Vývoj desktopových aplikací v Jazyce Java s bohatým grafickým uživatelským rozhraním.

Bakalářská práce

autor: **Petr Bálek**

Vedoucí práce: **RNDr. Jaroslav Icha**

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra Informatiky

akademický rok 2010 / 2011

Prohlášení

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

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

V Českých Budějovicích dne

Abstrakt

Práce se snaží ukázat na pokročilých příkladech možnosti, které nabízejí již standardně dodávané balíčky Javy, kde největší důraz je kladen na balíčky javax.swing a java.awt. V práci jsou samozřejmě vytvořeny a uvedeny ukázkové příklady, které jsou vybudovány na výše uvedených balíčcích. Okruhy, kterých se práce týká jsou například animace, efekty statické a dynamické, konvoluční filtry, přechody, alfa kompozice a jiná další témata.

Abstract

This work aims to show examples of the advanced options that offer longer supplied as standard Java packages, where the greatest emphasis is placed on the package javax.swing and java.awt. In this work, of course, created and given exemplary examples that are built on the above packages. Circuits, which work as the animation, effects static and dynamic convolution filters, gradients, alpha composition, and other another topics.

Klíčová slova

java, programování, grafické uživatelské rozhraní, GUI, swing

Keywords

java, programming, Graphical User Interface, GUI, swing

Poděkování

V prvé řadě bych tímto rád poděkoval panu RNDr. Jaroslavu Ichovi, za cenné rady a časté konzultace, které mi pomohly při psaní této práce. Dále také děkuji příslušníkům své rodiny, kteří mi zajišťovali materiální zázemí i duchovní oporu při mém studiu. V neposlední řadě patří dík mé přítelkyni Nele, která byla tolerantní v časech, kdy jsem se Jí nemohl plně věnovat a musel jsem pracovat. Rovněž Jí patří dík za Její designerské názory, které korigovaly vzhledy některých výstupních programů. Další dík patří všem bytostem (lidem, rostlinám i zvířatům), které na mne pozitivně působili celý můj život a dodávali mi sil do života.

Obsah

1 Úvod

Programovací jazyk Java nabízí v dnešní době, při vývoji desktopových aplikací velmi známý a v hojné míře také využívaný balíček javax.swing. Tento balíček slouží pro budování grafického uživatelského rozhraní. Pro Javu je samozřejmě možné vytvářet GUI, neboli grafické uživatelské prostředí i za pomocí jiných balíčků Javy, ať už standardních, jakým je například balíček java.awt, kterému je v práci věnována také část či nestandardních nedodávaných v základních balíčcích, jako jsou například knihovny OpenGL, v Javě pojmenované jako JOGL či DirectX v prostředí operačních systémů Microsoft Windows. Těmto nestandardním, jistě výkonným a schopným knihovnám nebude však věnována pozornost, neboť se domnívám, že standard Javy nabízí sám o sobě velký potenciál.

1.1 Cíle práce

Cílem práce je ukázat nějaké vybrané a zajímavé možnosti, které nabízí knihovna javax.swing pro vývoj desktopových aplikací a pokusit se stávající techniky vylepšit, tak že budou vytvořeny aplikace, které zaujmou uživatele prostřednictvím svých naimplementovaných grafických a animačních efektů. Práce si mimo jiné klade za cíl uvést čtenáře tváří v tvář úskalí, která nabízejí standardně dodávané balíčky javax.swing a java.awt, ale také mu nabídnout řešení oněch vyvstalých problémů. V bakalářské práci je samozřejmě dále vytvořeno několik původních ilustrujících aplikací, které budou reprezentovat uvedený

přístup. Těm nejdůležitějším částem zdrojových kódů aplikací je věnována samozřejmě pozornost i v textu.

1.2 Motivace

Hlavní motivací pro zadání této bakalářské práce bylo mé studium kurzů Javy na pedagogické fakultě JČU, které bylo rozdělené do třech semestrů. V těchto třech semestrech jsem absolvoval kurzy programování PGJ1, PGJ2 a PGJ3. V předmětech PGJ2 a PGJ3 byla výuka zaměřená mimo jiné také na tvorbu GUI. Vzhledem k tomu, že knihovna javax.swing a všeobecně grafika je velmi rozsáhlé téma, rozhodl jsem se, že se budu věnovat této problematice více. V rámci této snahy jsem se rozhodl vytvořit několik výukových projektů, které budou moci (dle uvážení učitele kurzu) doplňovat probírané příklady v kurzech.

2 Java 2D API

2.1 Co je Java 2D API?

„Java 2D API je sada tříd pro tvorbu pokročilé 2D grafiky a zpracování obrazu. Toto grafické rozhraní zahrnuje algoritmy pro tvorbu křivkové grafiky, textu a obrázků do jednoho komplexního modelu. API také poskytuje rozsáhlou podporu pro kompozici obrazu, včetně obrázků s alfa kanálem. Sada tříd rovněž poskytuje přesnou definici barevného prostoru, a bohatou sadu zobrazovacích promítačů. Tyto třídy jsou poskytovány jako dodatky k balíčkům java.awt a java.awt.image.“ [1]

2.2 Pokročilá grafika a obrázky

„Rané verze AWT poskytovaly jednoduhý renderovací balíček vhodný pro vykreslování běžných HTML stránek, ale neschopný pro složitější vykreslování složitější grafiky či obrázků. Oproti tomu Java 2D již poskytuje flexibilnější vykreslování. Například pomocí třídy **java.awt.Graphics** můžete vykreslovat obdelníky, elipsy a polygony. Třída **java.awt.Graphics2D** zlepšuje koncept geometrického vykreslování na jakýkoliv geometrický útvar. S Java 2D API můžete vykreslovat křivky různých tvarů, jakékoliv šířky či vyplňovat geometrické útvary jakoukoliv texturou. Všechny geometrické tvary jsou poskytovány třídami implementující rozhraní **java.awt.Shape**, například třída **java.awt.geom.Rectangle2D** či **java.awt.geom.Ellipse2D**. Křivky a oblouky jsou také konkrétní specifikací rozhraní **java.awt.Shape**. Vyplňování a styl křivek mají na starosti třídy, které implementují rozhraní **java.awt.Paint** a **java.awt.Stroke**. Například

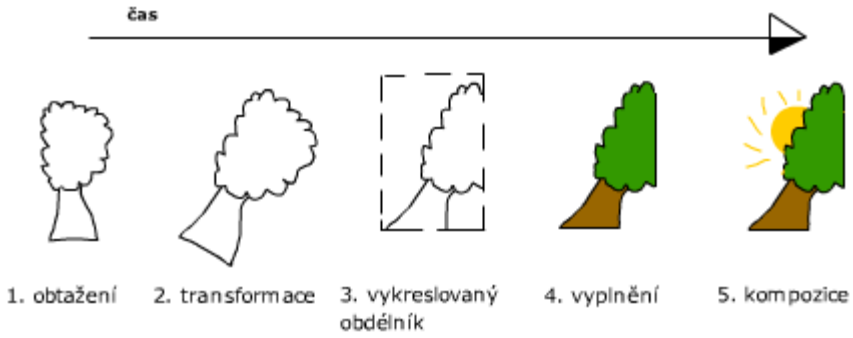
java.awt.BasicStroke, **java.awt.GradientPaint**, **java.awt.TextuterePaint**, **java.awt.Color**, od Javy 1.6 také abstraktní třída **java.awt.MultipleGradientPaint** a mnoho dalších vyplňovacích či vykreslovacích stylů. Třída **java.awt.geom.AffineTransform** definuje transformace 2D souřadnic, které zahrnují například změnu měřítka, překládání či otáčení. Ořezávací oblasti jsou definovány ve třídách, které implementují rozhraní java.awt.Shape. Například **java.awt.geom.GeneralPath** apod. Barevná kompozice je poskytována implementací rozhraní **java.awt.Composite**. Příkladová třída, která toto kritérium splňuje je třída **java.awt.AlphaComposite**. " [2]

2.3 Renderovací model

Základní model vykreslování grafiky se nezměnil s přidáním Java 2D API. K vykreslování grafiky nastavíte grafický kontext a vyzvete vykreslovací metodu objektu. Třída **java.awt.Graphics2D** rozšiřuje třídu **java.awt.Graphics** a poskytuje tudíž i nové vykreslovací metody. Java 2D API automaticky převádí rozdíly mezi souřadnicovými systémy různých grafických zařízení a navenek se tváří jako jednotný systém. S příchodem Javy 1.3 je také poskytována podpora pro více-obrazovková prostředí. [3]

2.3.1 Proces vykreslování

Vykreslování v AWT je možné shrnout do 5 kroků, jak ukazuje níže uvedený obrázek 1.



Ilustrace 1: Proces vykreslování, překresleno z [4]

Tyto kroky můžeme popsat:

1. útvar je obtažen
2. útvar je transformován (otočen, je mu změněno měřítko, atd.)
3. vykreslovanému objektu je nastaven překreslovací obdelník, který má být překreslen
4. objekt je vyplněn nějakou texturou
5. nakonec je aplikována na objekt kompozice, což zahrnuje vypočítávání, jak bude vypadat výsledná barva pixelu, který je sdílen dvěma či více grafickými objekty [4]

2.4 Souřadnicové systémy

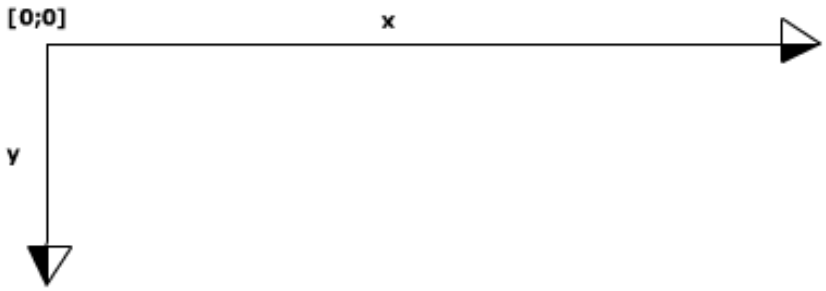
„Java 2D API podporuje dva souřadnicové systémy:

- uživatelský prostor

- prostor grafického zařízení “[5]

2.4.1 Uživatelský souřadnicový systém

„Uživatelský souřadnicový systém je logický systém operující ve dvou osách, kde hodnoty horizontální **osy x** se zvětšují směrem doprava a hodnoty vertikální **osy y** se zvětšují směrem dolů. Počátek tohoto souřadnicového systému je v levém horním rohu zařízení jak ukazuje obrázek číslo 2.



Ilustrace 2: Souřadnicový systém, překresleno z [5]

Tento systém je nezávislý na grafickém zařízení. Uživatelský prostor reprezentuje jednotný abstraktní model všech možných souřadnicových systémů různých grafických zařízení. Při vykreslování grafického objektu, Java sama zařídí převedení z obecného souřadnicového uživatelského prostoru na individuální prostor grafického zařízení.“[5] Z tohoto důvodu je to pro programátory jednoduchý systém, kde se Java postará o nízkoúrovňové operace a programátor se může soustředit jiným směrem.

2.4.2 Souřadnicový systém grafického zařízení

„Java 2D API definuje tři úrovně informací o konfiguraci, které slouží mimo jiné také při konverzi z uživatelského prostoru do prostoru grafického zařízení. Tyto informace jsou zapouzdřeny ve třech třídách:

- GraphicsEnvironment
- GraphicsDevice
- GraphicsConfiguration“ [5]

2.4.2.1 Třída java.awt.GraphicsConfiguration

„Tato třída popisuje kolekci renderovacích zařízení dostupných v Java aplikacích na konkrétní platformě. Renderovací zařízení zahrnuje:

- obrazovky
- tiskárny
- obrázkové buffery

Třída GraphicsEnvironment také obsahuje seznam všech dostupných fontů na platformě, grafických zařízení + ke každému zařízení výčet všech módů, které dokáže ten který hardware používat. “[5]

2.4.2.2 Třída java.awt.GraphicsDevice

„Tato třída popisuje pro aplikaci viditelné renderovací zařízení, jakým je například obrazovka či tiskárna. Každá možná konfigurace zařízení je reprezentována třídou java.awt.GraphicsConfiguration. Například SVGA obrazovka může pracovat v několika módech:

- v rozlišení 640 na 480 pixelů při 16 barvách
- v rozlišení 640 na 480 pixelů při 256 barvách
- v rozlišení 800 na 600 pixelů při 256 barvách

Tato fyzická SVGA obrazovka je reprezentována instancí třídy GraphicsDevice a všechny možné zobrazitelné módy touto obrazovkou jsou reprezentovány instancemi třídy GraphicsConfiguration“[5]

2.5 Výplně a styly obtahování

Příchod Java 2D API umožnil vykreslování křivek různými styly per a vyplňování ploch všelijakými texturami. Ačkoliv text je reprezentován sadou glyfů, tak textové řetězce mohou být také obtaženy a vyplněny jako ostatní grafika. Styly vykreslovacích per jsou definovány objekty, které implementují rozhraní **java.awt.Stroke**. Těmto obrysům je dovoleno specifikovat různé šířky a typy čar, jakými jsou například čerchovaná, tečkovaná čára, atd. Vyplňovací vzory neboli textury jsou definovány objekty, které implementují rozhraní **java.awt.Paint**. Letitá třída **java.awt.Color**, která byla přítomna v raných verzích AWT je jednoduchý příklad implementace java.awt.Paint. Nyní balík java.awt obsahuje 8 tříd, které implementují rozhraní java.awt.Paint. A jsou to konkréttně třídy:

- GradientPaint
- LinearGradientPaint
- MultipleGradientPaint
- RadialGradientPaint

- Color
- SystemColor
- ColorUIResoure
- TexturePaint

Nebudu všechny tyto třídy popisovat, vyberu jen část. Třída **java.awt.GradientPaint** umožňuje vytvářet výplně, které jsou spočítány jako přechody mezi 2 zadanými barvami. Třída **java.awt.MultipleGradientPaint** slouží obdobně jako výše zmiňovaná třída GradientPaint, jen s rozdílem, že je možné zadat více různých barev z kterých se bude barva výsledné výplně skládat. Třída **java.awt.TextuterePaint** vytvoří výplň z obrázku, který je předán v parametru. [6]

2.6 Kompozice

Když je vykreslován nějaký objekt, který překrývá nějaký jiný objekt, tak je potřeba určit jak zkombinovat barvy tohoto nového objektu s barvami které již zabírají plochu, kde se bude vykreslovat. Java 2D API zapouzdřuje pravidla, jak kombinovat barvy v rozhraní **java.awt.Composite**. Primitivní vykreslovací systémy poskytují jenom booleovské operace kombinování barev. Například booleovská pravidla by mohla umožnit logický součin, logický součet či exkluzivní disjunkci barev. Existuje několik problémů, z tohoto důvodu, příkladovým problémem nechť poslouží fakt, že není příliš snadné sčítat barvy. Java 2D API se vyhýbá těmto nástrhám a má již hotovou svou třídu

java.awt.AlphaComposite, která definuje pravidla, jak pracovat s pixely, které jsou sdíleny více objekty. [7]

2.7 Obrázky

„Obrázky jsou kolekce prostorově orientovaných pixelů. Jeden pixel definuje jeden bod obrázku. Dvourozměrné pole pixelů se nazývá raster. Vzhled pixelu může být definován přímo nebo jako index v barevné tabulce obrázku. V obrázcích, které obsahují více barev (více než 256), pixely obvykle přímo reprezentují barvu, alfa kanál a další informace.. Takové obrázky mají tendenci být mnohem větší než obrázky, které mají indexované barvy, ale vypadají realističtěji. U obrázků s indexovanými barvami jsou barvy limitovány specifikovanými barvami barevné tabulky, což často vede k menšímu počtu barev, které je možno použít v obrázku. Obrázky s indexované barvami typicky požadují méně úložného prostoru než barevná hodnota. Proto obrázky uchovávané jako množina indexovaných barev jsou obvykle menší. Tento pixelový formát je populární u obrázků, které obsahují jen 16 nebo 256 barev. Obrázky v Java 2D API mají dvě základní komponenty:

- obrazová data (pixely)
- informace důležité pro interpretování pixelů

Pravidla pro interpretaci pixelu jsou zapouzdřeny například ve třídě **java.awt.image.ColorModel**. Obrázkový balíček **java.awt.image** obsahuje také několik dalších implementací barevného modelu. Implementace třídy **java.awt.color.ColorSpace** ve třídě **java.awt.Color** reprezentuje nejpopulárnější barevné prostory, zahrnující například RGB

či stupně šedi. Nakonec bloku ještě poznámka: barevný prostor není kolekce barev, nýbrž pouze definice pravidel, která určují jak interpretovat barevnou hodnotu. Oddělení barevného prostoru od barevného modelu poskytuje větší flexibilitu v tom jak jsou barvy reprezentovány a převedeny z jedné barevné reprezentace do druhé.“ [8]

2.8 Transformace

Java 2D API má jednotný model transformace souřadnic. Všechny transformace souřadnic, včetně transformací z uživatelského souřadnicového prostoru do souřadnicového prostoru grafického zařízení jsou reprezentovány objekty třídy j**ava.awt.geom.AffineTransform.** Tato třída definuje pravidla pro manipulaci se souřadnicemi objektu pomocí matic. [9]

3 Balíček javax.swing

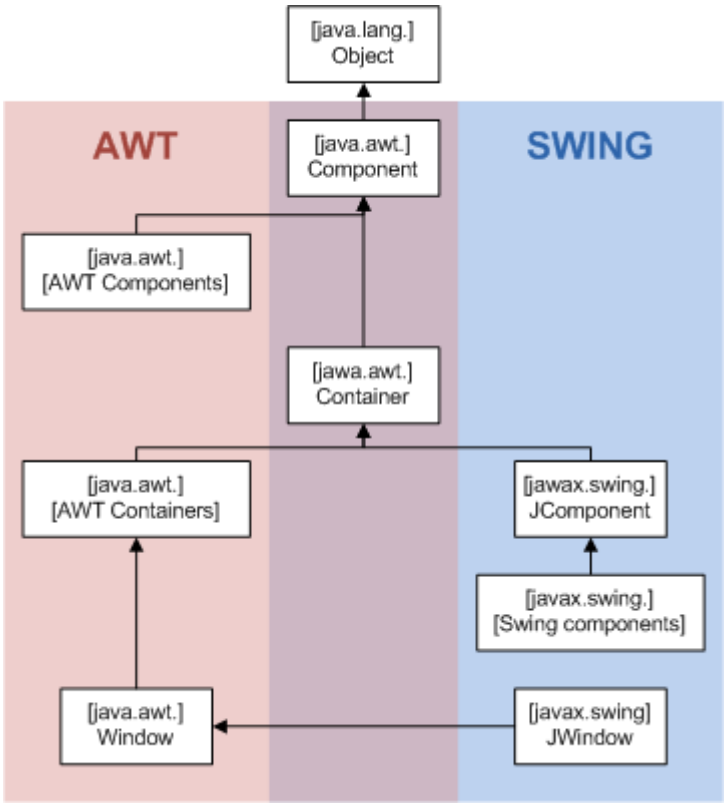
Swing obsahuje velké množství komponent pro tvorbu grafického uživatelského rozhraní a přidávání interaktivity do Java aplikací. Swing zahrnuje ve svých balíčcích snad všechny komponenty, které může uživatel očekávat od moderního grafikého rozhraní: tabulky, tlačítka, podporu drag and drop, rozklikávací stromové struktury a jiné další komponenty. Mimo jiné obsahuje také podporu pro zpětné operace, valné většině uživatelů známých pod klávesovou zkratkou CTRL + Z. [10]

3.1 Historie Swingu

„Původní nástroj pro tvorbu grafického uživatelského rozhraní bylo AWT, vyvíjené firmou Sun Microsystems jako součást Javy. První verze vyšla již v roce 1995, spolu s první verzí Javy. Během dalšího vývoje se ale u AWT projevily chyby v návrhu a další problémy. Např. bylo závislé na platformě a tím porušovalo jeden z hlavních principů, na kterém je jazyk Java postaven. A to platformní nezávislost, neboť AWT poskytuje pouze nativní komponenty konkrétního operačního systému, tzv. heavyweight neboli těžké komponenty. V roce 1997 Sun upustil od dalšího vývoje AWT a začal vyvíjet Swing. Jako základ sloužily Internet Foundation Classes (IFC), vyvinuté Netscape Communications Corporation na konci roku 1996, jako nástroj pro tvorbu grafického uživatelského rozhraní pro Javu. V roce 1997 došlo ke spojení IFC a dalších technologií od Netscape a Sun Microsystems do Java Foundation Classes (JFC), jehož je Swing součástí. Swing je vyvíjen jako kombinace

AWT, IFC a dalších technologií. Swing je nedílnou součástí Java SE od verze 1.2. Do té doby byl dostupný pouze jako knihovna k samostatnému stažení.” [11]

3.2 Hierarchie tříd



Ilustrace 3: Vztahy mezi balíčky AWT a SWING, převzato z [12]

„Základním stavebním kamenem všech grafických komponent je třída java.awt.Component. Instance třídy java.awt.Component představují objekty, které mají svoji grafickou reprezentaci a mohou být zobrazeny na monitoru a interagovat s uživatelem. Všechno, co Java na monitoru

nakreslí, je instance třídy java.awt.Component. Různé ovládací prvky (tlačítko, zaškrtávátko, seznam, atd…) v AWT jsou definovány jako potomci třídy java.awt.Component. Od třídy java.awt.Component je dále odvozena třída java.awt.Container (kontejner). Kontejner je taková grafická komponenta, která v sobě může držet a kreslit ostatní grafické komponenty. Kde se komponenty v kontejneru nakreslí, určuje tzv. správce rozložení (implementace rozhraní java.awt.LayoutManager). V AWT jsou od třídy java.awt.Container odvozeny různé prvky, např. třída java.awt.Window nebo java.awt.Panel. Základním kamenem knihovny javax.swing je třída javax.swing.JComponent. Tato třída je rodičem každé swingovské komponenty. Jak je patrno z obrázku 3, javax.swing.JComponent je potomkem java.awt.Container. Tudíž všechny swingovské prvky v sobě mohou držet další komponenty. Potomci třídy javax.swing.JComponent jsou samotné swingovské komponenty, jako je javax.swing.JButton (tlačítko), javax.swing.JList (seznam), javax.swing.JPanel (obecný panel), atd… Všechny swingovské komponenty mají před svým logickým názvem písmeno J, aby byly rozlišitelné od svých protějšků z AWT. K této obecné hierarchii existují vyjímky, a to jsou okna a obecně všechny nejvyšší kontejnery (okna, dialogy, applety, apod.). Okna jsou obecně potomci třídy java.awt.Window, a nemohou proto být potomci třídy javax.swing.JComponent (v Javě nelze dědit od více tříd). Swingovské protějšky k java.awt.Window a jeho potomků (java.awt.Frame, java.awt.Dialog, atd…) jsou definovány jako potomci těchto tříd s „J“ na začátku. Hlavní swingovské okno javax.swing.JFrame je definováno jako potomek java.awt.Frame” [11]

3.3 Architektura MVC

„Model – View – Controller (MVC) je typ softwarové architektury, kde je uživatelské rozhraní (View) odděleno od logiky programu (Model) řídící logikou (Controller). Tyto tři komponenty bývají často odděleny rozhraními, a tak modifikace jedné části má minimální vliv na ostatní části. Ve Swingu má každý grafický objekt (tlačítko, seznam, rozbalovací menu…) svůj model či více modelů v podobě rozhraní, které jsou na daném grafickém objektu nezávislé. Swing poskytuje defaultní implementaci modelu pro běžné využití, ale nic uživateli nebrání, aby si naprogramoval objekt s jiným modelem. Toto umožňuje uživateli změnit chování grafické komponenty, ale využívat její vzhled. Například JList (seznam) má definovaný model pomocí rozhraní ListModel. Protože rozhraní je definované pro danou komponentu, ListModel poskytuje metody pouze pro čtení dat (to je vše, co JList potřebuje k vykreslení prvků). Defaultní implementace ListModelu je DefaultListModel, který přidávání a odebírání prků ze seznamu již podporuje. Obecně ListModel představuje jednorozměrné pole prvků, ale programátor může dané rozhraní implementovat podle svých potřeb. Model dále nabízí rozhraní pro komunikaci se zbytkem programu pomocí vyvolávání událostí. Ve zkratce, komponenta nabízí, že si k ní kdokoliv může přihlásit posluchače (tzv. listener) na různé události. Listener je pro danou komponentu definován obvykle jako nějaké rozhraní. Když v komponentě proběhne daná událost (ať už ji vyvolal kdokoliv) – např. zmáčknutí tlačítka, vybrání položky, minimalizování okna, atd., komponenta všem přihlášeným posluchačům odešle zprávu o provedené akci. Tato technologie přihlašování a odesílání zpráv je založena na

návrhovém vzoru Observer. Např. tlačítko vyvolává jedinou událost – obecnou akci, která nastane při zmáčknutí tlačítka. Kdokoliv se může stát posluchačem mačkání tlačítka tím, že implementuje rozhraní java.awt.event.ActionListener. Po zaregistrování posluchače k tlačítku bude daný objekt dostávat zprávy o mačkání tlačítka prostřednictvím metod definovaných v rozhraní ActionListener. Jiné komponenty mohou definovat více událostí – např. strom (JTree) vyvolává události při rozbalení položky a při vybrání položky. Dále obecně všechny komponenty (protože jsou potomky různých tříd v komponentní hierarchii), definují další události – při změně vlastníka, při změně vlastnosti (property), při změně kontejneru, při klepnutí myší, při psaní textu, při změně zaměření (focus), atd…“ [11]

3.4 Těžké a lehké komponenty

„AWT bylo koncipováno jako rozhraní mezi Javou a grafickým API platformy. Všechny grafické komponenty byly přímo kresleny systémem a měly tedy nativní vzezření. Např. JVM na Windows využívala Win32 API (konkrétně knihovny gdi32.dll a další) ke kreslení AWT komponent. Tyto nativně vykreslené prvky se nazývají „těžké“ (v originále heavyweight), protože měly své vlastní nativní neprůhledné okno v systému. Na systém se spoléhaly i v dalších věcech, např. zajišťování pozice na ose Z (která komponenta je v popředí a která v pozadí). Swing přináší jiný přístup, a to ten, že každá komponenta je sama zodpovědná za svůj vlastní vzhled a nespoléhá se na systém. Každá komponenta sama definuje, jak vypadá, a na požádání se namaluje na monitor. Samotné kreslení se odehrává přímo v Javě, a operačnímu systému se pouze předá

hotový obrázek k namalování. Tomuto přístupu se říká „lehký“ (lightweight). To, jak komponenty budou vypadat, již tedy nezáleží na operačním systému. Swing navíc přidává podporu pro dvojitou vyrovnávací paměť (double buffering), která umožňuje plynulé kreslení grafických prvků; pro podporu průhlednosti a částečné průhlednosti; pro optimalizované kreslení v případě překrývajících se komponent; a další technologie. Pro kreslení využívá další část JFC, a to Java 2D API. Kreslení nejvyšších kontejnerů (oken, dialogů…) je ve Swingu převzato z AWT, tudíž okna jako taková jsou platformě závislá (nelze mít okno na platformě, která okna nepodporuje). Jejich vzhled ale konfigurovatelný je. “ [11]

Je však možné okno, tzv. „oddekorovat“ neboli skrýt jeho rám a zobrazit pouze obsah, kterému vytvoříme vlastní rám podle svých představ a vytvořit tak okno, které bude vypadat celé stejně na různých operačních systémech.

3.5 Proces malování

Proces malování se skládá z kroků:

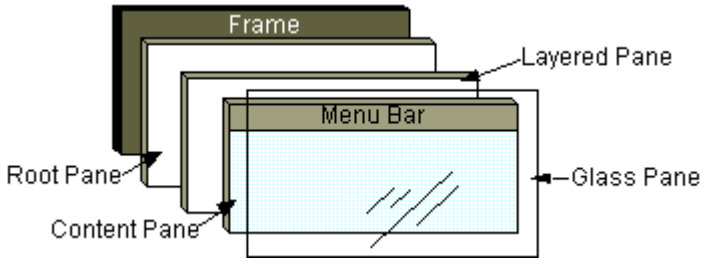
- EDT[1] volá metodu paint() předka
- Výchozí implementace Container.paint() rekurzivně volá metodu paint() na „lightweight“ potomka
- voláním metody JComponent.paint() se provede:

1 EDT je zkratka pro Event Dispatching Thread, což je vlákno, které běží na pozadí. Toto vlákno zpracovává události z AWT. [13]

- pokud je povolen „double-buffering", bude se vykreslovat pouze do paměťového bufferu a ne rovnou na obrazovku.
- volání metody **paintComponent()**[2]
- volání metody **paintBorder()**[3]
- volání metody **paintChildren()**[4]
- pokud je povolen „double-buffering", je překreslen obsah „obrázkového bufferu" na obrazovku. V případě, že nebyl povolen, tak metody paintComponent(), paintBorder() a paintChildren() vykreslují rovnou na obrazovku.

Poznámka: V programech by nemělo být volána přímo metoda paint(), nýbrž metoda repaint()![14]

3.5.1 Plátna



Ilustrace 4: plátna, převzato z [21]

2 *Volá metodu UI delegáta, v případě, že je delegát nenulový. [15]*

3 *Vykresluje rámeček komponenty.*

4 *Vymalovává potomky, které komponenta vlastní.*

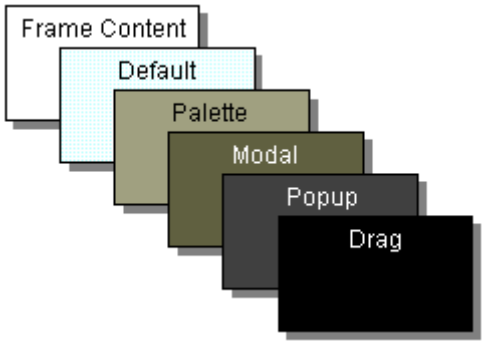
Jak ukazuje obrázek 4, tak je zřejmé, že každý **top-level kontejner** (například javax.swing.JFrame, javax.swing.JDialog, javax.swing.JApplet) může obsahovat až čtyři svá plátna. [21]

3.5.1.1 Glass Pane

Glass pane neboli „skleněné plátno“ je defaultně skryto. Pokud je však nějakému oknu nastaveno nějaké konkrétní glass pane pomocí veřejné instanční metody setGlassPane(javax.swing.JComponent component) nějaké třídy top-level kontejneru (například javax.swing.JFRame), tak se vykresluje i obsah, který vykresluje (paint()) komponenta předaná v parametru metody. Toto plátno má tu vlastnost, že je vykreslováno nejvýše nad všemi ostatními plátny.[22]

3.5.1.2 Layered Pane

Layered Pane neboli „vrstvené plátno“ je kontejner, kterému můžeme říct do jaké vrstvy se má přidat ta která komponenta. Sdílejí-li dvě



Ilustrace 5: vrstvy vrstveného plátna, převzato z [23]

komponenty jistou část obrazovky (pixel/y), a zároveň jsou vložené do různých vrstev, tak na sdílené části se vykreslí komponenta, která je ve

vyšší vrstvě.[23] Níže uvedená tabulka ukazuje do jaké vrstvy se konkrétní komponenty standardně vkládají.

Název vrstvy	**Hodnota**	**Popisek**
FRAME_CONTENT_LAYER	new Integer(-30000)	Do této hloubky vkládá root pane své content pane a menu bar
DEFAULT_LAYER	new Integer(0)	Není-li specifikována vrstva do které má být přidána komponenta, je vložena defaultně do nulté vrstvy.
PALETTE_LAYER	new Integer(100)	Do sté vrstvy jsou přidávány komponenty typu tool bar a palety
MODAL_LAYER	new Integer(200)	Do této vrstvy se vkládají vnitřní okna (internal-frame)
POPUP_LAYER	new Integer(300)	Do této vrstvy se vkládají vyskakovací dialogy.
DRAG_LAYER	new Integer(400)	Do této vrstvy jsou vkládány komponenty, jsou-li taženy myší.

Tabulka 1: používané vrstvy [23]

3.6 Swing a vlákna

Swing není vláknově bezpečný, může způsobovat problémy při přistupování ke svým komponentám. **Příklad:** Pokud jedno vlákno V1 v špatně napsaném programu přistupuje ke komponentě A, která je dejme tomu datového typu javax.swing.JList o 4 prvcích, a zároveň ke komponentě A přistupuje ještě jiné vlákno V2 či dokonce ještě další vlákna, může nastat problém. Například v modelové situaci: V1 přistoupí k seznamu A a odebere z něj prvek s indexem 3. V zápětí na to vlákno V2 pracuje také se seznamem. Shodou okolností chce získat objekt, který „je“ v seznamu na 3. indexu. Ten tam samozřejmě není, neboť byl již odebrán. Nezdar ve špatném výběru indexu nám potvrdí vyhozená vyjímka java.lang.ArrayIndexOutOfBoundsException. Samozřejmě takových vyjímek může být mnoho, ale mohou být samozřejmě i jiného typu při špatném použití Swingu. [11]

3.6.1 Časově náročné úlohy a Swing

Při používání vláken a Swingu zároveň používejte 2 pravidla:

- neswingová akce, která zabírá mnoho času (například I/O operace) by měla být spouštěna ve vlastním vlákně, nikdy v EDT
- ke swingové komponentě by mělo být přistupováno pouze z vlákna EDT

Při nedodržení prvního pravidla se aplikace tváří, že se zasekla, leč pracuje. Druhé pravidlo se nazývá „jednovláknové pravidlo“. [16]

3.6.1.1 Jednovláknové pravidlo

Každá javovská aplikace začíná hlavní metodou, která běží v hlavním vlákně. V swingovských aplikacích žije hlavní vlákno krátce. Naplánuje konstrukci uživatelského grafického rozhraní a ukončí se. Po vytvoření grafického uživatelského rozhraní vlákno EDT zpracováná události, které jsou volány nad komponentami. Jiná vlákna, jako například vlákna, která posílají vyvolané události do fronty událostí, běží za scénou a jsou neviditelná pro běžné programátory. [16]

3.6.1.2 Vyjímky jednovláknového pravidla

- můžete bezpečně přidávat a odebírat posluchače událostí.
- Malé množství swingovských metod je vláknově bezpečné. V případě, že jsou vláknově bezpečné, oficiální dokumentace API na tento jev upozorňuje. A jsou to například metody:
  - JTextComponent.setText
  - JTextArea.insert
  - JTextArea.append
  - JTextArea.replaceRange
  - JComponent.repaint
  - JComponent.revalidate [16]

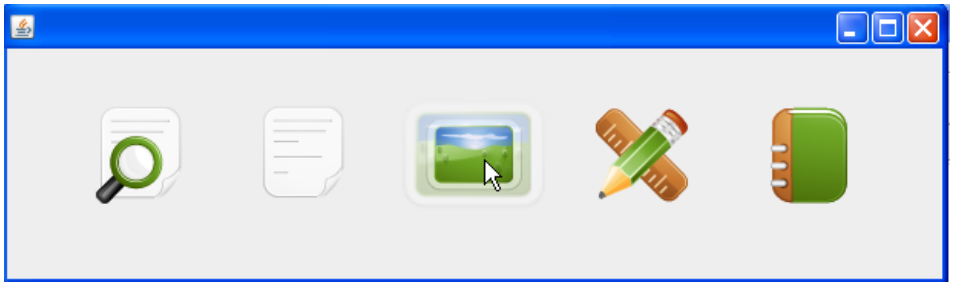
4 Příklady

V níže uvedené části práce jsou uvedeny výukové projekty, které vznikly jako podpůrné příklady pro vyučování předmětu PGJ2 a PGJ3 na pedagogické fakultě JČU. V těchto předmětech, které jsem oba úspěšně absolvoval se výuka zabývala obtížnějšími zákoutími J2SE, mimo jiné také javovským GUI, což je oblast velice obsáhlá a různorodá. Z tohoto pohledu soudím, že ukázkových příkladů může být vždy více, a že mohou přinést něco nového, v jiných projektech neřešeného, nebo zde řešeného jiným způsobem. Proto se domnívám, že níže uvedená praktická část mé bakalářské práce může být přínosná studentům a ostatním zájemcům tvorby grafických rozhraní v desktopových javovských aplikacích.

4.1 Jednoduchá animace

Tento projekt ukazuje způsob jakým vytvořit jednoduchou animaci obrázku, která může být použita například při najetí na vytvořené tlačítko. Tato animace spočívá v tom, že se při každém vykreslování vykreslí původní obrázek a nad ním jeho kopie, která je zprůhledněna pomocí nastavené alfa kompozice. Tato kopie je zvětšena a postupně mizí. Tento způsob implementace je samozřejmě poměrně náročný na čas procesoru, ale poměrně nenáročný na využití operační paměti. Samozřejmě je možné implementovat jinak algoritmus, aby byl méně využíván procesor a více operační paměť, ale to si může každý naprogramovat, dle svého uvážení. Tento projekt dále také ukazuje způsob jakým vytvořit vlastní lightweight komponentu.

4.1.1 Ukázka aplikace



Ilustrace 6: Výstup z aplikace „Jednoduchá animace“

4.1.2 Popis balíčku jednoduchaAnimace

Tento balíček se skládá pouze ze dvou tříd:

- Komponenta

- Okno

Třída **Okno** dědí od javax.swing.JFrame a slouží pouze jako místo kam se přidává panel s tlačítky. Rovněž tato třída obsahuje hlavní metodu v které se vytváří pouze nová instance okna a toto okno se zviditelní. V rámci tohoto projektu je zajímavější třída **Komponenta**, která dědí od třídy javax.swing.JComponent a má překryté metody nadtřídy:

- protected void paintComponent(Graphics g)
- public Dimension getPreferredSize()

V této třídě je aplikovaný celý algoritmus animace (zvětšování a zprůhledňování obrázku i vykreslování).

4.1.3 Algoritmus vytváření komponenty

Hlavní myšlenka tohoto projektu vyžaduje následující kroky:

- vytvořit třídu která dědí od javax.swing.JComponent (v projektu třída Komponenta)
- překrýt metodu public Dimension **getPreferredSize**() a nastavit tak velikost s kterou bude přidána komponenta na nějaké plátno.
- překrýt metodu protected void **paintComponent**(Graphics g), pomocí které bude komponenta vykreslována.
- vytvořit metodu, která bude v cyklu **zvětšovat obrázek** a postupně ho **zprůhledňovat** až do úplného zmizení. Tato metoda se v projektu jmenuje **kresli**(long cas)
- volat metodu kresli v samostatném vlákně.

4.1.4 Části zdrojového kódu

```
@Override
```

```
public Dimension getPreferredSize()
{
  return new Dimension(getObrazek().getWidth() + kroku,
                       getObrazek().getHeight() + kroku);
}
```

Komentář: Výše uvedená metoda **překrývá** metodu Dimension getPreferredSize() třídy javax.swing.JComponent. Tato metoda z vytvořené třídy **Komponenta** vrací rozměr, který je automaticky nastaven komponentě, když se přidá k jiné komponentě. Dále také určuje, jak velký bude obdelník, který bude překreslován voláním například metody repaint() bez parametrů na tuto komponentu.

```
@Override
protected void paintComponent(Graphics g) {
  Graphics2D g2 = (Graphics2D) g.create();
  g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
            RenderingHints.VALUE_INTERPOLATION_BILINEAR);
  g2.drawImage(getObrazek(), x, y, getObrazek().getWidth(),
                          getObrazek().getHeight(), null);
  g2.setComposite(AlphaComposite.SrcOver.derive(1.0f -
                                                getAlfa()));
  g2.drawImage(getObrazek(), x - getZoom() / 2, y -
        getZoom() / 2, getObrazek().getWidth() + getZoom(),
               getObrazek().getHeight() + getZoom(), null);
}
```

Komentář: Metoda paintComponent**(**Graphics g**)** **překrývá** metodu void paintComponent(Graphics g) třídy javax.swing.JComponent. Tato metoda z vytvořené třídy **Komponenta** je automaticky volána po zavolání metody repaint(). Tato metoda má všeobecně na starosti vykreslení komponenty. V tomto případě to znamená: vykreslit originální obrázek, nastavit alfa kompozici a vykreslit přes originální obrázek zprůhledněnou kopii obrázku, která je větší než originál a je rovněž průhlednější.

```
private void kresli(long cas)
{
    float inkrement = (1f / kroku);
    moznoVykreslit = false;
```

```
    int smycka = 0;
    while (smycka < kroku)
    {
        smycka++;
        repaint();
        setZoom(zoom + 1);
        alfa = inkrement * smycka;
        try
        {
            Thread.sleep(cas);

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
     }
     repaint();
     zoom = 0;
     alfa = 0;
     moznoVykreslit = true;
}
```

Komentář: Soukromá metoda kresli(long cas) ze třídy **Komponenta** přijímá jako parametr číslo, které určuje vláknu, jak dlouho má spát v milisekundách. Hlavní myšlenka této metody je vytvořit metodu, která bude volána ve vlákně, které bude uspáváno dle parametru "cas". Cyklus je řízen logickou relací smycka < kroku, kde "smycka" i "kroku" jsou číselné instanční proměnné. V těle cyklu se inkrementuje číselná instanční proměnná "zoom", která určuje faktor zvětšení poloprůhledného obrázku vůči originálnímu obrázku. Dále se nastavuje aktuální alfa. Rovněž se překresluje instance třídy Komponenta voláním metody repaint().

```
public void vykresli(final long cas)
{
     Runnable run = new Runnable()
     {
          public void run() {
              kresli(cas);
          }
     };
     new Thread(run).start();
```

```
}
```

Komentář: Veřejná metoda ze třídy Komponenta vykresli(final long cas) pouze vytvoří a spustí nové vlákno, ve kterém je volána výše popsaná privátní metoda kresli(long cas).

4.2 Obrázkové filtry

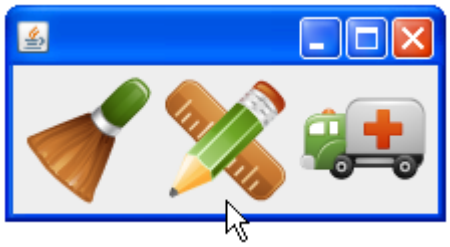
Na tomto příkladě (inspirace zdrojem [26]) je ukázáno, jak lehce jde pracovat v Javě s obrázkovými filtry. Konkrétně je zde ukázka, která ukazuje jakým způsobem prohodit barvy – například bílá bude černá, atd.. Příklad ukazuje jak vytvořit inverzní barvy. Samozřejmě je možné prohodit třeba bílou za červenou či provést jiné operace s barvami, ale to je ovšem na každém programátorovi, čeho chce docílit. Dále je zde ukázka konvolučního[5] filtru, který rozmazává obrázek. Aby aplikování filtrů na obrázek bylo působivější, jsou nové filtry vytvářeny a aplikovány v cyklu, který je volán v samostatném vlákně. Při každém průchodu cyklem se vytváří filtr s jinými parametry, kde je potom možné vidět, jak se obrázek na kterém aplikujeme filtr postupně rozmazává víc a víc, aplikujeme-li rozmazávací konvoluční filtr.

4.2.1 Zjištění

Při lazení tohoto projektu jsem zjistil, že grafický výstup při debugování v IDE (NetBeans IDE 6.7.1) se nemusí rovnat grafickému výstupu spustitelného jar souboru, který je generován pomocí IDE (viz. projekt v příloze). Bohužel, nebyl jsem schopen zjistit v čem je problém.

5 *Konvoluce je algoritmus, který počítá výsledné pixely jako vážené součty pixelů v jejich okolí. Toto provádí pro každou barevnou složku pixelu zvlášť. [11]*

4.2.2 Ukázka



Ilustrace 7: Výstup z aplikace „Obrázkové filtry“

Ilustrace 8: Výstup z aplikace „Obrázkové filtry“ (kliknutí na popisek)

Ilustrace 9: Výstup z aplikace „Obrázkové filtry“ (najetí na popisek).

4.2.3 Popis tříd

Tento projekt se skládá pouze ze 2 veřejných tříd, jelikož se jedná o poměrně jednoduchý příklad. Konkrétně jsou to třídy:

- AnimovanyPopisek

- Okno

Navíc třída AnimovanyPopisek má v sobě vložené další dvě své privátní třídy:

- **ZachytavacMysi**
- **ZachytavacPohybuMysi**

Třída ZachytavacPohybuMysi dědí od třídy java.awt.event.MouseMotionAdapter a třída ZachytavacMysi je podtřídou java.awt.event.MouseAdapter. Obě tyto třídy mají společné to, že slouží jako handlery událostí myši.

Nejzajímavější třída z projektu je však třída **AnimovanyPopisek** ve které se nachází implementovaný algoritmus filtrování obrázků. Tato třída dědí od javax.swing.JLabel, aby bylo jednoduché demonstrovat funkcionalitu filtrů obrázků, pomocí nastavování ikony popisku. Samozřejmě je možné vytvořit například obdobu tohoto příkladu, kde třída dědí od javax.swing.JComponent a má překrytou metodu paintComponent(Graphics g). Ale to by bylo obtížnější realizovat, proto jsem zvolil tento postup, aby více kódu nemátlo čtenáře, neboť dědění od třídy javax.swing.JComponent a překrytí potřebných metod je ukázáno v jiných příkladech. Třída **Okno** slouží jako ukázkové okno, kam se přidávají popisky pro demonstrovatelnost projektu.

4.2.4 Algoritmus

Hlavní myšlenka tohoto projektu vyžadovala následující kroky:

- **vytvořit** instanci třídy **AnimovanyPopisek** (třída má jediný konstruktor s jedním parametrem, který udává cestu k obrázku, který musí být připojen k projektu jako zdroj).

- tomuto obrázku nastavit booleovskou instanční proměnnou pomocí metody **setRozmazano(boolean)**, pokud jej chceme rozmazat a pomocí metody **setI(int)** nastavit instanční proměnnou i, která ovlivní matici konvolučního filtru (viz. zdrojový kód).
- případně použít inverzní filtr nastavením instanční proměnné **rozmazano** pomocí metody **setProhozeneBarvy(boolean)**, pokud je požadováno použití inverzního filtru
- zavolat metodu **aplikujFiltr()**, která vrátí filtrovaný obrázek v závislosti na instančních proměnných, které určují zda rozmazat obrázek či jej invertovat.
- v případě, že je požadováno vykreslování animované ikony popisku, volat metodu **blikej()** v samostatném vlákně.
- přidat instance třídy AnimovanyPopisek do nějakého kontejneru, například k instanci třídy **Okno**
- zviditelnit okno, kam byl přidán popisek

4.2.5 Fragmenty zdrojového kódu

```
public void blikej()
{
    while (i < 9)
    {
       try
        {
             BufferedImage filtrovany = aplikujFiltr();
             setIcon(new ImageIcon(filtrovany));
             Thread.sleep((int) (1000 / (i * 1.5)));
        }
        catch (Exception ee) {
                    ee.printStackTrace();
```

```
        }
        i++;
     }
     i = pocatek;

}
```

Komentář: Veřejná metoda blikej() ze třídy AnimovanyPopisek v cyklu pouze aplikuje filtr na obrázek, poté nastavuje pozadí popisku, které je tvořeno obrázkem, který vyšel po filtrování. Dále se uspí vlákno, po každém průchodu cyklem o kratší dobu.

```
public BufferedImage aplikujFiltr()
{
  BufferedImage pom = obrazek;
  if(isProhozeneBarvy())
  {
    byte pole[] = new byte[256];
    for (int j = 0; j < 256; j++)
    {
        pole[j] = (byte)(255-j);
    }
    ByteLookupTable tabulka = new ByteLookupTable(0,pole);
    LookupOp operace = new LookupOp(tabulka,null);
    pom = operace.filter(pom, null);
  }

  if(isRozmazano())
  {
      float matice[] = {
            1f/i,1f/i,1f/i,
            1f/i,1f/i,1f/i,
            1f/i,1f/i,1f/i};

      ConvolveOp operace;
      Kernel jadro = new Kernel(3,3,matice);
           operace = new ConvolveOp(jadro,
                          ConvolveOp.EDGE_ZERO_FILL,null);

               pom = operace.filter(pom, null);
   }
   return pom;
}
```

Komentář: Metoda aplikujFiltr() vrací filtrovaný obrázek z originálního obrázku, který je uložen v instanční proměnné „obrazek“. Druh filtrování je určen booleovskými instančními proměnnými „rozmazano“ a „prohozeneBarvy“. V závislosti na těchto proměnných se provádí či neprovádí bloky if. V prvním bloku, je-li nastavavena proměnná „prohozeneBarvy“ na true, provede se prohození barev. V druhém bloku se provede konvoluce. Matice, která je postupně přikládána zleva doprava a shora dolů na obrázek určuje objekt třídy Kernel.

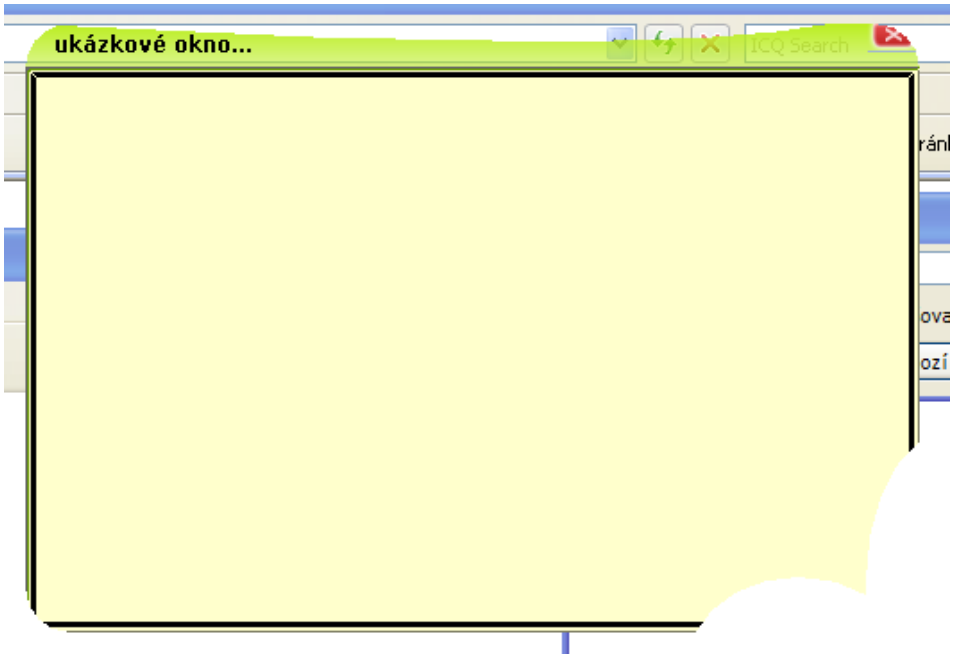
4.3 Průhledné okno libovolného tvaru

Tento příklad popisuje způsob, jak vykreslovat vlastní okna, která mají libovolný tvar a mohou být také průhledná či poloprůhledná. Tento příklad si žádá pro spuštění verzi JRE 6u10 či vyšší. Ve starších verzích nebude plně fungovat. Jelikož tato funkcionalita nebyla dříve v Javě podporovaná, muselo se jít jinou cestou, aby se získal výsledek, který je v tomto příkladě poměrně jednoduchý. Jeden ze způsobů, který mne napadá se skládá z těchto kroků:

- vytvořit okno, kam bude kresleno
- „oddekorovat okno“
- sejmout obrazovku, kde bude vykreslováno okno
- vykreslit na okno sejmutou obrazovku
- na okno vykreslit libovonou grafiku falešného okna
- zobrazit okno

Tento algoritmus je poměrně těžkopádný a neřeší ani klikání myší mimo naše okno, které jsme vykreslili na panel opravdového okna. Při klikání vedle našeho speciálního okna, se nám může stát, že se bude okno tvářit jakoby jsme klikali například na ikonu plochy, která se zdá být vedle našeho falešného okna. Ale nic se neděje. Proč? Pravděpodobně klikáme pouze na pozadí skutečného okna, které vypadá jako plocha pod ním. Následující příklad (inspirace zdrojem [24]) ukazuje jak ladně tyto nepříjemné jevy řešit.

4.3.1 Ukázkový výstup aplikace



Ilustrace 10: Výstup z aplikace „Tvarované okno libovolného tvaru“

4.3.2 Popis tříd

Tato aplikace se skládá z 5 tříd. Třída **UkazkovyPanel** je pouze předdefinovaný panel, který dědí od třídy javax.swing.JPanel a v aplikaci slouží jako místo s obsahem okna. Třída **Titulek** supluje skrytý „oddekorovaný” titulek klasického swingovského okna. Třída **VykreslovacOkna** pracuje v rámci reflexe a nastavuje průhlednost a tvar určitému oknu. Třída **PruhledneOknoLibovolnehoTvaru** je třída, která dědí od javax.swing.JFrame a reprezentuje naše okno, kterému je aplikovaný určitý tvar a průhlednost. Nakonec třída **Main** obsahuje metodu main a slouží tudíž jako vstupní bod programu.

4.3.3 Algoritmus

Zprovoznit tento projektu vyžadovalo následující kroky:

- **Vytvořit tvar okna** jaký chceme, aby mělo výsledné okno (tento tvar musí implementovat rozhraní **java.awt.Shape**).
- **Vytvořit titulek**, který bude fungovat obdobně jako titulek nativního či swingovského okna. V tomto příkladu to znamená vytvoření třídy **Titulek**, která **dědí** od třídy **javax.swing.JComponent,** následně **překrýt** metodu public void **paintComponent(Graphics g)**, která se stará o vykreslování komponenty.
- Přidat titulku posluchač, který reaguje na události myši a v tomto duchu se stará o přesouvání okna.
- Vytvořit okno, které dědí od javax.swing.JFrame. Zde je to třída **PruhledneOknoLibovolnehoTvaru**.
- Oddekorovat okno
- Přidat k oddekorovanému oknu titulek a obsah okna (třída UkazkovyPanel).
- Nastavit patřičně statické metody třídy **com.sun.awt.AWTUtilities**. Konkrétně metody:
    1. setWindowShape(Window, Shape)
    2. setWindowOpacity(Window, float)
    3. setWindowOpaque(Window, boolean)

V tomto příkladě jsou zmiňované metody volány ve statických metodách třídy VykreslovacOkna:

- Zviditelnit okno.

4.3.4 Popis zdrojového kódu

```
package pruhledneokno;

import com.sun.awt.AWTUtilities;
import java.awt.Shape;
import java.awt.Window;

public class VykreslovacOkna
{
  public static void nastavTvarOkna(Window okno,Shape tvar)
  {
         AWTUtilities.setWindowShape(okno, tvar);
  }

  public static void nastavPruhlednostOkna(Window okno,
                                           float pruhlednost)
  {
        AWTUtilities.setWindowOpacity(okno, pruhlednost);
  }


  public static void nastavNepruhlednostOkna(Window okno,
                                         boolean nepruhledne)
  {
        AWTUtilities.setWindowOpaque(okno, nepruhledne);
  }
}
```

Komentář: Výše uvedené veřejné statické metody mají jedno společné - volají statické metody třídy com.sun.awt.AWTUtilities, které ovlivňují jak se vykreslí okno. Metoda *setWindowShape* nastaví jaký tvar okna se vykreslí z originálního okna. Dále metoda *setWindowOpacity* určuje, zda bude celé okno neprůhledné, poloprůhlené či plně průhledné. Hodnota proměnné „pruhlednost“, proto může nabývat hodnot intervalu (0; 1), kde 1 = plná

neprůhlednost, 0.5 poloviční průhlenost celého okna. **Doporučení:** Zkuste nastavit neprůhlednost například na 0.3f, poté 0.8f a nakonec na 1.0f a sledujte, co se stane.

```
public class Titulek extends JComponent
{

    private String title;
    private JFrame vlastnik;


    public Titulek(String textTitulku, JFrame vlastnik) {
        this.vlastnik = vlastnik;
        this.title = textTitulku;
        inicializujKomponenty();
        AdapterMysi posluchacMysi = new AdapterMysi();
        addMouseListener(posluchacMysi);
        addMouseMotionListener(posluchacMysi);
    }

    @Override
    protected void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        Paint oldPaint = g2.getPaint();
        float rgb[] = new Color(188,239,34).
                        getRGBColorComponents(null);

        g2.setPaint(new GradientPaint(0f, 0f,
                        new Color(rgb[0], rgb[1],
                        rgb[2], 1f), 0.0f,
                        getHeight(),
                        new Color(rgb[0],rgb[1],rgb[2],
                        0.5f)));

        g2.fillRect(0, 0, getWidth(), getHeight());
        vykresliText(g2, 14);
        g2.setPaint(oldPaint);

    }
```

Komentář: Výše uvedený výřez zdrojového kódu třídy Titulek říká, že třída Titulek je podtřídou třídy javax.swing.JComponent a

překrývá ve svém těle metodu paintComponent(Graphics g). V této komponentě se vykresluje pouze obdélník, který má nastaven texturu, která se počítá jako barevný přechod určený 2 barvami. Na tomto přechodu je zajímavé to, že je poloprůhledný. Nakonec se přes texturu vykreslí titulek okna.

```
private class AdapterMysi extends MouseAdapter
{
   private Point aktualniPolohaKurzoru;

   @Override
   public void mousePressed(MouseEvent e)
   {
      aktualniPolohaKurzoru = e.getPoint();
      SwingUtilities.convertPointToScreen(
      aktualniPolohaKurzoru=(Component)e.getSource());

      Titulek.this.getParent().setCursor
      (Cursor.getPredefinedCursor(Cursor.MOVE_CURSOR));
   }


   @Override
   public void mouseReleased(MouseEvent e)
   {
       Titulek.this.getParent().setCursor(
                     Cursor.getDefaultCursor());
   }

   @Override
   public void mouseDragged(MouseEvent e)
   {
     Point poloha = e.getPoint();
     SwingUtilities.convertPointToScreen(poloha,
                            (Component) e.getSource());

     int vzdalenost_x=poloha.x - aktualniPolohaKurzoru.x;
     int vzdalenost_y = poloha.y-aktualniPolohaKurzoru.y;

     Point vzdalenostVlastnika = vlastnik.getLocation();
     vzdalenostVlastnika.x += vzdalenost_x;
     vzdalenostVlastnika.y += vzdalenost_y;

     vlastnik.setLocation(vzdalenostVlastnika);
     aktualniPolohaKurzoru = poloha;
```

```
        }

    }
}
```

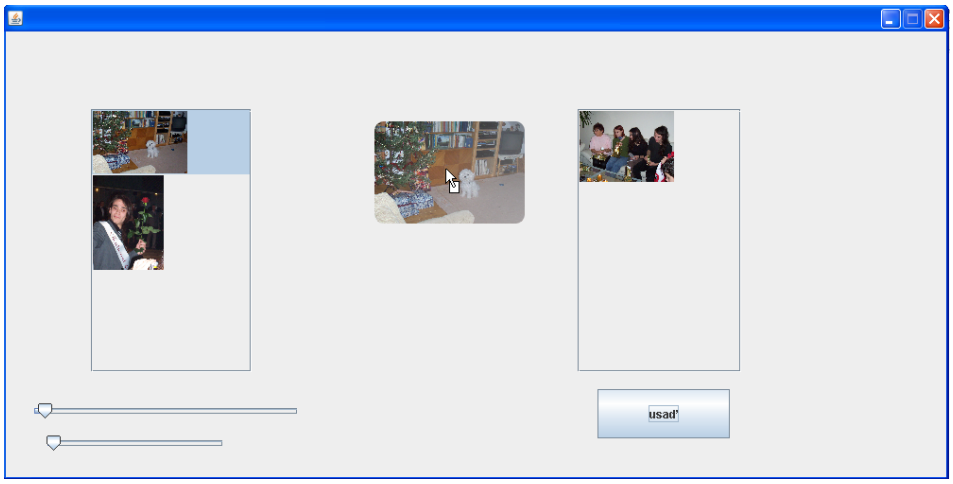
Komentář: Třída AdapterMysi dědí od třídy java.awt.event.MouseAdapter, která reaguje na události vyvolávané myší. Nejzajímavější ze 3 implementovaných metod jsou metody mousePressed(MouseEvent e) a mouseDragged(MouseEvent e). V překryté metodě mousePressed(MouseEvent e) se 1. příkazem **inicializuje** instanční proměnná „aktualniPolohaKurzoru“ **bodem**, kde byla vznikla událost. V druhém příkaze se převede „aktualniPolohaKurzoru“ **na globální souřadnice** obrazovky, neboť před převedením obsahuje pouze informaci o bodě, kde byla událost volaná v rámci komponenty. Metoda mouseDragged(MouseEvent e) je volána v případě, že je stisknuté tlačítko a zároveň se hýbe myší. V těle této metody je zajištěno, že se inicializuje „aktualniPolohaKurzoru“ a zároveň se přesouvá instance okna, která je uložena v instanční proměnné „vlastnik“ (instance javax.swing.JFrame).

4.4 Drag and Drop – obrázková animace

Na tomto příkladě je demonstrováno jakým způsobem pracovat s drag and drop, jsou-li přetahovány obrázky. Aby byl projekt zajímavější, přetahované obrázky jsou při přetahování animovány. Intervaly vykreslování jednotlivých obrázků animace má na starosti swingovský časovač, který je obsažen ve třídě javax.swing.Timer. Z důvodu rychlejšího vykreslování jednotlivých obrázků animace, jsou vytvořeny pouze v jednom kroku běhu programu náhledy celkového obrázku. Tento způsob sice klade vyšší nároky na operační paměť, ale minimální požadavky na čas procesoru, neboť procesor nemusí při každé změně obrázku animace vypočítávat znovu a znovu náhled velkého obrázku. Z tohoto důvodu jsou předpočítané obrázky uloženy v jednom seznamu a když přijde čas prohodit obrázek, vytáhne se ze seznamu nový obrázek, což jsem shledal jako nejrozumnější řešení. Tyto jednotlivé náhledy jsou vytvářeny pomocí alfa kompozice, tudíž výsledná animace je složena z poloprůhledných obrázků. Aby bylo ukázáno i jak fungují drop operace, obsahuje okno projektu 2 seznamy typu javax.swing.JList, kterým jsou povoleny pokládací operace a rovněž jsou jim implementovány transferové handlery. Do těchto seznamů je možné přetahovat soubor s obrázkem z plochy operačního systému, dokumentů či jiného místa OS. Také je možné přetahovat obrázky mezi seznamy samotnými. Tento projekt také ukazuje jak jednoduchým způsobem je možné vytvořit například GUI pro nějaký program, který by měl za úkol uploadovat obrázky například do webové galerie, jednoduchým přetažením z jednoho do druhého seznamu. V čase kdy je uploadován soubor z jednoho seznamu (obsah lokální složky) do druhého seznamu

(reprezentující cílovou složku na FTP serveru) se posouvá animovaný obrázek směrem ze zdrojového do cílového seznamu. V tomto projektu je možné rovněž měnit pomocí posuvníku rychlost přesunu animovaného souboru, čehož je docíleno jednoduchým způsobem, obyčejným posunutím o větší vzdálenost. V reálném nasazení by nejspíš nebyl žádný posuvník měnící rychlost posunu obrázku po ploše, ale rychlost přesunu by mohla být odvozena od uploadovací, případně downloadovací rychlosti na server či ze serveru. Rovněž by reálná aplikace neobsahovala projektové tlačítko „usaď“, které při stisknutí ukončí animování a usadí obrázek do cílového seznamu. Tento mechanismus by mohl být samozřejmě nahrazen v reálné aplikaci dejme tomu při úspěšném uploadu, voláním kódu, který je volán nyní při stisku tlačítka „usaď“.

4.4.1 Ukázka aplikace



Ilustrace 11: Výstup z aplikace „Drag and Drop – obrázková animace“

4.4.2 Popis balíčků a tříd

Tento projekt obsahuje:

- třídu Main
- balíček glassPaneDragAndDrop

Třída Main obsahuje hlavní metodu ve které se pouze vytváří instance třídy glassPaneDragAndDrop.Okno, která pouze slouží jako místo, kam „nalepit“ panel se seznamy, tlačítky a posuvníky. Dále projekt obsahuje balíček „glassPaneDragAndDrop“.

4.4.2.1 Balíček glassPaneDragAndDrop

Tento balíček je složen pouze z 5 tříd:

- AnimaceGlassPane
- ObrazkoveOperace

- Okno
- Polozka
- ZachytavacPrenosu

Třída **ObrazkoveOperace** obsahuje pouze statické metody, pomocí nichž se nahrávají externí obrázky a z velkých obrázků se vytvářejí náhledy. Třída **Okno** dědí od javax.swing.JFrame a přidává se do něj panel se seznamy (javax.swing.JList), tlačítko (javax.swing.JButton) a posuvníky (javax.swing.JSlider). Dále se také v této třídě nastavuje skleněné plátno, které je tvořeno instancí třídy AnimaceGlassPane. Třída **ZachytavacPrenosu** dědí od javax.swing.TransferHandler a slouží v projektu pro nastavení handlerů seznamů. Třída **Polozka** reprezentuje položky seznamů (javax.swing.JList). Tato třída dědí od javax.swing.ImageIcon a implementuje rozhraní java.awt.datatransfer.Transferable, z důvodu aby bylo možné přetahovat obrázky mezi seznamy. Třída **AnimaceGlassPane** dědí od javax.swing.JComponent a stará se o vykreslování animovaného obrázku při hýbání myší, či při přesouvání mezi seznamy. Také má v sobě nainstalován časovač, který se stará o měnění obrázků v zadaném intervalu (po určitém intervalu se vytáhne jiný obrázek).

4.4.3 Algoritmus

Myšlenka tohoto projektu vyžadovala následující kroky:

- **vytvořit** instanci třídy **Okno** dědící od javax.swing.JFrame

- vytvořit třídu, která může být použitá jako **glassPane** (musí dědit od javax.swing.JComponent). V projektu je to třída **AnimaceGlassPane** z balíčku glassPaneDragAndDrop.
- ve třídě AnimaceGlassPane nastavit, aby se po určité době změnil aktuální obrázek na jiný. Zde je toho docíleno pomocí časovače třídy **javax.swing.Timer**, který v určitém intervalu volá metodu **nastavNovyObrazek()**.
- ve třídě AnimaceGlassPane **nastavit, kde má být vykreslen** aktuální **obrázek**. Zde k tomu slouží metoda nastavAktualniPozici(java.awt.Point location)
- dále je nutné **překrýt** ve třídě AnimaceGlassPane metodu **paintComponent(java.awt.Graphics g)**, která se stará o vykreslování komponenty
- obdobným způsobem jako ve třech výše uvedených bodech, **zajistit stav**, **kdy není tažen** žádný **obrázek**, ale běží animace přenášení obrázku z jednoho do druhého seznamu.
- **vytvořit seznamy** (javax.swing.JList), jejichž položky (instance třídy Polozka) budou reagovat na drag operace. Třída **Polozka** v projektu je třída, která dědí od javax.swing.ImageIcon a implementuje rozhraní java.awt.datatransfer.Transferable.
- Vytvořit **přenášecí handlery**, které budou kontrolovat drop operace seznamů. Tuto funkčnost v projektu zajišťuje třída TransferZachytavac rozšiřující třídu javax.swing.TransferHandler.

- **přidat** na panel **seznamy**, kterým je nastaven transfer handler pomocí jejich instanční metody setTransferHandler(javax.swing.TransferHandler t).
- **přidat panel** na okno.
- v konstruktoru třídy Okno **nastavit glassPane** pomocí metody setGlassPane(javax.swing.JComponent).
- **zobrazit okno**

4.4.4 Části zdrojového kódu

```
public Polozka(List<BufferedImage> nahledy,
        BufferedImage nahledVSeznamu)
{
   super(nahledVSeznamu);
   this.reprezentant = nahledVSeznamu;
   this.nahledy = nahledy;
}
```

Komentář: Výše uvedený konstruktor volá konstruktor své nadtřídy javax.swing.ImageIcon, kde parametr „nahledVSeznamu“ reprezentuje obrázek, který je zobrazen při přidání položky do modelu seznamu ze třídy javax.swing.JList. Parametrem „nahledy“ je inicializována instanční proměnná, která je datového typu java.util.List<BufferedImage>. Do tohoto seznamu je ukládána sekvence obrázků k animování.

```
@Override
public DataFlavor[] getTransferDataFlavors()
{
   return new DataFlavor[]{DataFlavor.javaFileListFlavor};
}
```

Komentář: Výše uvedená metoda se vyskytuje ve třídě Polozka. Tato metoda implementuje konkrétní chování metody deklarované v rozhraní java.awt.datatransfer.Transferable. V této konkrétní

metodě se vrací pole typu java.awt.datatransfer.DataFlavor[6] o jednom prvku.

```
@Override
public Object getTransferData(DataFlavor flavor) throws
                  UnsupportedFlavorException, IOException
{
        List<Polozka> l = new ArrayList<Polozka>();
        l.add(this);
        return l;
}
```

Komentář: Výše uvedená metoda rovněž implementuje metodu z rozhraní java.awt.datatransfer.Transferable. Tato metoda vrací objekt, který reprezentuje přenášená data. V tomto konkrétním případě seznam (java.util.List<Polozka>) o jednom prvku.

operace

```
private static BufferedImage vytvorNahled(BufferedImage
                     puvodniObrazek, int sirka, int vyska)
{
   BufferedImage obrazek = new BufferedImage(sirka, vyska,
                               BufferedImage.TYPE_INT_ARGB);
   Graphics2D g2 = obrazek.createGraphics();

   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
              RenderingHints.VALUE_INTERPOLATION_BILINEAR);
   g2.fillRoundRect(0, 0, sirka, vyska, 20, 20);
   g2.setComposite(AlphaComposite.SrcIn);
   g2.drawImage(puvodniObrazek, 0, 0, obrazek.getWidth(),
                                  obrazek.getHeight(), null);

   GradientPaint mask = new GradientPaint(0, 0, Color.red,
                  obrazek.getWidth(), obrazek.getHeight(),
                  new Color(1.0f, 1.0f, 1.0f, 0.5f));
   g2.setPaint(mask);
   g2.setComposite(AlphaComposite.DstIn);
```

6 *Každá instance reprezentuje pojetí datového formátu, jak by se objevil ve schránce, při drag & drop operaci nebo v souborovém systému. [18]*

```
   g2.fillRect(0, 0, obrazek.getWidth(),
                                obrazek.getHeight());
   g2.dispose();
 return obrazek;
}
```

Komentář: Výše uvedená privátní statická metoda vytvorNahled(BufferedImage puvodniObrazek, int sirka, int vyska) ze třídy ObrazkoveOperace vytvoří náhled ze zadaného obrázku v parametru metody, o konkrétních rozměrech, které jsou definovány ve zbylých parametrech.

```
@Override
public boolean importData(TransferHandler.TransferSupport
                                                    support)
{
   Component komponenta = support.getComponent();
   if (support.getComponent() instanceof JDesktopPane)
   {
      glassPane.setKresli(false);
      glassPane.skryj();
      return true;
   }
   kam = (JList) komponenta;

   Polozka p = (Polozka) getKam().getSelectedValue();
   Point bodKamUsadit = globalniSouradnicePolozky(getKam(),
                                                        p);
   glassPane.setPoziceKamUsadit(bodKamUsadit, kam);
   glassPane.usazuj(this);
   return true;
}
```

Komentář: Výše uvedená metoda ve třídě ZachytavacPrenosu překrývá metodu své nadtřídy. Ona zmiňovaná nadtřída je javax.swing.TransferHandler. Průchod programu v této metodě může vjet do podmíněného bloku, pokud je volána nad komponentou, která je typu javax.swing.JDesktopPane. V tomto případě se přestane vykreslovat vlákno, skryje se a vrátí se true. V případě že komponenta, které jsou importována data není instance javax.swing.JDesktopPane, tak to musí být v našem projektu

javax.swing.JList, neboť jiná možnost zde není[7]. Nakonec se zjistí bod, kam usadit a začne se s usazováním. Dále by bylo také vhodné zmínit, že třída ZachytavacPrenosu v sobě má také implementovanou mimo jiné i metodu boolean canImport(ransferHandler.TransferSupport support), která je automaticky volána, když je taženo nad nějakou komponentou. Tuto metodu zde však neuvádím, protože její kód je poměrně rozsáhlý.

```
private Point globalniSouradnicePolozky(JList seznam,
                                                Polozka pol)
{
   if (polozka != null)
   {
      DefaultListModel model = (DefaultListModel)
                                          seznam.getModel();
      int index = model.indexOf(pol);
      Rectangle2D obdelnikPolozky =
                   seznam.getCellBounds(index, index + 1);

      Point vyslednyBodNaObrazovce = null;
      if (obdelnikPolozky != null)
      {
         vyslednyBodNaObrazovce = new Point(
                        (int)obdelnikPolozky.getX(),
                        (int) obdelnikPolozky.getY());
      }
      else {
         vyslednyBodNaObrazovce = new
                       Point(seznam.getX(), seznam.getY());
      }
      SwingUtilities.convertPointToScreen(
                        vyslednyBodNaObrazovce, seznam);
      return vyslednyBodNaObrazovce;
   }
   else{
        return null;
   }
}
```

7 *Samozřejmě to může být jiná komponenta, zajistí-li to vývojář. Ale v tomto případě to mohou být pouze tyto dvě možnosti.*

Komentář: Výše uvedená privátní metoda taktéž ze třídy ZachytavacPrenosu slouží k zjišťování, kde se nachází levý horní roh (bod), určité položky (instance třídy Polozka) v určitém seznamu (javax.swing.JList). Obsahuje-li seznam hledanou položku, vrátí se poloha hledaného bodu v souřadném systému celé obrazovky.

```
@Override
protected void paintComponent(Graphics g)
{
   if(aktualniObrazek != null)
   {
      int x = aktualniSouradnice.x -
                  aktualniPolozka.getNahledy().get(
                  indexAktualnihoObrazku).getWidth() / 2;
      int y = aktualniSouradnice.y -
                  aktualniPolozka.getNahledy().get(
                  indexAktualnihoObrazku).getHeight() / 2;
      g.drawImage(aktualniObrazek, x, y, null);
      stareSouradnice = new Point(x, y);
    }
}
```

Komentář: Výše uvedená metoda ze třídy AnimaceGlassPane překrývá metodu své nadtřídy (javax.swing.JComponent). Ve stručnosti jde konstatovat, že se zde akorát vykresluje aktuální obrázek na obrazovku.

```
private void nastavNovyObrazek()
{
    if(aktualniPolozka != null)
    {
       if (couva && indexAktualnihoObrazku > 0)
       {
          indexAktualnihoObrazku--;
       } else if (couva && indexAktualnihoObrazku == 0) {
                couva = false;
       }

       if (indexAktualnihoObrazku >=
                   aktualniPolozka.getNahledy().size() - 1)
       {
```

```
            indexAktualnihoObrazku--;
            couva = true;
        }

        if (!couva && indexAktualnihoObrazku <
                    aktualniPolozka.getNahledy().size() - 1)
        {
            indexAktualnihoObrazku++;
        }
        aktualniObrazek = aktualniPolozka.getNahledy().get(
                                indexAktualnihoObrazku);
        staryObrazek = aktualniPolozka.getNahledy().get(
                                indexAktualnihoObrazku);
    }
}
```

Komentář: Výše uvedená privátní metoda je taktéž ze třídy AnimaceGlassPane. V této metodě se dle výše uvedeného algoritmu vybírá nový obrázek, který se ukládá do instanční proměnné „aktualniObrazek“. Dále se inicializuje i starý obrázek, aby mohl být při překreslování překreslen.

```
public void nastavAktualniPozici(Point location) {
    SwingUtilities.convertPointFromScreen(location, this);
    this.aktualniSouradnice = location;
    prekresli();

}
```

Komentář: Nejdůležitější část těle této metody ukazuje jak převádět bod v souřadnicích obrazovky na souřadnice komponenty.

```
public Okno()
{
   initComponents();
   setGlassPane(glassPane);
   seznamA.setModel(new DefaultListModel());
   seznamA.setDragEnabled(true);
   seznamB.setDragEnabled(true);

   final TransferZachytavac t = new
                    TransferZachytavac(glassPane, false);
```

```
   addMouseListener(new MouseAdapter(){
      @Override
      public void mouseEntered(MouseEvent e)
      {
         if ((glassPane.isKresli() || t.getKam() == null ||
            t.getOdkud() == null) && !glassPane.isUsazuj())
         {
                     glassPane.setKresli(false);
                     glassPane.skryj();
         }

       }
    });
    seznamA.setTransferHandler(t);
    setTransferHandler(t);
    seznamB.setTransferHandler(t);
    seznamB.setModel(new DefaultListModel());
}
```

Komentář: Na příkladu výše uvedeného konstruktoru ze třídy Okno je ukázáno, jak je možné povolit nějakým komponentám (zde javax.swing.JList) provádět drag operace (tahání). Dále je zde ukázáno, jak zaregistrovat posluchače seznamům pro poslouchání, které poslouchají, jestli se někdo nesnaží položit nějaký tažený objekt na plochu seznamu.

```
public AnimaceGlassPane()
{
   super();
   setOpaque(false);
   setDoubleBuffered(true);

   jenVymenovaciCasovac = new Timer(intervalZmenyObrazku,
                                    new ActionListener()
   {
        public void actionPerformed(ActionEvent e) {
            nastavNovyObrazek();
        }
   });

   casovacUkladani = new Timer(intervalAutomatickehoPohybu,
                                  new ActionListener()
   {
```

```
      public void actionPerformed(ActionEvent e) {
            nastavNovyObrazek();
            posouvej();
            prekresli();
      }
   });
}
```

Komentář: V Konstruktoru třídy AnimaceGlassPane je vidět, jak se mohou vytvářet časovače ze třídy javax.swing.Timer.

4.5 Vrstvené plátno

Tento příklad je zaměřen na několik témat. V prvé řadě na to, aby demonstroval, jak lze používat **javax.swing.JLayeredPane**, což je speciální typ kontejneru, v kterém je možné kontrolovat do jaké vrstvy se vkládá ta která komponenta a tím zároveň kontrolovat, u které komponenty bude vidět pixel, sdílejí-li 2 a více komponent nějaký pixel či pixely (přerývají se). Mimo jiné tento projekt také ukazuje jak vytvořit animovanou obdobu swingovské komponenty javax.swing.JTabbedPane. Po najetí na vytvořené tlačítko změní vytvořená komponenta obsah (přepne na jinou zobrazovanou záložku), který je asociován s konkrétním tlačítkem. Pro lepší dojem je změna prováděná v rámci jednoduché animace. Tato **animace** se skládá z několika kroků:

- Starý obsah postupně mizí a nový se po zmizení starého zvýrazňuje.
- Zároveň se otáčí deaktivované „tlačítko“ do polohy směřující doprava, signalizující neaktivnost „tlačítka“.
- Rovněž se mění poloha aktivovaného „tlačítka“, které postupně směřuje do aktivované polohy, která je reprezentována, tím že rafička tlačítka směřuje dolů.

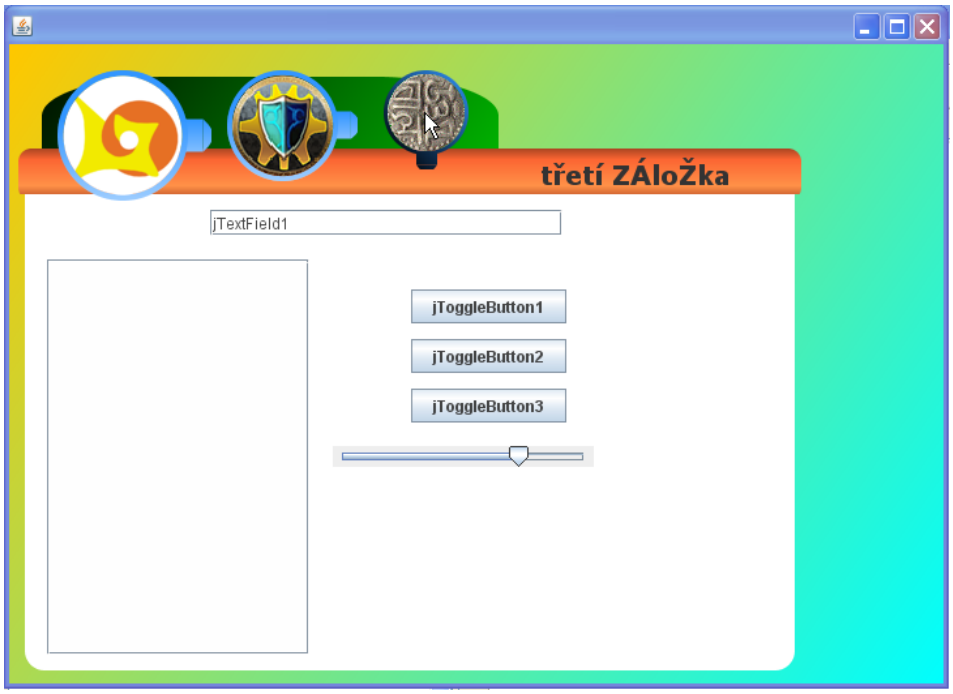
V tomto projektu je ukázáno taktéž jak vytvořit třídu vlastního **tlačítka**, která rozšiřuje třídu **javax.swing.JComponent** a dále také ukazuje jak používat třídu **java.awt.TexturePaint** (tato třída je použita jen v tomto projektu) pro vyplňování obsahu obrázkem. Jelikož „tlačítka“ jsou ve tvaru kolečka, tak když se zaregistruje posluchač myši k jakékoliv standardní swingovské komponentě, reaguje komponenta na události v

celém svém překreslovaném obdelníku. Ale „tlačítka“ v tomto projektu volají nějaký kód pouze, když je událost generována v určitém místě, které zahrnuje:

- obrys tlačítka
- výplň tlačítka
- rafička tlačítka

To je zajištěno tím, že jsou uchovány v instančních proměnných vykreslované tvary. Tyto tvary implementují rozhraní **java.awt.Shape**. Toto rozhraní nutí implementovat každému tvaru metodu **boolean contains(Point point)**, která vrací informaci zda tvar obsahuje či neobsahuje nějaký bod. Událost MouseEvent v sobě zapouzdřuje informaci o bodu, kde byla událost generována. S těmito informacemi je již snadné zajistit, aby se aktivovala jiná záložka například při události, kterou vyvolá najetí myši na kolečko reprezentující tlačítko. Tento příklad také ukazuje jak vytvořit mezikruží, čímž demonstruje jak pracovat s booleovskými operacemi (sloučení, rozdíl, atd.). Tyto logické operace v sobě zahrnuje třída **java.awt.geom.Area**.

4.5.1 Ukázka aplikace



Ilustrace 12: Výstup z aplikace „Vrstvené plátno“

4.5.2 Popis tříd a rozhraní

Tento projekt se skládá celkem z 9 veřejných tříd:

- DruhyObsah
- Kontejner
- Main
- Obsah
- Okno
- Platno
- PrvniObsah

- Tlacitko
- TretiObsah

Třída **Main** obsahuje hlavní metodu a slouží tedy jako vstupní bod programu. Dále udržuje statickou referenci do třídy Platno. Třída **Platno** se stará o vykreslování aktuálního obsahu. Tato třída dědí od javax.swing.JComponent. Třída **Okno** slouží jako místo kam se „lepí“ plátno a dědí od javax.swing.JFrame. Třída **Obsah** dědí od javax.swing.JComponent a implementuje rozhraní ISvetlajici. Tato třída reprezentuje konkrétní obsah jedné viditelné stránky záložky, který je spojen s nějakým tlačítkem. Třída **Kontejner** rozšiřuje třídu javax.swing.JComponent a slouží jako místo kam se přidává vždy jeden viditelný obsah. Třída **Tlacitko** obsahuje algoritmy, které vykreslují tlačítko a také jím otáčí. Dále v sobě obsahuje své vlastní privátní třídy:

- Kolo
- Raficka
- SpickaRaficky

Všechny tyto třída mají společné to, že **dědí od třídy java.awt.geom.Area** a slouží tak jako úložiště s informacemi o tvarech, které vznikaly pomocí logických operacích (průnik, rozdíl, atd.). Poslední 3 třídy (**PrvniObsah**, **DruhyObsah**, **TretiObsah**) mají společné:

- dědí od javax.swing.JPanel
- slouží jako konkrétní obsahy jednotlivých stránek záložkového plátna

Rozhraní **ISvetlajici** deklaruje 4 metody:

- public float getAlfa()
- public void setAlfa(float alfa)
- public String getTitulek()
- public void setTitulek(String titulek)

Toto rozhraní je naimplementováno třídou Obsah (viz. zdrojový kód implementující třídy).

4.5.3 Algoritmus

Hlavní myšlenka tohoto projektu vyžadovala následující kroky:

- vytvořit **okno**.
- vytvořit **vrstvené plátno** typu javax.swing.JLayeredPane.
- vytvořit třídu, která bude sloužit jako **tlačítko**, v projektu to je třída Tlacitko dědící od javax.swing.JComponent.
- ve třídě Tlacitko zajistit, jak bude vypadat **vykreslované tlačítko** pomocí překrytí metody paintComponent(Graphics g).
- tlačítkům zajistit, aby reagovala na události myši, v případě je-li kurzor pouze **v určitých oblastech** (java.awt.geom.Area).
- tlačítkům naimplementovat **handlery** událostí myši.
- vytvořenému plátnu **přidat tlačítka do 1. vrstvy**, zde jsou to instance třídy Tlacitko.
- vytvořit instance třídy **Obsah** (dědí od javax.swing.JComponent a implementuje rozhraní ISvetlajici) a naplnit je nějakým obsahem

(přidání grafických komponent na plátno). V projektu k tomu slouží ukázkové třídy PrvniObsah, DruhyObsah, TretiObsah.

- **spojit obsahy s tlačítky**. Tohoto je docíleno, vložením konkrétní instance tlačítka a konkrétního obsahu do mapy (java.util.HashMap), která je uložená v instanční proměnné „tlacitka“ třídy Platno.
- **nastavit** aktuálně **zobrazovaný obsah**.
- vytvořenému oknu **nastavit plátno,** pomocí instanční metody setContentPane(java.awt.Container platno) třídy javax.swing.JFrame, zde to je instance třídy Platno.
- **zviditelnit okno**.

```
@Override
public void mouseMoved(MouseEvent e)
{
      if((obrys.contains(e.getPoint()) ||
                        vypln.contains(e.getPoint()) ||
                        bambule.contains(e.getPoint()) ||
                        raficka.contains(e.getPoint()))
                        && !isTociSe())
      {
            if (!isDole()) {
                  otocDolu();
            }
      }
}
```

Komentář: První podmínka výše uvedeného kódu ukazuje jakým způsobem zjistit, zda bylo najeto kurzorem myši na jakoukoliv část tlačítka. Tvar tlačítka je reprezentován čtyřmi tvary, které jsou uloženy v instančních proměnných: „obrys“, „vypln“, „bambule“ a „raficka“.

```
private void otocDolu()
```

```
{
  Runnable r = new Runnable()
  {
    public void run()
    {
      setTociSe(true);
      if(!isDole())
      {
        float pom = 0.0f;
        for(int i = 0; i<=otoceni; i += inkrementOtoceni)
        {
          rotace = i;
          repaint();
          try
          {
            Thread.sleep(getCas());
          }
          catch (Exception e)
          {
            e.printStackTrace();
          }
          Main.getPlatno().nastavAlfuObsahu(pom,
                  Main.getPlatno().getAktualniObsah());
          pom += 0.1f;
        }
        ukazAktualniObsah();
        setDole(true);
      }
       setTociSe(false);
     }
   };

   Thread t = new Thread(r);
   t.start();
}
```

Komentář: Výše uvedená část kódu ukazuje celou privátní metodu otocDolu() ze třídy Tlacitko. V této metodě se vytváří a spouští nové vlákno. Kód, který je volán ve zmiňovaném vlákně se stará o otáčení tlačítka, postupným inicializováním instanční proměnné „rotace“ v cyklu. Proměnná „rotace“ je typu float a uchovává v sobě informaci o aktuálním otočení tlačítka. Dále se při každém průchodu cyklem překresluje celé tlačítko. Rovněž s každým průchodem cyklem je zprůhledňován aktuální obsah. Po skončení

cyklu je volána metoda ukazAktualniObsah(), která se postará o vykreslení nového obsahu.

```
Kolo(float x, float y, float prumer)
{
    x_pozice = x;
    y_pozice = y;
    stredX = x + (prumer / 2);
    stredY = y + (prumer / 2);

    Ellipse2D.Double v =
                  new Ellipse2D.Double(x,y,prumer,prumer);

    Ellipse2D.Double m = new Ellipse2D.Double(x + prumer /
                 22.059,y + prumer / 22.059,prumer / 1.1,
                 prumer / 1.1);

    obrys = new Area(v);
    vypln = new Area(m);

    obrys.subtract(vypln);
    super.add(obrys);
}
```

Komentář: Výše uvedený konstruktor třídy Kolo je obsažen v privátní třídě Kolo. Třída Kolo dědí od java.awt.geom.Area. Tuto třídu má v sobě třída Tlacitko. Na tomto příkladovém zdrojovém kódu je ukázáno, jak provádět booleovské operace nad tvary rozhraní, které implementují rozhraní java.awt.Shape. Tyto logické operace (průnik, sjednocení, apod.) můžeme provádět pomocí metod třídy java.awt.geom.Area (viz. zdrojový kód).

```
public Obsah(String textTitulku)
{
    this.titulek = textTitulku;
}

@Override
protected void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    g2.setComposite(AlphaComposite.SrcOver.derive(1.0f -
                                          getAlfa()));
```

```
}
```

Komentář: Ve výše uvedeném konstruktoru se pouze inicializuje instanční proměnná „titulek“. V překryté metodě paintComponent(Graphics g) se pouze nastavuje alfakompozice

```
public Platno()
{
   setLayout(new BoxLayout(this, BoxLayout.PAGE_AXIS));
   layeredPane = new JLayeredPane();
   layeredPane.setPreferredSize(new Dimension(450, 300));
   pocatekSouradnehoSystemu = new Point(7, 80);
   Tlacitko tlacitko1 =
      new Tlacitko(
            new Point(getPocatekSouradnehoSystemu().x + 30,
                  getPocatekSouradnehoSystemu().y - 60),
                  100, "obrazek.png", 20, 9);

   layeredPane.add(tlacitko1, new Integer(1));

   kontejner = new Kontejner(getPocatekSouradnehoSystemu(),
                                          600, 400);
   Obsah obsah1 = new Obsah("první záložka");
   PrvniObsah po = new PrvniObsah();
   po.setLocation(10,0);
   po.setSize(kontejner.getWidth() -
                     20,kontejner.getHeight() - 40);
   obsah1.add(po);

   tlacitka.put(tlacitko1, obsah1);
   nastavAktualniObsah(tlacitko1);

   layeredPane.add(kontejner, new Integer(0));
   add(layeredPane);
}
```

Komentář: Konstruktor třídy Platno je potomek třídy javax.swing.JPanel. V originálním zdrojovém kódu je kód v konstruktoru podstatně větší. Ale zde je uvedená jen ta důležitější část, protože určité části kódu se v originále opakují. Druhý příkaz v kódu vytvoří nové vrstvené plátno (javax.swing.JLayeredPane), které uloží do instanční proměnné „layeredPane“. Dále je zajímavý pátý příkaz ve kterém se vytvoří nová instance třídy Tlacitko.

Šestý příkaz zajistí, že se přidá instance tlačítka do první vrstvy. Dále se inicializuje instanční proměnná „kontejner“ typu Kontejner. Vytvoří se instance třídy Obsah. Ve čtvrtém příkazu od konce se vloží do mapy (java.util.HashMap<Tlacitko, Obsah>) vytvořená instance tlačítka jako klíč a jako hodnota klíče se uloží vytvořený obsah. Zvolil jsem mapu, aby odpovídalo určité tlačítko určitému obsahu. Ke konci kódu se volá metoda nastavAktualniObsah (Tlacitko tlacitko), která je zobrazena níže jako další ukázka. Nakonec se přidá vytvořené vrstvené plátno k instanci třídy Platno.

```
public void nastavAktualniObsah(Tlacitko tlacitko)
{
   if(getAktualniObsah() !=  null) {
        getKontejner().remove(getAktualniObsah());
   }
   setAktualniObsah(getTlacitka().get(tlacitko));
   kontejner.getTitulek().setText(
                       aktualniObsah.getTitulek());
   setAktualniTlacitko(tlacitko);
   getAktualniObsah().setLocation(new Point(0, 1));
   getAktualniObsah().setSize(
            getKontejner().getSize().width,
            getKontejner().getSize().height);

   getKontejner().add(getAktualniObsah(), null, -1);
}
```

Komentář: Tato metoda ze třídy Platno vymaže z modelu instanční proměnné „kontejner“, která je ze třídy Kontejner (podtřída javax.swing.JComponent) aktuální obsah. Nastaví aktuální obsah, titulek a seplé tlačítko. Nakonec přidá aktuální obsah do kontejneru.

```
public void nastavAlfuObsahu(float alfa, Obsah obsah)
{
   if (alfa >= 0.0f && alfa <= 1.0f)
   {
       if (alfa <= 1.0f) {
                obsah.setAlfa(alfa);
```

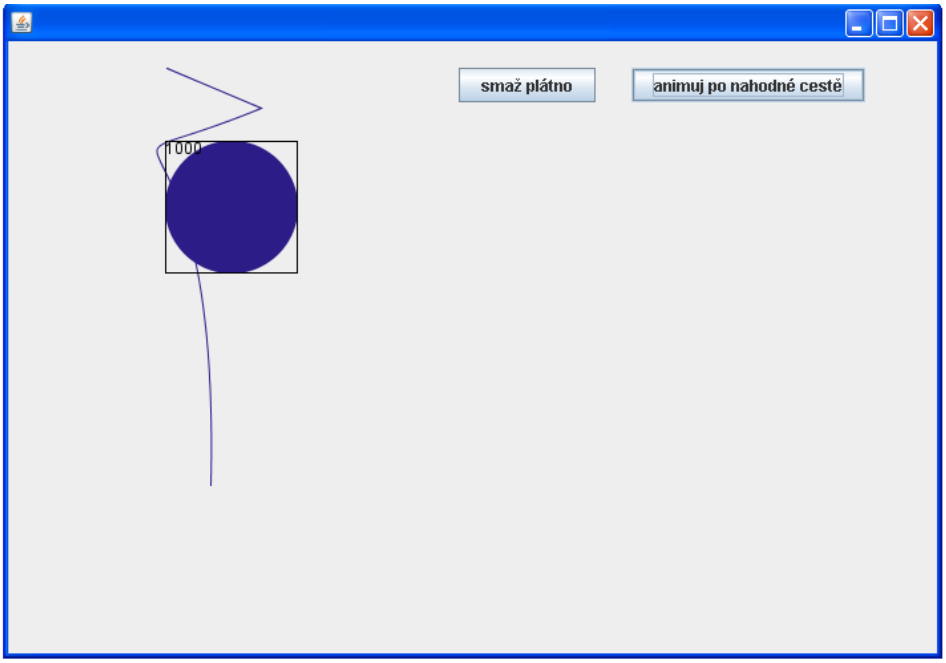
```
        }
        if (alfa == 0.9000001f) {
                obsah.setAlfa(1.0f);
        }
        Main.getPlatno().getKontejner().repaint();
    }
}
```

Komentář: Výše uvedená metodu ze třídy Kontejner nastaví nějakému obsahu hodnotu jeho alfa kanálu. Obsah i hodnota alfa kanálu jsou pochopitelně parametry metody. Nakonec se překreslí celá instance třídy Kontejner, uložená v instanční proměnné „kontejner“.

4.6 Animační Framework

Tento příklad byl vytvořen proto, aby bylo co nejjednodušší animovat javovské aplikace. Pomocí tohoto frameworku je **možné přesouvat** jakýkoliv objekt, který je podtřídou **javax.swing.JComponent**, **po dráze** která musí splňovat, že je podtřídou **java.awt.geom.Path2D**, například třída java.awt.geom.GeneralPath, která umožňuje vytvářet křivky, které jsou složeny z úseček, kubických a kvadratických křivek. Dále také je schopen framework zajistit, aby animovaná komponenta dorazila **od začátku** křivky **do cíle** za **určený čas**, neboť v sobě framework obsahuje i **časovač**. Komponenta, která má být animovaná je ukotvena ke křivce cesty za levý horní roh. Jako ukázkové komponenty k animování jsou vytvořeny komponenty, které vykreslují náhodně velké a barevné kolečko v rámečku s textem, který udává jak dlouho se má komponenta animovat(přejíždět křivku). Samo o sobě kolečko se náhodně zvětšuje a zmenšuje, aby byla demonstrovaná vlastnost, že mohou být animovány animované komponenty. V projektu je také možné nastavit, zda vykreslovat křivku po které se „jede“, aby bylo možné lépe nastavovat cestu kudy animovat.

4.6.1 Ukázka programu



Ilustrace 13: Výstup z aplikace „Animační framework“

4.6.2 Popis balíčků

Tento projekt se skládá ze 2 balíčků:

- animacni_framework
- ukazkova_data

V balíčku **animacni_framework** je samotná implementace animančního nástroje, jak název napovídá. Balíček **ukazkova_data** je k projektu připojen z důvodu ukázky práce animátoru.

4.6.2.1 Třídy balíčku animacni_framework

Balíček s implementovaným animačním algoritmem obsahuje 3 třídy:

- AnimovanaScena
- AnimovanaKomponenta
- PocitaniKrivek

Třída **AnimovanaScena** obsahuje časovač, který v určitou dobu hýbe určitou komponentou. Tohoto je dosaženo tím, že při tiku časovače je procházená kolekce celé scény, která je uložena v statické třídní proměnné pojmenované **prvky**, která je typu **java.util.ArrayList<AnimovanaKomponenta>** . Pomocí časových známek (instanční proměnné) instance AnimovanaKomponenta je určováno, zda má být pohnuto s konkrétní komponentou, která je momentálně vytažená z cyklu (viz. zdrojový kód).

Třída **AnimovanaKomponenta** je datová struktura, která obsahuje informace o cestě, kde se má vykreslovat určitá komponenta typu **javax.swing.JComponent**, jak dlouho má trvat celá cesta komponenty od začátku do konce. Dále v sobě také zapouzdřuje informaci o cestě kudy se má animovat. Tato informace je uložena v instanční proměnné **cestaAnimace**, která je typu **java.awt.geom.Path2D**. Také v sobě tato třída uchovává booleovskou instanční proměnnou, která říká zda má být vykreslena i cesta kudy se bude komponenta přesouvat, apod.

Třída **PocitaniKrivek** je v projektu protože se cesta animace může skládat z křivky, která je daná například jako instance třídy **java.awt.geom.GeneralPath**. Tato křivka se skládá z určitých segmentů.

Křivkové segmenty mohou být buď udány jako úsečka (vymezená 2 body koncovými), jako kvadratická či kubická křivka (vymezená 3 kontrolními body, potažmo 4 body). Jelikož je nutné znát body kudy křivka vede, je nutné spočítat podrobnější průběh křivky a toho je zde docíleno pomocí Bernsteinových polynomů[8] na intervalu [0,1]. Tyto algoritmy jsou implementované ve statických třídních metodách třídy PocitaniKrivek.

4.6.2.2 Třídy balíčku ukazkova_data

V tomto balíčku jsou třídy, na kterých není závislá funkčnost animačního nástroje. Jsou zde třídy, které mohou být nahrazeny podobnými třídami, která splňují určitá pravidla, která budou zmíněna níže u algoritmu. Konkrétně jsou to shodou okolností také tři třídy:

- Okno
- UkazkovaKomponenta
- VrstvenePlatno

Třída **Okno** dědí od javax.swing.JFrame a slouží jako okno celé aplikace.

Třída **UkazkovaKomponenta** je v projektu, aby bylo vidět, že se něco někde pohybuje. V tomto případě se v ní vymalovává animované kolečko a vykresluje obdelník, ale může se vykreslovat samozřejmě cokoliv,

8 *„V teorii numerické matematiky je Bernsteinův polynom, nebo také polynomu v Bernsteinově tvaru, polynomem, který je lineární kombinací Bernsteinových bázových polynomů.“[25]*

neboť tato třída dědí od **javax.swing.JComponent** a má přetíženou metodu **paintComponent(Graphics g)**. Poslední nepopsaná třída **VrstvenePlatno** funguje jako plocha na které se animují objekty. V tomto projektu tato třída dědí od javax.swing.JLayeredPane, tudíž je možné přidávat na ní komponenty v různých vrstvách. Vrstvenost kreslící plochy je zde proto, aby bylo vidět nově přidávané komponenty, které jsou přidávány vždy do vyšší vrstvy, než komponenta, která byla přidávaná dříve. Aby byla zachovaná funkčnost projektu, plátno na které jsou vykreslovány animované objekty musí splňovat jediné pravidlo, které diktuje podmínky funkcionality. Tato jediná podmínka říká, že plátno musí být hierarchicky pod třídou javax.swing.JComponent.

4.6.3 Algoritmus použití animačního frameworku

Myšlenka tohoto projektu žádala následující kroky:

- vytvořit **plátno** na které se bude kreslit. Toto plátno **musí dědit od javax.swing.JComponent**
- mít vytvořené **okno** na které se přidá plátno
- nastavit pomocí statické metody **setPlatno(JComponent platno)** ze třídy **AnimovanaScena** námi vybrané plátno, na které chceme kreslit
- pomocí statické metody **spustCasovac()** rovněž ze třídy **AnimovanaScena** spustit časovač, který po určitém čase překresluje scénu (jen co je třeba).

- v případě, že chceme vykreslovat nějaký objekt, který se přesouvá po určité dráze, vytvoříme instanci třídy **AnimovanaKomponenta**. Tato třída má pouze jeden konstruktor, který má parametry:
  - JComponent **kdeAnimovat** (zadáme referenci na plátno na které chceme vykreslovat náš objekt)
  - JComponent **coAnimovat** (zadáme referenci na objekt, kterým chceme hýbat)
  - Path2D **cestaAnimace** (zadáme cestu, po které chceme aby se přesouvaná komponenta pohybovala)
  - long **celkovaDelkaAnimace**(zadáme parametr za jak dlouho chceme, aby projel objekt po křivce od začátku do konce)
  - jako poslední krok zavoláme u komponenty, kterou chceme animovat po dráze instanční metodu **setAnimuj**(boolean animovat) ze třídy AnimovanaKomponenta s parametrem **true**

4.6.4 Fragmenty zdrojového kódu

```
public AnimovanaKomponenta(JComponent kdeAnimova,
                JComponent coAnimovat, Path2D cestaAnimace,
                                    long celkovaDelkaAnimace)
{
  this.zmenitZaMilisekund = 1000 /
                   AnimovanaScena.getPocetSnimkuZaSekundu();

  this.cestaAnimace = cestaAnimace;
  this.kdeAnimovat = kdeAnimova;
  this.komponenta = coAnimovat;
  this.iteratorCesty = cestaAnimace.getPathIterator((
                   (Graphics2D)kdeAnimovat.getGraphics())
```

```
                                                .getTransform());

  this.dobaAnimace = celkovaDelkaAnimace;
  ArrayList<ArrayList<Point2D>> sez =
            PocitaniKrivek.rozdelNaSegmenty(iteratorCesty);
  this.bodyCesty = PocitaniKrivek.getposloupnostBodu(sez);
  zjemniBody();
  AnimovanaScena.setCelkemAnimovanychPrvku(
             AnimovanaScena.getCelkemAnimovanychPrvku()+1);

  kdeAnimovat.add((JComponent)coAnimovat,
      new Integer(
            AnimovanaScena.getCelkemAnimovanychPrvku()));
}
```

Komentář: Výše uvedený konstruktor třídy AnimovanaKomponenta v sobě zahrnuje klasické incializace, které se obvykle provádí v konstruktorech. Tento fragment zdrojového kódu je zde, proto aby bylo jednodušší představit si strukturu třídy AnimovanaKomponenta.

```
static
{
  casovac = new Timer(1, new ActionListener() {

    public void actionPerformed(ActionEvent e)
    {
      for(AnimovanaKomponenta prvekKAnimovani : prvky)
      {
        if prvekKAnimovani.isAnimuj() &&
            (System.currentTimeMillis()-
                        prvekKAnimovani.getPosledniZmena())
            >= prvekKAnimovani.getZmenitZaMilisekund() )
        {
          if (prvekKAnimovani.getIndexAktualniPozice() <
                  prvekKAnimovani.getBodyCesty().size())
          {
            prvekKAnimovani.setAktualniBod(
                        prvekKAnimovani.getBodyCesty().get(
            prvekKAnimovani.getIndexAktualniPozice()));

            prvekKAnimovani.getKomponenta().setLocation(
            (int)prvekKAnimovani.getAktualniBod().getX(),
            (int)prvekKAnimovani.getAktualniBod().getY());
```

```
            prvekKAnimovani.setIndexAktualniPozice(
              prvekKAnimovani.getIndexAktualniPozice()+ 1);

            prvekKAnimovani.setPosledniZmena(
                              System.currentTimeMillis());

          }
          else
          {
           if(prvekKAnimovani.getBodyCesty().get(
              prvekKAnimovani.getBodyCesty().size()-1)==
              prvekKAnimovani.getAktualniBod())
             {
              prvky.remove(prvekKAnimovani);
              getCestyKVykresleni().remove(
                 prvekKAnimovani.getCestaAnimace());
                    return;
             }
           }
         }
        }
        platno.repaint();
       }
   });
}
```

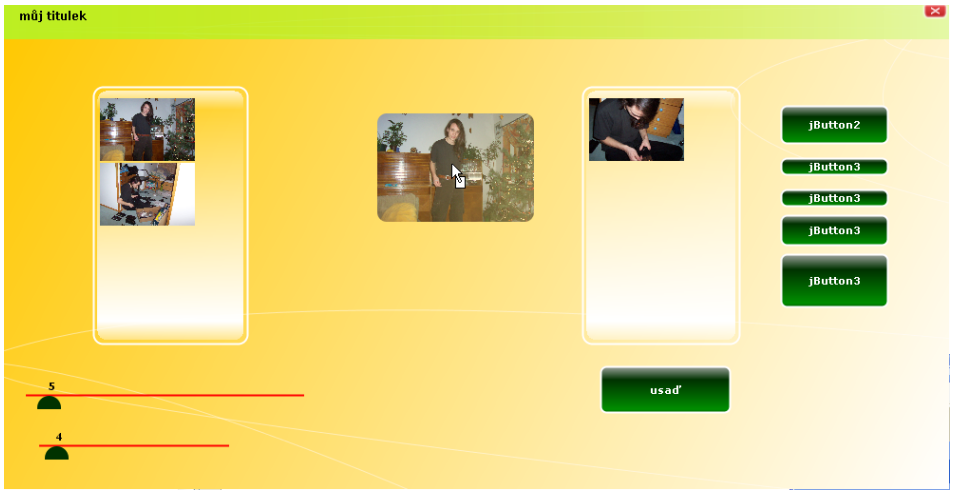
Komentář: Výše uvedený statický blok ze třídy AnimovanaScena ukazuje, kdy se překresluje nějaká komponenta. Tento blok v sobě zahrnuje vytvoření časovače (javax.swing.Timer), který volá každou milisekundu kód s cyklem ve kterém se procházejí prvky seznamu (java.util.ArrayList<AnimovanaKomponenta>), který je uložen ve statické proměnné „prvky“. Do této kolekce jsou přidávány komponenty, které mají být animovány po určité dráze. V případě, že komponenta je v cíli cesty, odebere se ze seznamu, aby seznam, kterým se prochází nebyl zbytečně velký a nezdržoval tak procházení cyklem. O tom **zda** má být některá komponenta **překreslena** rozhoduje **rozdíl aktuálního času** a **času** kdy byla **naposled překreslována**. Pokud je tento rozdíl větší nebo roven hodnotě, za kolik milisekund má být ta která komponenta překreslena, přesune se (překreslí se).

4.7 Vlastní vzhled Look & Feel

V tomto příkladě je popsáno, jak jednoduše změnit vzhled jednotlivých komponent aplikace bez složitého programování pomocí balíčku **javax.swing.plaf.synth**. Například všechna tlačítka budou mít jednotný vzhled. O tom jaké bude mít ta která komponenta vzezření rozhoduje typicky jeden **xml soubor**, v kterém jsou deklarovány elementy informující o tom jak vykreslovat region/-y[9] určité komponenty. Tento deklarativní přístup je velice podobný vytváření stylů pomocí CSS u webových stránek. Výše zmiňovaný xml soubor může obsahovat deklarace tagů, které říkají například že se má jako pozadí seznamu použít nějaký externě nahrávaný obrázek či informace o tom ve které třídě se nachází metoda a jak se jmenuje, která se stará o vykreslení komponenty. Tento druhý způsob se mi zdá lepší, protože nikterak nesvazuje vývojáři ruce v kreativitě. Je sice o něco složitější na realizaci, ale o to má větší možnost vytvořit zajímavý vzhled aplikace. Samozřejmě, že je možné kombinovat oba tyto způsoby. Například tlačítka vykreslovat programově v nějaké metodě a zároveň vykreslovat pozadí panelů externě načítaným obrázkem a nic neprogramovat.

9 *Region reprezentuje určitý typ vykreslované plochy určité komponenty. Každá třída dědící od javax.swing.JComponent může mít jeden či více regionů. Všechny standardní podporované regiony v sobě zahrnuje třída javax.swing.plaf.synth.Region ve statických proměnných. [17] Například pro javax.swing.JButton existuje pouze region Button. Oproti tomu pro komponenty typu javax.swing.JScrollBar existují hned 3 regiony: ScrollBar, ScrollBarTrack a ScrollBarThumb.*

4.7.1 Ukázka aplikace



Ilustrace 14: Výstup aplikace „Vlastní vzhled Look & Feel“

4.7.2 Popis funkcionality

4.7.3 Obsah balíčku „vzhled“

- třída **VykreslovacProStav1**
- třída **VykreslovacProStav2**
- soubor **konfiguracniSoubor.xml**

4.7.3.1 Popis tříd balíčku „vzhled“

O programové vykreslování komponent se starají třídy **VykreslovacProStav1** a **VykreslovacProStav2,** které dědí od třídy **javax.swing.plaf.synth.SynthPainter**. Zvolil jsem tyto názvy tříd, protože v konfiguračním xml souboru vzhledu existuje element „state“, který může obsahovat atribut „value“, který říká při jaké události kreslit.

Jelikož jsem chtěl oddělit třídy které budou vykreslovat stavy komponent v různých stavech (PRESSED, ENABLED, MOUSE_OVER, atd.) zvolil jsem tuto terminologii. V tomto projektu jsou deklarovány komponentám pouze 2 stavy:

- všeobecné vykreslení
- po najetí myší na komponentu (v příkladu pouze na tlačítko)

4.7.3.2 Soubor „konfiguracniSoubor.xml“

```
<synth>
  <style id="vse">
    <opaque value="false"/>
   <font name="VERDANA" size="11" style="BOLD"/>
  </style>
  <bind style="vse" type="region" key=".*"/>
 <style id="stylTlacitka">
      <property key="Button.textShiftOffset" type="integer"
            value="1"/>
    <insets top="3" left="20" right="20" bottom="3"/>
    <state value="MOUSE_OVER">
       <object id="pr" class="vzhled.VykreslovacProStav2"/>
       <painter method="buttonBackground" idref="pr"/>
    </state>
    <state>
       <object id="p" class="vzhled.VykreslovacProStav1"/>
       <painter method="buttonBackground" idref="p"/>
       <color value="white" type="FOREGROUND"/>
    </state>
  </style>
  <bind style="stylTlacitka" type="region" key="BUTTON"/>
</synth>
```

„výřez z konfiguračního souboru“

Výše uvedená část kódu je pouze výřez z celého konfiguračního souboru a je zde jen pro ukázku v jakém duchu se nese xml formát. Popis použitých tagů:

- **<synth>**- kořenový element souboru

- **<style>** - element s atributem id. V jeho těle se deklaruje vzhled jednotlivého regionu.
- **<bind/>** - element sloužící k připojení a používání nadeklarovaného stylu. Styl, který má být připojen je referencován na styl pomocí svého atributu „style“. Hodnota atributu „style“ se musí shodovat s atributem „id“ elementu <style>, který chceme připojit.
- **<state>** - je element a může obsahovat atribut „value“, který deklaruje při jaké události se má použít to které vykreslení. Například při pohybu kurzoru myši nad komponentou má atribut „value“ hodnotu „MOUSE_OVER“.
- **<object />** - tento element se používá, když je třeba získat referenci na třídu, která je udaná v atributu „class“ celým svým názvem. Kam se má uložit reference deklaruje atribut „id“
- **<painter /> -** tento element obsahuje atribut „method“. V tomto atributu je deklarováno jakou metodu volat pro vykreslování. Dále také obsahuje atribut „idref“. Hodnota tohoto atributu se musí shodovat s atributem „id“ elementu „<object>“. Hodnota atributu „method“ se musí shodovat s názvem existující metody třídy, která je referencována pomocí atributu „idref“, odmyslíme-li si prefix **paint**, neboť všechny použitelné vykrelovací metody tříd dědících od javax.swing.plaf.synth.SynthPainter začínají na „paint“. Proto je nutné zadávat název volané metody bez prefixu paint.

Samozřejmě je možné použít více elementů. Popis všech možných elementů a jejich atributů je možné získat ze zdroje [19]. Dále je také možné nadeklarovat komponentám různé vlastnosti. Přehled všech těchto vlastností je možné najít ve zdroji [20].

4.7.4 Algoritmus

Při vytváření tohoto projektu bylo potřeba:

- vytvořit konfigurační xml soubor.
- vytvořit třídu či třídy dědící od třídy **javax.swing.plaf.synth.SynthPainter**, pokud chceme vykreslovat programově
- pokud vykreslujeme programově, **překrýt metody** začínající na „paint“ u regionů, které chceme překreslit.
- pokud máme vytvořený svůj styl, načteme ho a nastavíme jako aktuální, voláním statické metody „setLookAndFeel“ třídy „javax.swing.UIManager“

4.7.5 Části zdrojového kódu programu

```
SynthLookAndFeel styl = new SynthLookAndFeel();
String cestaKeKonfiguracnimuSouboru
                                ="konfiguracniSoubor.xml";

try
{
  styl.load(VykreslovacProStav1.class.getResourceAsStream(
                             cestaKeKonfiguracnimuSouboru),
                                VykreslovacProStav1.class);

  UIManager.setLookAndFeel(styl);
```

```
}

catch (Exception e)
{
          e.printStackTrace();
}
```

Komentář: Výřez kódu ze statické metody inicializujVzhled() třídy Main ukazuje, jak jednoduše načíst xml soubor s deklaracemi designu a nastavit vzhled.

4.7.5.1 Ukázka dvou metod třídy VykreslovacProStav1

```
@Override
public void paintButtonBackground(SynthContext context,
Graphics g, int x, int y, int w, int h)
{
   LinearGradientPaint mnohoPrechodovaBarevnaTextura =
      new LinearGradientPaint(0.0f, 0f, y, h,
            new float[]{0f, .3f, .9f,1f},
            new Color[]{new Color(255, 255, 255).darker(),
                    new Color(0, 50, 0),
                    new Color(0, 100, 0, 0).brighter(),
                    new Color(0, 100, 0, 20).darker()
        });


   int sirka = 2;
    Graphics2D g2 = (Graphics2D) g;

   g2.setPaint(mnohoPrechodovaBarevnaTextura);
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,

RenderingHints.VALUE_ANTIALIAS_ON);
   g2.fillRoundRect(x + sirka, y + sirka, w - sirka * 2, h
                        - sirka* 2,12, 12);
   g2.setColor(Color.WHITE);
   g2.setStroke(new BasicStroke(sirka));
   g2.drawRoundRect(x + sirka,y + sirka,w - sirka*2, h
                                    - sirka * 2, 12, 12);
}
```

Komentář: Metoda paintButtonBackground(SynthContext context, Graphics g, int x, int y, int w, int h) je obsažena ve třídě

VykreslovacProStav1 balíčku vzhled. U této třídy je důležité zmínit, že dědí od třídy javax.swing.plaf.synth.SynthPainter. Zmiňovaná metoda překrývá metodu třídy javax.swing.plaf.synth.SynthPainter. Na tuto uvedenou metodu je odkazováno pro vykreslování tlačítka v deklaraci xml souboru. Konkrétně tato metoda má na starosti vykreslování regionu, který reprezentuje pozadí tlačítka.

```
@Override
public void paintSliderThumbBackground(SynthContext
      context, Graphics g, int x, int y, int w,int h, int
                                          orientation)
{
   Graphics2D g2 = (Graphics2D) g;
   g2.setPaint(Color.GREEN);

   int prumer = 25;
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setStroke(new BasicStroke(prumer,
      BasicStroke.CAP_SQUARE, BasicStroke.JOIN_MITER));

   g2.setPaint(new Color(0, 50, 0));

   Ellipse2D.Double ed = new Ellipse2D.Double(x, y, prumer,
                                                 prumer);
   Rectangle.Double rec = new Rectangle.Double(x, y, w, h);
   Area ctverec = new Area(rec);

   Area elipsa = new Area(ed);
   elipsa.intersect(ctverec);
   g2.fill(elipsa);

}
```

Komentář: Metoda paintSliderThumbBackground funguje obdobně jako uváděná metoda výše. S rozdílem, že vykresluje tahátko slideru (javax.swing.JSlider).

5 Závěr

5.1 Zhodnocení práce

Pro připomenutí, v zadání práce byly vytyčeny čtyři hlavní cíle, které zahrnovaly:

- zpracování stručného popisu obsahující nezbytné informace o Java Graphics API
- věnovat se pokročilým technikám vykreslování grafických komponent
- věnovat se animačním technikám
- vývoj vybraných aplikací a jejich transformace do podoby výukových projektů.

Subjektivně sám za sebe si troufám tvrdit, že cíle byly splněny, neboť část práce se zabývá obecnými informacemi o Java Graphics API. Pro tuto část byla většina textů tvořena překladem oficiálních dokumentů, neboť předpokládám, že výrobce určité technologie bude pravděpodobně nejvíce v obraze, publikuje-li nějaké informace o určitých svých produkovaných enginech, v našem případě grafických. Následující dva cíle byly splněny při vytváření výukových příkladů. Co se týká výukových projektů, tak samozřejmě některé příklady by bylo možné optimalizovat, dle různých kritérií, kterými mohou být menší paměťová náročnost, menší časová náročnost či jiná individuální kritéria. Příkladem individuálního kritéria může být snaha, zvýšit přesnost animačního frameworku, hlavně v případě, je-li animováno více objektů „zároveň“.

Těchto nepřesností a možností optimalizace jsem si vědom. Rovněž jsem si však vědom i toho, že dokonale fungující věci jsou obvyklé náročné na realizaci (je-li to vůbec možné), hlavně v ohledu na časový fond, který v mém případě nebyl závratně velký. Dále také byl problém, jakým konkrétním příkladům se věnovat (viz. Kapitola 1.2). V tomto ohledu je snad všem jasné, že jsem se mohl věnovat pouze několika určitým příkladům, které jsem uznal za vhodné vytvořit. Samozřejmě jiný člověk by volil zřejmě jiné příklady, kterým by se věnoval, ale já zvolil výše popsané, jelikož se domnívám, že demostrují několik různých přístupů a popisují, co je na těchto přístupech dobré a co nikoliv.

5.2 Zhodnocení použitých technologií

Při vytváření této práce se ukázalo, že pomocí balíčků awt a swing je možné dělat velice zajímavé grafické efekty, aniž by bylo potřeba používat nestandardních knihoven.

Reference

[1] *ORACLE Java Technology* [online]. Copyright © 1993, 2010 [cit. 2010-10-02]. Java 2D™ Graphics and Imaging. Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/index.html>.

[2] *ORACLE Java Technology : Java 2D™ API Overview* [online], © 1993, 2010 [cit. 2010-09-08]. Enhanced Graphics, Text, and Imaging, Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[3] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. Rendering Model, Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[4] HORSTMANN, Cay S.; CORNELL, Gary. *Core Java™ 2 Volume II - Advanced Features, Seventh Edition : The Rendering Pipeline*. [sill.] : Prentice Hall PTR, 2004. Advanced AWT, s. 1024. ISBN 0-13-111826-9.

[5] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. Coordinate Systems, s. . Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[6] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. 1.2.5 Fills and Strokes, s. . Dostupné z WWW:<http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[7] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. Composites, s. . Dostupné z WWW:

<http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[8] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. Images, s. . Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[9] *ORACLE Java Technology : Java 2D™ API Overview* [online]., © 1993, 2010 [cit. 2010-09-08]. Transforms, s. . Dostupné z WWW: <http://download.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html>.

[10] *ORACLE Java Technology : What is Swing? (The Java™ Tutorials Graphical User Interfaces A Brief Introduction to the Swing Package)* [online]., © 1993, 2010 [cit. 2010-09-10]. What is Swing?, s. . Dostupné z WWW: <http://download.oracle.com/javase/tutorial/ui/overview/intro.html>.

[11] Swing (Java). In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-09-15]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Swing_(Java)#cite_note-2>.

[12] Soubor:AWTSwingClassHierarchy.png. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-09-15]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Soubor:AWTSwingClassHierarchy.png>.

[13] Event dispatching thread. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-10-12]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Event_dispatching_thread>.

[14] *Painting in AWT and SWING : Paint Processing* [online]., © 1993, 2010 [cit. 2010-10-12]. Painting in Swing, s. . Dostupné z WWW: <http://java.sun.com/products/jfc/tsc/articles/painting/>.

[15] *Java™ Platform Standard Ed. 6 : paintComponent* [online]. © 1993, 2010 [cit. 2010-10-15]. JComponent (Java Platform SE 6). Dostupné z WWW: <http://download.oracle.com/javase/6/docs/api/javax/swing/JComponent.html>.

[16] HORSTMANN, Cay S.; CORNELL, Gary. *Core Java Volume I - Fundamentals*. Revised and Updated for Java SE 6, 2008. Threads and Swing, s. 811. ISBN 0-13-235476-4.

[17] *Java™ Platform Standard Ed. 6* [online], © 1993, 2010, [cit. 2010-12-05]. Region, s. . Dostupné z WWW: <http://download.oracle.com/javase/6/docs/api/javax/swing/plaf/synth/Region.html>.

[18] *Java™ Platform Standard Ed. 5* [online]. © 1993, 2010 [cit. 2010-12-06]. DataFlavor (Java Platform SE 5). Dostupné z WWW: <http://download.oracle.com/javase/1.5.0/docs/api/java/awt/datatransfer/DataFlavor.html>.

[19] *Download.oracle.com : File Format* [online]. 20?? [cit. 2010-12-06]. Synth File Format. Dostupné z WWW: <http://download.oracle.com/javase/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/synthFileFormat.html>.

[20] *Download.oracle.com : Component Specific Properties* [online]. 20?? [cit. 2010-12-06]. Dostupné z WWW: <http://download.oracle.com/javase/1.5.0/docs/api/javax/swing/plaf/synth/doc-files/componentProperties.html>.

[21] *The Java ™ Tutorials* [online]., 20?? [cit. 2010-12-12]. How to Use Root Panes, Dostupné z WWW: <http://download.oracle.com/javase/tutorial/uiswing/components/rootpane.html>.

[22] *The Java ™ Tutorials* [online], 20?? [cit. 2010-12-12]. The Glass Pane, Dostupné z WWW: <http://download.oracle.com/javase/tutorial/uiswing/components/rootpane.html>.

[23] *The Java ™ Tutorials* [online], 20?? [cit. 2010-12-12]. The Layered Pane, Dostupné z WWW: <http://download.oracle.com/javase/tutorial/uiswing/components/rootpane.html>.

[24] BONDARENKO, Dmitry; PETROV, Anthony. *SDN Home : Graphical User Interfaces* [online]. updatováno: srpen 2008., duben 2008 [cit. 2010-10-10]. How to Create Translucent and Shaped Windows, s. . Dostupné z WWW: <http://java.sun.com/developer/technicalArticles/GUI/translucent_shaped_windows/>.

[25] Bernsteinův polynom. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-12-13]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Bernstein%C5%AFv_polynom>.

[26] KUŽELKA, Ondřej. *Interval.cz : Java* [online]., 16. 12. 2003 [cit. 2010-11-18]. Java - pokročilá grafika, s. . Dostupné z WWW: <http://interval.cz/clanky/java-pokrocila-grafika-operace-s-obrazky/>.

Seznam ilustrací

Seznam příloh

Příloha A – DVD

V této příloze je možné nalézt složku „projekty“, obsahující ukázkové projekty, které byly vytvářeny v prostředí NetBeans IDE 6. 7. 1, jakožto praktická část této bakalářské práce. Samozřejmě každá složka konkrétního projektu obsahuje mimo jiné i podsložku „src“ ve které jsou obsaženy balíčky (jsou-li součástí projektu), obrázky a v neposlední řadě zdrojové kódy zahrnující rozhraní a třídy ve formátu textového souboru. Dále také příloha obsahuje složku „UML“, kde jsou vygenerovány obrázky class diagramů jednotlivých projektů, pomocí pluginu UML v prostředí NetBeans.