Technická univerzita v Liberci

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 – Elektrotechnika a informatika

Studijní obor: 2612R011 – Elektronické informační a řídicí systémy

Grafická aplikace pro internet v jazyce Java

Graphics application for the Internet in the Java language

BAKALÁŘSKÁ PRÁCE

Autor bakalářské práce:	Jan Šedivý
Vedoucí bakalářské práce:	Ing. Roman Špánek
Konzultant:	Ing. Pavel Pirkl

V Libereci 1. 5 2007

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Katedra softwarového inženýrství Akademický rok: 2006/2007

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: Jan Šedivý

studijní program: B 2612 – Elektrotechnika a informatika

obor: 2612R011 – Elektrotechnické informační a řídící systémy

Vedoucí katedry Vám ve smyslu zákona o vysokých školách č.111/1998 Sb. určuje tuto bakalářskou práci:

Název tématu:

Grafická aplikace pro intrnet v jazice Java

Zásady pro vypracování:

1. Student by se v první části měl věnovat možnostem jazyka Java

2. Měl by podrobně prostudovat využití Java jako „Applet“

3. Vytvořit aplikaci, která by zobrazovala 2D scény

4. Doplnit aplkaci o dynamické zobrazování ve 2D

Rozsah grafických prací: dle potřeby dokumentace

Rozsah průvodní zprávy: cca 40 stran

Seznam odborné literatury:

[1] Internetové stránky Sun: http//java.sun.com/

[2] Bruce Eckel: Myslíme v jazyku Java, Grada Publishing, ISBN: 80-247-9010-6, 2001

[3]Peter Druska: CSS a XHTML, Grada Publishing, ISBN: 80-247-1382-9, 2006

Vedoucí bakalářské práce: Ing. Roman Špánek

Konzultant: Ing Pavel Pirkl

Zadání bakalářské práce: **18.10.2006**

Termín odevzdání bakalářské práce: **18. 5. 2007**

V Liberci dne 19.10.2006

Prohlášení

Byl jsem seznámen s tím, že na mou Bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Poděkování

V úvodu své práce bych rád poděkoval Ing. Špánkovi za metodické pokyny a podnětné návrhy ohledně konceptu a struktury celé své práce.

Dále bych chtěl poděkovat celé své rodině za všestrannou podporu při mém vysokoškolském studiu.

Anotace

Tato bakalářská práce se převážně zabývá možnostmi jazyka Java pro vytváření 2D grafických scén. Tyto scény mají být schopné dynamického zobrazování jednoduchých grafických objektů. Je zde podrobně objasněno použití nejběžnějších tříd balíku Java2D API. Cílem není vysvětlení všech možností Java2D API, ale seznámení s možnostmi a postupy při jejich použití pro Java aplety. Dále jsou tu rozebrány právě tyto Java aplety, který slouží pro vytvoření zajímavějšího a poutavějšího uživatelského rozhraní ve webové stránce. Jedním z cílů této práce je ukázat jak pomocí Java apletu zobrazit 2D scény ve webovém prohlížeči a tak i v prostředí Internetu. Celkově tato Bakalářská práce obsahuje potřebné informace pro vytvoření Java apletu který zobrazuje jednoduché 2D animace.

Annotation

This Bachelor thesis deals mainly with the possibilities of the Java language for the creation of 2D graphic scenes. These scenes shall be capable of dynamically displaying simple graphic objects. The use of the most common classes of the Java2D API package is clarified herein in detail. The aim is not to explain all the possibilities of Java2D API, but to acquaint with the possibilities and procedures in using them for Java applets. Furthermore, just these Java applets that are used to create a more interesting and attractive user interface in a web site are analysed herein. One of the goals of this thesis is to show how to display 2D scenes in a web browser and thus also in the Internet environment by means of a Java applet. Overall, this bachelor thesis contains necessary information for the creation of a Java applet that displays simple 2D animations.

Obsah

Úvod

Java je jedním z nejrozšířenějších a nejúspěšnějších jazyků posledních let. Našla své uplatnění snad ve všech oborech, kde se programování využívá. Používá se pro běžné kancelářské aplikace, sofistikované matematické programy či lékařský software. A v neposlední řadě pak také na hry a to nejen pro mobilní telefony. Tento interpretovaný jazyk přinesl webu možnost psaní tzv. Java apletů, malých aplikací začleněných přímo do HTML stránky. Již první verze programovacího jazyka Java v sobě obsahovala balík, který umožňoval základní práci s grafikou pomocí primitivních grafických elementů a práci s nejrozšířenějšími grafickými formáty GIF a JPG. Ve skutečnosti však nebylo možné s grafickými daty nijak dále pracovat, pouze je načíst a zobrazit. Java 2D API rozšiřuje tyto možnosti.

Tato práce obsahuje potřebné informace pro vytvoření funkčního apletu. Dále je zde ukázáno jak vytvářet 2D scény pomocí Java 2D API v programovacím jazyce Java. Jsou tu také popsány běžně používané funkce Java 2D API a metody objektů. Tento souhrn má za úkol ukázat základní informace pro snazší orientaci v API a ukázat co Java 2D API nabízí. Popsány jsou například funkce pro práci s obrazovými daty a také které umožňují kontrolu nad vlastním renderováním grafiky. Java 2D přidává další třídy pomocí nichž lze definovat nejen barvy, způsoby vyplňování objektů ale i fonty a vlastní grafická primitiva. Nové třídy také podporují základní funkce pro zpracování obrazových dat, jakými jsou například geometrické transformace, ostření a rozmazávání obrazu, úprava kontrastu apod.

.

1. Java

1.1. Java historie

Java je poměrně nový programovací jazyk; veřejnosti byl představen v roce 1995 a v současné době se uvádí, že ročně přibudou po celém světě asi 4 miliony programátorů, kteří ho používají. Vytvořil ho vývojový tým firmy Sun Microsystems vedený J. Goslingem. Java byla původně určena jako programovací jazyk pro tvorbu softwaru spotřební elektroniky, jako jsou ledničky, mikrovlnné trouby, pračky, osobní digitální diáře, videa apod. Později se ukázalo, že Javu lze snadno a dobře používat v prostředí Internetu, zejména vzhledem k možnostem zabezpečení, a to se stalo základem jejího úspěchu. Dnes se používá v mnoha oblastech. Vedle běžných aplikací (programů) se využívá pro programování tzv. apletů (krátké programy, které lze vkládat do webových stránek), serviety (programy, které běží na webových serverech a vytvářejí dynamické stránky), aplikace tzv. střední vrstvy a mnohé další.

1.2. Vlastnosti Javy

Jazyk Java je jednoduchý, objektově, orientovaný, robustní, bezpečný nezávislý na architektuře, přenositelný, vysoce výkonný,.výceprocesní a dynamický jazyk. Na první pohled tato definice působí velmi formálním dojmem. Avšak každý z těchto přívlastků poměrně elegantně vyjadřuje jednu důležitou vlastnost tohoto programovacího jazyka. Dále jsou podrobněji popsány tyto jednotlivé vlastnosti Javy.

Jednoduchost

Java je jednoduchý jazyk. Jedním z cílů návrhu Javy bylo vytvořit jazyk, který se programátor velmi rychle naučí. Proto jazyk obsahuje jen minimální množství jazykových konstrukcí. Navíc jsou tyto konstrukce odvozeny od existujících jazyků, čímž je usnadněno chápání jazyka při migraci. Java nepodporuje ani struktury, sjednocení (union), přetížení operátorů ani vícenásobnou dědičnost. Není podporována ani ukazatelová aritmetika. Java tak předchází většině chyb vyskytujících se v jiných programovacích jazycích (C/C++, Pascal, Delphi).

Objektová orientace

Java byla od začátku koncipována jako objektově orientovaný jazyk. Java je plně objektově orientovaný jazyk, s výjimkou osmi primitivních datových typů jsou všechny ostatní datové

typy objektové. Třídy jsou uspořádány do hierarchií od obecnějších tříd směrem ke konkrétnějším implementacím. Java sama přichází s kompletním balíkem tříd, které se můžou snadno v programech použít.

Distribuce

Už vzhledem ke svému původnímu zaměření je Java navržena s podporou síťových aplikací. Java podporuje různé úrovně síťového spojení. Jedním z příjemných rysů Javy je, že otevření souboru vzdáleného objektu na Internetu není nic těžkého, prakticky je poskytována stejná funkčnost jako při manipulaci s lokálním zdrojem dat. V Javě je také snadné vytvářet programy používající kanály (sockets).

Interpretace

Zdrojové kódy Javy nejsou kompilátorem přeloženy přímo do strojového kódu, ale do tzv.bajt kódu. Je tím zajištěna přenositelnost programu. Ta se opírá o existenci virtuálního stroje Javy, který instrukce uvedené v bajt kódu vykoná. Java se řadí mezi interpretované jazyky, nicméně díky pokročilým technologiím není odstup rychlosti vykonávání programů od programů přeložených přímo do strojového kódu nijak propastný. Navíc program v Javě je možné spustit prakticky na jakékoli platformě, kde je dostupná příslušná verze virtuálního stroje.

Robustnost

Java byla od samých počátků navržena pro psaní softwaru určeného pro spotřební elektroniku. Proto jazyk sám musí podporovat vytváření spolehlivého a robustního softwaru. Samozřejmě lze v Javě napsat i nespolehlivý program, nicméně díky návrhu jazyka jsou vyloučeny některé nejčastější typy problémů. Důležitou skutečností je, že Java je silně typový jazyk, jsou vyžadovány explicitní deklarace proměnných a díky tomu je možné odhalit chyby již v době překladu do bajt kódu. Výhodou v tomto směru je existence správce paměti (garbage collector), který automaticky slučuje volné části paměti, stará se o odstraňování nepoužitých objektů a předchází vytváření „děr" v paměti. Java také podporuje strukturované ošetření výjimek, které vede k průhlednějšímu kódu.

Bezpečnost

Java obsahuje bezpečnostní mechanismy, které chrání uživatele proti kódu pracujícímu jako virus nebo červ, který by mohl napadnout systém. Všechny bezpečnostní mechanismy jsou založeny na předpokladu, že ničemu a nikomu se nedá věřit. Java nepodporuje práci s ukazateli, programátor se nemůže dostat za „oponu" a přesměrovat ukazatele na jinou oblast paměti. Stejně tak překladač nerozhoduje o rozmístění paměti, tak že programátor nemůže z deklarace třídy vytušit její umístění v paměti a pokusit se ji narušit. Podstatným rysem je také verifikace bajtového kódu načítaného ze sítě. Runtime systém přitom zajistí, že nebudou porušena jazyková omezení Javy.

Nezávislost na architektuře

Programy v Javě nejsou překládány přímo do strojového kódu daného procesoru, ale do formátu bajtového kódu. Hlavní výhodou této architektury je možnost spouštění aplikace na různých operačních systémech a hardwarových platformách bez nutnosti překladu pro cílovou platformu. Proto je v Javě snadnější vyhovět požadavku psaní aplikací použitelných bez větších rozdílů na MS Windows, Apple, Linuxu a klonech UNIXu.

Přenositelnost

Nezávislost na architektuře není to samé jako přenositelnost, ale pouze její velkou částí. Přenositelnost musí navíc zamezit existenci jakýchkoli implementačně závislých aspektů jazyka. Specifikace jazyka proto kromě jiného přesně určuje velikost primitivních datových typů a jejich chování při aritmetických operacích. Také samotné běhové (runtime) prostředí Javy musí být přenositelné na různé hardwarové platformy a systémy. Hranice přenositelnosti jsou opřeny o normu POSIX (Portable Operating System Interface). POSIX je standard pro rozhraní nezávislé na dodavateli mezi operačním systémem a aplikacemi včetně definice rozhraní a zdrojových souborů.[1]

[1] Bližší informace o normě POSIX jsou na webové adrese: http://standards.ieee.org/regauth/posix/.

Výkonnost

Java je interpretovaný jazyk. Přesto je její rychlost více než přijatelná pro vykonávání běžných programů. V dnešní době totiž mnoho programů často většinu času pouze nečinně očekává vstup od uživatele, nebo na data z databáze či ze sítě. Protože existují situace, kdy se hodí každý výkonnostní stupínek, obsahuje Java interpret typu Just-In-Time (právě včas, JIT). Ten překládá dynamicky na základě potřeb části kódu přímo do strojového kódu procesoru. Nestačí-li to, může se použít nástroje, které přeloží bajt kód přímo do spustitelné formy, nicméně s výhodou zrychleného programu ztrácíte přenositelnost a robustnost programu.

Podpora vláken

V dnešní době je více než běžné vytvářet více vláknové aplikace. Uživatel takové aplikace může stahovat nějaký soubor ze sítě a během této doby zpracovat jinou úlohu a ještě přitom poslouchat hudbu nebo sledovat animaci. Také Java podporuje psaní programů s vlákny, neboť má vestavěnu podporu tohoto typu úloh, včetně možnosti vyloučení souběžného vykonání úlohy dvěma vlákny současně.

Dynamičnost

Zajímavou vlastností je také dynamičnost. Program může zavádět třídy do paměti podle potřeby a ty ještě mohou být umístěny na síti. Třídy v Javě mají také svoji jedinečnou reprezentaci v době běhu programu. Předáte-li programu objekt, je možné zjistit, ke které třídě objekt patří. Děje se tak prověřením run-time informací o typu. Tento rys také umožňuje dynamické linkování tříd do běžícího systému. Díky dynamičnosti lze i za běhu aplikace zajistit upgrade softwaru.

1.3. Způsob zpracování programu v Javě

Standardní postup je ten, že program v Javě prochází pěti fázemi - editováním, překladem (kompilací), zavedením (load), ověřováním (verifikací) a prováděním. Čtyři z těchto fází jsou běžné i v ostatních programovacích jazycích. Fáze ověřování je něco nového, ale pro Javu (a zejména programování na WWW) velmi důležitého - umožňuje totiž dosáhnout velmi vysoké bezpečnosti spuštěného programu, čímž je míněna hlavně ochrana toho, kdo program spouští.

Další zvláštností Javy je, že překlad neprobíhá do jazyka relativních adres, který je v podstatě totéž, co strojový jazyk počítače, ale do pseudojazyka nazývaného *byte-code.* Tento jazyk je nezávislý na cílovém počítači, což prakticky znamená, že programátora nemusí vůbec zajímat, na jakém počítači jeho program poběží. Přeložený program „bajtkód“ je uložen v souborech s vyhrazenou příponou *class*. Tento soubor je pak z disku zaváděn do paměti počítače a současně probíhá ověření „bajtkódu“, což je možné provést jednotně díky nezávislosti „bajtkódu“ na platformě. Po ověření je program spouštěn pomocí interpreteru.

1.4. Java platformy

Existují tří základní typy Java platformy

Java Platform, Standard Edition (Java SE). Je standardní Java, toto označení vzniklo z důvodu odlišení od dalších verzí. Další její označení je Java2, J2SE. Před koncem roku 2006 společnost Sun Microsystems zveřejnila novou verzi platformy Java Standard Edition 6 (Java SE 6).

Java Platform, Enterprise Edition (Java EE), což je platforma pro psaní podnikových aplikací, která rozšiřuje Java .

Java Platform, Micro Edition (Java ME), která je určena speciálně pro psaní aplikací do mobilních zařízení (telefony, PDA apod.).

Java platforma se skládá ze dvou hlavních částí. První část tvoří tzv. virtuální stroj (Java Virtual Machine - JVM), který se skládá z části zajišťující vazbu na hardware a z části interpretující bajtkód. Tento interpreter může být nahrazen JIT kompilátorem. Druhou část Java platformy tvoří Java Core API.

1.4.1. Java Virtual Machine

Virtuální stroj jazyka Java je software, který vystupuje jako hardwarové zařízení a který umožňuje spouštění programů v jazyce Java . Převádí jejich bajtový kód do nativních instrukcí pro hostitelský počítač JVM je označován jako virtuální stroj, protože neexistuje žádný hardware, který by bajtový kód interpretoval. Problémem interpretovaných jazyků je jejich pomalost ve srovnání s kompilovanými jazyky. Java tento problém částečně řeší použití tzv. JIT kompilátorů (Just In Time), které v době zavádění programu z disku do paměti

počítače (po ověření správnosti bajtkódu) jej přeloží on-line do strojového jazyka konkrétního počítače, čímž z něj prakticky vyrobí v paměti .EXE program. Ten pak běží stejnou rychlostí, jako kterýkoliv jiný kompilovaný program napsaný třeba v C.

1.4.2. Java Core API

API (Application Programming Interface) je předem připravený kód, který je uspořádán do tématicky jednotných balíčků. Pod zkratkou API se skrývá značné množství knihovních tříd, které jsou považovány za standardní, čili musí se vyskytovat v každém prostředí, kde se Java používá. To znamená, že když program využívá metody z API, není jejich kód součástí programu, protože je součástí API. Prakticky to znamená, že programy obsahují pouze kód, který jsme napsali, a soubory, ve kterých jsou přeložené programy uloženy, mají proto poměrně malou velikost. Všechny třídy, jejich metody a proměnné jsou velmi dobře zdokumentovány a dokumentace je přístupná pomocí WWW prohlížečů.

2. Aplet

2.1. Možnosti apletů

Aplet (applets) je aplikace napsaná v Javě, která je umístěna. do HTML stránky. Pomocí apletů lze velmi dobře zobrazovat ve 2D téměř jakýkoli problém. Bitmapové obrázky lze nahrávat a dalším způsobem je zpracovávat. Je možné používat i vektorovou grafiku. K tomu, aby bylo možné tyto aplety vytvořit a zobrazit v prohlížeči, je třeba mít nainstalované Java Development Kit (JDK). JDK je soubor základních nástrojů pro vývoj aplikací pro platformu Java. Někdy bývá označován jako Java SDK. Java programy a aplety jsou spouštěné pomocí již zmíněného JVM (Java Virtual Machine), neboli jakéhosi virtuálního stroje. Ten je obsažen v SDK společně s API. Java nebývá standardně s prohlížeči dodávána. SDK je nutné mít nainstalované pro tvorbu apletu. Pro spuštění apletu nemusí být nainstalované celé SDK, postačí jen jeho část tzv. JRE (Java Runtime Enviroment). JRE obsahuje pouze nejnutnější nástroje pro spuštění Java aplikace nebo apletů a tak za pomocí něho nelze aplikace vytvářet.

Aplet se od běžné Aplikace především liší tím, že neběží samostatně, ale je začleněn v HTML stránce, kde je mu vyhraněna obdélníková oblast. Další rozdíly jsou nejen ve fázi zavedení kde je zaveden do paměti počítače pomocí prohlížečů (Netscape, MSIE, Mozila), ale i ve fázi ověření kde aplet má přísnější pravidla. Z bezpečnostních důvodů platí pro aplet některá omezení a naopak má aplet některé funkce rozšířeny:

- Aplet nemůže nahrávat knihovny ani definovat nativní metody.
- Aplet nemůže navazovat síťové spojení na jiný než domovský server.
- Aplet nemůže zapisovat do souborů na straně klienta (prohlížeče).
- Aplet nemůže spouštět programy na domovském serveru.
- Aplet nemůže číst některé systémové proměnné.
- Aplet může přehrávat zvuky.
- Aplet může požádat browser o zobrazení libovolné WWW stránky.
- Aplet může volat veřejné metody apletů umístěných na téže WWW stránce

Takové nepovolené metody obvykle vyhazují programovou výjimku java.lang.SecurityException. Některé prohlížeče umožňují uvedená omezení nastavit individuálně pro vybrané aplety. Každý aplet je odvozen od třídy Aplet z balíku java.aplet. Kromě toho je zapotřebí importovat balík java.awt, který umožní apletu používat a zobrazovat grafické uživatelské prostředí AWT(Abstract Windows Toolkit).

2.2. Metody apletu

Narozdíl od desktopové aplikace nepotřebuje aplet metodu *main(),* protože není prohlížečem spouštěn. Sám prohlížeč (nebo alespoň jeho část zodpovědná za applety) je naopak Java aplikace a jednotlivé applety pouze dynamicky připojuje do svého adresového prostoru. Aplet tedy nemusí (a zpravidla nemá) metodu *main ()*. Zato by měl mít metody *init(), destroy(), start(), stop()* a některé další. Prohlížeč volá podle potřeby jeho metody. Konstruktor volá prohlížeč při zavedení apletu do paměti (po natažení WWW stránky, která aplet obsahuje). Obvykle inicializuje nestatické proměnné apletu.

Metoda void init()

Tuto metodu zavolá prohlížeč ihned po konstruktoru. Jejím úkolem je postarat se o inicializaci apletu. Typicky načte parametry apletu z HTML. Je volána prohlížečem při inicializaci, start apletu. Typicky se v ní provádí inicializace proměnných, vykreslení potřebných komponent apod.

Metoda void destroy ()

Metoda *destroy ()* představuje v jistém smyslu opak metody *init()*. Připravuje aplet k zániku. Prohlížeč ji volá těsně před uvolněním apletu z paměti. Nemusí obsahovat žádné akce.

Metoda void start()

Spustí činnost apletu. Tuto metodu je volána ihned po skončení metody *init()* a pak ji volá vždy, když se aplet vrátí na plochu okna prohlížeče. Rozdíl mezi metodami *start()* a *init()* je v tom, že *init()* je volána pouze jednou, kdežto *start()* je pokaždé, když se aplet dostane do „zorného pole“ prohlížeče. Je volána např. při obnovení okna, ale také tím, že je aplet na konci delšího HTML souboru, který postupně prohlížíme. Metoda *start()* je proto vhodné místo pro spouštění různých vláken, která provádějí animaci, která se mají zastavit při minimalizaci a opět rozběhnout při obnovení okna. Jejím opakem je metoda *stop()*.

Metoda void stop()

Tato metoda zastaví činnost apletu. Prohlížeč ji zavolá vždy, když aplet opustí viditelnou plochu okna. Je např. zavolána proto, že se uživatel přesune na stránce o kus dále. Pokud aplet předvádí nějakou animaci, metoda *stop()* ji zastaví, protože je zbytečné, aby tato animace

zatěžovala systém, když ji uživatel stejně nevidí. Když se později uživatel vrátí zpět a plocha apletu se znovu objeví na obrazovce, zavolá prohlížeč metodu *start ()* a tím obnoví činnost apletu. Jestliže uživatel opustí HTML dokument, který tento aplet obsahoval, zavolá prohlížeč nejprve metodu *stop()*, která aplet zastaví, a teprve pak zavolá *destroy ()*.

Metoda StringgetAppletInfo

Tato metoda vrací znakový řetězec obsahující informace o apletu, které bychom mohli použít např. v dialogovém okně programu, pokud bychom aplet program spustili jako samostatnou aplikaci. Aplety obvykle nic podobného nenabízejí. Používání dialogových oken se v apletech nedoporučuje.

Metoda StringgetParametr

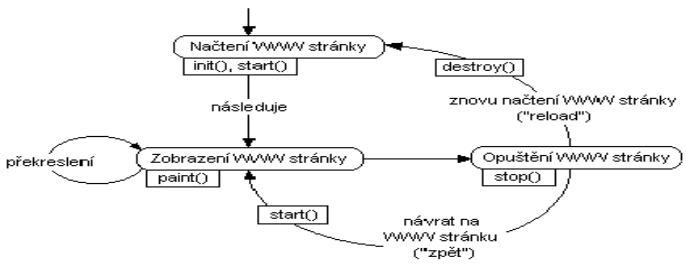
Hlavička této metody má tvar String getParameter(String jméno) a má za úkol přečíst hodnotu parametru uvedeného v příkazu PARAM v HTML stránce.

Metoda void paint( )

Tato metoda byla velice důležitá u apletů v JDK 1.x (odvozených od třídy Applet). Volala se vždy, když bylo potřeba překreslit obsah okna - např. proto, že uživatel část plochy apletů zakryl jiným oknem nebo když si to program vyžádal voláním metody *repaint()*.

Hlavička této metody má tvar *void paint(java.awt. Graphics g)*, kde Graphics je třída představující obecnou „kreslicí plochu". Její metody umožňují kreslit běžné geometrické tvary, vkládat text atd.

Následující obrázek znázorňuje volání metod apletu v závislosti na stavu prohlížeče:



obrázek 1: volání metod apletu

2.3. Vložení apletu do HTML stránky

Způsoby vložení apletu

Chceme-li spustit applet v prohlížeči, musíme vložit do stránky HTML tag. Jsou používány hlavně dva druhy tagu, se značkou <applet> a <object>. Tag <object> byl dříve doporučován a měl nahradit značku <applet>. Tag <object> je vhodný jen pro prohlížeč Intrnet Exploer, při použití v prohlížeči Mozila Firefox se daný aplet nezobrazí. Tento problém je řešen použitím Javascriptu pro zjištění typu použitého prohlížeče. Pokud Javascrip pozná Microsoft Internet Explorer použije tag <object>, v případě webových prohlížečů typu Netscape je použit tag <embed>. Tag <embed> Je tedy vhodný v případě vložení apletu pro prohlížeče typu Mozila. V současné době je doporučován tag <applet> dovoluje aby se vkládaný applet zobrazil bez ohledu na použitý prohlížeč. Tento tag je nejpoužívanější, díky nezávislosti na použitém webovém prohlížeči. Pro jeho funkční aplikaci v prohlížeči musí být nainstalovaná verze Java Plug-in 1.3.1 nebo pozdější[2]. Nyní je už k dispozici Java Plug-in 1.6.0 .

Jakmile prohlížeč načte tento tag, spustí tzv. Class Loader (součástí JVM), jenž ze serveru stáhne kód apletu a kód všech potřebných tříd, na než se aplet odkazuje a které ještě na klientovi nejsou dostupné. Jsou-li všechny požadované soubory dostupné, je aplet spuštěn. Aby byla zajištěna nezbytná bezpečnost při běhu apletu, omezuje dostupnou funkcionalitu tzv. Security Manager, který umožňuje uživateli zvolit, jaké skupiny funkcí mají být povoleny či zakázány. Obvykle nemají applety přístup do souborového systému a nemohou navazovat spojení s jinými servery, než odkud byly nahrány. Apletu je ve stránce vyhrazen obdélníkový prostor (velikosti určené v atributech tagu) který je plně pod jeho kontrolou a prohlížeč jeho vykreslování nijak neovlivňuje.

Značka <applet>

Aplet lze vložit do HTML dokumentu tím, že do jeho kódu vložíme tag <applet>. Obsah tohoto tagu ukazuje následující HTML kód.

```
<APPLET>
CODE     =“Aplet1.class“
WIDTH    = “šířka“
HEIGHT   = “výška“
[CODEBASE = "URL"]
```

[2] Přehled podpory tagu <applet> v jednotlivých webových prohlížečích je na webové adrese http://kofler.dot.at/browsersupport/java.html. Přehled pro tagy <object> a <embed> je na adrese http://kofler.dot.at/browsersupport/plugin.html.

```
[ARCHIVE = "JARsoubor1, JARsoubor2"]
[NAME     =“jméno instance apletu“]
[HSPACE   =“vodorovné odsazení“]
[VSPACE   =“svislé odsazení“]
[ALIGN    =“zarovnání“]
[ALT      =“alternativní text]

[<PARAM NAME="jméno1" VALUE="hodnota1">]
[<PARAM NAME="jmeno2R" VALUE="hodnota2">]
[alternativní HTML]
</APPLET>
```

Značka <*applet*> slouží k stažení a spouštění apletu z HTML stránky. Tato značka zároveň definuje oblast, kterou bude v ploše dokumentu zabírat zobrazený aplet. Vše za značkou <*applet*> až po ukončení, tedy po značku <*/applet*>, je zadání apletu. Atribut code je povinný určuje přitom jméno souboru. Atribut *codebase* definuje alternativní nebo základní adresu URL, z níž se pokusí načíst třídy a JAR soubory apletu. Není-li použito, nebo je tam tečka, je hodnota *codebase* URL adresáře, odkud byl načten HTML soubor odkazující na applet. *Name* je jméno, kterým bude tento aplet označován v hlášeních některých prohlížečů. Udává jméno instance apletu, na které je pak možné odkazovat metodu getApplet(), která slouží pro komunikaci apletů. *Width* udává šířku grafického výřezu (v pixelech) na WWW stránce, pro applet. *Height* udává výšku grafického výřezu. *Hspace* a *vspace* určují velikost odsazení (v pixelech) kterou prohlížeč vynechá okolo plochy apletu. *Align* znamená horizontální zarovnání apletu ve stránce; hodnota:left, right, top, texttop, middle absmiddle, baseline, bottom, absbottom. *Archive* specifikuje jeden nebo více komprimovaných souborů JAR, které se budou nahrávat při načtení HTML dokumentu. Tyto archívy mohou obsahovat *.class* soubory a další data apletu. Jejich použitím se výrazně zkrátí celková doba stahování ze sítě k souborům z JAR archívů se z programu přistupuje jako by se nalézali v adresáři relativně od *cudebase*. *Alt* obsahuje text, který prohlížeč zobrazí v případě, že rozeznává značku <*applet*> , ale nepodporuje grafický režim.

Specifikace parametrů začíná slovem *param*. Pak následuje klíčové slovo *name* určující jméno parametru, rovnítko a jméno v uvozovkách. Pod tímto jménem bude aplet hledat tento parametr pomocí funkce *getParameter()*. Pak následuje klíčové slovo *value* a za rovnítkem hodnota parametru.

2.4. JAR archív

Použití JAR archívu

JAR je zkratka od Java Archiver a označuje způsob archivování souborů včetně jejich komprese. Je použit kompresní algoritmus stejného principu jako je ZIP. Je samozřejmé, že soubory JAR jsou nezávislé na platformě, tj. na počítači s operačním systémem. Typický soubor JAR obsahuje všechny přeložené. *class* soubory plus všechny potřebné další zdroje, jako jsou obrázky, pomocné sobory, konfigurační soubory, audiosoubory atd.

Archív JAR se vytváří z několika dobrých důvodu:

1) Vše potřebné je vjednom souboru. Když je vytvořen jakýkoliv program o málo složitější než triviální, bude v něm použito několik tříd a vnitřních anonymních tříd. Přenést vzniklé .class soubory (a případné další soubory ikon) a na žádný nezapomenout není zcela triviální záležitost, zejména když je použito více balíků, tzn. i více podadresářů.

2) Soubor JAR může být pojmenován dle našich potřeb. To je neocenitelná výhoda zejména tehdy, je-li v distribučním řetězci něco, co vyžaduje jen osmiznaková jména souborů a tříznakovou příponu nebo nerozlišuje velká a malá písmena.

3) Úspora místa. U souborů *.class* se zcela běžně dosahuje o 40 % (tj. na 60 % původní velikosti souboru), což samo o sobě nemusí být tak ohromující číslo. Je třeba si ale uvědomit i to, že na disku je pro uložení souboru potřeba větší místo. To ve skutečnost znamená, že pro mnoho malých souborů (což jsou typicky soubory *.class*) na velkém disku dosáhneme zmenšení až na jednu čtvrtinu místa.původně využitého místa.

4) Zmenšení času natahování a natažení na jeden přístup. To má význam zejména u apletů, přenášených postupně HTTP protokolem, kdy u nezapakovaných souborů *class* je každý soubor přenášen v jedné HTTP transakci.

5) Zjednodušené spouštění pro uživatele apletů i aplikací. Chceme-li spustit aplikaci, není nutné archiv vůbec rozbalovat, ani se starat, ve kterém souboru je hlavní třída.

Spuštění programu z JAR archivu

Pro spuštění programu nemusíme archiv nutně rozbalovat. Toto je důležité zejména u apletů, kdy si WWW prohlížeč příkazem uvedeným v HTML souboru natáhne soubor archivu a není nikdo, kdo by zadal příkaz k jeho rozbalení.

Spuštění apletu je jednoduší než u aplikace. Stačí jen napsat jednu řádku v HTML souboru. Do značky <*applet*> se přidá parametr *archive* s názvem JAR archívu a značka code s označením „hlavní“ soubor .*class* (viz kapitola 3.3 Značka <*applet*>). Při použití JAR archívu záleží samozřejmě na tom, zda WWW prohlížeč podporuje *archive*. Pokud ne, pak nelze tento způsob použít. MS Internet Explorer minimálně od verze 4.0,prohlížeč Opera od verze 7, a od verze Firefox 1.5 tento parametr podporuje.

Je samozřejmé, že jak u apletů, tak i u aplikace dojde před samotným spuštěním programu ke skrytému rozbalení archivu, protože JVM ke své činnosti potřebuje soubory . *class*. Ale toto rozbalení je dočasné a uživatel se o něj jednak nemusí starat a také se mu na disku nevytvářejí další soubory ani adresáře.

3. 2D zobrazení v Javě

3.1. Třída Graphics

Graphics je abstraktní třída která reprezentuje rozhraní pro práci s grafikou . Práce s grafickým výstupem probíhá tzv. grafickým kontextem, což je instance třídy java.awt.Graphics. Grafický kontext má přiřazen každá zobrazená komponenta. Výstupním zařízením je pak například zobrazovaná plocha komponenty Canvas (prázdná uživatelsky definovaná komponenta), jenž je reprezentována třídou Canvas. Při požadavku na vykreslení komponenty je zavolána metoda paint nebo update tohoto objektu a předá se mu instance grafického rozhraní to jest třída Graphics (*update(java.awt.Graphics g)* a *paint(java.awt.Graphics g)*).

Třída Graphics definuje metody pro kreslení grafických primitiv (čára - *drawLine()*, ovál - *drawOval()*, polygon - *drawPolygon()* apod.), výstup textu - *drawString()*, vykreslení obrázku - *drawImage()* atd. Třída Graphics umožňuje kreslit jen na pravoúhlé komponenty s tím, že kreslení se provádí pomocí souřadnic x a y. Počátek souřadnic je v levém horním rohu, souřadnice x se zvětšuje směrem vodorovně doprava. Souřadnice y se zvětšuje svislým směrem dolů. Maximální hodnota x a y je dána velikostí komponenty metodami *getWidth()* pro x a *getHeight ()* pro y. Hodnoty u celočíselné (typu *int*) a představují počet pixelů.

3.2. Třída Graphics2D

Pro vytváření programů s poměrně jednoduchou grafikou sice třída java.awt.Graphics postačí, ale pro náročnějšího uživatele má poměrně omezené použití. Od JKD 1.2 je k dispozici rozšíření java.awt.Graphics2D. Java2D obsahuje oproti předchozí verzi mnoho vylepšení. Mezi nimi je především vylepšena práce se základními tvary, možnost definice barev a jejich skládání, jednotný kreslicí model pro výstup na tiskárny i grafické zařízení (doposud bylo různé), podpora načítání a ukládání výstupu obrázků z a do souborů či umožnění filtrování obrázku. Tato knihovna byla navržena nejen tak, aby byla zpětně kompatibilní se svým předchůdcem, ale aby dokázala spolupracovat s jinými API, jenž pracují s grafikou to jest Java Media Framework, Java Advanced Imaging API a jiné Java Media API. Jak jsme již zmínili, Java2D je zpětně kompatibilní a tak se nezměnil ani způsob získávání instance grafického rozhraní. Na místo toho je nám dávána instance třídy Graphics2D avšak přetypována na třídu Graphics. Třída Graphics2D reprezentuje nové grafické rozhraní Java2D.

Proto tedy před vlastní prací s rozšířeními Java2D musíme Graphics přetypovat na Graphics2D.

```
public void paint (java.awt.Graphics g) {
java.awt.Graphics2D g2d = java.awt.Graphics2D) g;
…}
```

3.3. Grafická primitiva

Java 2D API nabízí třídy které definují základní geometrické prvky jako bod, přímka, oblouk obdélník. Tyto třídy jsou v balíku Java.awt.geom. Užitím geometrických tříd můžeme lehce definovat geometrická 2D tělesa a poté s nimi manipulovat. Grafická primitiva implementují především dvě rozhraní a to *Shape* a *PathIterator*

PathIterator definuje metody pro načítání souborů do paměti. Obsahuje metody, kterými "říká", jak má být tvar vykreslován (teď rovná čára, teď parabola atd.).

Shape poskytuje metody pro popis geometrických objektů, implementuje většinu geometrických tříd

3.3.1. Třída Point2D

Třída point2D představuje bod, který je reprezentovaný svou x-ovou y-ovou souřadnicí. Pomoci podtříd Point2D lze využít zvětšení přesnosti bodu. Podtřídy zvyšují přesnost bodu tím, že jednotlivé souřadnice nemusí být typu int, ale i typu float nebo double. Třída Point2D obsahuje také metody jak vypočítat vzdálenosti mezi dvěma body.

```
//vytvoření bodu Point2D.Double
Point2D.Double point = new Point2D.Double(x, y);
```

3.3.2. Třída Line2D

Jedná se o nejjednodušší tvar který je vykreslen metodou *drawLine(int xl, int yl, int x2, int y2)*. Ta vykreslí úsečku z [xl, yl] do [x2, y2]. Třída má také své podtřídy pro zlepšení přesnosti Souřadnice můžou být typu float(Line2D.Float) nebo double (Line2D.Double).

```
// vytvoření Line2D.Double
Line2D.Double usecka = new Line2D.Double(x1, y1, x2, y2);
```

Přímku se nemusí pokaždé vytvářet znovu. Obsahuje několik nastavovacích metod pro změnu koncových bodů.např. *setLine(Point2D p1, Point2D p2);*.

3.3.3. Křivky

Balík java.awt.geom umožňuje vytvořit kvadratickou nebo kubickou křivku

Třída QuadCurve2D

Kvadratická křivka je implementována v třídě QuadCurve2D. Tato třída představuje Beziérovu křivku určenou dvěma koncovými body a jedním řídícím bodem. Obsahuje opět podtřídy QuadCurve2D.Float a QuadCurve2D.Double.

```
//vytvoření
 new QuadCurve2D.Float QuadCurve2D q = new QuadCurve2D.Float();
// vykreslení QuadCurve2D.Float s nastavením souřadnic
g2.draw(q.setCurve(x1, y1, x2, y2, ridicix, ridiciy));
```

Třída CubicCurve2D

Reprezentuje kubické parametry křivky v homogenních souřadnicích. Také obsahuje podtřídy CubicCurve2D.Float a CubicCurve2D.Double pro zlepšení přesnosti. Od kvadratické křivky se hlavně liší tím, že kubická křivka má dva kontrolní body. Jinak tato křivka má metody podobné jako kvadratická křivka.

```
// vytvoření  nové CubicCurve2D.Double
CubicCurve2D c = new CubicCurve2D.Double();
// vykreslení CubicCurve2D.Double s souřadnicemi
g2.draw(c.setCurve(x1, y1, x2, y2, ctrlx1, ctrly1, ctrlx2, ctrly2));
```

3.3.4. Třída Rectangle2D

Třída Rectangle2D slouží pro vykreslení obdélníku. Má také své podtřídy Rectangle2D.Double a Rectangle2D.Float. Pro vykreslení můžeme požít několik metod. *g2.draw(Shape s)* tato metoda vykreslí obrys, s předem definovaným typem čary a danou barvou. Metoda *g2.fill(Shape s)* vykreslí tvar s definovanou výplní. Výplň nemusí být jen jednobarevná Java2D umožňuje také vytvářet textury nebo gradientní barevné přechody.

```
// vykreslení Rectangle2D.Double
g2.draw(new Rectangle2D.Double(x, y, sirkaobdelniku, vyskaobdelniku));
```

3.3.5. Třída RoundRectangle2D

Třída RoundRectangle2D slouží pro vykreslení obdélníku s zaoblenými rohy. Tato třída je podobná třídě Rectangle2D. Třída RoundRectangle2D navíc pro své vykreslení potřebuje informaci o zaoblení rohů a to šířka zaoblení rohu, výška zaoblení rohu.

```
// vykreslení RoundRectangle2D.Double
g2.draw(new RoundRectangle2D.Double(x, y, rectwidth, rectheight, 10, 10));
```

3.3.6. Třída Ellipse2D

Třída Ellipse2D je metoda pro vytvoření elipsy její parametry jsou stejné jako u třídy Rectangle2D

```
// vykreslení Ellipse2D.Double
g2.draw(new Ellipse2D.Double(x, y, rectwidth, rectheight));
```

3.3.7. Třída Arc2D

Tato třída Arc2D vytvoří oblouk jako část elipsy. Tvar křivky definuje počáteční úhel, úhel rozevření a její typ. Pro větší přesnost slouží další dvě třídy Arc2D.Double a Arc2D.Float. Arc2D je definován třemi konstantami který označují její typ a to OPEN, PIE, CHORD.

- Open-otevřený oblouk bez spojeni mezi konci oblouku.
- Chord-(struna) uzavřený oblouk úsečkou.
- Pie-(výsečový graf) vykreslí oblouk jako výseč elipsy.

```
// vykreslení Arc2D.Double
g2.draw(new Arc2D.Double(x, y, rectwidth, rectheight, 90, 135,
Arc2D.OPEN));
```

3.3.8. třída GeneralPath

GeneralPath je obecně křivka, která může být uzavřená (*polygon*) nebo otevřená (*polyline*). Typicky se vytváří tak, že se nejdříve definují pole souřadnic x a y. Dále se vytvoří objekt třídy GeneralPath, kterému se určí způsob vyhodnocení, zda je nějaký bod uvnitř (*winding rule*). Způsoby vyhodnocení jsou even-odd a non-zero. Při vytvoření složitější křivky, která se v několika místech kříží, dojde ke vzniku objektu, který se skládá z několika menších uzavřených oblastí. Způsoby vyhodnocení even-odd nebo non-zero určí těmto jednotlivým

oblastem zda budou považovány za vnější nebo vnitřní části vytvořeného objektu. Tyto metody, způsoby vyhodnocení, jsou zejména důležité při použití výplně.Většinou se používá even-odd. Při nastaveni objektu třídy GeneralPath na parametr even-odd, dojde k přiřazení pořadového čísla pro každou uzavřenou oblast, z kterých se skládá objekt. Číslem jedna je vždy označena oblast mimo objekt a pak každá část označená lichým číslem je považované za vnější část. Vyhodnocení způsobem non-zero je složitější. Zde jsou také přiřazeny jednotlivým částem čísla. Ty jsou voleny v závislosti na hodnotě sousední části a na směru vedení přímky, která je od sebe odděluje. Jestliže je hraniční přímka vedena ve směru hodinových ručiček, tak je té části přiřazena hodnota sousední části a je k ní přičtena jednička. V případě kdy je hraniční přímka vedena v protisměru hodinových ručiček, tak je té části přiřazena hodnota sousední části a je snížena o jedničku. Části s hodnotou nula jsou pak považovány za vnější oblast. Druhý parametr pro vytvoření objektu třídy GeneralPath je počet bodů, ze kterých se bude křivka skládat. Třetím krokem je nastavení výchozího bodu pomocí *moveTo()*. Pak následuje spojení bodů pomocí metod.

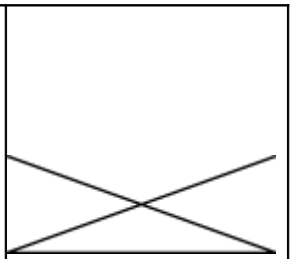
- *moveTo()*. Pak následuje spojení bodů pomocí metod.
- *lineTo()* – spojení úsečkou
- *quadTo()* – spojení Beziérovou křivkou
- *closePath()* což je poslední krok který uzavře křivku a tím vytvoří polygon.

Následující příklad ukazuje jak kreslit *polyline* s pomocí GeneralPath:

```
// vykreslení GeneralPath (polyline)
int x2Points[] = {0, 100, 0, 100};
int y2Points[] = {0, 50, 50, 0};
GeneralPath polyline =
new GeneralPath(GeneralPath.WIND_EVEN_ODD,
x2Points.length);

polyline.moveTo (x2Points[0], y2Points[0]);

for (int index = 1; index < x2Points.length; index++) {
   polyline.lineTo(x2Points[index], y2Points[index]);
};
g2.draw(polyline);
```



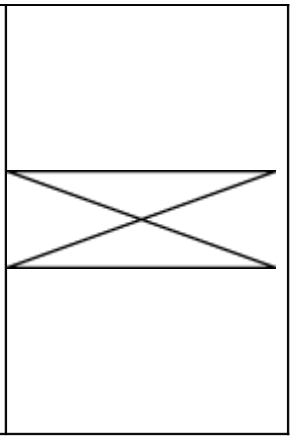
tabulka 1: Ukázka vytvoření polyline

Následující příklad ilustruje jak kreslit *polygon* pomocí GeneralPath

```
// vykreslení GeneralPath (polygon)
int x1Points[] = {0, 100, 0, 100};
int y1Points[] = {0, 50, 50, 0};
GeneralPath polygon =
        new GeneralPath(GeneralPath.WIND_EVEN_ODD,
x1Points.length);
polygon.moveTo(x1Points[0], y1Points[0]);

for (int index = 1; index < x1Points.length; index++) {
        polygon.lineTo(x1Points[index], y1Points[index]);
};

polygon.closePath();
g2.draw(polygon);
```

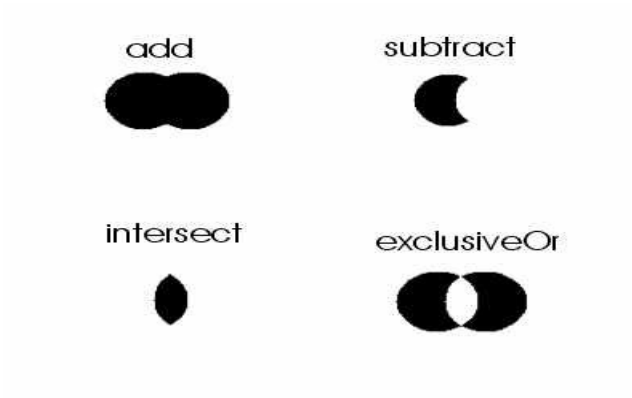


tabulka 2: Ukázka vytvoření polygonu

3.3.9. Třída Area

Třída Area je pro práci s geometrickými objekty, přesněji řečeno s uzavřenými oblastmi. Na rozdíl od výše zmíněných tříd, poskytuje operace jako je průnik, logickou operaci XOR atd. Třída Area obsahuje dva konstruktory *Area()* a *Area(Shape s)*. Při použití bezparametrického konstruktoru *Area()* dostáváme prázdnou oblast, do níž můžeme další tvary přidávat pomocí metody add. Konstruktor *Area(Shape s)* přebírá jako parametr objekt Shape, podle něhož objekt Area vytvoří. Pokud objekt Shape předávaný konstruktoru *Area(Shape s)* reprezentuje otevřený tvar, uzavře se automaticky.

Nezbytnými metodami pro operace s oblastmi jsou *subtract(Area a)*, *intersect(Area a)*, *add(Area a)* a *exclusiveOr(Area a)*. Vyjádřeno v množinové terminologii je metoda subtract odčítáním, metoda intersect průnikem, metoda add sjednocením a metoda exclusiveOr je logickou operací XOR. Jak vypadá výsledek použití těchto metod můžete vidět na obrázku č.2.



obrázek 2: Operace s oblastmi pomocí třídy Area

```
Area shape =new Area()
shape.add(new Area(new  RoundRectangle2D.Double(200,251,50,55,30,60)))
shape.intersect(new Area(new RoundRectangle2D.Double(155,145,98,140,50,80)))
shape.subtract(new Area(new Ellipse2D.Double(150,200,145,55)))
```

Třída Area obsahuje i několik metod sloužících k dotazování se na povahu tvaru, který reprezentuje. Metoda *isEmpty()* vrací true, pokud objekt Area neobsahuje žádný geometrický tvar. Metoda *isRectangular()* zjišťuje, zda je reprezentovaný tvar obdélníkem, a analogicky metoda *isPolygonal()* zjišťuje, zda je reprezentovaný tvar mnohoúhelníkem.

3.4. Nastavení pera pro kresbu

K nastavení stylu pera slouží v Javě metoda *setStroke(java.awt.Stroke s)* třídy Graphics2D, která jako parametr přebírá instanci třídy implementující rozhraní Stroke. Pro vytvoření nového stylu čáry je zapotřebí třída BasicStroke.(java.awt.BasicStroke)

Ukázka pěti variant konstruktoru BasicStroke:

```
BasicStroke()
BasicStroke(float width)
BasicStroke(float width, int cap, int join)
BasicStroke(float width, int cap, int join, float miterlimit)
BasicStroke(float width, int cap, int join, float miterlimit, float[] dash,
float dash_phase)
```

Bezparametrický konstruktor *BasicStroke()* vytváří pero o šířce jednoho bodu, se standardními hodnotami zbývajících atributů (viz dále). Konstruktor *BasicStroke(float width)* vytváří pero se šířkou rovnou width a zbývajícími atributy rovnými standardním hodnotám. Konstruktor *BasicStroke(float width, int cap, int join)* přidává k předchozímu ještě styly zakončení čar a styly spojení mezi jednotlivými navazujícími čarami. Hodnota cap určuje styl zakončení a může nabývat jedné z hodnot CAP_BUTT, CAP_ROUND a CAP_SQUARE definovaných ve třídě BasicStroke. Výsledek jejich použití můžete vidět na obrázku č.3.

obrázek 3: Ukázka typů zakončení čar

Stejně jako parametr cap i parametr join určující styl spojení jednotlivých čar může nabývat tří různých hodnot: JOIN_BEVEL, JOIN_MITER a JOIN_ROUND. Jak vypadá výsledek jejich použití, je opět patrné z obrázku:

obrázek 4: Ukázka typů spojení čar

Parametr miterlimit se používá pouze ve spojení s hodnotou JOIN_MITER parametru join a určuje, za jakých podmínek bude spojení čar zakončeno oseknutím, jako je tomu při použití hodnoty JOIN_BEVEL, a kdy naopak bude spojení ostré, jak je to možné vidět u hodnoty JOIN_MITER na obrázku. Pokud je poměr vzdálenosti vnitřního a vnějšího vrcholu spojení ku šířce pera větší než hodnota miterlimit, dojde k oseknutí, jinak bude realizována druhá z výše uvedených možností (ostré zakončení). To je velice užitečné, protože při velmi malých hodnotách úhlu svíraného spojovanými čárami by se nám spojení začalo protahovat k nekonečnu (minimálně mimo obrazovku).

Poslednímu konstruktoru třídy BasicStroke, kterému se vedle již zmíněných parametrů předávají další dva. Tento konstruktor slouží k vytváření čárkovaných, tečkovaných a jinak přerušovaných čar. Hodnoty uložené v tomto poli určují délku jednotlivých vykreslovaných a nevykreslovaných úseků v pořadí, jak jdou za sebou. Hodnota *dash_phase* pak určuje, kdy se má začít s vykreslováním přerušovaných úseků.

Vytvoření a nastavení pera šířky 5, které bude vykreslovat čárkované čáry s vykreslovanou částí délky 20 a mezerou délky 10, se spojením JOIN_MITER (miterlimit=10) a zakončením CAP_ROUND, vypadá následovně:

```
//...
float dash[] = {20,10};
BasicStroke bs = new BasicStroke(5, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_MITER, 10, dash, 0);
public void paint(Graphics g){
 Graphics2D g2 = (Graphics2D)g;
 g2.setStroke(bs);            }
```

Ještě zbývá říct, jaké jsou základní hodnoty jednotlivých atributů pera vytvořeného voláním konstruktoru *BasicStroke(float width)*. Jsou to JOIN_MITER jako styl spojení s hodnotou miterlimit rovnou 10 a CAP_SQUARE pro zakončení.

3.5. Nastavení štětce pro kresbu

Kromě nastavování per umožňuje třída Graphics2D rovněž nastavovat různé štětce (štětcem v tomto případě rozumíme způsob, jakým je volena barva jednotlivých bodů). K nastavení štětce slouží metoda *setPaint(Paint p)* třídy Graphics2D, která jako parametr přebírá instanci třídy implementující rozhraní Paint, například Color, TexturePaint nebo GradientPaint.Rozhraní Paint implementují tyto třídy které popisují styl výplně.

- java.awt.Color (výplň s jednou barvou)
- java.awt.GradientPaint (výplň s barevným přechodem)
- java.awt.TexturePaint (výplň texturou)

Vlastnosti se nastavují pomocí *g2.setPaint(Paint p);*.Takto nastavený štětec může vykreslit vybraným stylem obrys, nebo křivku, dokonce i text a samozřejmě výplň uzavřeného tělesa. Pokud postačí výplň s jednou barvou bez nějakých efektů tak postačí metoda *setColor(Color c);*.Třída Color má mnoho možností jak zadat barvu. Typické je *Color (int r,int g,int b);* pro vytvoření libovolné neprůhledné barvy, nebo použít jedné z již přednastavených základních barev (black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow) Také lze použít i průhledné barvy (transparenci).

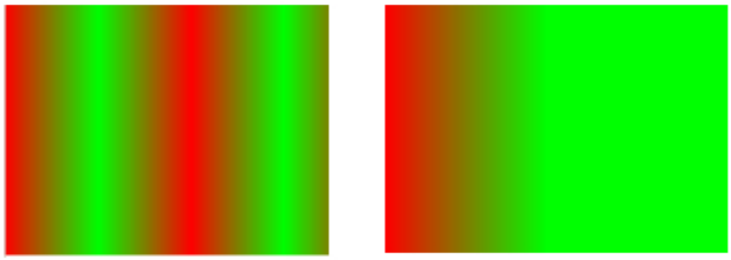
3.5.1. GradientPaint

Třída GradientPaint vytváří štětec, pomocí něhož můžeme vykreslovat barevné přechody. Konstruktor této třídy je *GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic).* Parametry x1, y1 a x2, y2 jsou souřadnice prvního a druhého řídícího bodu. Řídící body určují úsečku, na které se postupně mění barva z *color1* (na prvním řídícím bodu) na *color2* (na druhém řídícím bodu). Pokud je parametr *cyclic* roven true, pak se na prodloužení této úsečky osově promítnou body úsečky a s takto vzniklou novou úsečkou se provede totéž. Pokud je naopak parametr cyclic roven false, mají body na prodloužení této úsečky barvu bližšího řídícího bodu. Ostatní body jsou určeny barvou bodu ležícího na přímce dané řídícími body, který leží na kolmici k této přímce vedené bodem, jehož barva nás zajímá.

Výsledek použití následujících konstruktorů ukazuje obrázek níže:

```
//...
//vytvoření objektu GradientPaint pro levý obrázek
GradientPaint gr = new GradientPaint(0,0,Color.red,100,0,Color.green,true);

//vytvoření objektu GradientPaint pro pravý obrázek - zatím jako komentář
//GradientPaint gr = new
GradientPaint(0,0,Color.red,100,0,Color.green,false);
//...
//V metodě paint pak nastavíte štětec takto:
g2.setPaint(gr);
//...
```



obrázek 5: Vyreslení s třídou GradientPaint

3.5.2. TexturePaint

Třída TexturePaint vytváří štětec, pomocí něhož můžeme kreslit texturu. Pro konstruktor TexturePaint potřebujeme třídu BufferedImage s vytvořeným vzorem textury. Dále je zapotřebí použít třídu Rectangle2D která nám určí velikost jedné dlaždice textury. *TexturePaint(BufferedImage obraz, Rectangle2D velikostdlazdice);*.

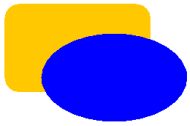
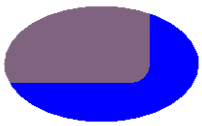
```
BufferedImage bi =new BufferedImage (20,20,BufferedImage.TYPE_INT_RGB);
    Graphics2D big =  bi.createGraphics();

big.setColor(Color.RED);
big.fillRect(0,0,20,20);
Rectangle r = new Rectangle (0,0,20,20);
g2d.setPaint(new TexturePaint(bi, r));
```

3.6.Kompozice grafických objektů

Další nedílnou součástí API je i podpora kompozice grafických objektů. Toto skládání objektů je umožněno přes třídu Composite. Zde jsou předem definována základní Boolean operace (AND, OR, XOR). Dále pak objekt AlphaComposite, jenž umožňuje použití skládání obrazců za použití alpha kanálu, který určuje stupeň průhlednosti. Úplná průhlednost je při

Alpha = 0.0. Naopak naprostá neprůhlednost je pokavaď s Alpha = 1.0. Ukázky nadefinovaných stylů kompozice v závislosti na alpha kanálu jsou v následující tabulce.

Alpha composite	Alpha kanál		
	1	0,5	0
DST_IN			
DST_OUT			
DST_OVER			
DST_ATOP			
SRC_IN			
SRC_OUT			
SRC_OVER			
SRC_ATOP			
CLEAR			
XOR			

tabulka 3: Ukázky kompozice grafických objektů

Nastavení stylu kompozice se provádí konstruktorem s voláním AlphaComposite.getInstance

```
AlphaComposite ac = AlphaComposite.getInstance(AlphaComposite.SRC_IN);
```

Při změně kompozici nebo hodnoty alpha je možné znovu použít AlphaComposite.GetInstance a přiřadit nový AlphaComposite. AlphaComposite nastavuje způsob interakce barvy původního pixelu obrázku s barvou pixelu vkládaného obrázku. Hodnota alpha je použita pro nastavení vlastnosti pixelů.

```
AlphaComposite.getInstance. ac =
AlphaComposite.getInstance(AlphaComposite SRC_IN, alpha)
```

Propojení nastavené kompozice v objektu AlphaComposite a grafického kontextu se provádí metodou Graphics 2D set Composite.

```
BufferedImage buffImg =
new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
Graphics2D gbi = buffImg.createGraphics();
...
gbi.setComposite(ac);
```

Objekty jsou kopírovány na obrazovku, tak jak jsou nastaveny vlastnosti kompozice v kontextu Graphics2D pro BufferedImage

3.7. Operace s obrázky

3.7.1. Načtení a vykreslení obrázku

Třídy balíčku javax.imageo implicitně podporují rastrové formáty GIF, PNG, JPEG. Další kodeky, jako například TIFF, JPEG2000 a BMP už jsou součástí volitelného balíčku pro IO operace s obrázky. Obrázky si do apletu můžeme načíst z souboru nebo adresy URL. Pro získání obrázku pro aplet, který není ve stejném adresáři jako je HTML soubor slouží metoda *getCodeBase()*. Tato metoda vrací objekt třídy java.net.URL.

```
try {
     URL url = new URL(getCodeBase(), "http://www.adresa.com/obrazek.jpg");
         img = ImageIO.read(url);
         } catch (IOException e) {         }
```

Následující metoda se dá také použít i pro získání obrázku který je ve stejném adresáři jako je HTML.

```
Image obr = getImage (getCodeBase(),"obrazek.JPG")
```

Metoda *getImage()* netestuje, zda soubor s obrázkem skutečně existuje, a ani okamžitě nenatahuje obrázek do paměti. K fyzickému načtení obrázku dojde až v okamžiku, kdy jej chceme zobrazit. Tento způsob je jednoduchý, ale vyhovuje jen malým obrázkům. U větších je nepříjemná časová prodleva od pokynu načíst obrázek do skutečného zobrazení obrázku.

Pro vykreslení obrázku slouží metoda *drawlmage ()*, která je ve třídě Graphics šestinásobně přetížena. U všech je parametr Image img, což je obrázek, který má být vykreslen a ImageObserve. ImageObserver je asynchronní rozhraní které slouží pro přijímání informací o zobrazení obrázku. Pokud nepotřebujeme pracovat s těmito informacemi můžeme na toto místo zadat null.

Nejjednodušší ze všech šesti metod je:*drawlmage(Image img, int x, int y, ImageObserver io);*. Tato metoda zobrazí obrázek s levým horním rohem na souřadnicích x, y a ponechá původní velikost obrázku. Pokud se obrázek nevejde celý do okna programu, *drawlmage ()* neprovádí žádnou další činnost a jedna z mála možností, jak si lze celý obrázek prohlédnout, je zvětšení hlavního okna.

Stejně funguje další metoda:*drawlmage(Image img, int x, int y,Color bgcolor, ImageObserver io);*. Zde přibyl zdánlivě zbytečný parametr barvy pozadí. Tu oceníme, pokud je obrázek zobrazen proporcionálně (tj. při zobrazení je zachován poměr stran obrázku) a plátno, na němž je obrázek zobrazen má jiný poměr výšky a šířky. Další použití je v tom případě, kdy by části obrázku byly transparentní. Pak oceníme možnost mít definovanou barvu pozadí.

Další metoda se podobá první s tím rozdílem, že stanovuje šířku a výšku kresleného obrázku: *drawlmage(Image img, int x, int y, int width,int height, ImageObserver io);*. To tedy znamená, že se obrázek zmenší nebo zvětší na tyto rozměry. Pokud šířka a nebo výška přesahují rozměry okna, je opět zobrazena jen část obrázku.

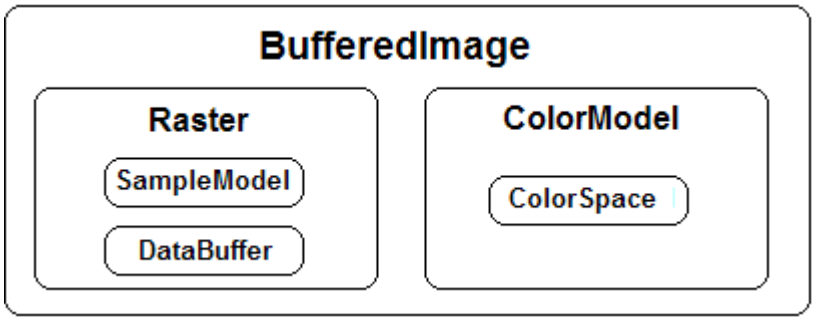
Této metodě oproti předchozí přibývá pouze barva pozadí.*drawlmage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver io);*. Pokud byl obrázek větší než okno, dokázaly všechny předchozí metody zobrazit jen levou horní část. Následující metody dokáží zobrazit a současně zvětšit či zmenšit jakoukoliv část obrázku. Souřadnice v níže uvedené ukázce, které začínají *d* jsou souřadnice pro okno (d = *destination). S*ouřadnice

s určují poloho výřezu obrázku (s = *source)*. Pro určení výřezu obrázku se používají jen souřadnice, nikoliv výška a šířka, jako u předchozích metod.

```
drawImage(Image   img, int dx1, int dy1, int dx2, int dy2,
                       int sx1, int sy1, int sx2, int sy2,
                                         ImageObserver io);
drawImage(Image img,   int dx1, int dy1, int dx2, int dy2,
                       int sx1, int sy1, int sx2, int sy2,
                       Color bgcolor, ImageObserver io);
```

3.7.2. Třída BufferedImage

Základem pro operace s obrazem je třída BufferedImage. Rozšiřuje třídu Image a to o možnost přímo pracovat s obrazovými daty. Například získat nebo změnit barvu jednotlivých bodů. BufferedImage je v podstatě bitmapa s přístupnou vyrovnávací pamětí. Obrázky BufferedImage je obecně vzato možné upravovat buď prostřednictvím objektu Graphics, respektive Graphics2D. BufferedImage se skládá z ColorModel a rastru. Rastr se stará o rozložení a popis obrazu do jednotlivých bodů, které jsou uspořádány do mřížky. Pravidla pro reprezentování dat jsou dány třídou ColorModel. Toto umožňuje podporu nejen indexovaných barev, ale i použití barevných formátů jako je RGBA (RGB s alpha kanálem).



obrázek 6: Složení třídy BufferedImage

3.7.3.Použití filtrů

Jednou z jednodušších úprav obrázků jsou filtry z balíčku java.awt.image, které implementují rozhraní BufferedImageOp. Kromě nich existují ještě filtry implementující rozhraní ImageFilter. Pro použití filtru v rozhraní BufferedImageOp je nejpodstatnější metoda *public BufferedImage filter(BufferedImage zdroj, BufferedImage cíl)*, po jejímž zavolání dojde k vlastnímu filtrování obrázku. Výsledný obrázek se pak uloží buď do objektu "cíl" předaného jako parametr, nebo do nově vytvořeného obrázku, pokud bude parametr "cíl" roven "null".

Java2D nabízí čtyři základní filtry a to ConvolveOp, AffineTransformOp, LookupOp, RescaleOp. S pomocí těchto filtrů lze provádět geometrické nebo barevné transformace.

Filtr ConvolveOp

Filtr ConvolveOp je často používaná operace na zpracování obrazu. Slouží k barevnému přepočtu pixelů. Nový barevný bod je spočítán z okolních pixelů. Konstruktor tohoto filtru vypadá následovně: *ConvolveOp(Kernel kernel, int okraje, RenderingHints vlastnosti)*, kde kernel je matice popisující změnu vstupních pixelů na výstupní. Dalším atributem je příznak, který určuje zda se mají okraje nahradit nulami (EDGE_ZERO_FILL) nebo zda se mají zkopírovat ze zdrojového obrazu (EDGE_NO_OP). Objekt RenderingHints pak obsahuje informace o tom, jak má být vše provedeno, může však být i null, jak je v ukázce. Vytvoření objektu Kernel. Konstruktor Kernel(int šířka, int výška, float[] matice) očekává tři parametry: výšku a šířku matice a matici v podobě jednorozměrného pole, jehož délka musí být šířka * výška.

Následující ukázky kódu provádí rozmazání, zaostření, vytvoření reliéfu obrázku v objektu typu BufferedImage. Ve všech příkladech je použito jádro o rozměrech 3, 3.

Rozmazání obrázku v objektu typu BufferedImage

```
Kernel  kernel = new Kernel  (3,3,new float[]{1f/9f, 1f/9f, 1f/9f,
                                              1f/9f, 1f/9f, 1f/9f,
                                              1f/9f, 1f/9f, 1f/9f, })
BufferedImageOp op= new ConvolveOp(kernel)
BufferedImage = op.filter (bufferedImage, null)
```

Zaostření obrázku v objektu typu BufferedImage

```
Kernel   kernel  = new Kernel (3,3,new float[]{-1, -1, -1,
                                               -1, 9, -1,
                                              -1, -1, -1})
BufferedImageOp op= new ConvolveOp(kernel)
BufferedImage = op.filter (bufferedImage, null)
```

Tvorba reliéfu obrázku v objektu typu BufferedImage

```
Kernel   kernel  = new Kernel (3,3,new float[]{-2, 0, 0,
                                               0, 1, 0,
                                               0, 0, 2})
BufferedImageOp op= new ConvolveOp(kernel)
BufferedImage = op.filter (bufferedImage, null)
```

Filtr LookupOp.

Tento filtr nahrazuje jednotně barvy pixelů za jinou barvu podle tabulky. Pro jednotné úpravy jako třeba vytvoření negativu. Pro záměnu barev, lze tedy použít třídu LookupOp. Konstruktoru LookupOp(LookupTable table, RenderingHints hints) se předává objekt LookupTable, v němž jsou uloženy nové hodnoty pro jednotlivé složky barvy. Máme-li například pixel s červenou složkou 123, bude tomuto pixelu přiřazena hodnota uložená v tabulce na indexu 123. Následující kód demonstruje použití této třídy pro vytvoření negativu.

```
byte lut[] = new byte[256];
          for (int j=0; j<256; j++) {
              lut[j] = (byte)(256-j);
          }
          ByteLookupTable blut = new ByteLookupTable(0, lut);
          LookupOp lop = new LookupOp(blut, null);
          g2.drawImage(bi, lop, 0, 0);
```

Filtr AffineTransformOp

Další běžnou operací, prováděnou s obrázky, jsou rotace posun zkosení či změna měřítka. K tomu se využívá třída AffineTransformOp, jejíž konstruktor vypadá takto: *AffineTransformOp(AffineTransform at, RenderingHints hints)*. Tento filtr zmapuje pixely ze zdrojového obrazu a změní jejich pozici.

```
public void paint (Graphics g){

    Graphics2D g2d = (Graphics2D)g;
    AffineTransform tx = new AffineTransform();
    //zvětšení
    double scalex = .5;
    double scaley = .1;
    tx.scale(scalex,scaley);
    //zkosení
    double shiftx = .1;
    double shifty = .3;
    tx.shear(shiftx,shifty);
    //posun
    double x = 50;
    double y = 50;
    tx.translate(x,y);
    //otočení
    double radians = -Math.PI/4;
    tx.rotate(radians);

    g2d.drawImage(image, tx, this);
```

Filtr RescaleOp

RescaleOp. Tento filtr ovlivňuje barvi podle faktoru .Umožňuje obraz zesvětlit nebo ztmavit oproti originálnímu obrazu nebo také redukovat jeho neprůhlednost, atd. Operaci tohoto filtru lze popsat následujícím předpisem: vezmi pixel, vynásob jeho každou barevnou složku příslušným parametrem "scale" a pak k ní přičti parametr "offset" - pokud je výsledek větší než maximální hodnota, použij maximální hodnotu. Konstruktor má tvar *RescaleOp(float scale, float offset, RenderingHints hints)*, přičemž parametr "hints" může být i "null". Následuje kód s použitím filtru pro zesvětlení obrázku.

```
    //zesvětlit obrázek o 30 procent
       float scaleFactor = 1.3f;
       RescaleOp op = new RescaleOp(scaleFactor, 0, null);
       bufferedImage = op.filter (bufferedImage, null);
    //ztmavit obrazek  o 10 procent
       scaleFactor = .9f;
       Op = new RescaleOp(scaleFactor, 0, null );
        bufferedImage = op.filter (bufferedImage, null);
                }
```

3.7.4. Vytvoření obrázku

Pro sestavení obrázku je důležité nejdříve v paměti vytvořit prostor do kterého se bude provádět grafický výstup. Nejlepším řešením je použití třídy BufferedImage. Jak již bylo zmíněno, třída BufferedImage je bitmapa, do které je možné kreslit. Kreslit do této třídy můžeme přímo prostřednictvím objektu Graphics2D získaném voláním její metody *createGraphics()*. BufferedImage je možné vytvořit jedním z těchto třech konstruktorů této třídy. První příklad ukazuje základní konstruktor s předdefinovaným typem obrazu

```
new BufferedImage(width, height, type);
```

`V dalším typu konstruktoru` je možné si zvolit z dvou typů třídy ColorModel a to z typu *byte bininar a byte index.*

Typ byte bininar - V tomto modelu lze určit každé barevné složce (R,G,B) jinou bytovou šířku. Tento typ nepoužívá složku alpha, která slouží k nastavení průhlednosti.

`Typ` *byte index* - Nabízí kompaktnější způsob ukládání informace o barvách. Neobsahuje informace o barvě, ale používá index do tabulky barev obsahující rozložení složek RGB a alpha.

```
new BufferedImage(width, height, type, colorModel);
```

U posledního konstruktoru kromě nastavení typu colorModel je možné specifikovat ještě rastr.

```
new BufferedImage(colorModel, raster, premultiplied, properties)
```

3.8. RenderingHints

Kromě nástrojů na vlastní kreslení objektů na scénu je také umožněna jejich kontrola kvality. To se provádí za pomoci objektu RenderingHints.Většinou lepší kvalita vykreslení je na úkor rychlosti vykreslení a naopak. Některé třídy mají nastavení tohoto atributu rovnou v konstruktoru. Například při kreslení základních primitiv tenkou čárou, je vhodné zapnout antialiasing, který často významně vylepší vzhled obrázku. RenderingHints se nastavuje pomocí metody *setRenderingHint(RenderingHints.Key hintKey, Object hintValue)*. Metoda obsahuje dva parametry *hintKey a hintValue*. Parametr *hintKey* určuje typ úpravy jako např. antialiasing, interpolace, dithering. Pro bližší přesné definováni stavu hintKey slouží parametr *hintValue.* Nejčastější nastavení výběru *hintValue* je zapnuto, vypnuto nastavení na kvalitu, rychlost atd. S pomocí kolekce a metody *setRenderingHints(Map hints)* lze nastavit rovnou celou skupinu vlastnosti pro vykreslení a tak se nemusí nastavovat po jedné. Další ukázka kódu předvádí zapnutí antialisingu pro text.

```
public void paint (graphics g){
      Graphics2D g2 = (Graphics2D)g;
      RenderingHints rh = new RenderingHints(
            RenderingHints.KEY_TEXT_ANTIALIASING,
            RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
      g2.setRenderingHints(rh);
   ...
   }
```

Třída RenderingHints podporuje následující vlastnosti neboli *hintKey*, kterými lze ovlivnit způsob vykreslování.

Antialiasing hint key

Obrázky mají diskrétní povahu, tj. jsou složeny z konečného počtu bodů (pixelů) umístěných do sítě řádků a sloupců (rastru), to způsobuje zubaté zobrazení šikmých čar a křivek, ztrátu detailu u malých objektů. Tento jev se objevuje zejména při příliš malém rozlišení a nazývá se aliasing. Antialiasing je technika, která se snaží odstranit nebo snížit výsledek aliasingu.

Typickou metodou pro redukci je vyhlazování přechodů mezi dvěma částmi obrazu.Pro vyhlazení hran se například používá průměr barev v místě aliasingového přechodu a tím je dosaženo rozmazání. *Antialiasing hint key* může být nastaveno na výchozí, zapnuto či vypnuto.

Alpha interpolation hint key

Alpha interpolace *Alpha interpolation hint key* řídí míšení barev podle hodnoty alpha kanálu. Způsob metody pro získání hodnot barvy záleží na tom, jestli je kladen důraz na rychlost nebo na kvalitu. Pokud tolik nezáleží na kvalitě je výsledná barva zjištěna podle tabulky. V případě, kdy je kladen důraz na kvalitu, ale ne na rychlost jsou použity SIMD instrukce. Alpha interpolace může být nastavena na výchozí či optimalizované na kvalitu či rychlost.

Color rendering hint key

Bez použití barev se neobejde žádný grafický systém, proto bývá obrazový model často složitý. Informace o barvách jsou uvedeny v ColorSpace reprezentuje systém pro měření barev. Tato třída obsahuje metody pro konverzi mezi přímo zadanou hodnotou barvy a RGB systémem. *Color rendering hint key* ovlivňuje výpočty pro převody mezi jednotlivými barevnými schématy pro konečné zobrazení. Kalkulace pro zobrazování barev může být nastaveno na výchozí či optimalizované na kvalitu či rychlost.

Dithering hint key

Dithering hint key ovládá způsob jak se nejlépe přiblížit k barvě na systémech s nižším barevným rozlišením než je vytvořený obraz. Odstíny, které nejsou k dispozici se simulují pomocí rastru dvou nejbližších odstínů, procentuálně namíchaných tak, aby se věrně napodobila požadovaná barva. Java standardně používá barevný model, který ukládá barvu pixelu jako 32 bitové číslo. Tento model používá 8 bitů pro uložení každé ze složek red, green, blue a alpha kanál(alpha definuje transparentnost pixelu- 0- pixel je transparentní, 255- je neprůhledný). Dithering hint key může být nastaveno na výchozí, povoleno či nepovoleno.

Fractional metrics hint key

Fractional metrics slouží pro možnosti přístupu třídy FontMetrics. Třída FontMetrics dává aktuální informace o použitém fontu. Standardně jsou tyto informace typu integer, ale s

nastavením *Fractional metrics* mohou být i typu float. Zapnutím *Fractional metrics* získáme větší přesnost informace o umístění jednotlivých znaků. Fractional metrics může být nastaveno na výchozí, zapnuto či vypnuto.

Interpolation hint key

Interpolace obrazu udává metodu překreslování bitmapových objektů při jejich zvětšení , zmenšení nebo jiných transformacích. Interpolace může být nastavena na Nearest Neighbor interpolation, Bilinear interpolation a Bicubic. Principem *Nearest Neighbor interpolation* je okopírování hodnoty pixelu nejbližšího souseda v okolí vzorku. Metoda je geometricky nejméně přesná, ale rychlá. Výsledný snímek může obsahovat nespojitosti a dochází ke ztrátě některých detailů obrazu. *Bilinear interpolation* dosahuje lepších výsledku. Hodnota pixelu v novém obraze je vypočtena jako vážený průměr čtyř nejbližších pixelů z původního obrazu. *Bicubic interpolation*, zajistí nejvěrnější překreslení rastru s velmi dobrou kvalitou výsledného zobrazení a přiměřenou ostrostí hran, ale je nejnáročnější na výpočet.

Rendering hint key

Rendering hint key řídí soubor algoritmů, který mají vliv na výsledný obraz. Lepší obraz lze získat nastavením kvality, ale je v závislosti na rychlosti vykreslení. Zde si mužem definovat čemu dáváme přednost, jestli rychlosti nebo kvalitě. *Rendering hint key* lze nastavit na výchozí či optimalizované na kvalitu nebo rychlost.

Stroke normalization control hint key

Stroke Normalization Control hint key slouží pro modifikace geometrických objektů. Při vykreslení obrysu například elipsi může dojít ke špatnému vystředění pixelu, které tvoří danou elipsu. Tento nežádoucí efekt je potlačen zlepšením přesnosti výpočtů, což vede k zpomalení procesu. Stroke Normalization Control nabízí tří možností nastavení. Po nastavení hodnoty na *Normalize* by se měla zlepšit jednotnost čáry i celkový vzhled. Při nastavení na hodnoty na *Pure* není prováděna normalizace. Poslední typ nastavení je *Default* pro výchozí nastavení.

Text antialiasing hint key

Antialising v textu je technika užívaná pro vyhlazení okrajů u textu. Java 2D API dokáže sama na základě použitého fontu rozhodnou, zda by měla text graficky upravit. Tato vlastnost

se získá nastavením Text antialiasing hint key na hodnotu *GASP*. Nejběžnější úpravou antialiasingu je míchání barvy z okraje textu s barvou pixelu z pozadí. Nevýhoda antialiasingu je zřejmá hlavně u menších velikostí fontu, kde dochází díky rozmazání k nečitelnosti písma. LCD displeje mají schopnost, že můžou používat java2D API k vytvoření textu který nebude tak rozmazaný jako je při klasickém použití antialiasingu a bude tak lépe čitelný i při malých fontech. Pro tuto optimalizaci složí hodnota LCD_HRGB. Dále *Text antialiasing hint key* lze také nastavit na výchozí, zapnuto čí vypnuto.

LCD text kontrast hint key

Hodnoty by měli být přirozené číslo v rozsahu 100 až 250. Nižší hodnoty odpovídají vyššímu kontrastu textu, při zobrazení tmavé pasáže na světlém pozadí. Vyšší hodnota než 200 odpovídá nižšímu kontrastu textu, při zobrazení tmavého textu na světlém pozadí. Běžná hodnota je v úzkém rozsahu 140-180. Jestliže žádaná hodnota je povolena systémem pak bude hodnota použita.

3.9. Vykreslování textu

3.9.1. Použití fontů

S pomocí Javy 2D API nejen že možné vykreslovat obyčejný text, ale je možné mít text vyplněný texturou nebo jen vykreslit obrys písma atd. Java 2D také nabízí metody pro měření fontů, formátování textu použití antialiasingu a jiné grafické úpravy textu.

Psaní do grafického okna a operace s textem v komponentách, které jsou pro to specielně určeny jako je TextField, mají shodné jen práce s nastavením fontů. Všechny fonty, které se skutečně v počítači vyskytují, se nazývají fyzické fonty. Při jejich použití může vzniknout problém s přenositelností programů na jinou platformu, protože MS Windous a Unix se ve jménech fyzických fontů zásadně liší. Z tohoto důvodu používá Java od JDK1.1 systém tzv. symbolických jmen (též virtuálních fonty/jména nebo logické fonty/jména), kdy jsou místo jmen fyzických fontů, jako jsou například Arial nebo Helvetica, používána symbolická jména. Tento způsob má výhodu, že program je pak nezávislý na platformě. Konečné přiřazení symbolického jména konkrétnímu fyzickému fontu je pak provedeno pomocí jejich mapování uvedené v souboru font.properties. Symbolické jméno je platné pro celou rodinu písem a lze z něj lehce odvodit všechny čtyři řezy písma. Jeden řez písma je vždy základní nazývaný plain (*Font.PLAIN*). Dalšími řezy jsou kurzíva (*Font.ITALIC*) a tučný řez (*Font.BOLD*). Čtvrtým

řezem je tučná kurzíva pro ni musíme použít součet(*Font.BOLD + Font.ITALIC*). Java umožňuje použití pěti symbolických jmen rodin fontů které jsou uvedeny v prvním sloupci tabulky. V druhém a třetím sloupci tabulky jsou uvedeny názvy nejčastěji použitých rodin fontů.

Symbolický název	Linux	MS WINDOWS
Serif	Times Roman	Times New Roman
SansSerif	Helvetica	Arial
MonoSpaced	Curier	Courier New
Dialog	Helvetica	Arial
DialogInput	Courier	Courier New

tabulka 4: Srovnání názvů fontů

Vykreslení textu je umožněno pomocí metody Graphics.drawString a třídy Font, jenž obsahuje detailní informace o fontu. Třída Font má metody *getFont()* a *setFont()* pro zjištění a nastavení fontu. Metoda *drawString(String str, int x, int y)* provádí samotné vykreslení textu. Souřadnice x a y označují počáteční bod, odkud se bude text vypisovat.

```
Font font = new Font("Dialog", Font.BOLD+Font.ITALIC, 12)
      g2d.setFont(font)
      g2d.drawString("vypsaný text",20,20)
```

3.9.2.Metrika fontu

Pro detailnější informace o aktuálním fontu slouží třída FontMetrics. Tato třída dává i informace „šířkové“, kdy např. můžeme zjistit, zda se nějaký text v použitém fontu a v zadané velikosti ještě vejde na obrazovku. Instance FontMetrics se získává pomocí *getFontMetrics (Font f)* třídy Graphics2D. Metoda *getFontMetrics (Font f)* vrátí metriku libovolného fontu.

```
// Získání metriky
    FontMetrics metrics = graphics.getFontMetrics(font);
    // získání výšky fontu
    int hgt = metrics.getHeight();
    // vrací šířku použitého textu
    int adv = metrics.StringWidth("text");
```

3.9.3. Grafické úpravy textu

U fontu je umožněno použít několik přídavných vlastností jako je například napojování jednotlivých stylů znaků . Čehož lze využít hlavně u psacích fontu, kde je potřeba na sebe napojovat jednotlivá písmena ve slově. Dále je umožněno definování vzdálenosti mezi dvojicemi písmen, aby bylo odstraněno mezer vzniklých například mezi velkými tiskacími písmeny A a V. Vzdálenost těchto mezer ovlivňuje vlastnost fontu TextAttribute.KERNING. Zapnutím TextAttribute.KERNING je využita. možnost zkrácení mezery mezi jednotlivými písmeny. Toto ukazuje následující ukázka kódu a také další možnosti grafických úprav textu jako třeba: Underline (podtržení), Strikethrough (přeškrtnutí), Superscript nebo Subscript (horní nebo dolní index).

```
Hashtable<TextAttribute, Object> map =
            new Hashtable<TextAttribute, Object>();
map.put(TextAttribute.STRIKETHROUGH, TextAttribute.STRIKETHROUGH_ON);
map.put(TextAttribute.UNDERLINE, TextAttribute.UNDERLINE_ON);
map.put(TextAttribute.KERNING, TextAttribute.KERNING_ON);
   Map.put(TextAttribute.FOREGROUND, Color.BLUE);
        font = font.deriveFont(map);
        g.setFont(font);
        g.drawString(text, 45, 150);
```

Další funkcí je použití různých orientací textu to jest doprava, jenž se používá například u češtiny, nebo doleva, jako je tomu u arabštiny. Java2D API dále umožňuje zjišťování pozice kurzoru v textu (může být převedeno na problém zjištění pozice, na kterou se bude sázet další písmeno v textu). K tomuto účelu slouží třída TextLayout, která se stará o umísťování jednotlivých písmen textu. Tato třída dále slouží k posunu kurzoru po jednotlivých písmenech zajišťuje zvýraznění označené části textu. Obsahuje také třídu LinkeBreakMesure. Tato třída umožňuje naformátování delší textové řetězec do řádek. Každý text v řádku je pak vrácena jako objekt TextLayout.

Text nebo i geometrické tvary můžou také sloužit jako hranice pro výřez. Pro definici cesty výřezu lze použít jakýkoli objekt, jenž implementuje objekt Shape (to jest lze použít všech geometrických tvaru definovaných v Java2D API - package java.awt.geom) a nastaví se pomocí metody Graphics2D.setClip na instanci grafického rozhraní. Další funkcí kterou umožňuje Java2D API je odvozování fontů. Nový odvozený font je pak stejný jako původní font, na který je aplikována transformace AffineTransformation. Tím můžeme kreslit text především různým směrem a zkosením. Uvedeme zde malý příklad, jenž ukazuje jednoduchost

vytváření. Nový font je vytvořen metodou Font.deriveFont (AffineTransform). Dále je na příkladu ukázáno vyříznutí a aplikace transformace na konečný text.

```
AffineTransform fontAT = new AffineTransform();
fontAT.setToShear(-1.2, 0.0);
FontRenderContext frc = g2.getFontRenderContext();
Font f = new Font("Helvetica", Font.PLAIN, 1);
theDerivedFont = f.deriveFont(fontAT);
String s = new String("Text pro oříznuti a novým fontem");
TextLayout tl = new TextLayout(s, theDerivedFont, frc);
AffineTransform transform = new AffineTransform();
Shape outline = t1.getOutline(null);
Rectangle r = outline.getBounds();
transform = g2.getTransform();
transform.translate(w/2-(r.width/2), h/2+(r.height/2));
g2.transform(transform);
g2.setColor(Color.blue);
g2.draw(outline);
```

4. Dynamické zobrazování 2D scény

V apletu je možné vytvářet animace. Tyto animace mohou být vytvořený jako pole obrázků, nebo vykreslováním vytvořených tvarů v programu. Protože aplet by měl mít možnost reagovat na vstupy uživatele během animace, je zapotřebí animaci spustit v samostatném vláknu. Nejednodušším způsobem je implementace rozhraní Runnable třídy apletu.

4.1. Rozhraní Runnable

Pomocí rozhraní Runnable může definovat jádro vlákna, přičemž vytvářená třída nemusí být potomkem třídy Thread. Java nepodporuje vícenásobnou dědičnost a tak aplet nemůže získat vlákno děděním z třídy Thread, protože vytvořený aplet musí už dědit z třídy java.applet.Applet. Nejednoduší řešení je tedy pro získání vlastností vlákna implementací rozhraní Runnable (z balíku java.lang). Toto rozhraní je užitečné v případě, že aplet bude potřebovat pouze jedno další vlákno (kromě hlavního programu). Je zbytečné vytvářet kvůli jedné instanci třídu s vlákny, když je možnost využít rozhraní Runnable. Instance této třídy je pak předána jako argument konstruktoru třídy Thread nebo jejího potomka. Toto rozhraní obsahuje pouze jednu metodu *run()*. Tato metoda musí být samozřejmě ve vytvářené třídě implementována. Při spuštění vlákna se používá jednotně metoda *start()*, která spustí metodu *run()*. Bohužel při implementaci Runnable není povinné metodu *start()* vytvořit, což znamená , že se vlákno může z vnějšího pohledu jevit nestandardně. Obvyklý postup je, že při implementaci Runnable se vytváří i metoda *start()*.

Následující ukázka kódu slouží k přehrávání animace na pozadí Apletu. Pro animaci je vytvořeno vlákno konstruktorem `Thread(Runnable r)`, kterému předá odkaz na sebe, což je na instanci implementující rozhraní `Runnable` (*metoda run()*). Vykreslení je zajišťováno metodou *paint()*. Vlákno je uspáno vždy na 80 ms a pak se celý cyklus se opakuje, čímž vzniká animace

```
public class Animace extends java.applet.Applet
implements Runnable {
   Thread animator = null;
   int xpos = 0;

   public void init() {
      animator = new Thread(this);
      animator.start();
   }
```

```
   public void run() {
      while(animator != null) {
         repaint();
         try {
            Thread.sleep(80);
         } catch(InterruptedException e) {}
      }
   }

   public void paint(java.awt.Graphics g) {
      g.drawRect(xpos, 0, 20, 20);
      xpos = (xpos+1) % 100;
   }
}
```

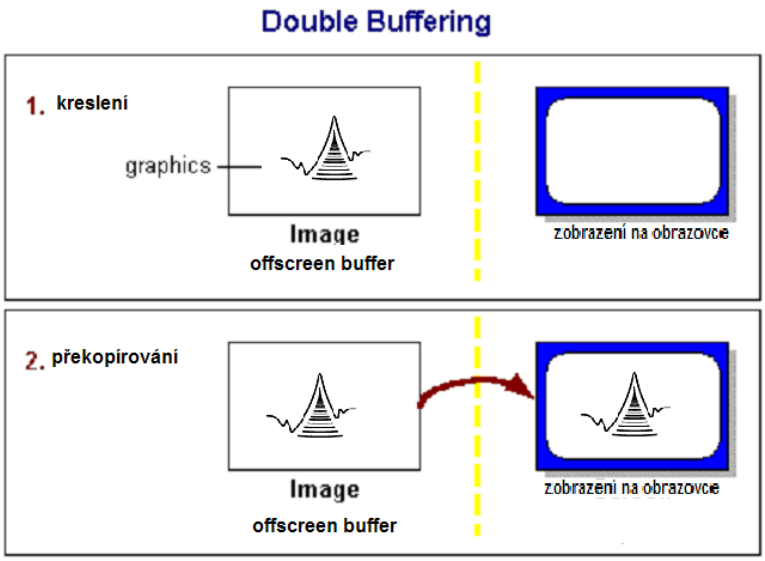
4.2.Principy zobrazení dynamické scény

Pro dynamické zobrazování objektů v apletu se používá několik metod. Nejednoduší metodou je postupně kreslit objekty na plátno (kreslící plátno okna). Na další snímek pak přejít tím, že je přes celé plátno nakreslen objekt s barvou pozadí, tím jsou smazány všechny objekty ze snímku. Po smazání plátna už nic nebrání postupně vykreslit jednotlivé objekty z nového snímku. Jestliže je tento princip opakován tak sice dojde k animaci, ale dosažený efekt bude velmi špatný. Výsledný efekt je takový, že obraz problikává. Kdykoliv mezi jednotlivými kroky můžeme spatřit stav plátna. Můžeme tedy vidět jen pozadí, jen jeden objekt, nebo v lepším případě i několik objektů. Občas prostě některý z objektů neuvidíme.

Další možností je podobný postup. Jednotlivé objekty jsou také postupně vykreslovány na plátno, ale rozdíl je v přechodu na další snímek. Při tomto přechodu tentokrát není překresleno celé plátno, ale jen pohyblivé objekty. Tyto objekty jsou překresleny stejnými nebo většími objekty s barvou pozadí. Po smazání starého snímku jsou postupně vykresleny objekty z nového snímku. Tento postup je také velmi nedokonalý. Jde aplikovat pouze pro animace s minimálním množství objektů. Tyto objekty také musí být malé jinak opět bude docházet k problikávání.

Pro minimalizaci blikání animace je zapotřebí použít tzv. *offscreen buffer*. V něm je nejprve vytvořen obrázek s nakreslenými objekty v paměti (mimo prostoru obrazové části = *offscreen*) a následně je najednou vložen přímo na plátno. Nejběžnější vykreslovací technika využívající. *Offscreen buffer* se nazývá *double buffering*. Tato technika je vhodná, když je grafický kontext složitý nebo opětovně používaný. Dokáže redukovat čas vykreslení obrazu tím, že využívá *offscreen buffer*, který je pak následně zkopírován překreslen na obrazovku. Tento *double*

buffering je často používán pro animace. Pro *offscreen buffer* je vhodné použít třídu BufferedImage. Přes tuto třídu lze získat objektu Graphics2D voláním metody createGraphics(). Pak lze přímo přistupovat k obsahu obrazu. Při tvorbě Bufferedimage jako *offscreen buffer* mohou být použity všechny nástroje java2D API. Překopírování z Bufferedimage na obrazovku se provede jednoduše, zavoláním metody *drawimage(Image offscreenbuffer, 0, 0, this);*.



obrázek 7: Princip double bufferingu

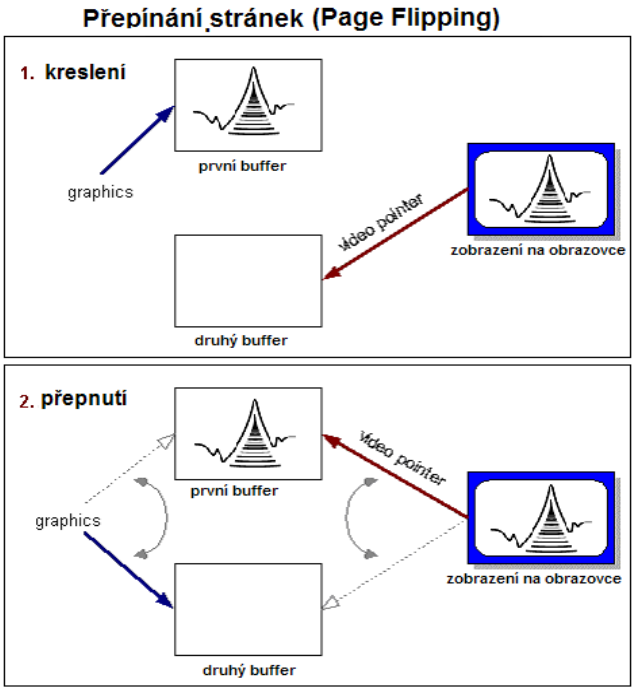
```
public class Buff extends Applet{
    BufferedImage bi;
    Graphics2D big;
    public void init(){
        Dimension dim = getSize();
        int w = dim.width;
        int h = dim.height;
        bi = (BufferedImage)createImage(w, h);
        big = bi.createGraphics();
    }


    public void paint(Graphics g){
        update(g);
    }

    public void update(Graphics g){
        Graphics2D g2 = (Graphics2D)g;
//zde se provádí kreslení do grafického kontextu který je pak vykreslen
        Big. …
        Big. …
        g2.drawImage(bi, 0, 0, this);
    }

}
```

Existuje ještě účinnější obrana proti blikání, a to *přepínání stránek*. Tento způsob je vhodnější pro grafické aplikace přes celou obrazovku. U tohoto principu se vykresluje do prvního bufferu a zobrazuje se druhý. Pak se přepne zobrazování na druhý a vykresluje se do prvního. Tato technika je implementována ve třídě BufferStrategy pouze pro samostatná okna a ne pro aplety.



obrázek 8: Princip přepínání stránek.

5. Experimentální aplikace

5.1. Aplet kyvadlo

Kyvadlo je jedním z často používaných systémů pro demonstraci mechanického kmitání. Matematické kyvadlo je pak zvláštní případ, kdy je zkoumán pouze hmotný bod zavěšený na tenkém vláknu zanedbatelné hmotnosti. Perioda, tedy doba kmitu matematického kyvadla, je přímo úměrná druhé odmocnině z délky závěsu. Perioda kmitání kyvadla nezávisí na hmotnosti hmotného bodu. Pohyb kyvadla je definován pomocí nelineární pohybové rovnice

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}t^2} + b\frac{\mathrm{d}\phi}{\mathrm{d}t} + \omega_0^2 \sin\phi = F\sin\Omega t, \qquad (1)$$

kde Φ je okamžitý úhel vychýlení kyvadla, b je člen zahrnující tlumení (tření, odpor vzduchu) a F je amplituda vnější budící síly o úhlovém kmitočtu Ω. V tomto apletu se předpokládá, že vlastní kruhový kmitočet kyvadla $\omega_0 = 1$. Parametry lze v apletu nastavit a tak pozorovat chováni matematického kyvadla při změně budící síly nebo tlumení.

Jelikož kyvadlo je popsáno nelineární pohybovou rovnicí, což je diferenciální rovnice druhého řádu, je zapotřebí jí řešit některou z numerických metod. Numerické řešení obyčejných diferenciálních rovnic je postup, jak získat přibližné řešení obyčejných diferenciálních rovnic, když nejsme schopni rovnice vyřešit přesně (analyticky). Jednotlivé kroky pro pohyb kyvadla jsou vypočítané pomocí metody Runge-Kutta. Metody Runge-Kutta patří mezi krokové metody a jsou velmi často používané. Jsou výpočetně poněkud složitější, ale také přesnější - metoda Runge-Kutta 2.řádu má přesnost h^2, metoda Runge-Kutta 4.řádu má přesnost h^4. Pro diferenciální výpočet v apletu je zvolena, metoda Runge-Kutta 4.řádu. Vzorce pro diferenciální rovnici jsou obecně zapsány v následujícím tvaru.

$$\begin{aligned} k_1 &= f\left(t_n, y_n\right) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f\left(t_n + h, y_n + hk_3\right) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \qquad (2)$$

5.1.1. Ovládání apletu

Pomocí tlačítek start\stop se spouští a vypíná výpočet včetně vykreslování polohy kyvadla. Do textových polí je možné zadávat následující údaje:

- Φ - počáteční úhel vychýlení kyvadla (v radiánech),
- dΦ/dt- počáteční úhlovou rychlost kyvadla,
- *b* - tlumení kyvadla,
- Ω - úhlovou frekvenci budící síly působící na kyvadlo,
- *F* - amplitudu této síly.

Stisknutím tlačítka „restart“ se načtou všechny údaje z textových polí. Zatím co při stisknu tlačítka „změna“ nedojde k načtení počátečních podmínek (počáteční úhel, počáteční úhlová rychlost).Tím se může simulovat situace, kdy se změní pouze budící síla nebo tlumení a kyvadlo tak může na tuto změnu za běhu reagovat. Dále pomocí třídy java.awt.checbox lze přepínat mezi jednotlivými grafy, na kterých jsou zobrazovány funkce vyplívající z pohybu kyvadla.

5.1.2. Grafické řešení apletu

Grafická uživatelské prostředí jsou v Javě dvě – starší AWT (Abstact Window Toolkit) a novější JFC Swing (Java Foundation Classes). AWT je součástí Jawa Core API od JDK 1.0 a Swing od JDK 1.2. Pro tento Aplet jsou zvoleny komponenty typu AWT.

Samotné kyvadlo je přímo vykreslováno na aplet. Jelikož se skládá pouze ze dvou geometrických tvarů nebylo zapotřebí použít *double bufferingu*. Technika *double bufferingu* byla použita pro vykreslování grafů. Pro jednotlivé zobrazování grafu je použit layout manager CardLayout. Správce rozvržení CardLayout umožňuje přidávat do kontejneru mnoho komponent, avšak v jednu chvíli zobrazit pouze jednu dceřinou komponentu. Toto je způsobeno tím, že každá komponenta je přidána na stejné místo. Zobrazovanými komponentami v apletu kyvadlo jsou panely s grafem. Bohužel v AWT není přepínání karet přímo podporováno, a proto, pokud se CardLayout použije, zkombinuje se nejčastěji s *choice*, aby mohl uživatel řídit přepínání karet.

5.2. Aplet sedačka

Tento aplet představuje sedačku, jejíž hmotnost je soustředěna do těžiště. Předpokládá se, že sedačka může pohybovat lineárně ve svislém směru podél vedení bez tření a v klidovém stavu na ni působí síly

Soustava sedačky je tvořena tlumičem, pružinou a samotnou sedačkou. Na tomto apletu lze sledovat vývoj polohy sedačky při zatěžování soustavy a její dynamické chování v přechodu mezi klidovými stavy. Lze i simulovat případ kdy na sedačku usedne za neustáleného stavu řidič, souřadnice *z* se dynamicky mění až do dalšího nového ustálení. Samotné početní řešení vychází z principu rovnováhy sil.

Na sedačku působí tyto síly:

tíha řidiče	F1= M*g	(3)
setrvačná síla	F2= (m+M)*a	(4)
síla tlumiče	F3= c*v	(5)
síla pružiny	F4= k*s	(6)
Lze tedy sestavit rovnici:	F1 = F2+F3+F4	(7)
Po rozepsání:	M*g = (m+M)*a +c*v+k*s	(8)

Následně jsou do rovnice (8) dosazeny vztahy pro zrychlení (a = y´´(t)), rychlost (v = y´(t)) a dráhu (s = y(t)).

Po dosazení vznikne diferenciální rovnice druhého řádu.

$$M.g = (m+M) \,.\, y''(t) + c.y'(t) + k.y(t) \qquad (9)$$

Dále je řešení podobné jako u apletu kyvadlo. Pro výpočet jednotlivých kroků je opět použitá metoda Runge-Kutta 4.řádu.

5.2.1. Ovládání apletu

Pomocí tlačítek start\stop se spouští a vypíná výpočet spolu s vykreslování polohy sedačky. Do textových polí je možné zadávat následující údaje:

- Y_0 - Počáteční polohu
- g - Tíhové zrychlení
- m - Hmotnost sedačky
- M - Hmotnost řidiče
- k - Tuhost pružiny
- c - Tlumení

K načtení údajů pro výpočet dojde v apletu až po stisku tlačítka změna, nebo restart. Při použití tlačítka změna nedojde k načtení počáteční polohy. To umožňuje měnit zátěž v průběhu simulace. Tím je nahrazena situace, kdy člověk vstane nebo sedne na sedačku v době, kdy systém není ustálený. Posledním tlačítko slouží pro otevření okna s grafem. V grafu jsou zobrazeny funkce rychlost a vertikální polohy sedačky v závislosti na čase. V apletu je také možnost regulovat rozsah animace. Tímto rozsahem se nastavuje konstanta, kterou je násobena poloha sedačky. Tím si uživatel může sám nastavit měřítko. Po zvětšení této konstanty je možné i sledovat minimální změny. V případě kdy jsou působící síly na sedačku natolik velké, že by se sedačka ocitala mimo hranice apletu, je možné tento rozsah zmenšit a tak udržet sedačku na viditelné ploše.

5.2.2. Grafické rešení apletu

Oproti předchozímu apletu jsou tu použity nejen objekty typ AWT, ale i typu swing. Grafické uživatelské rozhraní AWT není příliš vzhledné. Hlavní výhodou je jeho podpora na všech platformách které jsou Javy , ale na druhou stranu, vzhledově se jedná spíše o podprůměrné rozhraní. Oproti tomu Swing má velmi dobrý grafický model[3]. Má všechny komponenty, které by mělo mít každé moderní uživatelské rozhraní. Při vývoji Swingu byl ovšem obětován výkon.

Pro dynamické zobrazování sedačky je použita třída Image. S pomocí této třídy je vytvořen snímek obsahující samotnou sedačku, schéma sedačky a pozadím. Takto vytvořený obraz je najednou vykreslen na plochu apletu. Pro zobrazení grafu je využita možnost vytvoření okna, které se tváří samostatně a nezávisle na prohlížeči. Pro vytvoření samotného okna stačí

[3] Příklady povedených Swing aplikací lze najít třeba na http://jgoodies.com nebo na http://java.sun.com/products/jfc/tsc/sightings/S01.html.

vytvořit instanci třídy *Frame*. Do nově vytvořeného okna se pak může přidat menu a další dialogové prvky, nebo do něj kreslit přímo předefinováním metody *paint()*.

Závěr

Technologie Java apletů byla bleskově přijata a jazyk si záhy získal ohromnou popularitu. V současné době jsou aplety používány jen ve velmi malé míře. Své použití si našli zejména jako demonstrační prostředek pro fyzikální animace. Za zmínku určitě stojí software CabriJava, který umožňuje publikovat na internetu dynamické geometrické obrázky právě pomocí apletů. Tyto aplety pak slouží zejména pro výuku, kde pomohou k lepšímu pochopení dané látky. Aplety se také používají jako uživatelské rozhraní pro připojení k servletu.

Za výhody apletu se udává hlavě jejich nezávislost na platformě, automatická instalace, bezpečnost. Jako nejčastější problém při nezobrazení apletu je stará, nebo dokonce chybějící instalace (plug-in), pro podporu Javy. Pro spuštění apletu je nutné mít nainstalované minimálně Java runtime, které obsahuje JVM a API. V případě použití jiných API pro aplet, než je standardní např. Java Media Framework, Java Advanced Imaging API a Java 3D API, je koncoví uživatel přinucen si tyto knihovny doinstalovat, aby mohl být aplet spuštěn. Další problém může být v podpoře webového prohlížeče nebo v jeho nastavení.

V bakalářské práci jsme se zaměřili hlavně na knihovny Java 2D. Toto API nabízí řadu nástrojů pro zpracování grafiky. Přesto, že vzhledem k rozsahu knihovny nebylo možné popsat všechny dostupné třídy a jejich metody, které Java2D nabízí, pokusili jsme se alespoň popsat základní třídy pro tvorbu 2D grafiky, případně třídy mající možné využití v apletech. Java2D je napsána přímo v programovacím jazyku Java, což nabízí zajímavé možnosti při rozšiřování o další funkce a samozřejmě tím přináší jednotné rozhraní na všech platformách, na nichž je možno provozovat Javu. Avšak zároveň je implementace v Javě důvod pro zpomalení výsledné aplikace. Použití Java2D API lze tedy doporučit pokud nezáleží příliš na výsledné rychlosti programu a aplikace má především demonstrativní funkci.

Použita literatura a informační zdroje:

[1] Brůha, L: *JAVA, hotová řešení*. Computer Press, 2004

[2] Herout, P: *Učebnice jazyka Java*. KOPP, 2000

[3] Herout, P: *Java - grafické uživatelské prostředí a čeština*. KOPP, 2001

[4] Kiszka, B: *1001 tipů a triků pro programování v jazyce Java.* Computer Press, 2003

[5] Musciano, Ch. Kennedy, B: *HTML a XHTML Kompletní průvodce*. Computer Press, 2003

[6] Virius, M.: Java pro zelenáče, Neocortex, 2005

Webove zdroje:

[w1] Bosák, R. Martin Fanta, M. Peřina, M. *Pár kapek Javy.*
[online]. [cit 19. 2. 2007] URL: <http://www.ataco.cz/perina/par-kapek/intro.html>

[w2] David, A. *Java2D API*.
[online]. [cit 15. 3. 2007] URL:
<http://www.cgg.cvut.cz/~apg/apg-tutorials03/ch04s72.html>

[w3] Evans, R. *Java 2D API*.
[online]. [cit 28. 2. 2007]. URL: <
http://webdev.apl.jhu.edu/~rbe/java/Java_2D/Java2D.pdf>

[w4] Herout, P. *Grafika*.
[online]. [cit 17. 4. 2007] URL: <http://home.zcu.cz/~honzik1/oop/grafika-slajdy.pdf>

[w5] Kotala, Z. Toman P. *Java.*
[online]. [cit 04.03.2007] URL: <http://dione.zcu.cz/java/sbornik/toc.html>

[w6] Kuželka, O. *Java - pokročilá grafika.*
[online] [cit 16. 12. 2006] URL: <http://interval.cz/serialy/java-pokrocila-grafika/>

[w7] Miška, O. *Java 2D a Java Advanced Imaging.*
[online] [cit 15. 3. 2007] URL: <http://www.cgg.cvut.cz>

[w8] Prokop, L. *Java 2D API.*
[online] [cit 29. 3. 2007] URL:
<http://web.quick.cz/prokop.l/skola/3rocnik/6semestr/java/prezent/java.htm>

[w9] Semecký, J. *Naučte se Javu.*
[online] [cit 9. 1. 2007] URL: <http://interval.cz/serialy/naucte-se-javu/>

[w10] Sun Microsystems, Inc. *The Java™ Tutorials.*
[online]. [1. 5. 2007] URL: <http://java.sun.com/docs/books/tutorial/2d/TOC.html>

[w11] Sun Microsystems, Inc. *Java 2D™ API Specification*.
[online] [26. 8. 2005] URL: <http://java.sun.com/j2se/1.4.2/docs/guide/2d/spec.html>

[w12] Terber, R. *Stručný úvod do jazyka JAVA*.
[online]. [24. 9. 2006] URL:
<http://www.gapo.cz/doc/docs/java/uvod_do_jazyka/index.html>