tomu tak není, test okamžitě selže. Dále se registruje ovladač události a nastaví FPS, přičemž *_counter* je lokální instance třídy *CallbackHelper.* Další blok kódu slouží k přijmutí prvního snímku po vytvoření kamery. Ten totiž přichází s určitou odezvou, která je několikanásobně větší než prodleva mezi snímky tak, jak by měla být podle určené rychlost FPS a tudíž narušuje měření doby. Metody *StartDma* a *StopDma* slouží k zapnutí, respektive vypnutí snímání kamery. Po přijmutí prvního zkušebního snímku se vynuluje čítač a začne se odměřovat čas, který trvá kameře k přijmutí deseti

snímku. Vzhledem k tomu, že je nastavená rychlost na dvacet snímků za vteřinu, předpokládá se, že deset snímků bude přijato „asi“ za půl vteřiny. Výraz „asi“ je vyjádřen v podobě tolerance ve výši sto milisekund. Pokud je doba přijetí snímků menší než tato tolerance, test uspěje a předpokládá se, že nastavení rychlost FPS funguje v pořádku.

4.4 Shrnutí testování

Princip Test-Driven Development nabízí programátorovi nástroj nejen na testování, ale hlavně na kvalitní implementaci softwaru. Tím, že ho nutí napsat nejdřív test na neexistující funkcionalitu se vytváří rozhraní třídy a dodává se příklad jejího použití. Vývojář je nucen se zamyslet nad tím, co chce vytvořit, do velkých detailů ještě předtím, než to začne vytvářet. Taktéž narazí na množství chytáků které mohou znepříjemnit samotný vývoj. Fakt, že je aplikace testovatelná a po přidání nové funkcionality lze lehce pomocí testovacího nástroje zjistit, zda jsou kritické funkce ve správné činnosti, je jen bonusem k již tak kvalitnímu prostředku agilního programování. Čas, který se stráví psaním testů se bohatě vrátí při úpravě funkcionality, která byla napsána před několika měsíci a není jasné, jak byl každý jednotlivý řádek zamýšlen. Zásahem do systému a sledováním vyhodnocení testů lze silně takovému porozumnění pomoci. Neposlední výhodou je také snadnější obnova do bodu funkčnosti aplikace.

5 Interoperabilní vrstva pro práci s obrazem a externími zařízeními

Tato kapitola prezentuje použitý nástroj pro interoperabilní vrstvu, ukazuje úskalí přechodu mezi nativním kódem a kódem spravovaným a prezentuje způsoby řešení těchto problémů. Detailně uvádí techniky spjaté s použitým nástrojem a jejich použití.

5.1 Použitý jazyk – C++/CLI

V aplikaci IMPRODEMO se používají výpočetní metody pro dvě hlavní funkce zpracování obrazu: je to nalezení korelace mezi dvěma obrazy; tato metoda slouží k určení jemných až středních deformací v materiálu, a dále je to metoda nalezení markerů či objektů v obraze. Pro obě tyto metody je potřeba obraz z kamery, proto se využívají nástroje pro práci s kamerovým vstupem. Tyto výpočetní a kamerové metody jsou implementované v KORSYSu a tak bylo rozhodnuto, že v aplikaci IMPRODEMO budou taktéž využívány. Vzhledem k tomu, že jsou napsané v jazyce C nebo C++ a aplikace IMPRODEMO je aplikace psaná v .NET frameworku WPF za použití programovacího jazyka C#, bylo potřeba zvolit způsob, jak napsanou funkcionalitu přenést do prostředí jazyka C#. K tomuto účelu byl vybrán jazyk C++/CLI, který poskytuje možnost využití veškeré funkcionality napsané v C++, což je statický jazyk s neomezeným přístupem ke stroji nad kterém běží, ale omezenými možnostmi přístupu k typům aktivním v běžícím programu a žádnou možností přístupu k infrastruktuře programu. Na rozdíl od C++ však C++/CLI obsahuje onu část nazvanou CLI nebo-li *Common Language Infrastructure*, která reprezentuje dynamický model komponent. Je vrstvou nacházející se mezi programem a operačním systémem a poskytuje tak přístup a konstrukci typů a infrastruktury programu. C++/CLI tak poskytuje vazbu mezi statickým modelem použitým v C++ a dynamickým používaným v .NET jazycích. C++/CLI obsahuje mnoho rozšířených funkcionalit a ty budou ukázány na konstrukci konkrétního wrapperu pro C#.

5.2 Wrapper pro třídu Matrix32

Používání a počítání s maticemi je kritická oblast potřebná v korelačním systému IMPRODEMO. Pro tento účel byla z KORSYSu přenesena třída *at_matrix* a veškerá její funkcionalita byla zabalena do příhodného C++/CLI *wrapperu*. V této kapitole si popíšeme způsob, jakým se takový *wrapper* píše a rozebereme si jednotlivá úskalí práce na pomezí mezi „*managed*“ a „*unmanaged*“ [17] světem.

Jako první je potřeba definovat právě tyto dva světy, tedy *managed* a *unmanaged*. Pokud je program zkompilován v *managed* módu, zkompilovaná jednotka, nazvaná *assembly* je při spuštění převzata strojem CLR (*Common Language Runtime*). Ta kód převádí do strojového ve chvíli, kdy je

použit. Tomuto procesu se říka JIT (*just in time compiling*). Fakt, že *managed* kód je spravován pomocí CLR má několik zásadních výhod. Poskytuje spoustu automatických služeb, jako například GC (*garbage collector*), což je jednotka starající se uvolňování nepotřebné paměti, poskytuje bezpečností služby, služby pro práci s vlákny a podobně. Na druhou stranu *unmanaged* program je takový, který je zkompilován přímo do strojového kódu a o všechny tyto služby si musí sám explicitně říkat operačnímu systému, například prostřednictvím WinAPI nebo volání COM objektů. Výhoda takového kódu je absolutní kontrola nad vytvořeným strojovým kódem, neboť některé optimalizace prováděné CLR nemusí být zrovna žádoucí.

Managed, resp. *unmanaged* kód se indentifikuje pomocí direktivy *#pragma managed* resp. *#pragma unmanaged*. Může tak sloužit například k vkládání hlavičkových souborů napsaných v C++, nebo libovolných knihoven z WPF.

5.2.1 Vytvoření a zánik Matrix32

V této podkapitole bude popsáno, jakým způsobem se v C++/CLI tvoří metody obhospodařující důležité akce týkající se života objektu. Třída Matrix32 obsahuje tři druhy konstruktorů: bezparametrický, s dvěma parametry určující jeho rozměry a s jedním parametrem, kterým je jiný objekt typu Matrix32. První dva inicializují svůj obsah na 0, poslední zkopíruje obsah druhé matice do sebe a podle toho i nastaví svoje rozměry. Takto vypadá kód pro druhou verzi konstruktoru:

```
Matrix32(unsigned int cols, unsigned int rows)
{
    m_matrix = new at_matrix();
    at_alloc_matrix(m_matrix, cols, rows);
    memset(m_matrix->data, 0, sizeof(at_matrix_value) *cols*rows);
}
```

Nejdříve se vytvoří instance matice pomocí alokování struktury *at_matrix* z hlavičkého souboru napsaného v jazyce C (a vzhledem k tomu, že C++ podporuje drtivou většinu funkcionality z C, lze takový soubor použít). Použije se zde klasický operátor *new*, což značí, že takto alokovaná paměť nebude spravována GC a je tedy zodpovědnost programu, aby si ji správně uvolnil. Lze také použít operátor *gcnew*, který naopak říká, že takto alokovaný objekt se má v GC zaregistrovat, být jím spravován a v případě, že na tento objekt již nikde není reference, se má postarat o jeho zánik.

Dále se použije alokační funkce, opět z C knihovny, která alokuje paměť daných rozměrů. Jako poslední příkaz inicializujeme všechny buňky matice na 0.

Destruktor, neboli metoda volající se automaticky při zániku objektu pomocí operátoru *delete* je na první pohled velmi prostý. Vypadá takto:

```
~Matrix32()
{
    this->!Matrix32();
}
```

Volá se zde jen jedna metoda a to takzvaný finalizér. Ten působí podobně jako další destruktor. Rozdíl mezi klasickým destruktorem ve formátu *~NázevTřídy* a finalizérem ve formátu *!NázevTřídy* je ten, že zatímco destruktor je volán deterministicky přímo programátorem na určeném místě pomocí operátoru delete, finalizér je volán pomocí GC nedeterministicky jako součást procesu uvolňování paměti. Samotný finalizér již vypadá takto:

```
!Matrix32()
{
    at_dealloc_matrix(m_matrix);
    delete m_matrix;
}
```

Zavolá se funkce na uvolnění veškeré paměti matice a poté se pomocí operátoru *delete* objekt matice, nad kterým se wrapper staví, uvolní.

5.2.2 Další komponenty Matrix32

Za účelem zlepšení možnosti odstranění chyby byla zavedena výjimka informující ostatní komponenty, popř. uživatele o chybě při počínání s maticí. Vypadá takto:

```
[SerializableAttribute]
public ref class CameraException : public Exception
{
public:
    CameraException(String^ msg) : Exception(msg) {}
};
```

Její definice využívá standartní .NET třídu *Exception*, která poskytuje možnost získat text vyjímky, většinou zde bývá důvod jejího vyvolání, resp. selhání. Za povšimnutí zde hlavně stojí atribut *[SerializableAttribute]*, který podle definice standartu C++/CLI musí být přítomen u uživatelsky vytvořených výjímek. Je to za účelem tzv. *remotingu*, tedy ve chvíli, kdy by se výjímka posílala přes síti, je nutné ji serializovat a také deserializovat. Více o serializaci v [30].

Matrix32 dále obsahuje speciální třídní proměnné, v C# nazývané properties. Kromě klasické třídní proměnné mají několik dalších funkcionalit implementujících OOP paradigma nazvané zapouzdření. Pomocí klíčových slov *get* a *set* vytváří jednotky kódu, ve kterých lze před předáním nebo nastavením předávané proměnné provést např. konverzní operace. Property v C++/CLI je zapsaná takovýmto způsobem:

```
property int Width
{
    int get() { return m_matrix->width; }
}
```

Klíčové slovo *property* deklaruje blok ve kterém je kromě typu proměnné a jejího nazvu také ve složených závorkách očekáván buď zpřístupňující blok nazvaný *get* nebo nastavující blok s názvem *set*, nebo oba, nebo žádný. Když je nějaký z těchto bloků nepřítomen, znamená to, že daná operace není podporována.

Poslední metodou ilustrující některé zajímavé a důležité vlastnosti je metoda na získávání extrémních hodnot z matice. Začneme přímo prezentováním kódu:

```
void GetMinMax(GrayImage8^ mask, [Out] double% min,[Out] double% max)
{
    at_matrix_value _min, _max;
    at_matrix_min_max(m_matrix, mask, &_min, &_max);
    min = (double)_min;
    max = (double)_max;
}
```

Nejzajímavější část je na prvním řádku, v signatuře metody na získávání minima a maxima z matice. První parametr obsahuje znak „^” mezi určením typu a názvu proměnné, který se bude používat ve funkci. Ten říká, že daná třída bude předána referencí, tedy že při volání metody se nevytvoří nový objekt, ale předá se ten existující. To je důležité z důvodu, že pracujeme v *managed* světě a tímto kompilátoru řekneme, že přejímaný objekt je spravován GC. Další parametry jsou obklopeny tagem *[Out]* a znakem „%”. Tag říká, že předaná proměnná může být při předávání prázdná, protože se předpokládá, že bude v metodě naplněna. Znak informuje o tom, že proměnná má být opět předána referencí. Důvod, proč je zde použit znak „%” místo „^“ je ten, že se jedná o hodnotový typ. Zbytek funkce již zavolá nativní funkci a výsledek předá do připravených proměnných s dvojitou přeností polovoucí desetinné čárky.

5.3 Wrapper pro práci s kamerou

V předchozím *wrapperu* pro třídu *Matrix32* byla ilustrována většina potřebných vlastností jazyka C++/CLI pro to, abychom mohli napsat obalový kód pro většinu tříd přenášených z KORSYSu. Třída pro práci s kamerou však vyžaduje vyzkoumání ještě několika dalších nových postupů. Jako první je způsob, jak poskytnout zachycené snímky. Bylo by možné je všechny, jeden po druhém, ukládat ve třídním poli, nebo poskytovat vždy ten poslední, potom by ale byl příliš komplikovaný proces zjišťování nových snímků. Každý, kdo by měl o nový snímek zájem, by se musel opakovaně třídy dotazovat, zda nějaký nepřišel. Mnohem pohodlnější je zavedení systému event handleru, nebo-li ovladače událostí. Ten zajišťuje notifikování všech zaregistrovaných metod a funguje následujícím způsobem: Předpokládejme, že existuje třída, která chce všechny zájemce notifikovat o určité události. Potom si tato třída deklaruje zpracovač událostí takto:

```
public event SampleEventHandler EventHandler;
```

Tento řádek říká, že ve třídě, ve které je umístěn, existuje člen nazvaný *EventHandler*, což je událost (definováno klíčovým slovem *event*), která je typu *SampleEventHandler*. Tento typ může být nějaký z přednastavených a dostupných v používaném frameworku nebo to může být vlastní. V tom případě může být definován následujícím způsobem:

```
public delegate void SampleEventHandler(object sender, EventArgs e);
```

Ve chvíli, kdy definujeme typ ovladače událostí tímto způsobem, je tím explicitně řečeno, že každá metoda, která má zájem být notifikována o události prostřednictvím člena *EventHandler* musí mít tuto signaturu. Se členem *EventHandler* nyní můžeme vykonávat pouze dvě operace: přichycení metody zpracovávající událost, nebo naopak její odstranění. To se vykoná pomocí operátorů „+=“, resp „-=“ následujícím způsobem:

```
EventHandler += EventHandlerNumberOne;
EventHandler -= EventHandlerNumberOne;
```

za předpokladu, že metoda *EventHandlerNumberOne* je definována následujícím způsobem:

```
public void EventHandlerNumberOne(object sender, EventArgs e)
{
    // tělo metody
}
```

Takto funguje systém ovladače událostí v jazyce C# a prostředí .NET. Cílem obalové třídy pro kamerový systém je poskytnout ovladač události notifikující zájemce o nově vytvořeném snímku. Naráží se zde na problém přechodu z *managed* do *unmanaged* světa. Na straně nativního kódu, pocházejícího z KORSYSu, je nutné předat ovladač událostí jako ukazatel na funkci, která má jako parametr strukturu s informacemi o kameře. Toto se děje při započetí snímání pomocí kamery při použití této funkce:

```
at_cam_start_dma(at_cam_id* cam_id, at_cam_cb cb, void* param)
```

přičemž *at_cam_cb* je definován takto:

```
typedef void (*at_cam_cb)(struct _at_cam_id* cam_id);
```

Je to tedy ukazatel na funkci, která nic nevrací a za parametr má ukazatel na strukturu obsahující informace o kameře. Tímto způsobem se tedy notifikuje uživatelský kód, že byl pořízen nový snímek. Jak ale propojit tyto dva světy? Začneme od *managed* kódu a budeme postupně sestupovat k *unmanaged* nativnímu kódu. Událost, k jejímž odběru se můžou třídy z pohledu *managed* kódu zaregistrovat je následující:

```
event EventHandler<CameraEventArgs^>^ OnNewFrame;
```

Dva znaky stříšek informují překladač, že se jedná o reference. Tento ovladač událostí a tím pádem všechny metody, které se k němu zaregistrují, volá interně funkci, která vypadá takto:

```
void RaiseOnNewFrameEvent(at_cam_id* cam_id)
{
    Frame^ frame = gcnew Frame();
    int rc = at_cam_copy_image8(cam_id, frame->GetImage());
    AT_THROW_WHEN_FAILED( rc, CameraException, "Could not copy image from camera: ");
    OnNewFrame(this, gcnew CameraEventArgs(frame));
}
```

Tato funkce přijímá argument obsahující informace o kameře. Z tohoto argumentu pomocí funkce *at_cam_copy_image8* se vykopíruje snímek a předá se do třídy *Frame*. Další řádek je makro, které vyvolá výjimku v případě jakéhokoliv problému při kopírování obrázku z kamery a v poslední řadě se vyvolá již výše uvedený ovladač událostí. Zbývá doplnit, jakým způsobem se vyvolá výše uvedená funkce. K tomu je ovšem potřeba vysvětlit si několik dalších aspektů. Hlavním z nich je *unmanaged* třída *CameraCallbackHelper*. Ta poskytuje onen předěl mezi nativním a *managed* světem. Jejím jádrem je *managed* člen *m_managedDelegate*, který je definován takto:

```
gcroot<CameraCallbackDelegate^> m_managedDelegate;
```

Tento člen je typu *CameraCallbackDelegate*, který si vysvětlíme záhy. Vzhledem k tomu, že *CameraCallbackHelper* je *unmanaged* třída, nejde v ní člena typu *CameraCallbackDelegate*

uchovávat jen tak. Je nutno ho zabalit pomocí šablonové třídy *gcroot<T>*, která poskytuje jednoduchý přístup ke třídě *GCHandle*, která poskytuje způsob, jak přistupovat ke *managed* objektu z *unmanaged* paměti. Dále je zde další důležitá komponenta a tou je *callback* funkce v nativním kódu, která se z managed třídy typu *CameraCallbackDelegate* získá následovně:

```
m_nativeCallback = (at_cam_cb)
    Marshal::GetFunctionPointerForDelegate(call_delegate).ToPointer();
```

Zde se využije třída *Marshal* z .NET jmenného prostoru *System.Runtime.InteropServices.Marshal*. Tato funkce poskytuje možnosti manipulace s *unmanaged* pamětí a možnosti konvertování z *managed* do *unmanaged* kódu. Funkce *GetFunctionPointerForDelegate* vygeneruje nativní kód, který vnitřně volá předaný *managed handler*. Tento kód je dále přetypován na typ *at_cam_cb*. Zde se vracíme zpět k metodě *at_cam_start_dma*, která začne přenos snímků z kamery. Ta přijímá jako parametr odkaz na funkci typu právě *at_cam_cb*. A to je právě onen člen *m_nativeCallback*, který se dá získat z třídy typu *CameraCallbackHelper*. Poslední důležitý aspekt procesu tohoto převodu je definice typu *CameraCallbackDelegate*. Vypadá následovně:

```
[UnmanagedFunctionPointer(CallingConvention::Cdecl)]
delegate void CameraCallbackDelegate(at_cam_id* cam_id);
```

Za povšimnutí zde především stojí první řádek informující překladač, že následující řádek bude *managed* delegát (což je C# verze ukazatele na funkci) reprezentující ukazatel na *unmanaged* funkci s volací konvencí *Cdecl*, která říká, že volaný uklízí zásobník. To je potřeba specifikovat, protože *managed* funkce mají jako výchozí volací konvenci StdCall, která říká, že volaný uklízí zásobník [19].

Jak je tedy vidět, přenést událost z *unmanaged* do *managed* světa je poměrně netriviální úkol. Jazyk C++/CLI však nabízí dostatek prostředků k práci s dvěma druhy pamětí, konverzní funkce a jiné nástroje, které tento úkol do značné míry zjednodušují.

6 Závěr

Tato kapitola obsahuje shrnutí výsledků práce v návrhu a použitých softwarových metodologiích a technologiích. Shrnuje jejich přínos a diskutuje možnosti dalšího rozšíření a použití dalších nástrojů.

6.1 Dosažené výsledky

Byla navržena a implementována aplikace IMPRODEMO za použítí návrhového vzoru MVVM. Návrh aplikace vyplýval z logického členění do tří vrstev: Model, View a ViewModel, přičemž na úroveň ViewModel byla přidaná část *Controller* obsahující způsob řízení logického toku aplikace. Vrstvy komunikují podle nástrojů dodaných v použitém *frameworku* WAF (rozhraní, které implementují tříd z vrstvy View) a pomocí nástroje *EventAggregator*.

Implementace probíhala v trendu použité agilní metody vývoje Test Driven Development. Pro většinu tříd byly napsány testy, které určovaly jejich funkcionalitu a poté byla iterativně doplňována funkcionalita tříd a rozšiřovány testy. Podařilo se implementovat testy tříd z vrstvy ViewModel za použití Mock objektů tříd z vrstvy View.

Dále byla implementována interoperabilní vrstva pro práci s obrazem a externími zařízeními za účelem využití rychlé implementace v C++. Zde se ukázalo několik zádrhelů, jako například způsob propagace události z nativního kódu do .NET prostředí, ty byly však vyřešeny a funkčně implementovány.

6.2 Přínos práce

V aplikaci IMPRODEMO se ukázala možnost skloubení rychlého výpočetního jádra napsaného v jazyce C++ s grafickými možnostmi nejnovějšího způsobu prezentace dat na Windows platformě, a to frameworku WPF. V práci je popsán způsob použití moderních nástrojů pro vývoj komplexních GUI aplikací jako jsou návrhové vzory a agilní metody vývoje.

6.3 Možností využití a dalšího rozvoje

Poznatky získané v této práci lze využít pro vývoj aplikací, které jsou vyvíjeny s myšlenkou neukončeného vývoje, tedy že budou neustále dodávány další funkce a aplikace bude průběžně vylepšována a rozšiřována. Taktéž se výborně hodí pro aplikace, kde je požadován moderní a čistý vzhled s možnostmi vytváření vlastních konkrétních grafických komponent na vizualizaci dat.

Jako rozšíření či námět na další práci v budoucnu je například způsob testování i části View, zasazení celého vývoje do širšího kontextu agilní metody SCRUM či použití extrémního programování. Dále by bylo zajímavé vývoj na základě použitých technik sledovat dále a za nějakou dělší časovou jednotku (např. rok) vyhodnotit jejich účinnost a užitečnost. Vývoj aplikace IMPRODEMO totiž může pokračovat dále směrem k ještě větší modularitě (např. jednotlivé části panelů nástrojů mohou mít vlastní model prezentace). Taktéž řidící logika programu může být více strukturovaná do samostatných příkazů (implementace návrhového vzoru Command).

Literatura

[1] Intel.com [online]. 2006-06-26, 2011-09-02 [cit. 2011-05-23]. Intel® microprocessor export compliance metrics. Dostupné z WWW: <http://www.intel.com/support/processors/sb/cs-023143.htm>.

[2] NATHAN, Adam. Windows Presentation Foundation Unleashed. USA : SAMS Publishing, 2007. 656 s. ISBN 0-672-32891-7.

[3] Big ball of mud. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2008 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Big_ball_of_mud>.

[4] SMITH, John. WPF Apps With The Model-View-ViewModel Design Pattern. MSDN Magazine Issue [online]. Leden 2009, [cit. 2011-04-05]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>.

[5] KOIRALA, Shivprasad. Design pattern – Inversion of control and Dependency injection. The Code Project ASP.NET articles [online]. 2010-06-13, 1, [cit. 2011-04-05]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>.

[6] ASTELS, David. Test-Driven Development: A Practical Guide. USA : Prentice Hall, 2003. 592 s. ISBN 0-13-101649-0.

[7] ERDOGMUS, Hakan; MORISIO, Maurizio; TORCHIANO, Marco. On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering. 2005, Volume 31 Issue 3, s. 226-237.

[8] NUnit – framework pro testování .NET aplikaci [online]. 2007 [cit. 2011-05-23]. NUnit.org. Dostupné z WWW: <http://www.nunit.org/>.

[9] Cross Correlation [online]. 1996 [cit. 2011-04-16]. 2D Pattern Identification using Cross Correlation. Dostupné z WWW: <http://paulbourke.net/miscellaneous/correlate/#2d>.

[10] WPF Application Framework (WAF) [online]. 2011-01-17 [cit. 2011-04-17]. Dostupné z WWW: <http://waf.codeplex.com/>.

[11] FOWLER, Martin. Patterns of Enterprise Application Architecture. USA : Addison-Wesley Professional, 2002. 560 s. ISBN 0321127426.

[12] MSDN Library [online]. 2010 [cit. 2011-04-18]. Properties (C# Programming Guide). Dostupné z WWW: <http://msdn.microsoft.com/en- us/library/x9fsa0sw.aspx>.

[13] Patterns & practices: Prism [online]. 2010-11-12, 2011-03-24 [cit. 2011-04-18]. Prism 4 Documentation.Dostupné z WWW: <http://compositewpf.codeplex.com/releases/view/55580>.

[14] MSDN Library [online]. 2010 [cit. 2011-05-11]. Assemblies. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.71).aspx>.

[15] POOLE, Charlie. NUnit.org [online]. 2010 [cit. 2011-05-11]. NUnit documentation. Dostupné z WWW: <http://www.nunit.org/index.php?p=documentation>.

[16] STAMATAKIS, William. Microsoft Visual Basic Design Patterns (Microsoft Professional Series). USA : Microsoft Press, 2000. 262 s. ISBN 1572319577.

[17] SIVAKUMAR, Nishant. C++/CLI in Action. USA : Manning Publications, 2007-04-17. Mixing managed and native code, s. 416. ISBN 1932394818.

[18] HOGENSON, Gordon. C++/CLI: The Visual C++ Language for .NET. USA : Apress, 2008-12-08. 448 s. ISBN 1590597057.

[19] FOG, Agner. Agner.org [online]. USA : Copenhagen University College of Engineering, 2011-01-30 [cit. 2011-05-23]. Calling conventions for different C++ compilers and operating systems. Dostupné z WWW: <http://agner.org./optimize/calling_conventions.pdf>.

[20] JONES, Wendy. Beginning DirectX 10 Game Programming. USA : Course Technology PTR, 2007-08-27. 384 s. ISBN 1598633610.

[21] BECK, Kent. Test Driven Development: By Example. USA : Addison-Wesley Professional, 2002-11-18. 240 s. ISBN 0321146533.

[22] Managed Extensibility Framework [online]. 2010 [cit. 2011-05-18]. Dostupné z WWW: <http://mef.codeplex.com/>.

[23] CsUnit.org [online]. 2001 [cit. 2011-05-18]. Dostupné z WWW: <http://www.csunit.org/>.

[24] BECK, Kent. Extreme Programming Explained: Embrace Change. USA : Addison-Wesley Professional, 1999-10-05. 224 s. ISBN 0201616415.

[25] Junit.org [online]. 1997 [cit. 2011-05-18]. Dostupné z WWW: <http://www.junit.org/>.

[26] FOWLER, Martin. Martin Fowler Software development web [online]. [cit. 2011-05-18]. Xunit. Dostupné z WWW: <http://www.martinfowler.com/bliki/Xunit.html>.

[27] WxWidgets [online]. 1992 [cit. 2011-05-20]. Dostupné z WWW: <http://www.wxwidgets.org/>.

[28] Agile software development. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2005 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Agile_software_development>.

[29] Command pattern. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2006 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Command_pattern>.

[30] Serialization. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2005 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Serialization>.

[31] Model-view-controller. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2009, last modified on 2011 [cit. 2011-05-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Model-view-controller>.

Seznam příloh

Příloha A. Návrh grafického uživatelského rozhraní.
Příloha B. Manuál k programu IMPRODEMO
Příloha C. DVD s aplikací IMPRODEMO.

Příloha A: Návrh grafického uživatelského rozhraní

Tento dokument obsahuje návrh grafického uživatelského rozhraní, jak byl proveden v aplikaci Pencil (http://pencil.evolus.vn/en-US/Home.aspx). Ukazuje jednotlivé obrazovky aplikace včetně relevantních popisků.

Tools Panel

Toggle buttony jsou označeny tučným písmem v tlačítku. Netučným jsou označeny klasické tlačítka

Adresar projektu. Urcuje jak projektovy soubor (*.xml), tak adresar pro ukladani obrazku a vysledku.

Ukaze modalni dialog "Advanced Options"

IMPRODEMO

Project Path: ./data/default.xml

Time Units: frames ms s m h

Advaced Options

Camera Capture

Camera | Capture Image | Record | Flow Mode

Frame Rate: 0.45 s 1s 60s

Sequence Operations

12.43s 22.53s Current Frame: 15.24s

Crop | Erase

Fwd | Rev | Stop | Loop

Prvky části "Sequence Operation" budou dostupné jen v případě, že bude vyplé snímání z kamery (Tlačítko "Camera")

Area Selection

Auto Select | Show | Polygon | Subtract | Line | Point | Delete

Results Display

Output Variable: total/xy x y | Graph

Panel umožňující nastavit druh zobrazovaných dat nebo vypočtená data vynést do grafu

iBreak

Compute | Show scale

Toggle button, který při stisknutí spočítá posunutí ve zvolené oblasti. Pokud není zaplá kamera, automaticky bere snímky ze záznamu. Při zapnutí zároveň vypne zobrazování vyznačené oblasti a zapne zobrazení scale meteru

iMechanics

Track Markers | Select Markers | Pair Markers

Info Panel

Track Markers

Zapnuto/Vypnuto automatická metoda pro nalezení markerů

Select Markers

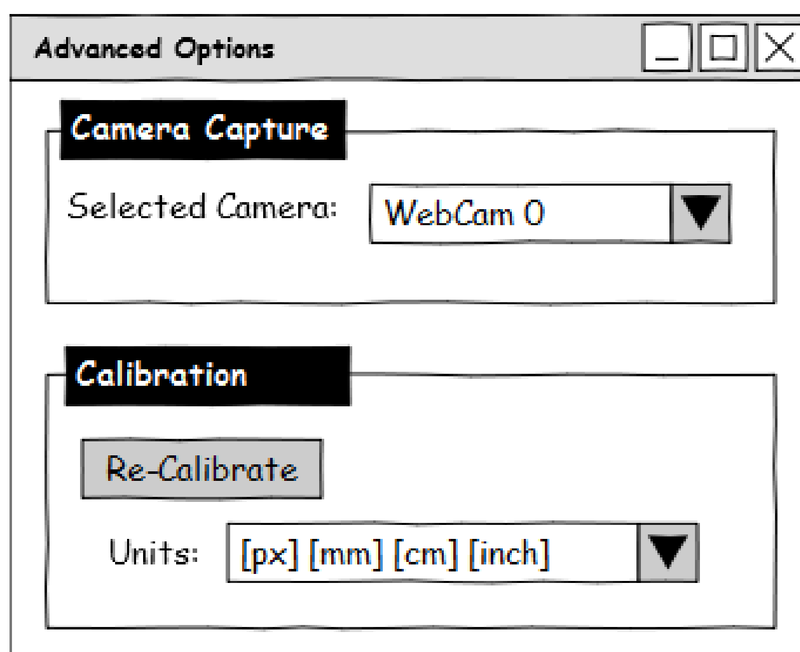
Zapnuto/Vypnuto manuální metoda pro nalezení markerů

Pair Markers

Zapnuto/Vypnuto zobrazení a zadávání sparovaných markerů

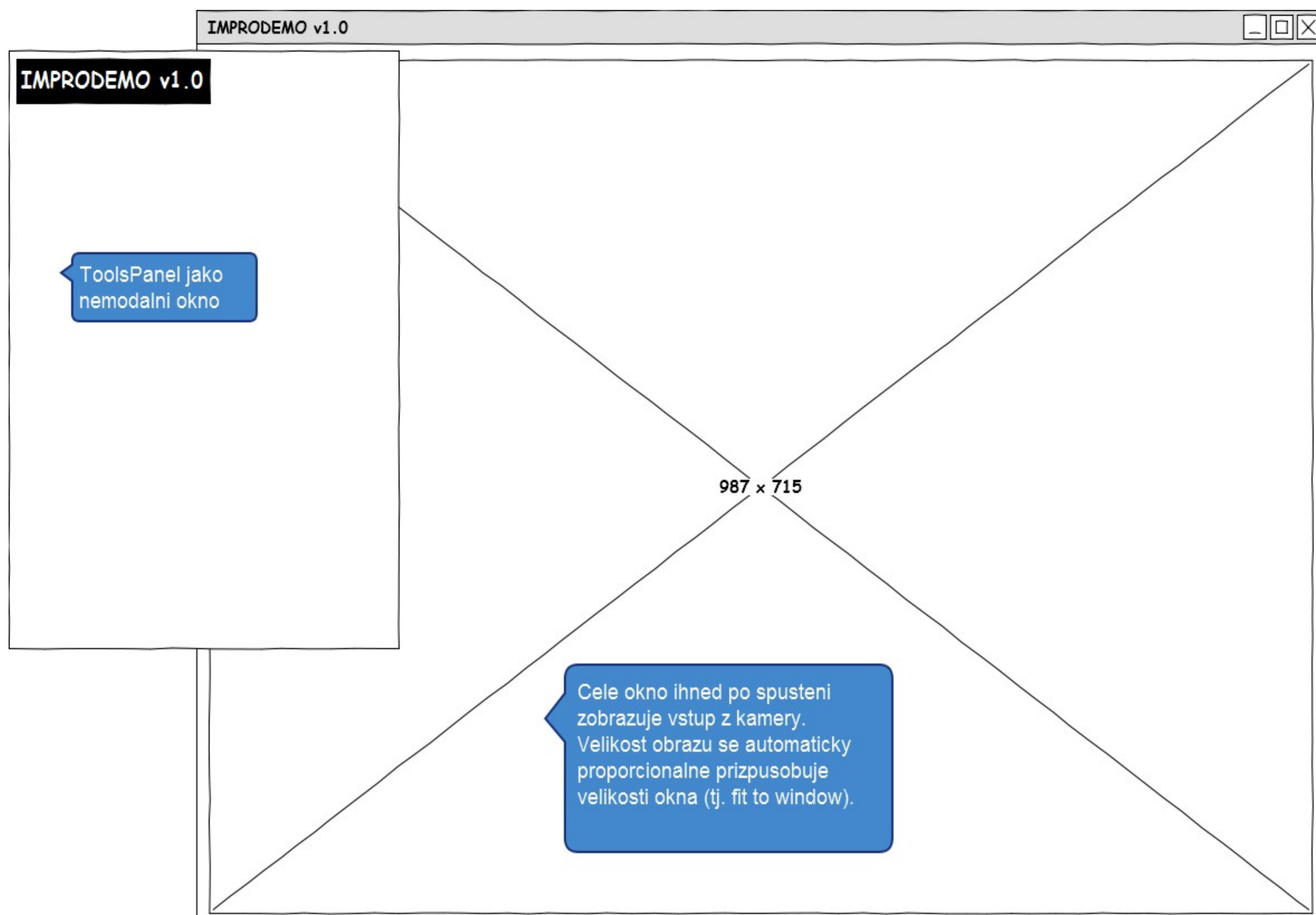
Obrázek A.1: Návrh ToolsPanel.

Advanced Options Panel



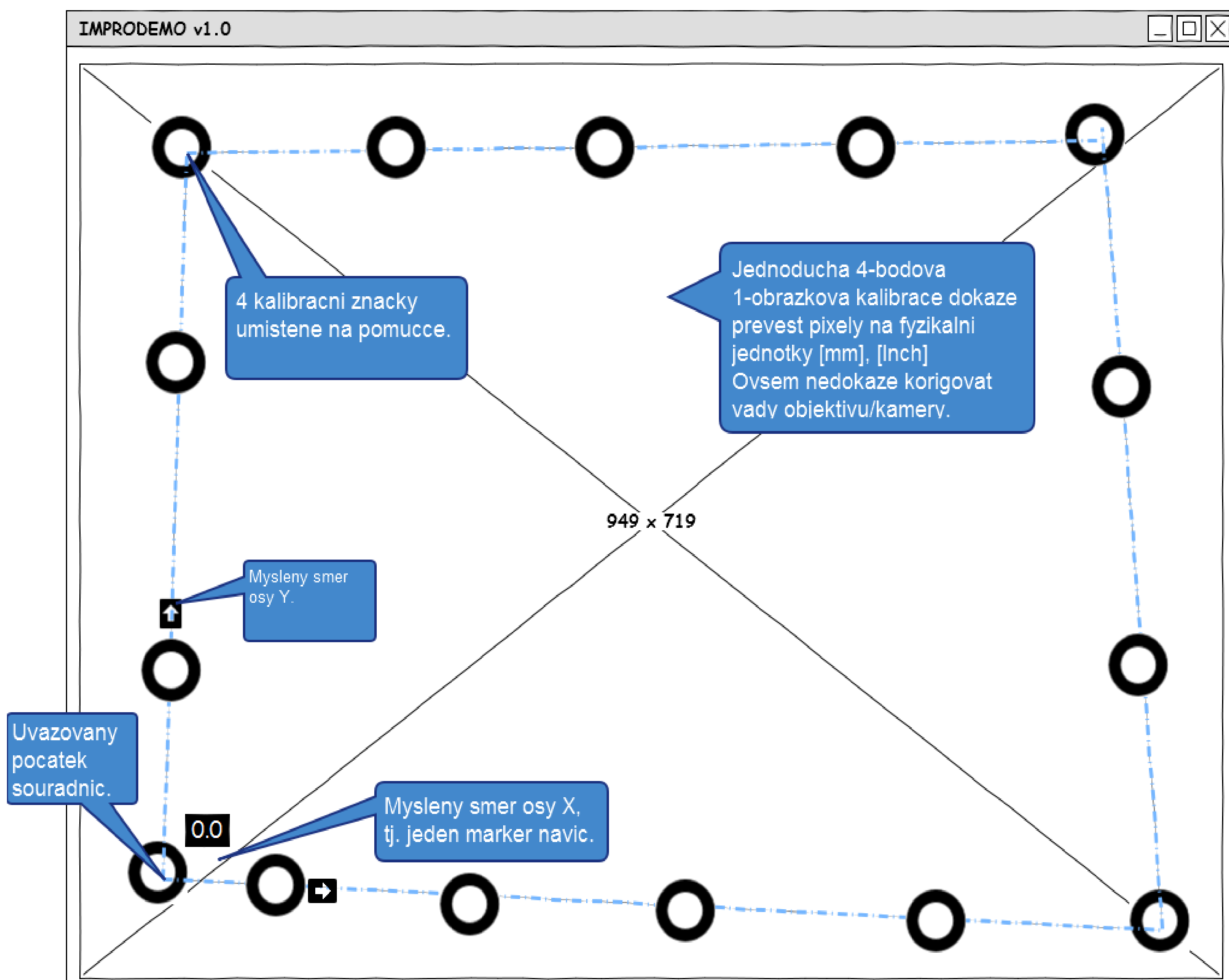
Obrázek A.2: Návrh Advanced Options Panel.

Stav po spuštění



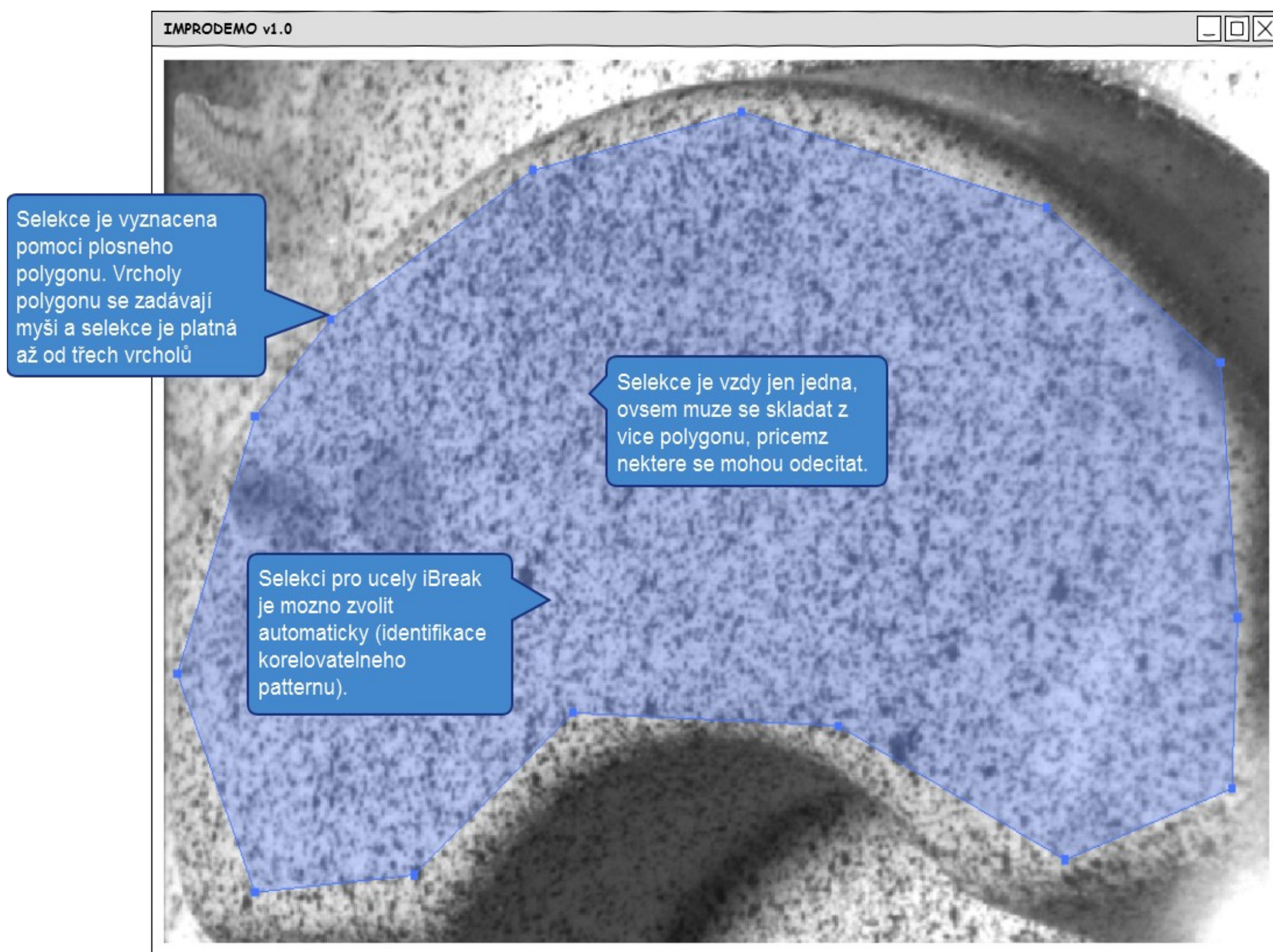
Obrázek A.3: Stav hlavního okna po spuštění.

Kalibrace



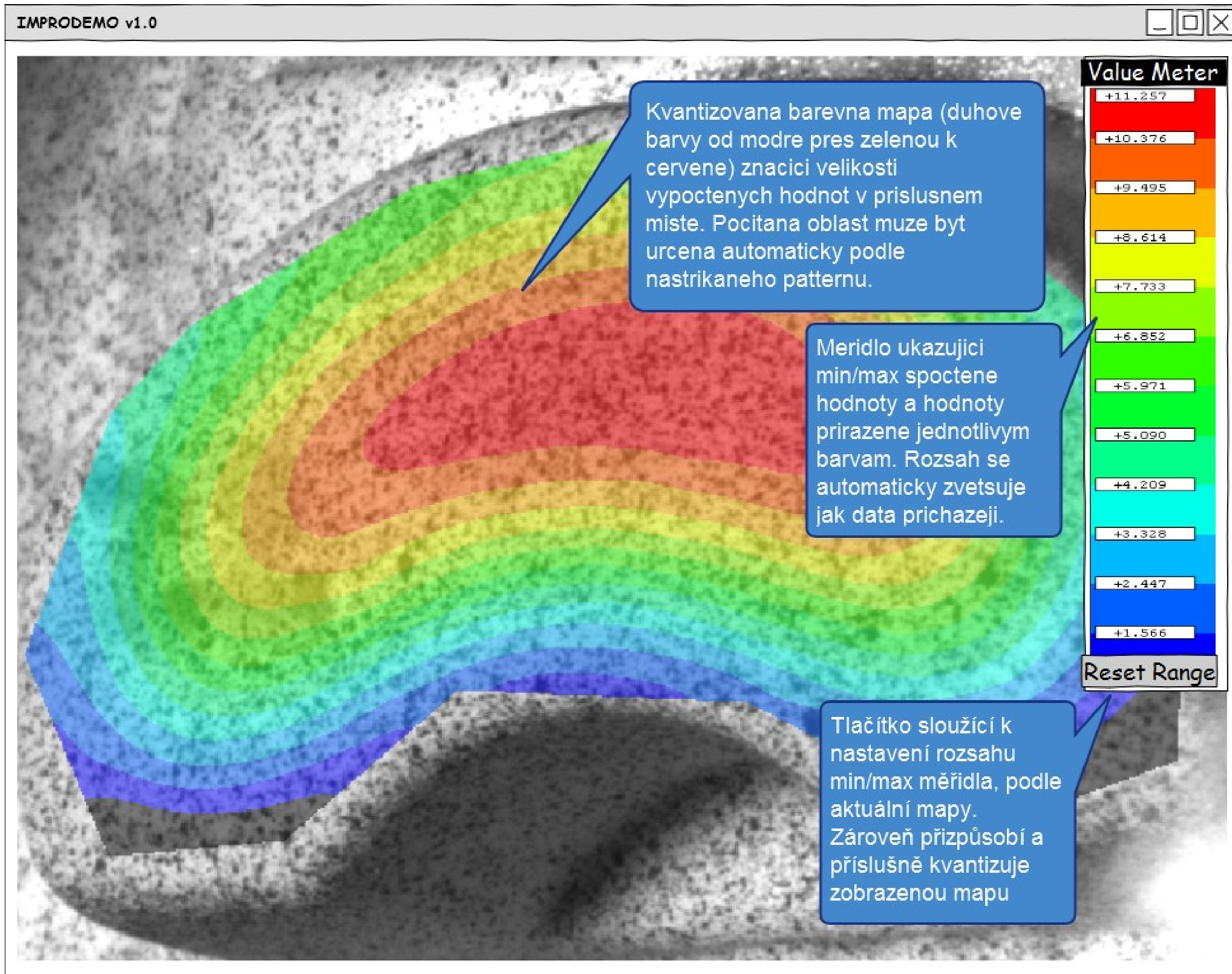
Obrázek A.4: Návrh kalibračního systému.

Selekce



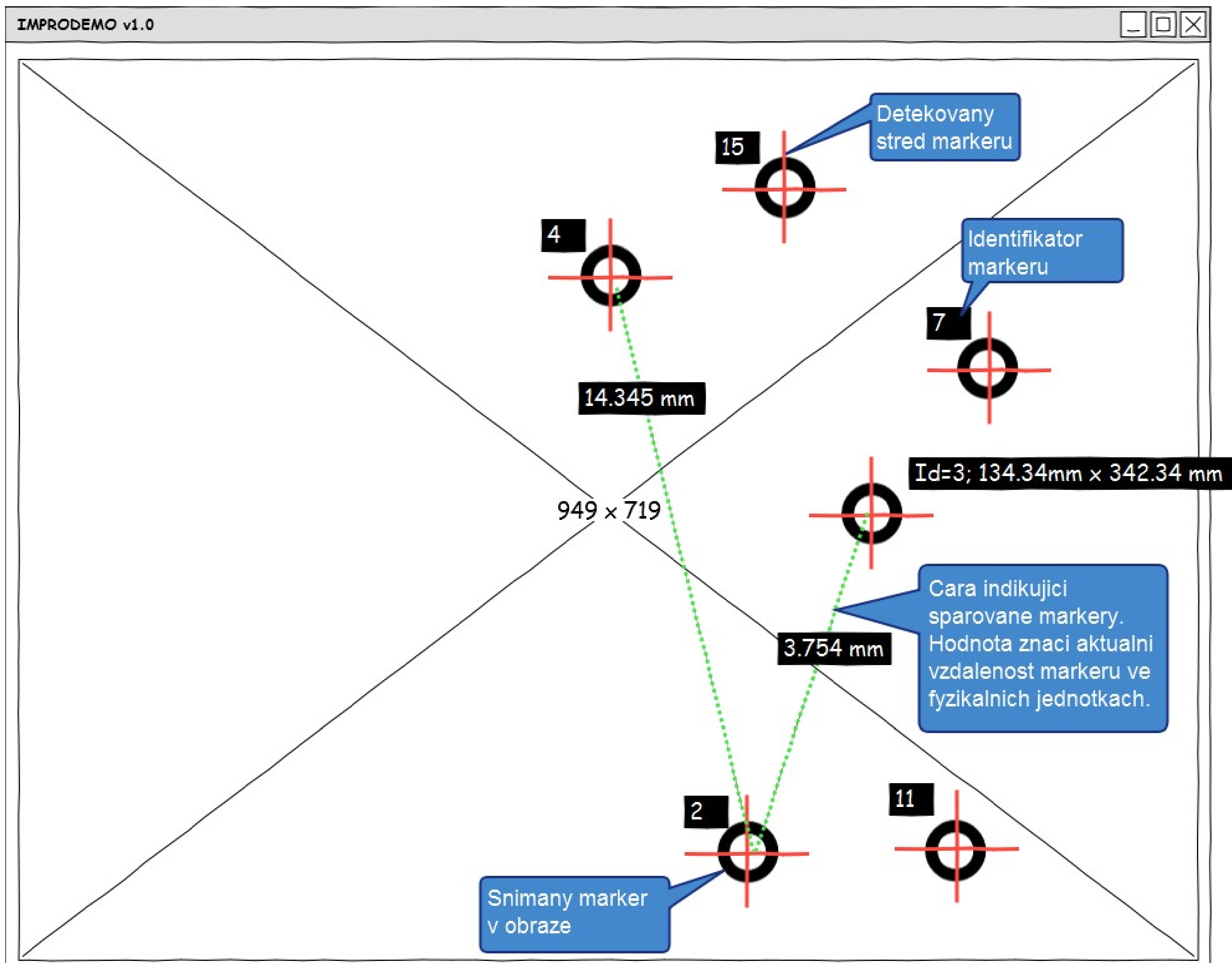
Obrázek A.5: Návrh systému selekcí.

Zvoleno: iBreak



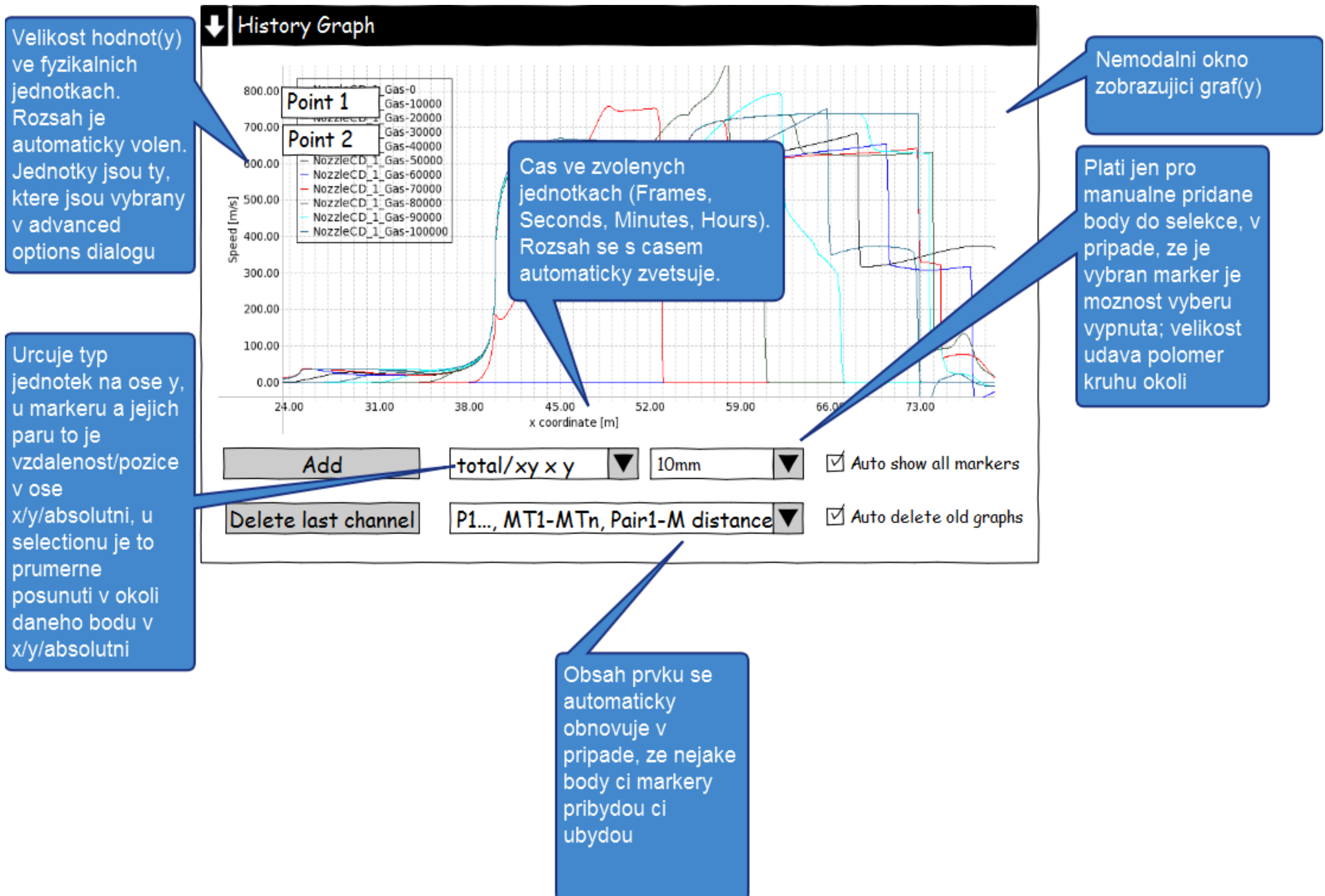
Obrázek A.6: Návrh okna v módu iBreak.

Zvoleno: iMechanics



Obrázek A.7: Návrh marker tracking systému.

Grafy



Obrázek A.8: Návrh zobrazování pomocí grafů.

Příloha B: Manuál k programu IMPRODEMO

Tato příloha popisuje ovládání a práci s aplikací IMPRODEMO. Popisuje podrobně všechny funkce tak, jak byli naimplementovány a předány na DVD v příloze C. Všechny zde popisované cesty se vztahují k adresářové struktuře vytvořené na tomto DVD.

Část 1: Horní informační a ovládací panel

Tento panel slouží k informování uživatele o některých důležitých nastaveních programu. Zároveň poskytuje možnost tyto údaje měnit. Také poskytuje přístup do pokročilých nastavení programu. Všechny prvky tohoto panelu si nyní popíšeme podrobněji.

Obrázek B.1: Horní část a ovládací panel.

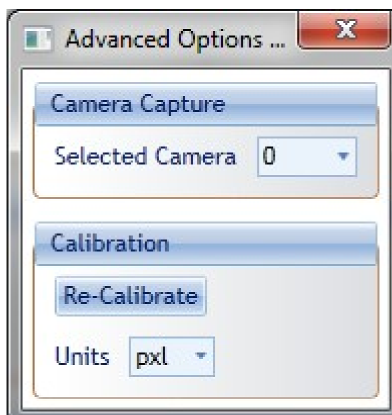
Horní informační panel obsahuje odleva tyto prvky

- Název a aktuální verzi programu.
- Cestu k aktuálnímu projektu. Je formou odkazu a proto se kliknutím na něj dá měnit. Otevře se okno pro výběr souboru a po vybrání souboru se změní cesta ukládání nahrávaných obrázků. Celý projekt popisuje jeden xml soubor. Dále je spoučástí projektu sada nahraných obrázků, kterými lze procházet pomocí části „Sequence operations“, viz příslušná část níže.
- Tlačítko „Load Project“ nahraje již uložený xml soubor s projektem. S tím se také nahrají příslušné obrázky a upraví se rozsah, ve kterém se lze procházet pomocí „Sequence operations“.
- Výběr zobrazovaných časových jednotek. Jsou k dispozici následující možnosti:
    - Frames (snímky)
    - Miliseconds
    - Seconds
    - Minutes
    - Hours

  Tyto budou ovlivňovat zobrazování pozice posuvníku v části „Sequence operations“.
- Tlačítko „Options“. Po jeho stisknutí se vyvolá dialogové okno s pokročilým nastavením programu. Jeho obsah si popíšeme v následující části.

Část 2: Okno s pokročilým nastavením aplikace

Okno s pokročilým nastavením aplikace obsahuje některé detaily, se kterými se běžný uživatel IMPRODEMA nebude zabývat. Většině uživatelů bude stačit výchozí nastaven, ti, co mají zájem o pokročilejší funkce sem ovšem zavítají.



Obrázek B.2: Okno s pokročilým nastavením aplikace.

Okno s pokročilým nastavením aplikace obsahuje následující prvky:

- Výběr kamery. Zde je možno vybrat si ze všech dostupných kamer připojených k počítači. Výběrem kamery se okamžitě aktivuje kamerový vstup vybrané kamery.
- Tlačítko pro kalibraci. Kalibrace je proces, při kterém se naleznou ve snímaném obraze kalibrační markery. Kalibrace probíhá tak, že se před kameru příblíží kalibrační papír (nachází se v obrázku IMPRODEMO_VS2010/TestImages/autocalibration4.bmp) a poté se stiskne toto tlačítko. Kalibrační systém okamžitě začne hledat příslušné markery. Pokud je ihned nenajde, je možné, že jsou některé rozmazané, nebo jsou špatné světelné podmínky. Zkuste tedy lépe nasvítit oblast před kamerou, případně papírem pohnout. Ve chvíli, kdy kalibrační systém najde markery, ohlásí se to zprávou a nyní se jednotky při sledování markerů zobrazují ve fyzikálních jednotkám, které se vyberou v následujícím prvku.
- Výběr jednotek. Uživatel má na výběr z následujících jednotek:
    - pxl. Standartní pixely tak, jak odpovídají zobrazení na obrazovce od levého horního rohu. Standartní volba v případě nekalibravaného obrazu.
    - mm. Milimetry.
    - cm. Centrimetry.
    - inch. Palce. Použití zejména v anglosaských zemích.

Část 3: Ovládání kamery

Tato část obsahuje několik prvků sloužících k ovládání kamerového vstupu a jeho zachycování. Je těsně svázaná s částí 1, ale i s části 4, neboť zde se ukládají zachycené snímky.

Obrázek B.3: Oblast pro ovládání kamery.

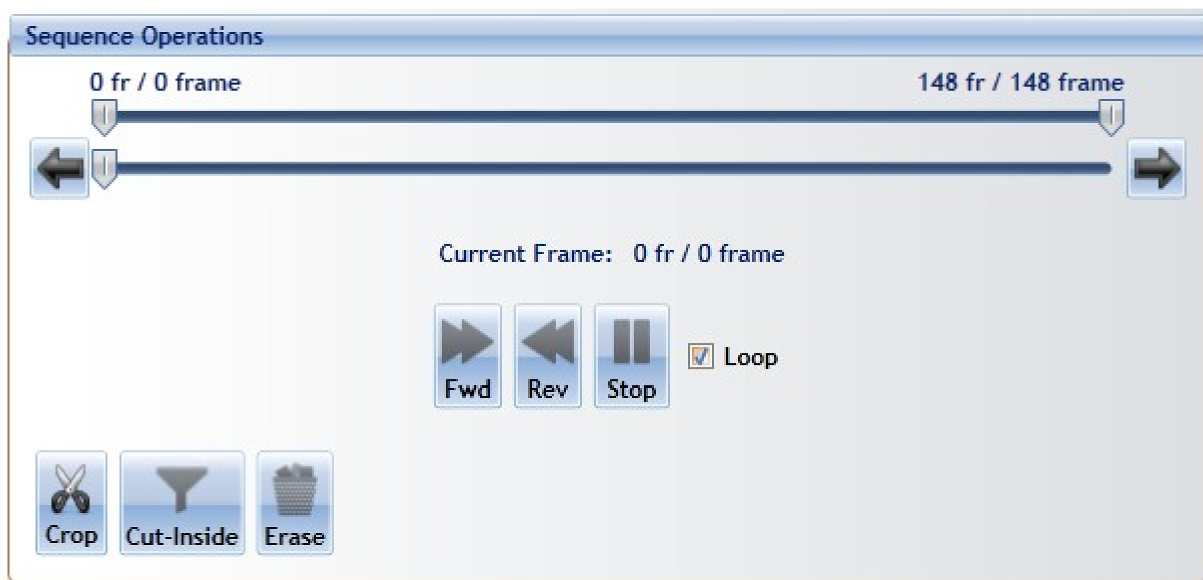
Oblast pro ovládání kamery obsahuje následující prvky:

- Toggle tlačítko pro aktivování kamerového vstupu. Po jeho zamáčknutí dojde k aktivování kamerového vstupu, který je poté zobrazován v hlavním okně. Toto tlačítko je po spuštění programu zapnuté (v případě, že je nalezená na počítači nějaká kamera).
- Tlačítko pro zachycení jednoho snímku. Po jeho stisknutí se zachytí právě jeden snímek z kamery a uloží se do nastaveného projektu. Pokud není zaplý kamerový vstup, stisknutím tohoto tlačítka se zároveň zapne.
- Toggle tlačítko pro nahrávání z kamerového vstupu. Po jeho zamáčknutí se začne veškerý vstup z kamery nahrávat do vybraného projektu. Pokud není zaplý kamerový vstup, stisknutím tohoto tlačítka se zároveň zapne.
- Ovládání rychlosti nahrávání. To může probíhat ve dvou módech
    - Vypsání cílené hodnoty. Po vypsání hodnoty do textového pole a stisknutí klávesy Enter je požadovaná rychlost zaznamenávání snímků uložena a nastavena.
    - Posun pomocí posuvníku. Posuvník umožňuje plynule, tahem myši, měnit rychlost zaznamenávání. Změnou rychlosti v posuvníku se ovlivňuje hodnota v textovém poli a naopak.

  Jednotka rychlosti, která se ovlivňuje, je svázána s vybranou jednotkou v části 1. Kromě rychlostí ve snímcích za vteřinu lze nastavovat rychlost jako jeden snímek za „x“ časových jednotek, přičemž tyto časové jednotky můžou být jakékoliv, které se vyberou v části 1.

Část 4: Ovládání záznamu

Tato část je srdcem celého programu. Obsahuje ovládání záznamu společné pro obě aktivní funkcionality programu (část 6 a 7). Zároveň sem putují veškeré zaznamenané snímky z části 3 a je ovlivňována také vybranými jednotkami z části 1.



Obrázek B.4: Oblast ovládání záznamu.

Oblast ovládání záznamu obsahuje následující prvky:

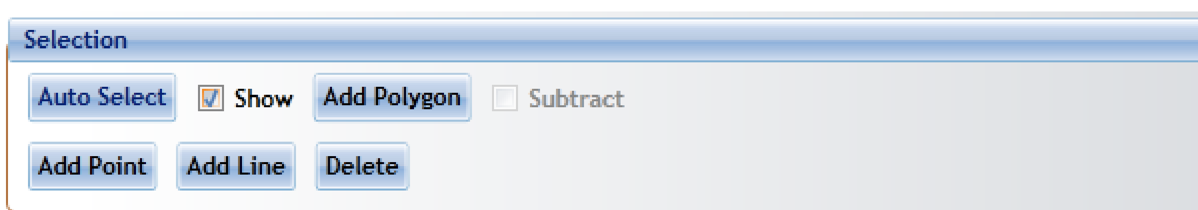
- Posuvník rozsahu záznamu. Ten určuje v jakých intencích se může pohybovat posuvník zobrazeného snímku. Pokud nastavíme nějaký rozsah, pomocí posuvníku zobrazeného snímku se nikdy nemůžeme dostat za tyto nastavené hranice. Zároveň jsme informováni o rozsahu i pomocí textových polí zobrazujících údaj o snímku před lomítkem. Ten se opět chová podle vybrané časové jednotky v části 1. Každý snímek má totiž svoje časové razítko a tak pokud jsou vybrány sekundy, milisekundu, minuty, nebo hodiny, vidíme zde, před lomítkem, kde v čase se vybraný snímek, vůči prvnímu snímku, nachází. Za lomítkem se nachází vždy údaj o pozici snímku v jeho pořadovém čísle.
- Posuvník pozice zobrazeného snímku. Kde se momentálně ve vybraném rozsahu v záznamu nacházíme určíme právě tímto posuvníkem. Pozici vidíme v textu pod ním, v textu označeném „Current Frame“. Nikdy se nemůžeme dostat za nastavené hranice prvním posuvníkem.
- Opakovací tlačítka. Jedná se o tlačítka na stranách posuvníku zobrazeného snímku. Jejich funkce je posouvání do příslušného směru v záznamu. Krome standartní tlačítkové funkce nabízejí možnost přidržení a kontinuálního posunu v záznamu daným směrem.
- Ovládací tlačítka pro přehrávání záznamu. Jsou jimi:
    - Toggle tlačítko pro dopředné přehrátí záznamu. Stisknutím se začne záznam přehrávat rychlostí vybranou v části 3.

- Toggle tlačítko pro zpětné přehrátí záznamu. Slouží pro přehrátí záznamu v rychlosti dané částí 3, ale v opačném směru než předchozí tlačítko.
- Tlačítko pro přerušení přehrávání záznamu. Kliknutím na něj se přeruší jakékoliv přehrávání záznamu.
- Zaškrtávátko určující zda, po přechodu na poslední snímek, bude pokračovat přehrávání záznamu na první snímek či nikoliv.

- Ovládací tlačítka pro manipulaci se záznamem:
  - Tlačítko implementující funkci „Crop“. Snímky nacházející se mimo rozsah určený posuvníkem pro určení rozsahu jsou okamžitě smazány z projektu i z disku.
  - Tlačítko implementující funkci „Cut inside“. Jedná o doplněk předcházející funkce. Snímky nacházející se uvnitř rozsahu určeného posuvníkem pro určení rozsahu jsou okamžitě smazány z projektu i z disku.
  - Tlačítko Erase. Okamžitě smaže z disku i z projektu všechny zaznamenané snímky.

Část 5: Zadávání selekce

Tato oblast slouží k zadávání zájmové oblasti pro počítání deformací a posunutí. Umožuje různou manipulaci sloužící k určení zájmové oblasti a k zobrazení.



Obrázek B.5: Oblast sloužící k zadávání selekce.

Oblast sloužící k zadávání selekce obsahuje následující prvky:

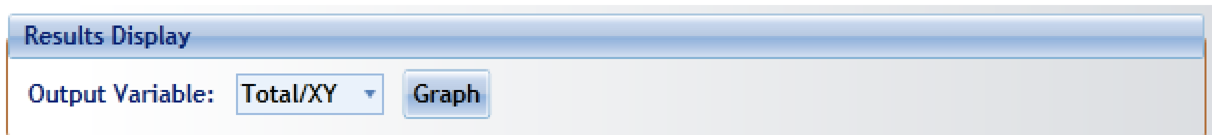
- Tlačítko Auto Select. Po jeho stisknutí dojde k vypočítání automatické selekce na právě zobrazeném obraze. Systém automatické selekce se snaží rozpoznat oblasti nastříkané speciálním postřikem, který byl vybrán kvůli korelační metodě. Cílem je, aby obsahoval velké množství malých částeček, které jsou vhodné pro korelační algoritmus. Typického zástupce tohoto postřiku lze vidět na obrázku IMPRODEMO_VS2010/TestImages/0000_0.tif. Všechny polygony se po vypočítání přidají do hlavního okna a lze s nimi poté manipulovat jako s normálně vytvořenými polygony.
- Zaškrtávací prvek „Show“. Tento prvek určuje, zda-li se mají polygony zadané v obraze zobrazovat.
- Toggle tlačítko „Add Polygon“. Toto tlačítko po zamáčknutí vyvolá mód zadávání polygonů do hlavního okna. Levým kliknutím do hlavního okno se poté zadá vrchol polygonu. Pravým

kliknutím se dokončí zadávání polygonu. Polygon lze kliknutím a táhnutím také posouvat. Vrcholy zadaného polygonu nelze měnit či přesouvat. Polygon lze mazat.

- Zaškrtávací prvek „Substract“. Tímto prvkem se určí mód vykreslovaných polygonů. V případě, že je toto tlačítko zaškrtlé, kreslí se šedivý polygon, který zájmovou oblast odebírá. V případě že není zaškrtlé, kreslí se modrý polygon, který zájmovou oblast přidává.
- Toggle tlačítko „Add Point“ a „Add Line“. Tyto prvky nemají v současné době žádnou funkcionalitu. Jsou připravené kvůli budoucímu rozšíření.
- Toggle tlačítko „Delete“. Po zamáčknutí tohoto prvku dojde k aktivování módu mazání. Po kliknutí na polygon levým tlačítkem v tomto módu se polygon smaže.

Část 6: Zobrazování výsledků

Tato menší oblast nabízí možnosti zobrazování.



Obrázek B.6: Oblast zobrazování výsledků.

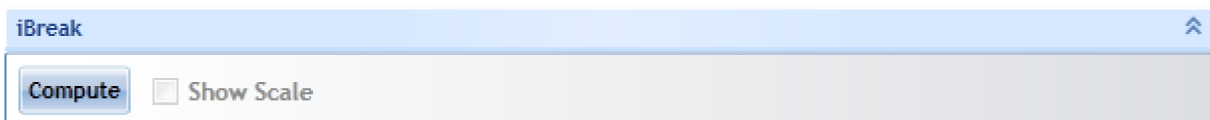
Oblast zobrazování výsledků obsahuje následující prvky:

- Výběr výstupní hodnoty pro zobrazení posunutí. Uživatel má na výběr ze tří možností:
  - Total/XY. Ukazuje euklidovoskou vzdálenost posunutí.
  - X. Ukazuje posunutí v ose X.
  - Y. Ukazuje posunutí v ose Y.

  Výběr této hodnoty ovlivní zároveň již zobrazenou hodnotu a mapu v případě průběhu či pozastavení módu iBreak.
- Toggle tlačítko Graph. Vyvolá nemodální okno zobrazující graf. Podrobnosti v části 9.

Část 7: Ovládání počítání posunutí

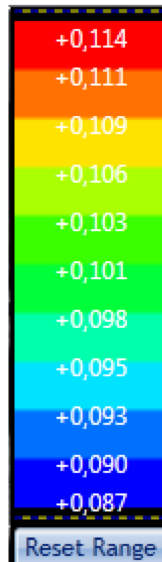
Tato část obsahuje dva prvky pro počítání posunutí.



Obrázek B.7: Oblast ovládání počítání posunutí.

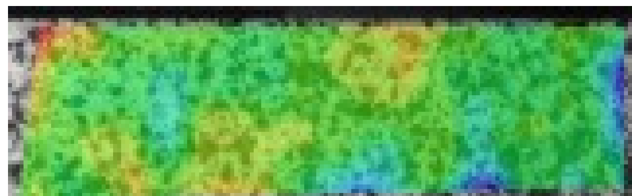
Oblast ovládání počítání posunutí obsahuje dva prvky:

- Toggle tlačítko „Compute“. Zamáčknutím tohoto tlačítka se aktivuje počítání posunutí. Od této chvíle pokud se na hlavním okně objeví nový snímek, okamžitě je poslán na výpočet. Po vypočítání oblasti vyznačené pomocí polygonů se na obrazovce objeví barevná mapa indikující posunutí v pixelech. Zároveň se zapnutím počítacího módu se automaticky skryjí polygony a zobrazí prvek „Scale Meter“. Pokud je vypnuto přehrávání z kamery, automaticky se po stisknutí tlačítka záznam posune na další snímek.
- Zaškrtávátko „Show scale“. Pokud je zaplé předchozí tlačítko pro ovládání počítání, je možné vypnout či zapnout si prvek „Scale Meter“. Ten se zobrazuje v hlavním okně.



Obrázek B.8: Prvek „Scale Meter“.

Prvek „Scale Meter“ ukazuje význam barev vyskytujících se v barevné mapě. Barevná mapa, která přísluší prvku „Scale Meter“ zobrazeném na Obrázku B.8 je zobrazená na obrázku B.9.

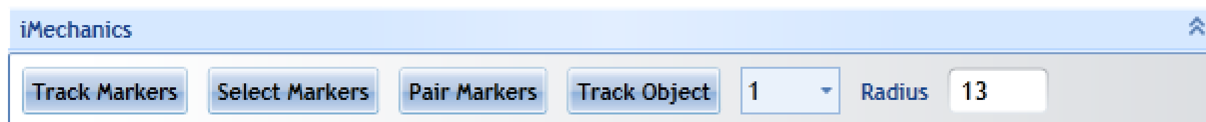


Obrázek B.9: Barevná mapa posunutí.

Každá barva vyjadřuje určitý rozsah hodnot posunutí vysktujících se v barevné mapě. Jak přicházejí další a další snímky, maximální a minimální hodnota prvku „Scale Meter“ se rozšiřuje a tak se můžou objevit snímky, které nebudou obsahovat plné spektrum barev. Kliknutím na tlačítko „Reset Range“ se tento rozsah vždy nastaví na rozsah aktuálně zobrazené mapy.

Část 8: Ovládání zobrazení mechaniky

Tato část obsahuje několik tlačítek určujících rozdílné módy sledování objektů v obraze. Jejich rozdílnou a výlučnou funkcionalitu si v této části podrobně popíšeme.



Obrázek B.10: Ovládání zobrazení mechaniky.

Oblast ovládání zobrazení mechaniky obsahuje následující ovládací prvky:

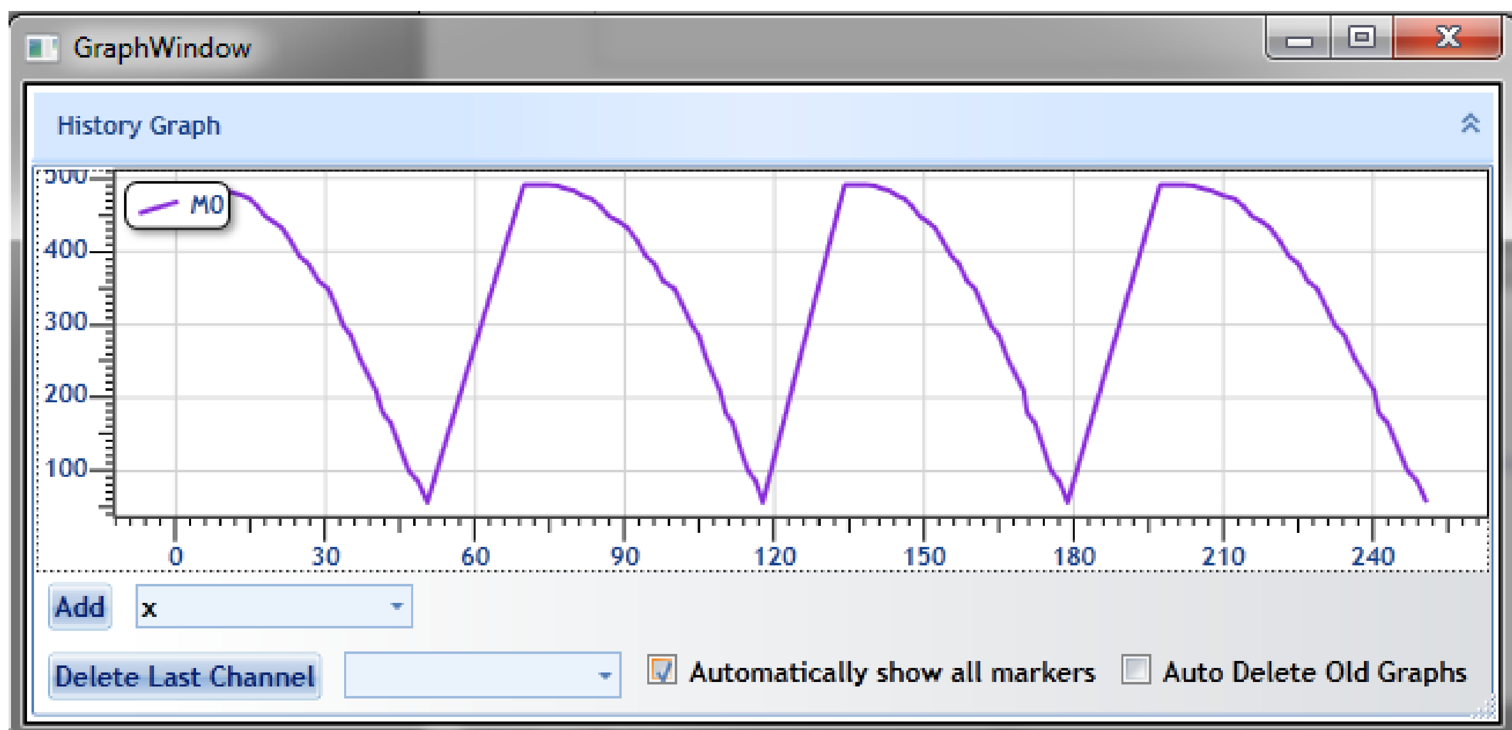
- Toggle tlačítko „Track Markers“. Po jeho zamáčknutí se bude snažit systém sledování markerů snažit nalézt markery v obraze. Marker, který se snaží systém nalézt je možno vidět v souboru IMPRODEMO_VS2010/docs/marker001.bmp. V případě nalezení jejich polohu označí křížem s pozicí. Pokud byla předtím provedena kalibrace (viz část 2) jsou pozice zobrazeny ve vybraných jednotkách, pokud kalibrace není uskutečněna, jsou pozice zobrazeny v pixelech.

Obrázek B.11: Nalezený marker v obraze a jeho vypsaná poloha.

- Toggle tlačítko „Select Markers“. Po jeho zamáčknutí dojde k aktivování módu výběru vlastní sledované oblasti. Ve chvíli, kdy chceme vybranou oblast začít sledovat, je nutno zamáčknout také „Track Markers“. Během sledování markerů může být již „Select Markers“ odmáčklé.
- Toggle tlačítko „Pair Markers“. Slouží k budoucímu rozšíření funkcionality o možnost párovat markery. V současnosti nelze použízt.
- Toggle tlačítko „Track Object“. Po jeho zamáčknutí se bude snažit sledovací algoritmus najít objekt ve tvaru černé koule. Pomocí výběrového prvku vpravo od tlačítka lze nastavit maximální počet nalezených objektů a pomocí textového pole označeného textem „Radius“ lze určit i velikost hledaných objektů.

Část 9: Okno s grafem

Okno s grafem umožňuje sledovat průběhy pozic sledovaných markery v obraze. Obsahuje několik možností nastavení, které si v této části popíšeme.



Obrázek B.12: Okno s grafem.

Okno s grafem obsahuje následující prvky:

- Samotný graf. Ten se dá pomocí táhnutí tlačítka libovolně posouvat, kolečkem myši přibližovat a oddalovat a pravým tlačítkem lze vyvolat menu, které obsahuje kompletní nápovědu ovládání.
- Tlačítko „Add". Slouží k manuálnímu přidání markerů do grafu. Marker, který je cílem vybrat, se vybere z výběrového prvku vedle tlačítka „Delete Last Channel".
- Výběrový prvek určující typ zobrazované hodnoty. Může jím být hodnota x, y, nebo celková euklidovská vzdálenost nazvaná „total xy".
- Tlačítko Delete Last Channel. Po jeho stisku se z komponenty grafu odebere posledně přidaný graf.
- Zaškrtávátko „Automatically show all markers". Pokud je zaškrtlé, tak všechny markery nalezené v obraze se automaticky přidají jako grafy do grafové kompomenty. Pokud zaškrtlé není, je nutné všechny požadované grafy přidat ručně.
- Zaškrtávátko „Auto Delete Old Graphs". V případě, že marker se na chvíli ztratí je v standartní situaci okamžitě odebrán ze zobrazovaných grafů. Pokud je tato možnost zaškrtnuta, zůstávají zobrazeny v grafové komponentě vždy, i přestože už zmizí z obrazu.