Copyleft: Kryštof Pešek (2013)

Původní elektronický dokument:
https://github.com/K0F/processing_1

Vydalo Nakladatelství Akademie múzických umění v Praze
ISBN 978-80-7331-224-4

Vysázeno textovým systémem XƎLaTeX.

Poděkování

Rád bych poděkoval všem lidem, kteří se podíleli na vzniku této publikace.

Nejprve děkuji Tomáši Pospiszylovi, vedoucímu této práce, za neúnavný dohled nad podobou následujícího textu. Velmi rád bych poděkoval Miloši Vojtěchovskému za oponenturu práce, Andree Slovákové a Zdeňku Holému za konzultace a pomoc při vydávání publikace. Rovněž bych rád poděkoval Janě Křížové a Olze Stehlíkové za jejich korektorskou práci a usměrnění příručky do mezí čitelnosti. Naposledy bych rád velmi poděkoval Janu Omastovi za podobu výsledného tisku.

– Kryštof Pešek

Abstrakt

Příručka *Processing Beta* je základním úvodem do jazyka *Processing*, ten byl navržen v akademických dílnách *Massachusetts Institute of Technology* k snazšímu a co možná nejsvobodnějšímu přístupu tvůrců – obecně vzato – všech netechnických oborů k pokročilým technologiím. Základ pro tuto příručku vznikl při sérii otevřených dílen o programování vedených Kryštofem Peškem na pražské FAMU.

Publikace vychází v tištěné podobě v době, kdy vývoj programovacího prostředí *Processing* opouští vývojovou fázi 1.0 a přibližuje se k fázi 2.0. Problematika, které se tento průvodce dotýká, je natolik základní, že se vztahuje na obě verze *Processingu*. Název příručky *Beta* by tedy měl co nejobecněji shrnovat základy všech dostupných verzí do dnešních dnů.

Podobné příručky v posledních letech vznikly pouze v anglickém jazyce, např. Casey Reas a Ben Fry: *A Programming Handbook for Visual Designers and Artists* (2007), Ira Greenberg: *Processing Creative Coding and Computational Art* (2007), Daniel Shiffman: *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction* (2008), Tom Igoe: *Making Thinks Talk: Practical Methods for Connecting Physical Objects* (2007). V českém jazyce žádná podobná kniha doposud nevyšla.

Příručka by měla přiblížit, ve shodě s původním smyslem *Processingu*, technologie i netechnicky zaměřeným oborům, hlavně pak studentům výtvarných škol bez předešlé zkušenosti – nebo jen s malou praxí s programováním.

Publikace byla zpracována jako diplomová práce studijního oboru Centra audiovizuálních studií na pražské FAMU a vznikla za podpory grantu *SGS AMU* (2011 – 2012).

Obsah

1 O knize . . . 9
1.1 Na úvod . . . 9
1.1.1 Cíle příručky . . . 10
1.1.2 Uspořádání informací . . . 11
1.1.3 Pravidla formátování v knize . . . 12

2 Od textu k obrazu . . . 13
2.1 Krajiny a světy . . . 13
2.2 Toxilibs . . . 16
2.3 Local Bilateral Symmetry . . . 17

3 Dobré zdroje v začátcích . . . 21
3.1 Síť . . . 21
3.2 Examples, příklady přímo v Processingu . . . 22
3.3 Knihy a průvodci . . . 22
3.4 Experiment . . . 22

4 K čemu slouží programovací jazyk? . . . 25
4.1 Počátky programovacího jazyka . . . 25
4.2 Jednoduchý programovací jazyk . . . 27
4.3 Dokonalost jazyka . . . 28
4.4 Volba vhodného jazyka . . . 29
4.5 Proč zrovna Processing? . . . 30
4.6 Tvorba softwaru . . . 32
4.6.1 Otevřenost softwaru . . . 33
4.6.2 Syntetický obraz . . . 34

4.6.3 Empirický přístup k programování 34

5 Processing jako prostředí 37
5.1 Základní prostředí 38
5.1.1 Základní diskové operace 39
5.1.2 Sketch 39
5.1.3 Sketchbook 40
5.1.4 Editor 41
5.2 Klávesové zkratky 42
5.3 Soustředěná činnost 42
5.4 Základní pravidla a zvyklosti 43

6 Stavba programu a syntax 45
6.1 Logika programování 45
6.1.1 Komentář 45
6.1.2 Základní datatypy 46
6.1.3 Seznam základních datatypů 47
6.1.4 Barva 48
6.1.5 Tisk do konzole 48
6.1.6 Základní operace s datatypy 50
6.1.7 Základní struktura programu 55
6.2 Zobrazení 56
6.2.1 Orientace v prostoru 56
6.3 Hodnota a její zobrazení 59
6.4 Kresba 60
6.4.1 Výplň a obrys 61
6.5 Pohyb 63
6.5.1 Proměnlivost 63
6.5.2 Animace 65
6.5.3 Dynamika pohybu 67
6.6 Podmínka 68
6.6.1 If 69
6.6.2 Else 73
6.6.3 ? : 74
6.7 Interakce 74
6.8 Pole 79

6.9 Smyčka 81
6.10 Uspořádání struktury programu 83
6.10.1 Funkce 83
6.10.2 Funkce a jejich datatypy 87
6.10.3 Třída a objekt 89
6.10.4 Práce s objekty 92
6.11 Náhoda 95
6.11.1 Perlinův šum 96

7 Metody zobrazení 99
7.1 Renderovací mody 99
7.2 3D zobrazení 100
7.2.1 Kamera 101
7.2.2 3D objekty 103
7.3 Transformace 105
7.3.1 Použití transformací 108

8 Práce s daty 111
8.1 Ukládání informací 111
8.1.1 Ukládání obrazových dat 112
8.1.2 Ukládání holých dat 114
8.1.3 Tisk do PDF, ukládání vektorů 116
8.2 Načítání informací 119
8.2.1 Načítání obrázků 120
8.2.2 Načítání textových souborů 121
8.3 Operace s textem 122
8.3.1 Parsing, získávání hodnot z externích dat 122
8.4 Vizualizace hodnot 124

9 Rozšíření Processingu 127
9.1 Knihovny 127
9.1.1 Vestavěné knihovny 127
9.1.2 Komunitní knihovny 129
9.1.3 Práce s knihovnou 129
9.2 Poznávejte, experimentujte! 134

Slovník 134

Rejstřík 148

Kapitola 1 O knize

1.1 Na úvod

Vážení čtenáři,

tato kniha by pro vás měla být povzbuzením při osvojování si vašeho prvního programovacího jazyka. Autor této publikace nepředpokládá žádnou vaši předchozí zkušenost s programováním. V případě, že již určitou praxi máte, některé kapitoly pro vás mohou být jednoduché. Kniha by vás měla postupně provázet při získávání dovedností programování stroje od základních výpočetních operací k tvorbě pokročilejších nástrojů.

Příručka je psána ve sledu, který bych uvítal, kdybych se sám měl učit programovat. Programování je bezesporu myšlenkově náročná operace. Mým záměrem je postupně tuto náročnost redukovat a proměnit psaní programu v lehkost.

Cestu k této zručnosti vám neumím jednoduše předat, pedagogický talent jsem nikdy nepocítil. Tato kniha by proto měla být odložena vždy, bude-li mít čtenář nutkání si něco sám vyzkoušet. To považuji za absolutně nejrychlejší a zároveň nejlepší způsob učení.

Proces tvůrčího programování obsahuje prvek intuice, která vychází podstatnou měrou ze zkušenosti. Pro podporu vaší intuice při psaní programu je nutné ovládnout základní jazyk natolik, abyste měli určitou jistotu. Přestanete-li se zabývat funkčností programu a získáte-li v ní jistotu, přijmete jazyk za vlastní nástroj, a pak můžete nechat hovořit právě vaši intuici. Intuici považuji za avantgardu logiky, protože je před ní vždy minimálně o krok napřed.

K vybudování intuice je především nezbytná schopnost pozorování. Samotná intuice nezmůže v komunikaci se strojem nic. Stroj neví, co počítá, ale dokáže spočítat i to, co sami nevíte.

Stroj je rychlý nástroj pro ověření vaší intuice a je zapotřebí být pozorný. Můžeme to nazvat experimentální přístup, výsledky by měly být vždy intuitivně a hlavně kriticky zkoumány. Zpětnou dekonstrukcí intuice, rozborem vaší jistoty, získáváte postupně dar rozumění věci. Dar je to danajský, rozumění je jen ustálená podoba znalostí, která musí být opět zpochybněna intuicí; nutno podotknout, že opakovaným prověřením se často jen více utvrzuje.

Není-li tento programovací jazyk vaším prvním, prosím vás o trpělivost ohledně podrobností, do kterých v úvodu této knihy zabíhám. Šíře znalostí, které se mohou vyskytovat u potenciálních čtenářů tohoto průvodce, je pro autora první velkou neznámou.

V obou případech prosím o shovívavost ve způsobu popisu programovacího jazyka a prostředí *Processingu*. Sám jako samouk a člověk zaměřený zejména výtvarným směrem nemohu zaručit absolutní, stoprocentní a všeobecnou platnost všech tvrzení. Tímto vás tedy žádám o věcné podněty pro pozdější doplnění nebo přeformulování jednotlivých tezí.

Má snaha provést začínající i středně pokročilé uživatele programovacím jazykem bude vždy nedostatečná, berte tuto knihu, prosím, spíše jako průvodce nesnadnými začátky. Čtete-li knihu z jiných důvodů, než abyste se naučili programovacímu jazyku, pokusím se text proložit poznatky nabytými mojí několikaletou zkušeností se sdílením života se stroji. Dostala-li se vám tato publikace do rukou jinou cestou, nebo dokonce náhodou či omylem, tedy věru netuším, co v ní dále najdete, a proto bych i vás rád pobídl alespoň k začtení se do světa skriptů a kódů.

1.1.1 Cíle příručky

Cílem této knihy je provést uživatele-začátečníka sérií příkladů, které budou ilustrovat znalosti nezbytné k naprogramování jednotlivých stadií programu.

Prvním cílem této příručky je popsat základní terminologii používanou v jazyce *Processing*, tato pasáž je nezbytná pro další pokračování. Jestliže

se cítíte v tento okamžik připraveni, můžete si rovnou nalistovat začátek průvodce *(viz Základní prostředí, str. 38)*.

Následující krok je pochopení datatypů a operací s nimi. Proměnné představují základní stavební jednotku *Processingu*, orientujete-li se například v jazyce *Java*, směle tento krok rovnou přeskočte, všem ostatním čtenářům ovšem stěžejní pasáž příručky důrazně doporučuji přečíst *(viz Logika programování, str. 45)*.

Dalším cílem publikace je uživatele *Processingu* uvést do zábavnější části programování: experimenty s obrazovým výstupem. Kapitoly týkající se základních kreslicích funkcí *(viz Zobrazení, str. 56)* jsou dobré pro naprosté začátečníky. Těm lehce pokročilým bych doporučil sekci věnovanou pohybu *(viz Pohyb, str. 63)* a kapitolu věnovanou interakci *(viz Interakce, str. 74)*.

Všechny kapitoly postupně čtenáře vedou k osvojení si objektově orientovaného jazyka. Zde se *Processing* stává opravdu mocným nástrojem. Pasáže v příručce věnující se podrobněji práci s objekty najdete zejména v pokročilejších kapitolách *(viz Uspořádání struktury programu, str. 83)*.

Všem čtenářům ovšem doporučuji alespoň si pročíst úvod, a tím získat základní představu o kontextu programovacího jazyka *Processing*.

1.1.2 Uspořádání informací

Text je členěn do jednotlivých kapitol a dále strukturován do dvou úrovní podkapitol. Pořadí kapitol by mělo odpovídat sledu informací nezbytných k plynulému učení se programovacímu jazyku *Processing*.

Řazení a obsah jednotlivých kapitol vychází z mé vlastní zkušenosti. Z veliké míry koresponduje s podobnými průvodci, jež píšou samotní tvůrci a širší komunita uživatelů kolem programovacího jazyka *Processing*.

Zvolená forma textu odpovídá ověřeným postupům. Následující text bude podléhat určitým zákonitostem. V knize se objeví několik typů textu, které budou vždy odlišeny formou zápisu.

1.1.3 Pravidla formátování v knize

Pravidla jsou následující:

Obyčejný text: popis ve formě klasického textu

kód:
```
/**
* Barevny text v sedem ramovani bude znacit
* strojovy kod ve forme ktera je srozumitelna
* Processingu.
*/

boolean pravda = true;
```

CONTROL + T: klávesové zkratky, které jsou psány „verzálkami“

GNU / GPL: výrazy a pojmy, jejichž definici můžete nalézt ve slovníku na konci knihy

boolean: jednotlivé příkazy a oficiální příkazy *Processingu*, které jsou zároveň zařazeny do slovníku

promenna: proměnné a názvy použité v kódu, které bude potřeba dovysvětlit v textu

cvičení: Pomocí cvičení a otázek si můžete kontrolovat v průběhu textu nabyté znalosti. Bublina bude vždy značit důležité informace, například ty, které si musíte zapamatovat, abyste se mohli orientovat v následujícím textu.

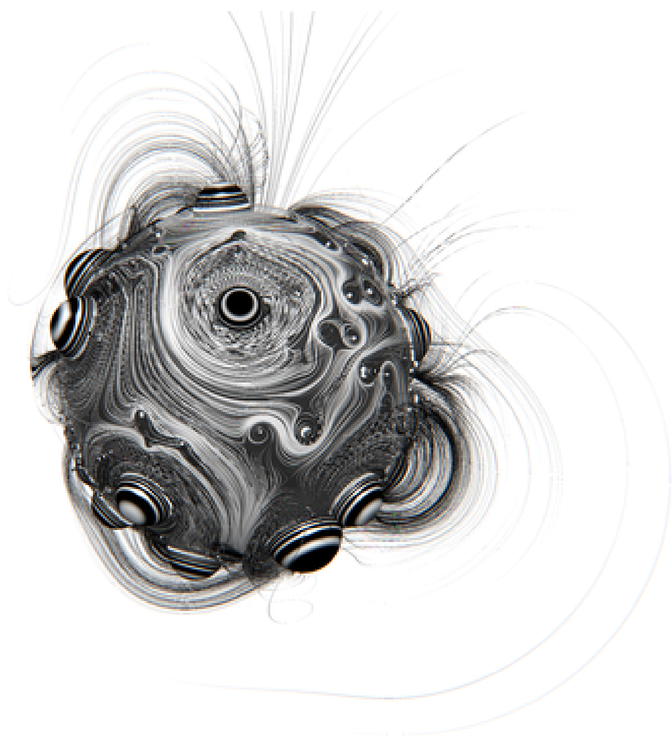
Zapamatujte si, prosím, jak budou znázorněna cvičení.

Kapitola 2 Od textu k obrazu

2.1 Krajiny a světy

Robert Hodgin, také známý pod pseudonymem *flight 404*, zůstává jednou z oslavovaných ikon processingové komunity. Jeho práce s vizualizacemi je vždy velmi dynamická a technicky obdivuhodná.

Hodgin pracuje s jazyky *Processing* a *Cinder*. V obou případech skrze tato prostředí ovšem programuje přímo aplikaci *OpenGL*, která zprostředkovává pokyny samotnému grafickému jádru.



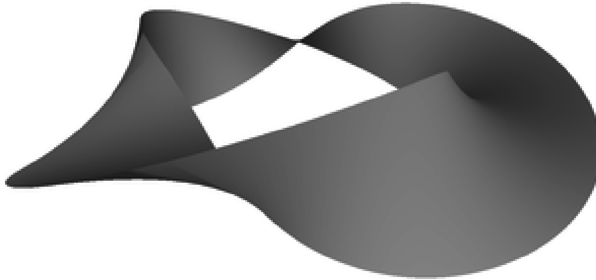
Ilustrace pochází z Hodginovy grafické práce, která byla vybrána jako jedna ze 42 výtvorů do prvního vydání knihy *Written Images*.

Publikace je kurátorským projektem a samostatným výtvarným počinem dánského tvůrce Marcina Ignace.

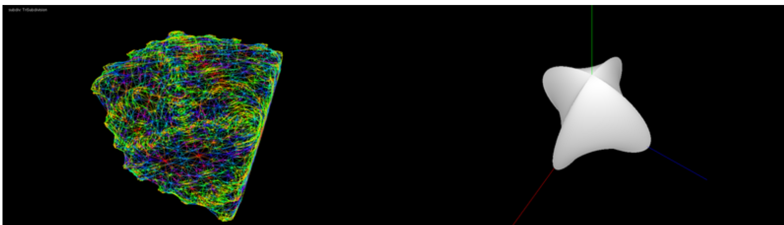
První výtisk knihy byl vydán v roce 2011, kniha byla zpracována softwarem *Processing*. Jde v podstatě o konceptuální dílo, automatizací vychází každý výtisk s posunem o jedno okénko, a tím ilustruje ducha generativní tvorby.

Kniha obsahuje generativní obrazy následujících umělců: *386dx25, Antoni Kaniowski, Ariel Malka, Carl-Johan Rosén, Casey Reas, clone, David Bollinger, David Bouchard, e m o c, flight404, Golan Levin, Jonathan McCabe, Jörg Piringer, Julien Deswaef, Kim Asendorf, kraftner, Leonardo Solaas, Lia, Luke Sturgeon, Marcin Ignac, Marius Watz, Michael Zick Doherty, Mitchell Whitelaw, Moka, Nervous System, Oliver Smith, paolon, Perceptor, rhymeandreason, Ricard Marxer, Roberto Christen, Rui Madeira, Ryan Alexander, Ryland Wharton, Sansumbrella, sojamo, Stefano Maccarelli, Szymon Kaliski, Victor Martins, W:Blut, William Lindmeier, zenbullets.*

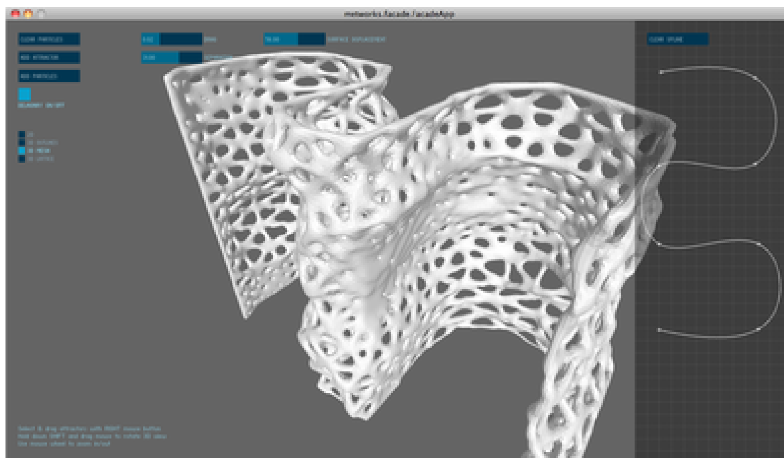
2.2 Toxilibs



Karsten Schmidt působí v processingové komunitě dlouhodobě a je více znám pod pseudonymem *Toxi*. Schmidt je autorem sady rozsáhlých knihoven nazvaných souhrnně *Toxilibs*. Jeho tvorba je zaměřena technicky na vývoj samotných knihoven, tedy jednotlivých modulů rozšiřujících funkce *Processingu (viz* Knihovny*, str. 127)*. V případě knihovny *Toxilibs* například rozšířením o funkce pro konstrukci trojdimenzionálních objektů.



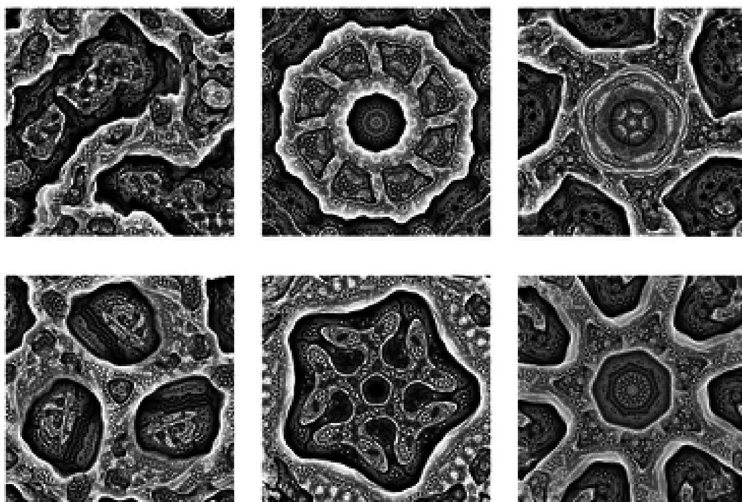
Schmidtova práce vyniká zejména pro možnost recyklovatelnosti a další užitné hodnoty samotného kódu. Schmidtovy algoritmy jsou propracované a matematicky pokročilé. Práce Karstena Schmidta dobře ilustruje podstatu komunitní práce ve formě knihoven, které volně cirkulují mezi uživateli a jsou volně využívány v duchu svobodného softwaru.



Tvorba Karstena Schmidta se dá generalizovat jako užitá nebo aplikovaná. Pomocí bohaté zásoby vlastního softwaru, veliké zdatnosti a technické inovativnosti Karsten Schmidt svým softwarem plynule navazuje na potřeby architektů a inženýrů.

2.3 Local Bilateral Symmetry

Pozoruhodné algoritmy Jonathana McCaba vynikají svou spektakulárností, která je založena na podrobné studii a rozkladu Turingových vzorců. Jednotlivé variace téhož algoritmu vytvářejí struktury, které se vzdáleně podobají uspořádání organických materiálů v kostech nebo jiným přírodním vzorům.



I když se to zprvu nabízí, McCabovy obrazy se nedají posuzovat pouze výtvarně. Nejvíce pozoruhodné jsou způsobem svého vzniku. Obrazy přímo odkazují na výzkum Alana Turinga, konkrétně na text vydaný v roce 1952 s názvem *The Chemical Basis of Morphogenesis*, který pojednává o vlivu chemických sloučenin na uspořádání růstu tkáně vícebuněčných organismů. Turing podobnou difuzi prvků v textu rozpracoval matematicky a po téměř padesáti letech na bázi původních algoritmů vznikají tyto obrazy. Podobnost v uspořádání lze nalézt například ve snímcích tkáně pod elektronovým mikroskopem, a i když se jedná o ryze výtvarnou interpretaci algoritmu, může dovést k zamyšlení i vědecké kruhy.

Kapitola 3 Dobré zdroje v začátcích

3.1 Síť

Autor knihy předpokládá, že čtenář má přístup ke globální síti internet.

Nejlepší referencí pro práci s *Processingem* zůstává tato síť. *Processing* byl vyvinut síťovou komunitou a je také v síti nejlépe zdokumentován. Existuje několik dobrých obecných stránek s průvodci. Za všechny lze jmenovat domovskou stránku projektu *processing.org*. Zde naleznete téměř vše, co byste mohli potřebovat. Nacházejí se zde zdokumentované veškeré příkazy *Processingu*. Je zde i záložka *Learning* s jednotlivými průvodci různých oblastí.

Nejadresněji se vám dostane pomoci prostřednictvím webových fór nebo IRC-kanálů. Na fórech se již řešil téměř jakýkoli problém, stačí proto vyhledávat klíčová slova konkrétního dotazu a velmi rychle byste měli nalézt řešení.

Na síti je dále obrovské množství uživatelů volně sdílejících své programy včetně zdrojových kódů. Nejpočetnější komunita sídlí zřejmě na portálu *openprocessing.org* založeném a spravovaném Sinanem Asciogluem.

Veškerá dokumentace je v angličtině včetně webových fór i dalších knih o programování. Nemáte-li problém s anglickým jazykem, míst k zdokonalování se v *Processingu* bude vždy dostatek.

Doporučuji se nejprve seznámit s projekty na internetu, získáte tím představu o tom, k čemu se vlastně Processing využívá.

3.2 Examples, příklady přímo v Processingu

Tato pomoc je vždy po ruce. *Processing* je navržen k výuce a je připraven poskytovat pomoc ze své podstaty. Přímo do aplikace *Processingu* jsou naimplementovány základní postupy programování. Témata jsou pokryta od základních operací až po pohyb v trojdimenzionálním prostoru nebo manipulaci s pixely.

Příkladů je na webu dostatek a jsou přehledně uspořádány. Vřele doporučuji si je čas od času projít.

Ke všem příkladům se dostanete z *Processingu* skrze nabídku *File* → *Examples* *(viz Základní prostředí, str. 38)*.

3.3 Knihy a průvodci

Ano, jednoho průvodce právě držíte v rukou (nebo čtete na obrazovce). V minulosti vyšla bezmála desítka podobných knih na stejné téma. Jsou bezesporu obsáhlejší, než kdy měla tato příručka v úmyslu.

Z nich bych vám nejvíce doporučil základní příručku od samotných tvůrců *Processingu* Caseyho Rease a Bena Frye nazvanou *Processing: A Programming Handbook for Visual Designers and Artists* (2007). A rozhodně pak knihu Daniela Shiffmana *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction* (2008). Je jednou z velmi podrobně napsaných knih o *Processingu*.

Knihy jsou cenově vcelku dostupné a dají se opatřit přes internet. Rozhodně bych to doporučoval všem, kteří potřebují mít opravdu dobrou dokumentaci vždy při ruce. V neposlední řadě bych také upozornil na to, že koupí knihy podporujete tvůrce svobodného softwaru, a tím potažmo i svobodný software jako takový *(viz Otevřenost softwaru, str. 33)*.

3.4 Experiment

Nejdůležitějším zdrojem vašich vlastních zkušeností bude vždy samotné experimentování s programováním. Na tuto část kladu největší důraz a v průběhu této knihy to i několikrát zopakuji: je vždy dobré odložit knihy

a reference a odvážit se do neznámých vod. Zkušenost, kterou takto naberete, bude vždy ta nejhodnotnější. Nebojte se proto sami experimentovat *(viz Empirický přístup k programování, str. 34)*.

Příštích několik kapitol se věnuje obecnějším úvahám o programovacích jazycích, již nyní si můžete Processing nainstalovat.

Kapitola 4 K čemu slouží programovací jazyk?

4.1 Počátky programovacího jazyka

Výchozím bodem v tvorbě počítačové logiky, jak ji známe dnes, byla přirozená lidská komunikace v podobě řeči. Člověk používá jazyk a při tvorbě zcela nového systému pravidel zákonitě sahá po známém prostředku.

Na počátku programování ale nebyl lidský jazyk. Se stroji se v jejich raných fázích vývoje hovořilo čistě strojovým jazykem, v podobě číselných řad nebo později holého strojového kódu, který přešel na logiku pokročilejších obvodů. Programovací jazyk se objevil již v padesátých letech minulého století a osvědčil se jako prostředek pro člověka, který začínal být limitován oproti kapacitě stroje schopného obsáhnout složitější úlohy. Řešením pro vzrůstající složitost logiky programů se stala lidská řeč.

Řeč se v počátcích tvorby struktur pro program jevila jako člověku nejpřirozenější řešení. Velmi brzy tvůrci zjistili, že pomocí jazyka lze sice definovat složitý problém pro člověka, vyvstává tím ovšem problém definovat význam takového jazyka pro stroj.

Počítačový jazyk není jeden, počítačových jazyků jsou celé rozvětvené rodiny. Konceptů, jak lépe hovořit srozumitelně pro člověka a zároveň ke stroji, je celá řada, žádný z nich nedokáže plně kopírovat lidský jazyk v jeho přirozené struktuře. Důvodů je několik, jeden z nejpodstatnějších je fakt, že lidská řeč neodpovídá organizaci ve struktuře stroje prostě proto, že člověk své řeči sám nerozumí natolik, aby popsal všechny její zákonitosti logickou cestou.

Jednoduše řečeno, lidský jazyk není popsán tak dokonale, abychom ho byli schopni vysvětlit stroji. Pokusy o průnik lidské řeči a logického obvodu sahají do absolutních počátků interakce člověka se strojem a představují pro tvůrce strojů závažný problém. Tento problém v padesátých letech minulého století definoval Alan Turing, jeden ze zakladatelů dnešní výpočetní techniky. Alan Turing spolu s Gordonem Wechmanem stáli v době druhé světové války u vývoje stroje k rozkrývání německých šifer nazvaného *Bomba (The Bomb)*. Alan Turing v poválečné éře definoval formální rámec počítače, jak jej známe dnes. Koncepce později nazvaná Turingův stroj vycházela zejména z matematických potřeb a jednalo se o snahu vytvořit kompletní výpočetní jednotku schopnou řešit veškeré známé matematické operace.

Je pozoruhodné, že Alan Turing se již v raných stadiích vývoje počítačů zabýval takzvanou umělou inteligencí. Z toho je zřejmé, že stroje podobné počítačům se od počátku svého vzniku připodobňují struktuře lidského uvažování. Alan Turing sestavil pro stroje známý experiment nazvaný Turingův test, jenž měl prověřit schopnosti stroje replikovat lidské chování. Není náhodou, že prostředkem pro komunikaci v testu byla zvolena právě řeč. Test spočíval v modelové situaci stroje schopného napodobit lidskou komunikaci tak dokonale, aby člověk nebyl schopen rozeznat, že komunikuje se strojem. Zajímavá na této definici lidské inteligence je zejména její vágnost, která v podstatě ilustruje míru porozumění logického uvažování lidské mysli jako takové.

K tomu, abychom sdělili informaci, používáme jazyk. Jazyk musíme umět přizpůsobit tomu, aby informoval, sdělil jistou skutečnost – myšlenku sdělovanému subjektu. Jazyk má nutně několik úrovní, zdaleka ne všechny jsme schopni reflektovat. Logicky popsaným jazykem a vnitřně uceleným systémem jsme schopni vysvětlit pouhý fragment skutečnosti.

Programovací jazyk je prostředek pro zápis algoritmů, jež mohou být provedeny na počítači. Zápis algoritmu ve zvoleném programovacím jazyce se nazývá program.

4.2 Jednoduchý programovací jazyk

Snaha o zpřístupnění programování širší veřejnosti dala již na konci 20. století vzniknout rodině jazyků, které jsou patřičně zjednodušeny tak, aby je mohli obsluhovat i laici.

Zjednodušit programování je odpověď na situaci, kdy programovací jazyky vytvářeli především lidé se zvláštním nadáním pro ryze technické uvažování. Pro technické myšlení nemá každý člověk správné predispozice. Programování v současnosti znamená zejména určitý stupeň svobody při komunikaci se strojem.

K míře svobody, která má své silné kritiky[1], se nyní nechci vyjadřovat, ale zjednodušeně z pohledu pouhého uživatele, který používá daný nástroj, schopnost programovat činí z uživatele již potenciálního tvůrce vlastních nástrojů.

Hovořím-li o stroji, mám na mysli spotřební počítač. Termín **stroj** používám záměrně pro zdůraznění jisté formy strojového přemýšlení v historickém kontextu.

Nástroje jsou pak programy zkonstruované pro jistou činnost. Nástroj je obvykle vyvíjen za jedním účelem, který plní uživatelsky co nejpřívětivější cestou. Tato cesta je pro uživatele schůdná a nabízí mu standardní škálu dovedností nástroje.

Aniž bychom si to často uvědomovali, současná vizuální kultura je ovlivněna těmito nástroji daleko více, než je na první pohled zřejmé. Technické možnosti jsou současným tržním hladem po inovaci patentovým systémem vlastněny a proměňovány ve zboží. V této situaci je důležité znát nástroje i jejich vznik pro reflexi nebo kritiku v širších souvislostech.

Tato kniha je spíše nežli jednomu nástroji věnována programu pro tvorbu takových nástrojů. Jak již vyplývá z této definice, použití *Processingu* není limitováno jen úhlem pohledu autora tohoto textu. Návod by se měl stát spíše pobídkou k co nejrozmanitější tvorbě vlastních nástrojů sloužících opět k co možná nejširší škále možných účelů.

Processing vychází z koncepce snadného přístupu k programování. Za běžných okolností by se vnímavý člověk měl být schopen naučit jednoduché struktuře programu a vytvořit vlastní program v průběhu několika

[1] Včetně mne samotného.

dní. Na druhou stranu, právě predispozice našeho uvažování jsou natolik rozmanité, že takovou prognózu nelze brát jinak než jen jako orientační.

4.3 Dokonalost jazyka

Co je to dokonalý jazyk? Obecně je zapotřebí říci, že absolutně dokonalý jazyk neexistuje. Jazyk si můžeme představit jako systém vzájemných vztahů, který je schopen popsat jednotlivé symboly nebo objekty. Symboly můžeme nazvat předměty, tyto předměty dále mají své vlastní hodnoty a vlastnosti. Jazyk kromě definic takových vlastností operuje a popisuje jednotlivé jevy a vztahy mezi těmito předměty. Jednodušší popis jazyka je v pojetí výpočetní techniky určitý ucelený systém schopný popsat rozmanité problémy řešitelné strojem.

Co předem činí jakýkoli jazyk absolutně nedokonalým, je fakt, že jakýmkoli jazykem nedokážeme vyjádřit původ jazyka, tj. jeho strůjce, člověka. Jazyk použitý pro instruktáž stroje je vnitřně konzistentní a funguje logicky hermeticky, tedy nepřipouští jiný než jeden výklad konkrétního textu. Pojetí dokonalosti jazyka ve smyslu vnitřní logické konzistence je naprostou nezbytností v pojetí interpretace strojem, na druhou stranu téměř nepřekonatelnou překážkou v případě abstraktnějších úvah o programování jako takovém.

Použiji zde pro názornost rozdíl programovacího jazyka a jazyka českého. Český jazyk je jazykem organicky ustáleným po staletích užívání. Jazyk, jak ho známe, slouží ke komunikaci mezi lidmi, lze jej tedy použít například pro popis krajiny. Přestože k dokonalému popisu krajiny můžeme stěží dojít, slovy, která se opírají o určitou sdílenou zkušenost, lze poměrně dobře přiblížit určitý obraz věcí.

Kdybychom se pokusili pro popis krajiny použít jazyk programovací, dostaneme se velmi rychle do nesnází. Programovací jazyk není jazykem určeným pro předávání informací mezi lidmi, ale pro komunikaci člověka se strojem. Jeho vnitřní logická konzistence, tvrdá logická struktura, která nedovoluje v jeden okamžik jinou než jednu interpretaci, je jeho velikou předností při definici exaktních parametrů. Podobnost s řečí spočívá ve vazbě slov, která reprezentují jednotlivé hodnoty a operace. Hlavní odliš-

nost je v jeho syntetickém původu, jedná se o jazyk umělý. Programovací jazyk je přednostně zkonstruován pro definici známého a pochopeného. V případě neznámých nebo nepoznaných veličin je programovací jazyk víceméně k ničemu.

Chceme-li komunikovat se strojem, musíme tedy svůj způsob vyjadřování přizpůsobit logicky dokonalému jazyku – vnitřní logice fungování stroje. Počítač není navržen k tomu, aby něčemu rozuměl. Počítač je navržen k řešení jasně definovaných otázek. Tato příručka se pokusí srozumitelnou formou popsat jeden z možných způsobů, jak si takový jazyk osvojit a potažmo způsob uvažování, který vede k jasné definici problému. Hovoříme-li o programování, máme na mysli proces tvorby jisté logické struktury. Osvojení si programování spočívá ve schopnosti definovat problém nebo jasně formulovat otázku tak, aby na ni stroj mohl odpovědět.

Vtip spočívá v tom, že ovládneme-li formálně jazyk určený stroji, můžeme prostřednictvím tohoto stroje hovořit i ke člověku, tj. popisovat i pocity z rozkvetlých luk.

4.4 Volba vhodného jazyka

Ve výpočetní technice se nachází celá škála programovacích jazyků i prostředí. Tyto jazyky mají svoji genezi a byly historicky vyvíjeny především počítačovými odborníky. Jejich dokonalost lze těžko ocenit z vnějšího pohledu, a to právě z důvodu jejich konstrukce, která odpovídá a částečně podléhá určitým účelům, ke kterým byly tyto jazyky původně navrženy. Celistvý pohled na vývoj programovacích jazyků zde není možné obsáhnout. Programovací jazyk dle historického vývoje můžeme rozdělit na dvě skupiny, jazyky imperativní a objektově orientované. Toto základní rozdělení programovacích jazyků popisuje dva rozdílné přístupy v popisu událostí.

Processing se svou stavbou na základech *Javy* řadí k objektově orientovaným jazykům. Programovací jazyk *Java* je relativně mladým jazykem. Historie tohoto jazyka sahá přibližně do roku 1995. Mezi jeho hlavní výhody, ze kterých i *Processing* velmi těží, jsou zejména důraz na multiplatformnost a relativní jednoduchost syntaxe. Příbuznými jazyka *Java*

jsou další objektově orientované programovací jazyky jako *C++*, *Perl*, *Object Pascal*, *Visual Basic*, *C#*, *PHP*, *Python*, *Ruby*, *Smalltalk*, nebo *Lisp*.

Rozdíl mezi imperativním pojetím programování a objektově orientovaným pojetím spočívá především v logice kódu a jeho následném uspořádání. Imperativní jazyk se dá považovat za předchůdce objektově orientovaného pojetí. Nedá se obecně říci, který přístup je výhodnější, jedná se o dvě rozdílná paradigmata. Dnes mezi nejpoužívanějšími programovacími jazyky masivně převládá objektově orientované paradigma.

Rozdílné pohledy lze ilustrovat na popisu nějakého jevu. Jev se dá popsat různými způsoby. Jedna perspektiva bude více hovořit o aktérech jevu, ty rozdělí do objektů, jejich vlastností a možností. Druhá perspektiva popíše celistvou situaci jako sérii událostí, které je možné v tomto pořadí kdykoli zopakovat.

V posledních přibližně dvaceti letech se mezi programovacími jazyky postupně objevuje tendence po větší srozumitelnosti a potažmo zjednodušení programování jako takového. Programování v této koncepci již není jazykem odborníků, ale je demokraticky přístupné širší veřejnosti z rozmanitých – prioritně netechnických oborů.

Tato tendence postupně dala vzniknout celé rodině programovacích jazyků, které se snaží přiblížit potenciál výpočetní techniky netechnickým oborům, v neposlední řadě i oborům výtvarným. V technických kruzích je ovšem již sama disciplína psaní programů považována za tvůrčí činnost. V pojetí výtvarného umění dnes převažuje pohled na programování jako na velmi technickou zdatnost, rigidní a notně limitovanou.

Jazyk, který je nutně limitován svou nezbytnou formální dokonalostí, ovšem nemusí nutně limitovat svého uživatele ve sdělení. Uživatel se musí k uskutečnění takové zdárné komunikace přizpůsobit stroji.

4.5 Proč zrovna Processing?

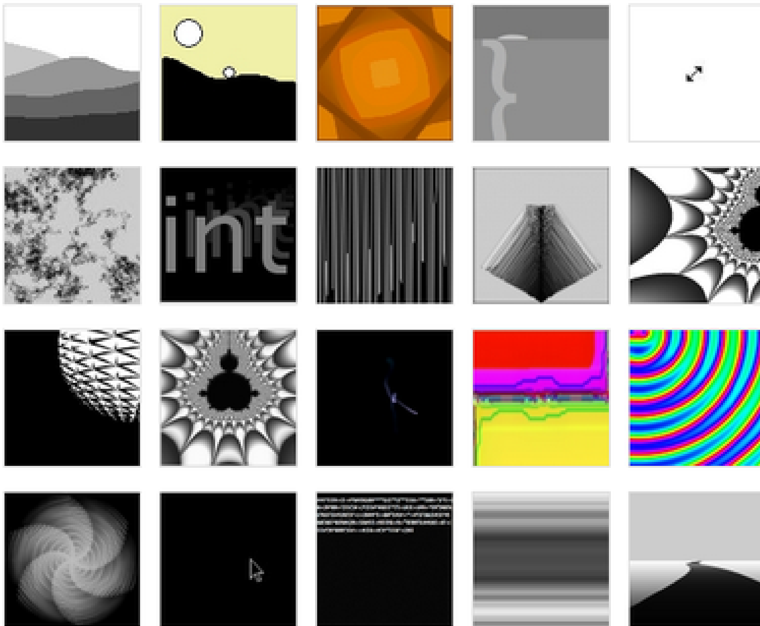
Processing je vhodný nástroj pro první experimenty s programováním z několika důvodů.

Prvním důvodem je jednoduchost pro uživatele, *Processing* se snaží začátečníky co nejméně zatížit zbytečnými informacemi. *Processing* prezentuje programování jako přístupnou a zvladatelnou schopnost. Dosahuje

toho především rychlou odezvou vizuálního prostředí. Jednoduše řečeno, v momentu, kdy spustíme kód, vidíme okamžitě jeho výsledek. Taková ilustrativnost je v začátcích podle mého názoru téměř nezbytná.

V *Processingu* lze velmi krátkým kódem programovat sofistikované programy. Pro příklad zde uvedu soutěž nazvanou *Tiny Sketch* pořádanou portály *rhizome.org* a *openprocessing.org*. *Processing* dokáže být natolik úsporný, že pomocí maximálně dvou set znaků byli tvůrci schopni naprogramovat svébytné programy.

Máte-li počítač u sebe a jste-li připojeni k internetu, vřele doporučuji si nyní projít aktuální podobu stránky openprocessing.org.



Snad největší předností pro začínající programátory je výtečná dokumentace a velmi přívětivá komunita lidí poskytující neúnavně rady začátečníkům.

4.6 Tvorba softwaru

Programy jsme v současnosti zvyklí spíše využívat než sami vytvářet. Tvoření programu je náročný proces a tvorba uživatelsky přátelského prostředí je velmi složitá.

Processing se svým způsobem neliší od žádného jiného programu, který běžně využíváme. Jde o sadu příkazů a programovací prostředí, které nám dovoluje určitou formou vytvářet svébytný program. I když se již jedná o programování, nelze jej běžně zaměňovat s klasickou tvorbou vyspělého nástroje.

Zde si musíme uvědomit, že náš potenciální výsledek – program bude vždy spíše banální v porovnání například se samotným prostředím *Processingu*. Processingový kód je výčet všech možností, které můžeme při tvorbě našeho programu využít. Obecně se dá říci, že *Processing* je nástroj pro tvorbu speciálních nástrojů. Výsledek našeho programování bude vždy pravděpodobně obsahovat poměrně větší část *Processingu* samotného.

Je dobré od začátku pochopit, k čemu lze *Processing* využít a k čemu jej opravdu využívat chceme. *Processing* se především hodí k rychlé tvorbě programu, ověření teze nebo spontánního nápadu. Pro podobné programování se také vžil pojem „rapid engineering“. Tvorba dospělejších nástrojů není sice nemožná, obecně ale platí, že *Processing* bude svými zkrácenými zápisy a jednoduchostí prostředí velmi nápomocný v začátcích.

Processing se svou konstrukcí zejména hodí pro práci s obrazem, a tedy obrazu bude také věnována nejrozsáhlejší část této knihy. Výstupy z *Processingu* lze nadále zpracovávat, *Processing* tak může sloužit například jako mezičlánek ve výrobním procesu.

Dnes je *Processing* využíván hlavně grafickými designéry, designéry užitého umění, tvůrci webových aplikací, softwarovými návrháři, vizuálními umělci a umělci, kteří se věnují instalacím v prostoru nebo živé projekci. K *Processingu* si ovšem nacházejí cestu i více technické obory, zejména architekti, badatelé v oboru přírodních věd, statistici a všeobecně tvůrci softwaru. Software nachází využití i v oborech zabývajících se humanitními vědami, jako například v sociologii nebo jazykovědě.

Processing se neomezuje na jeden obor, svou koncepcí spíše nabízí společnou platformu pro mezioborovou komunikaci.

4.6.1 Otevřenost softwaru

Processing je jedním z jazyků, které byly vytvořeny v diskurzu zjednodušování programování. Jako každý jiný programovací jazyk je i *Processing* navržen pro jisté účely. V případě tohoto programovacího jazyka se nejvíce jedná o důraz na rychlý vývoj a zjednodušené nakládání s obrazem i prostorem. Z více technického pohledu pak *Processing* vyniká otevřeností zdrojového kódu a důrazem na multiplatformovost.

Z pohledu vývojáře je velmi důležité, že jazyk i programovací prostředí *Processing* jsou v současnosti **otevřený software**, což znamená, že prostředí včetně samotného zdrojového kódu jsou volně k dispozici. *Processing* je dále šiřitelný pod **MIT licencí**. Pro vývojáře otevřenost zdrojového kódu znamená zásadní věc pro dosažitelnost celého zdroje, který se dá následně kupříkladu implementovat do různých prostředí. Další možnost vývojáře je rozšířit jazyk o vlastní funkce, a tím participovat na projektu, například tvorbou takzvaných knihoven *(viz Knihovny, str. 127)*. Otevřenost kódu teoreticky navyšuje počet možných participantů a de facto celý projekt udržuje v dlouhodobém horizontu naživu.

Z hlediska uživatele je velmi příjemné, že software je k dostání zdarma na stránkách projektu. Za jeho užívání není nutné platit žádné poplatky, a to ani v případě komerčních užití. V případě potřeby vyjádření vděku za práci autorů je možné zmínit kdekoli ve vašem produktu **Built with Processing**.

V začátcích nemusíte věnovat velikou pozornost této zvyklosti, dostane-li se vám po čase dobré pomoci z komunity, nebo budete-li jednoduše Processing využívat ke své práci, můžete jej někde zmínit, a tím vlastně projekt podpořit.

Programovací jazyk *Processing* není na první pohled zvláštní. V podstatě by se dalo říci, že se jedná pouze o rozsáhlou knihovnu původního jazyku *Java*.

To, co *Processing* řadí mezi oblíbené softwary pro tvorbu, je přívětivá komunita uživatelů s velmi odlišnými stupni znalostí a úhly pohledu. Zvláštní důraz je v komunitě kladen na poskytnutí co největší podpory právě začínajícím uživatelům. Tomu odpovídá i počet rozmanitých průvodců a rozsáhlá, velmi dobře a stručně napsaná dokumentace ke každému z příkazů v *Processingu*.

K otevřenosti kódu v neposlední řadě přistupuje i celá řada velmi zkušených tvůrců. Tím se uživatel na jakémkoli stupni znalostí může kdykoli naučit nové postupy nebo může svobodně recyklovat algoritmy ostatních

uživatelů. Čím dál více uživatelů *Processingu* ctí filozofii otevřeného software, která (mimo jiné) hlásá: „Vědomosti nesmí být privatizovány!“

4.6.2 Syntetický obraz

Hlavní doménou *Processingu* je schopnost vytvářet „živé“ programy, tj. programy běžící v reálném čase. Již název programovacího jazyka *Processing* napovídá akcent v čase se odvíjejících událostí.

Obraz vytvořený tímto způsobem, na první pohled zaměnitelný s videem, nemusí například podléhat časové omezenosti nebo může určitým způsobem reagovat na své okolí. Obecně jev běžícího programu v čase můžeme pojmenovat *Generovaný obraz* (nebo *zvuk*). Rozdělení je zde trochu problematické, v případě digitálního obrazu se již dnes nejedná o nic jiného než o generovaný obraz. Sám proto raději používám výraz *Syntetický obraz*. Syntetický obraz je takový obraz, který byl vytvořen uměle, tedy skladbou logických prvků, které obraz nakonec vytváří. Pro ilustraci zde lze například použít rozdíl mezi syntetickým zvukem a zvukem pořízeným nahrávkou.

4.6.3 Empirický přístup k programování

I když bychom zásadní odlišnost od jiných programovacích jazyků hledali stěží, *Processing* proslul zejména snadností použití. Kompilovat program není otázkou nastavování kompileru a veškerých jeho parametrů, program jednoduše po stisku tlačítka *RUN* běží (je-li správně napsán). Samozřejmě tento redukcionistický přístup má své nevýhody, speciálně při rozsáhlejších projektech tato jednoduchost může dokonce omezovat. *Processing* ovšem lze jako svobodný software naimplementovat do řady jiných prostředí, a potřebujete-li si kompilační proces nastavit sami, nic vám nebrání použít *Processing* jen jako knihovnu do *Javy*.

Tato jednoduchost na druhou stranu nezdržuje uživatele od myšlenkového toku psaní programu. Častou kontrolou výsledku kódu může uživatel lépe sledovat postupné změny v programu.

Nazývám tento způsob programování empirický, je to přístup, kdy je podle zkušenosti s běžícím programem dotvářen i samotný zdrojový kód.

Mnoho technicky zaměřených lidí by zřejmě mohlo tento postup kritizovat pro přílišnou reduktivnost a amatérský přístup. Zde bych oponoval faktem, že kreátora vytvářejícího například instalaci do galerie příliš nezajímá formální dokonalost programu. Program je v pojetí tvůrců jen prostředníkem pro další sdělení, a jestliže toto sdělení předá, je to dobrý program.

Proces poznávání struktur jazyka při tvorbě obrazu prostřednictvím *Processingu* bych přirovnal k vývoji od začátečnické malby na tkaninu k postupnému tkaní gobelínu. Zde je nutno podotknout, že ne každý autor využívající *Processing* chce tkát gobelín, a najde-li nástroj vhodný „jen" pro „malbu" na plátno, je to dobrý nástroj.

Zamyslete se, k čemu byste Processing rádi využívali, účel světí prostředky. V Processingu tomu bývá, bohužel, často právě naopak.

Kapitola 5 Processing jako prostředí

Processing představuje ucelené programovací prostředí, tzv. PDE[1]. Jedná se o kompletní prostředí určené především k rychlému vývoji aplikací. Samotný program je jednak otevřeným softwarem[2], jednak je zdarma ke stažení pro všechny majoritní platformy. *Processing* je k dispozici pro **GNU / Linux**, Mac OS i pro Microsoft Windows na stránkách projektu *processing.org*. *Processing* je teoreticky možné spustit v jakémkoli prostředí umožňujícím chod virtuálního stroje *Javy*. Celý jazyk i samotné PDE vychází pod licencí **GNU / GPL**, jedná se tedy o svobodný software *(viz Otevřenost softwaru, str. 33)*.

[1] Processing Development Enviroment.
[2] Tedy s otevřeným veřejně dostupným zdrojovým kódem.

5.1 Základní prostředí

Základní rozhraní tvoří textový editor s několika nezbytnými funkcemi. Patrně nejdůležitější je v liště tlačítko *(1)* RUN, které kompiluje a spouští program aktuálně rozpracovaný v textovém editoru. *(2)* Tlačítko STOP naopak program zastaví. V případě chyby v programu je lze též využít jako vynucené zavření programu. Zbylá tlačítka slouží k diskovým operacím.

Zkuste nyní spustit Processing, bude dobré ho mít při ruce. Nemáte-li jej po ruce, doporučuji přeskočit následující kapitoly věnované prostředí.

5.1.1 Základní diskové operace

Jsou to operace, jež něco zapisují či načítají z disku. Tyto operace zahrnují veškerou manipulaci se sketchí, její vytvoření, uložení, načtení či export do webového formátu či kompilaci do spustitelné aplikace.

Processingová sketch nemusí být nutně uložena, abyste ji mohli spustit. Nové úpravy, které v textovém poli napíšete, budou dočasně uloženy v závislosti na vašem operačním systému jinde. Tímto způsobem můžete experimentovat s kódem, aniž byste přišli o předchozí verzi.

Mezi základní operace patří vytvoření nové sketche *(3)*, otevření předešlé *(4)*, uložení *(5)* a export programu do webového appletu *(6)*.

Tlačítka mají dále speciální vlastnosti s podržením tlačítka: tlačítko s novou sketchí otevře také nové okno editoru. V kombinaci s načtením předchozí sketche *(4)* otevřete též sketch v novém editoru, aniž byste ztratili předchozí okno. U uložení se modifikace projevuje funkcí *uložit jako*. Při exportu přepínáte stisknutím klávesy SHIFT mezi exportem samostatné aplikace nebo appletu[3], aplikace schopné běžet v prohlížeči nebo obecně na webových stránkách.

5.1.2 Sketch

Sketch je v *Processingu* označení pro jednotlivé projekty. Sketch je ve své podstatě složka obsahující alespoň jeden stejně nazvaný soubor s příponou **.pde*. Sketch při vytvoření nevyžaduje název, je jí přidělen pouze aktuální datový kód zaznamenávající datum vytvoření. Tímto decentním způsobem *Processing* toleruje například nejasnost záměru autora při vytváření nového díla, jeho název či pracovní název lze tímto způsobem přiřadit až později, po nabytí jasnějších obrysů.

Program si vystačí pouze se samotným zdrojovým kódem, tj. souborem (soubory) **.pde*. Ovšem operujeme-li s externími daty stojícími mimo zdrojový kód, můžeme využít dvě možnosti. Za prvé soubor, se kterým potřebujeme operovat, tažením myši přesunout na textové pole *Processing PDE*. Touto operací bude soubor zapsán do naší sketche automaticky. Nebo lze operaci provést manuálně, stačí vytvořit v adresáři sketche adresář

[3] Pouze ve verzi 1.x, *Processing 2.0* již export do appletu nepodporuje.

s názvem **DATA**. *Processing* tuto složku automaticky rozpozná a soubory v této složce budou dobře dostupné pro pozdější operace.

Typy standardních adresářů, které se často objevují ve struktuře sketche, si stačí zapamatovat jen dva:

1. adresář nazvaný *DATA* – Tento adresář je využíván při práci s jakýmikoli vnějšími daty, jako jsou například obrázky, zvuky, videa, vektorové grafiky nebo textové soubory.

2. adresář nazvaný *CODE*, obsahující externí zdrojový kód, případně kód kompilovaný v podobě knihovny mající nejčastěji přípony **.java, *.class, *.jre.* – Tento adresář se nalézá v systémové cestě daného projektu, a umístíme-li zde soubory, budou též dobře dostupné z daného projektu. (Adresář *CODE* nás zatím nemusí příliš zajímat, později se jeho prostřednictvím můžeme pokusit rozšiřovat funkce *Processingu* externím kódem.)

5.1.3 Sketchbook

Místo na disku, které *Processing* využívá k uskladnění sketchí, se nazývá povšechně *sketchbook* a je v podstatě pouze adresářem obsahujícím jednotlivé projekty. Koncepce *sketchbooku* spočívá v uspořádání jednotlivých projektů do organizované formy, a ačkoli nevnáší do jednotlivých projektů sám o sobě řád, může napomoci k vytvoření řádu vlastního. Ze své zkušenosti mohu říci, že organizace jednotlivých projektů do jakékoli struktury má opravdu smysl. Mnou preferovaná struktura obsahuje jednotlivé roky následně ještě dělené do měsíců, ale možnosti organizace záleží čistě na uživatelských preferencích. Tímto chci spíše ilustrovat možnost uspořádání nežli nezbytnost, ale přítomnost vašeho systému od počátku vřele doporučuji.

Ve *sketchbooku* se dále nachází jeden speciální adresář nazvaný *libraries*. Adresář v sobě uchovává externí rozšíření *Processingu* o komunitní knihovny. Standardně *Processing* již své základní knihovny nese v sobě, tj. jsou běžně přidány do samotného *Processingu*. V tomto adresáři se nacházejí knihovny, které budete moci použít později. Na stránkách projektu *http://processing.org* můžete nalézt velké množství komunitních knihoven

s dobrou dokumentací volně ke stažení. Veškeré tyto knihovny tedy *Processing* potřebuje nalézt v tomto adresáři.

5.1.4 Editor

Editor je textové pole a je vaším hlavním nástrojem komunikace s *Processingem*. Veškeré informace, které zadáte do tohoto pole, budou interpretovány samotným *Processingem*. Textový editor není nijak dokonalý, pro začátek si s ním ovšem vystačíme. Formát, který tento editor produkuje, je ve své podstatě textový soubor s příponou **.pde*.

Textový editor má několik funkcí, které vám mohou usnadnit práci. Odlišuje například mezi příkazy *Processingu* barvami – tato zdánlivá drobnost speciálně v začátcích napomůže, nebo v případě objemnějších kódů může výrazně zpřehlednit, organizaci programu. Jedná se o programátorskou konvenci obecně v anglickém jazyce nazývanou **syntax highlighting**.

Další konvencí, se kterou se často setkáte, je tzv. **indenting**, zarovnávání kódu do úhledných paragrafů. V *Processingu* si ze začátku vystačíte se standardním zarovnáním, posléze lze zarovnávání aktivovat stiskem kombinace kláves CONTROL (APPLE) + T.

Horní lišta editoru označuje záložky, kliknutím na šipku na pravé straně lišty se vám zobrazí možnosti operací se záložkami. Pro začátek s nimi nebudeme pracovat. V případě delšího kódu záložky slouží k lepší organizaci struktury programu. V podstatě se dá říci, že každá záložka odpovídá jednomu souboru ve sketchi *(viz Sketch, str. 39)*.

Jak můžete vidět na obrázku, editor operuje i s českými znaky, ačkoli bych je z důvodů internacionalizace neradil používat. V komentářích, tj. textem uvozeným dvěma dopřednými lomítky „//“, což je text, který kompiler ignoruje, jsou diakritická znaménka přípustná.

Naopak v případě názvu proměnných nebo jakýchkoli funkčních definic je použití diakritiky nepřípustné.[4]

5.2 Klávesové zkratky

CTRL + r	spustí program
CTRL + SHIFT + r	takzvaný present mode spustí program a zatemní zbytek obrazovky
CTRL + t	automatické formátování textu, zarovnává text do odrážek
CTRL + k	otevře adresář konkrétního projektu
CTRL + f	vyhledávání v textu a nástroj pro hromadné nahrazování
CTRL + SHIFT + f	vyhledávání u označeného příkazu v referencích
CTRL + s	uloží projekt na disk
CTRL + SHIFT + s	uložit jako
CTRL + e	export jako webový applet[5], otevře exportovaný projekt
CTRL + SHIFT + e	export jako aplikace, zobrazí okno s výběrem platforem pro export

* CTRL značí klávesu CTRL na PC a APPLE na Mac

5.3 Soustředěná činnost

Než začneme doopravdy programovat, musíme si uvědomit, že tento proces vyžaduje velikou dávku koncentrace. Obecně se nedá říci, jaké fyzické prostředí je k programování ideální. To se samozřejmě může lišit. Ale platí, že fyzický prostor, který k programování potřebujete, je prostor, ve kterém se můžete sami soustředit na práci.

---

[4] Standardní jazykové kódování, ve kterém *Processing* operuje, je *UTF-8*.
[5] Pouze v *Processingu 1.0*.

Koncentrace při psaní programu je specifická dovednost, kterou se začátečník učí jen stěží. Bohužel pouhým čtením této knihy se již vyvádíte z potenciální koncentrace. Učení se programovat je náročný proces a vyžaduje velikou dávku koncentrace sám o sobě. Okamžité zkoušení nových poznatků pomáhá zápisu informací do vaší dlouhodobé paměti a opakování je samozřejmě nezbytné k utužení těchto struktur. Co je, bohužel, při počátečním programování nezbytné, je například hledání v referencích a spolu se soustředěním se na program samotný je celý proces učení se velmi náročný na koncentraci.

V problematice programování obecně platí, že není nic složitějšího než číst cizí kód, nebo dokonce kód vlastní s odstupem času. Při programování se lidská mysl dostává do stavu mimořádné bdělosti a koncentrace. Tento stav je často popisován programátory jako tzv. **flow**. Jedná se o stav, ve kterém mysl vědomě udržuje celý program[6] v paměti.

Pokaždé, když si programátor takový stav přivodí, vše v kódu[7] se pro něj jeví srozumitelné. Formátování a standardy velmi napomáhají k rychlému navození bdělého stavu, protože samy o sobě jsou jistou formou jazyka.

5.4 Základní pravidla a zvyklosti

V programování obecně platí jisté zákonitosti. Není tomu jinak ani u pravidel, která nejsou důležitá pro čtení kódu strojem, ale spíše pro člověka. Speciální zákonitosti formátování kódu mají ryze praktickou funkci. Jedná se o standard dodržovaný programátory, aby byli schopni sdílet své úsilí.

Otevřete některý z příkladů z lišty File → Examples, *a letmo se seznamte s kódem. I když mu nebudete rozumět, podívejte se, jaká znaménka a barvy se v kódu opakují.*

[6] Nebo větší části programu.
[7] Podle míry ovládnutí programovacího jazyka.

Kapitola 6 Stavba programu a syntax

6.1 Logika programování

Pro začátek je dobré si představit program jako sadu instrukcí. Každá instrukce má svůj význam a své místo. Instrukce se píšou v programovacím jazyku a jejich interpretace je vždy pro stroj jednoznačná. Stroje podle svého návrhu neudělají nic, kromě toho, co mají instruováno.

Veškeré programy, které používáte, dokonce i programy, o nichž nevíte, že používáte, byly někdy naprogramovány lidmi pomocí programovacích jazyků. Programovací jazyk je pouze plánovací forma zápisu programu. K tomu, aby byl program spuštěn, musí být v případě jazyka *Processing* převeden do strojového kódu. Strojový kód se liší od toho zdrojového, našeho čitelného plánovacího jazyka tím, že není čitelný pro člověka, je čitelný pro stroj.

Počítač je pouhý stroj. Umí opravdu rychle vykonávat sled banálních operací. Programovací jazyk slouží k vytvoření smyslu v těchto operacích. Proces převodu ze zdrojového kódu do strojového se nazývá kompilace.

O kompilaci toho nemusíme naštěstí mnoho vědět, vše bylo již tvůrci *Processingu* a jazyku *Java* shrnuto pod tlačítko RUN.

Budete-li mít otevřený nějaký kód z Examples, zkuste ho nyní spustit, uvidíte, jak se program chová. Vyzkoušejte několik příkladů.

6.1.1 Komentář

Komentář je vše, co program ignoruje. Je to místo v programu určené ke čtení člověkem. Komentáře dále často slouží k rychlé editaci programu.

Odkomentováním řádky funkčního kódu lze vynechat některé funkce *Processingu*.

Následuje zápis tak, jak jej již můžete vepsat do editoru *Processingu*. Vřele doporučuji mít v této chvíli *Processing* čtení knihy zapnutý a po ruce, abyste si příklady v rámečcích mohli vyzkoušet. Komentář je řádek textu nebo jeho část uvozená dvěma dopřednými lomítky.

```
// komentar je uvozen dvema dopřednymi lomitky
```

Spustíte-li nyní program tlačítkem RUN, objeví se šedé okénko prvního processingového programu. Jelikož jste zatím nedefinovali nic funkčního, program také nic nedělá.

V praxi je pak často využíván i druhý způsob komentáře, takzvaný víceřádkový komentář. Ten začíná dopředným lomítkem a hvězdičkou a končí těmito znaky v obráceném pořadí.

```
/* viceradkovy komentar je uvozen
doprednym lomitkem a hvezdickou
ukoncen hvezdickou a lomitkem
(v obracenem poradi)

toto je viceradkovy KOMENTAR
toto je viceradkovy KOMENTAR
toto je viceradkovy KOMENTAR
toto je viceradkovy KOMENTAR
*/

/*zacatek .............. konec*/
```

Komentáře v *Processingu* poznáte vždy nejrychleji tak, že je bude editor obarvovat šedou barvou.

Vyzkoušejte si napsat nějaký text do textového editoru, spusťte program, následně před něj vložte dvě dopředná lomítka, pozorujte rozdíl.

6.1.2 Základní datatypy

K stavbě programu potřebujeme stavební materiál. Pro základní pochopení fungování programu je nezbytné nejdříve pochopit základní datatypy. Datatyp si lze obecně představit jako obálku na informaci.

Processing rozlišuje mezi jednotlivými datatypy. Informace, které se nacházejí v paměti, se musejí nacházet pod správným datatypem, aby program věděl, jak s nimi operovat.

K datatypu je možné si představit různé paralely, má oblíbená je podobnost s obálkami nebo nádobami. Různé datatypy si můžeme představit jako tvary nádob. Do rozličných nádob lišících se tvarem se směstná rozmanitý obsah. Typy obsahů se dají názorně představit na rozdílu mezi textovou informací a informací číselnou.

Processing potřebuje nejdříve vědět, zda nádoba obsahuje text, nebo číslo, aby s ní mohl operovat. Například nelze provést matematickou operaci mezi dvěma slovy. Sčítat, odčítat, dělit či násobit se dají pouze čísla.

6.1.3 Seznam základních datatypů

Datatypů je jen několik, dají se vcelku snadno zapamatovat:

```
int mojeCeleCislo = 1;

float mojeCisloSDesetinnouCarkou = 1.33;

boolean mojePravdaCiNepravda = true;

String mujSlovniRetezec = "Ahoj svete!";

char mujJednotlivyZnak = 'A';
```

Takovému zápisu již bude *Processing* rozumět. Vepsáním těchto řádků do processingového editoru již dochází k alokaci vašich informací ve správných datatypech v paměti. Vysvětleme si nyní několik nejasností.

První slovo na každém řádku označuje odlišný datatyp (náš tvar nádoby), následuje jméno proměnné (název pro naši nádobu), poté může následovat rovnítko, které již přiřazuje obsah – hodnotu do proměnné (nádoba je naplněna).

Každý příkaz je ukončen středníkem „;“.

Podivné názvy takzvaných proměnných, jako například *mojeCeleCislo*, jen ukazují, že název pro svoji proměnnou můžete zvolit téměř podle libosti. Proměnná ale nesmí mít v názvu mezeru, tedy prázdný znak. Podle zvyklostí by dále proměnná měla začínat malým písmenem, a potřebujete-li nutně použít víceslovný název, napište velké písmeno místo mezery.

Proměnná nesmí být číslo, ale slovní název. Programátorské konvence při pojmenovávání proměnných jsou ryze praktické. Proměnná by měla

mít co možná nejkratší a nejvýstižnější název. Toto je spíše dobré doporučení nežli pravidlo. Správným pojmenováním proměnných docílíte větší přehlednosti v kódu, kratší délkou si ušetříte zbytečné psaní.

Bedlivě si prostudujte jednotlivé datatypy, přepište je do Processingu a zkuste pojmenovat po svém. Podle toho, co mohou reprezentovat, přiřaďte patřičné hodnoty.

6.1.4 Barva

Zvláštním datatypem v *Processingu* je datatyp určený pro barvy, *color()*. Barvy lze definovat následujícími způsoby.

```
// skala sede, 0 = cerna, 255 = bila
color mojeBarvaSeda = color(127);

// skala sede s pruhlednosti
color mojeBarvaSeda2 = color(255,127);

// cervena, modra, zelena
color mojeBarvaOranzova = color(255,127,0);

// cervena, modra, zelena, pruhlednost
color mojeBarvaOranzova2 = color(255,127,0,127);

// barva vyjadrena stylem HTML, v hexadecimalnim tvaru
color mojeBarvaOranzova3 = color(#FFCC00);
```

Pro výběr oblíbené barvy lze využít nástroj z lišty *Tools* → *Color Selector*.

Základní barevné nastavení je 24bitové RGB, to znamená, že každá barevná hloubka jednoho kanálu je 8 bitů, což značí 256 možných stupňů. Nejvyšší stupeň je ovšem 255, jelikož ten nejnižší není 1, nýbrž 0.

Kolorimetrické prostory a jejich rozmezí se dají volně definovat pomocí příkazu *colorMode()*.

Otevřete si lištu Tools → Color Selector, zkuste vybrat několik barev a přepsat je do správného tvaru.

6.1.5 Tisk do konzole

Tento jednoduchý program zatím jen definoval několik proměnných do správných datových typů a nedělá nic zajímavého. Celá alokace probíhá uvnitř programu. *Processing* provádí většinu svých operací skrytě. Program kreslí nebo jinak interaguje s uživatelem, jen je-li o to požádán.

Nejjednodušší výstup z programu je tisk do tzv. konzole. Konzoli jsme si již krátce uvedli v kapitole *Základní prostředí*. V *Processingu* se konzole

nachází pod textovým editorem. Toto černé pole má pouze textový výstup a slouží k doladění programu.

Tiskem do konzole si nyní můžeme například zkontrolovat obsah našich proměnných. To můžeme provést následujícími dvěma způsoby:

```
print(mojeCeleCislo);

println(mujSlovniRetezec);
```

Oba příkazy tisknou obsah našich proměnných. Jediný rozdíl mezi příkazy je ten, že druhý příkaz končí znakem:

```
"\n"
```

Jedná se o tzv. *newline character*, přidáním speciálního charakteru příkaz tiskne pokaždé na nový řádek. Není nutné si pamatovat tento speciální znak, postačí, když si zapamatujeme příkaz

```
println( cokoli );
```

Spustíme-li program nyní, v konzoli se nám ukáže výstup z našeho programu:

```
1Ahoj svete!
```

Zde je názorně vidět, že tisk pomocí pouhého příkazu *print()* nepoložil další příkaz na nový řádek, a tedy vytiskl *1Ahoj* dohromady.

Tisk do konzole se hojně používá při ladění programu. Kdykoli se na něco potřebujete kódu zeptat, můžete tak učinit tímto prostým příkazem.

V konzoli se tisk projevuje bílou barvou. Další možný obsah konzole jsou chyby. Ty jsou označeny barvou červenou. *Processing* vám chybou naznačuje, že něco není v pořádku s vaším kódem. V případě takto jednoduchého programu se v drtivé většině případů bude jednat o překlep či zapomenutý znak.

Programovací jazyk, bohužel (naštěstí?), neodpouští překlepy. Tedy bude stačit jeden chybný znak v programu, aby celý program nebyl spustitelný.

Bohužel chybové hlášení se nedá označit za dokonalé a pro začátečníky bude velmi složité dobrat se pomocí chybových hlášení původu chyby. Na zlepšení chybových hlášek se pracuje již dlouhou dobu a některé základní chyby *Processing* v angličtině umí dobře popsat. Většinou ale *Processing* tiskne řetězec svých chyb, které byly nastartovány chybou vaší, a chybová

hlášení čítající několik desítek řádků vám toho nakonec o původní chybě moc nesdělí.

V případě chyby se vás *Processing* bude snažit přesměrovat na řádek, kde chyba pravděpodobně vznikla. V případě takto jednoduchých programů, jaké si zde ukazujeme, chybu identifikuje zcela bezchybně, bohužel tomu tak nebude ve všech případech.

Vyzkoušejte si vytisknout různé proměnné, ověřte, zdali tisknou hodnoty, které jste jim zadali.

6.1.6 Základní operace s datatypy

Náš program nyní neprovádí nic světoborného. Definuje několik proměnných a následně některé z nich tiskne do konzole. Operujeme zde stále s „abstraktními“ hodnotami, které se zapisují do paměti programu a pak je z paměti zpětně získáváme.

Nyní si ukážeme, co s již definovanými proměnnými můžeme dělat. Jednotlivé datatypy mají různé přípustné operace. Zkusme si letmo projít možnosti našich proměnných.

- *Integer* a *float* neboli čísla: *int* a *float*

  První proměnná *mojeCeleCislo*, které byla přiřazena hodnota *1*, je takzvaný *Integer*, tedy celé číslo. Tento typ může mít hodnotu[1] v rozsahu -2147483648 až 2147483647. V rozmezí těchto hodnot můžeme provádět různé matematické operace s čísly:

  ```
  int prvniCislo = 1;
  int druheCislo = 5;
  int tretiCislo;

  tretiCislo = prvniCislo + druheCislo;

  println(tretiCislo);
  ```

  Tímto jsme provedli základní matematickou operaci, sečetli jsme dvě čísla a následně jsme výsledek vytiskli do konzole.

[1] Na 32bitových strojích.

Další možné operace jsou:

```
// aritmeticke operace
a + b;
a - b;
a * b;
a / b;

// prirustky
a += b;
a -= b;

// prirustek, ubytek o 1
a ++;
a --;

// modulo (prebytek po deleni)
a % b;

// a porovnavani
// vetsi, mensi
a < b;
a > b;

//vetsi nebo rovna se, mensi nebo rovna se
a <= b;
a >= b;

// shoda, neshoda
a == b;
a != b;

// pozor, vysledkem porovnavani jiz neni cislo
// ale odpoved ANO, nebo NE, TRUE, nebo FALSE
// pravda, nebo nepravda, datatyp boolean
```

S celými čísly neboli *int* nebude problém se sčítáním, odčítáním a násobením. V případě dělení může nastat logicky problém, výsledek nemusí být celé číslo. Budeme-li chtít výsledek takové operace uchovat v paměti, tj. zapsat pod naši proměnnou, musíme použít datatyp operující s desetinnou čárkou nebo výsledek zaokrouhlit.

Například:

```
int a = 10;
int b = 3;
float c;

c = a / b;

println(c);
```

Vytiskne:

```
3
```

Pozor, to není správný výsledek! Kde se stala chyba? *Processing* operující s celými čísly speciálně při dělení předpokládá výsledek opět celé číslo. Tedy ono zaokrouhlení provádí již sám. Stačí si nyní pamatovat, že pro přesné dělení čísel bychom měli dělit vždy číslem s desetinnou čárkou, tj. s datatypem *float*.

Tedy správně by dělení mělo proběhnout takto:

```
float a = 10;
float b = 3;
float c;

c = a / b;

println(c);
```

Tisk do konzole již ukazuje „správnou" hodnotu:

```
3.333333
```

Pro další operace s čísly bych pro přesnost výsledků důrazně doporučil používat jen *float*. Další operace mohou vypadat například takto:

```
float a = 3;
float b;

// sq je funkce pro "square", cislo na druhou
b = sq(a);
println(b);

// sqrt je funkce pro "square root", odmocninu
b = sqrt(a);
```

```
println(b);

// pow je funkce pro "power", cislo na N-tou
// minusova cisla v N jsou odmocniny
b = pow(a,3);
println(b);

b = pow(a,-3);
println(b);

// atp.
```

- *char* a *String*, znak a řetězec znaků, text

  S textem nelze operovat stejně jako s čísly, je logické, že nemůžeme násobit texty mezi sebou. *String* je speciální datatyp pro uchovávání textů v paměti a nakládá se s ním speciálními funkcemi. Prozatím nám bude stačit, když si ukážeme velmi jednoduchou operaci s textem, spojení dvou řetězců dohromady.

  Pro označení hodnoty řetězce se používají dvojité uvozovky. S textem se pracuje následovně:

```
String prvniSlovo = "Ahoj";
String druheSlovo = "svete!";
String slovniSpojeni;

slovniSpojeni = prvniSlovo + " " + druheSlovo;

println("Dve spojena slova: " + slovniSpojeni);
```

  Všimněte si, prosím, mezery vložené mezi slova. Uvozená mezera je také řetězec textu. Výsledným tiskem do konzole tedy dostaneme následující řetězec textu:

```
Dve spojena slova: Ahoj svete!
```

  Zde jsme provedli jednu ze základních operací s textem, spojování řetězců. *String* lze také spojit s jednotlivými znaky nebo i s čísly, výsledkem bude ovšem vždy další *String*.

```
int a = 1;
int b = 2;
```

```
String slova = "test";

// "slova = slova + neco" lze take nahradit znamenkem
    "+="
// stejnym znamenkem, ktere u cisel znamena prirustek,
// tedy namisto:
// slova = slova + " " + a + " " + b;
// muzeme zkratit na:

slova += " " + a + " " + b;

println(slova);
```

Výsledkem bude:

```
test 1 2
```

Pospojujte vlastní větu z jednotlivých slov, vyzkoušejte do věty spojit i čísla.

Řetězce se dají dále porovnávat, seřazovat, lze v nich vyhledávat znak či slovo a tak podobně. Pro naše účely zatím postačí si uvědomit rozdílné operace mezi textem a číslem.

- *boolean* neboli pravda nebo nepravda

Boolean je nejjednodušším datatypem v *Processingu*, může vyjadřovat pouze dva stavy. Pravdu *(true)* nebo nepravdu *(false)*. Žádnou jinou hodnotu *boolean* nepřijímá, a proto je, co se týče paměti, velmi úsporným datatypem. Operace s *booleany* se nazývají logické operace a vypadají následovně:

```
boolean prvniTvrzeni = true;
boolean druheTvrzeni = false;
boolean tretiTvrzeni;

// "&&" znaci logickou operaci AND
// vysledek bude TRUE, pravda, JEN pokud
// prvniTvrzeni a druheTvrzeni bude pravda
tretiTvrzeni = prvniTvrzeni && druheTvrzeni;

println(tretiTvrzeni);

// "||" dve svisle cary je OR, tj "nebo"
// vysledek bude TRUE, pravda, pokud
// prvniTvrzeni nebo druheTvrzeni je pravda
tretiTvrzeni = prvniTvrzeni || druheTvrzeni;
```

```
println(tretiTvrzeni);

// posledni zakladni operaci je porovnani
// muzeme porovnat dva booleany pomoci "=="
// dvojiteho rovnitka
tretiTvrzeni = (prvniTvrzeni == druheTvrzeni);

println(tretiTvrzeni);

// pro negativni vysledek je zaporne porovnani "!="
// vykricnik pred booleanem vzdy znaci opak

tretiTvrzeni = (prvniTvrzeni != druheTvrzeni);

println(tretiTvrzeni);
```

Boolean je možné si představit jako vypínač světla. Zapnuto a vypnuto jsou jediné dva stavy podobného vypínače. *Booleany* často řídí tok programu, je možné si je představit jako přepínače mezi jednotlivými stavy programu.

6.1.7 Základní struktura programu

Processing rozlišuje mezi jednotlivými programovacími přístupy. Textový editor se automaticky přizpůsobí způsobu programování. Tímto přístupem se *Processing* snaží co nejvíce přiblížit začátečníkovi. Základní prostředí nám dovoluje pouze nakreslit tvary a ukončit program. K pohybu v čase musíme *Processingu* definovat základní strukturu programu.

Dvě nejzákladnější funkce, se kterými se budeme při programování setkávat, jsou *setup()* a *draw()*:

- *setup()* je funkcí, která bude spuštěna jednou, na začátku programu – zde se nejčastěji nastavují počáteční parametry, které si potřebujeme připravit, než se spustí kreslicí funkce
- *draw()* je funkce pro opakované kreslení, bude spouštěna stále po dobu chodu programu – tady budeme nejčastěji kreslit tvary a provádět proměnlivé výpočty pro animaci

Základní funkce pro vyvolání překreslování výstupního okna je funkce *draw()*. Funkce je uvozena slovem *void*. O funkcích se více dozvíme později *(viz Funkce, str. 83)*. Jedná se o základní smyčku, která zajišťuje jakoukoli proměnlivost, tedy animaci v obraze. Smyčka je de facto operace, která v programu vyjadřuje rozměr času.

Základní struktura bude vypadat asi takto:

```
void setup(){

  size(200,200);
  frameRate(15);
  println("tisk z funkce setup()");

}


void draw(){

  background(0);
  println("tisk z funkce draw() " + frameCount);

}
```

Po spuštění programu uvidíte v konzoli následující výstup:

```
tisk z funkce setup()
tisk z funkce draw() 1
tisk z funkce draw() 2
tisk z funkce draw() 3
tisk z funkce draw() 4
```

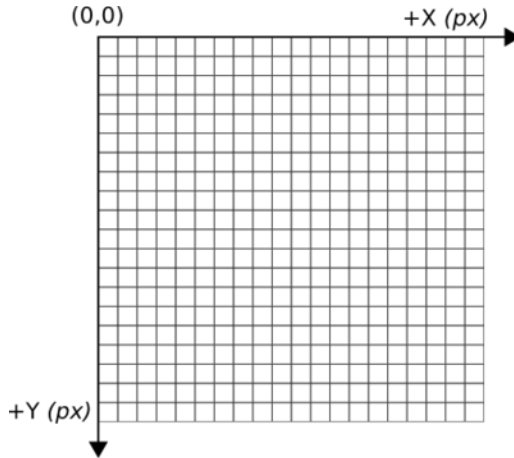
Velmi pozorně si pročtěte předchozí dva příklady. Ujistěte se, zda rozumíte dobře rozdílu mezi setup() a draw(), pro další pokračování bude pochopení klíčové.

6.2 Zobrazení

6.2.1 Orientace v prostoru

Okno programu by se dalo přirovnat k listu papíru, popřípadě k malířskému plátnu. Ve standardním dvoudimenzionálním modu má dva základní parametry, X a Y, označující souřadnice, ve kterých se veškeré kreslicí operace

pohybují. Důležité je zmínit, že oproti jisté konvenci v matematických grafech nese levý horní roh hodnotu $X = 0$, $Y = 0$. Směrem dolů hodnota Y přibývá, stejně tak jako hodnota X přibývá směrem doprava:



Tato konvence je převzata ze standardu počítačové grafiky, kdy první pixel v levém horním rohu nese souřadnicovou hodnotu právě $X = 0$, $Y = 0$. Obrácená osa Y se může ze začátku jevit matoucí. Důvody pro zdánlivé převrácení os pochopíme později, například právě při operacích se samotnými obrazovými body, pixely, které jsou standardně uspořádány od levého horního rohu doprava a níže.

Plochu je možné si představit jako prázdný prostor, na kterém lze zobrazovat grafiku. Tvary nebo kupříkladu text se zobrazují právě prostřednictvím zdrojového kódu, tj. instrukcemi psanými v editoru.

Na první pohled by se mohlo zdát, že je kód například při zobrazení obdélníka příkazem:

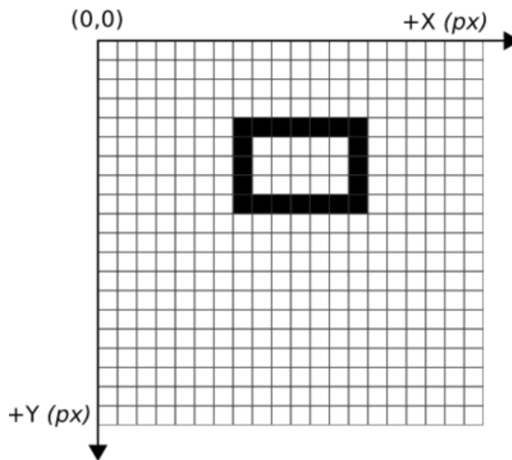
```
rect(x,y,sirka,vyska);
```

jehož výsledkem je kresba pouhého obdélníka, zbytečně komplikovaný oproti jiným zobrazovacím metodám. Důležité je si zde uvědomit koncepci *Processingu*, který tímto zobrazením primitivních objektů sestavuje celý obraz. To, co se zpočátku může zdát jako nadbytečná práce, tedy psaní

koordinátů každého ze zobrazovaných objektů, se posléze ukáže jako sofistikovaný a velmi ulehčující způsob přemýšlení o obraze. Veškeré parametry oddělené čárkou v kulatých závorkách příkazu *rect* náleží samotným vstupním hodnotám. Příkaz *rect* očekává čtyři parametry. Parametry mohou být celá čísla nebo čísla s desetinnou čárkou. Pro názornost, zadáme-li do parametrů „natvrdo“ hodnoty:

```
rect(8,6,7,5);
```

výsledné zobrazení na naší pomyslné ploše bude zhruba následující:



Zamyslíme-li se kupříkladu nad tím, že bychom v jakémkoli jiném nástroji chtěli zobrazit vícero, například tisíc obdélníků, na první pohled jednodušší GUI [2] grafické editory nám nedovolí tuto operaci učinit jinak, než že musíme všech tisíc obdélníků nakreslit. V případě *Processingu* nám bude stačit vytvořit rutinu pro kreslení libovolného počtu obdélníků a pak tuto rutinu spustit.

Zde se již dostáváme k samotnému jádru processingového přemýšlení. Programování obecně dokáže velmi ulehčit operace, jejichž pravidelnost dovedeme popsat. Veškeré umění psaní programu tedy spočívá v definicích těchto chování a redukci složitých jevů na jednoduché rovnice.

---

[2] Zkratka pro Guided User Interface klasické grafické editory jako Gimp nebo Photoshop.

Celé řemeslné umění psaní kódu v podstatě záleží na eleganci zápisu složitějších vztahů mezi různými parametry. Dovednosti se člověk učí postupně, osvojení si gramatické korektnosti a logické posloupnosti se mohou zpočátku jevit zbytečně komplikované, ovšem už po ovládnutí pouhých pár jednoduchých pravidel lze *Processing* využít kreativním způsobem.

Zkuste si na kus papíru, nejlépe čtverečkovaného, nakreslit jednotlivé objekty. Uvědomte si, v jakých dimenzích se pohybují.

6.3 Hodnota a její zobrazení

K čemu jsou vlastně hodnoty dobré? Tisk do konzole je jen kontrolní mechanismus, většinou se nejedná o výslednou podobu programu.

Po celou dobu sčítání a odčítání hodnot jsme nevyvolali žádnou funkci, která by kreslila na plátno.

Nyní je na čase ukázat si, jakým způsobem *Processing* rozumí kresbě. Tytéž hodnoty, které máme nyní v paměti, mohou být použity pro jakýkoli kreslený výstup. Řekněme, že chceme z těchto hodnot zobrazit například elipsu. K tomu potřebujeme vyvolat funkci pro tvorbu elipsy, pokud ji nechceme zrovna z nějakých důvodů popisovat matematicky (to je samozřejmě dobře možné).

Processing nemá předdefinované žádné složité tvary. Pracuje sám o sobě pouze s tvary primitivními, jako je bod, linka, trojúhelník, obdélník a elipsa. Pomocí těchto tvarů lze zkonstruovat nepřeberné množství obrazů. Představíme-li si nyní digitálně zpracovanou fotografii, můžeme kupříkladu říci, že je zkonstruována z bodů. Jednotlivé body, tj. pixely, mají jinou barevnou hodnotu a takto poskládaný obraz se ve výsledku jeví jako fotografie.

Problém v případě konstrukce syntetické fotografie nespočívá v geometrii; to je známá mřížka bodů s počtem šířky krát výšky obrazových bodů digitální fotografie. Problém je v barevných hodnotách, které neznáme a synteticky je jakoukoli matematickou funkcí velmi těžko obsáhneme.

Dat pro výpočet fotorealistického obrazu potřebujeme opravdu hodně, nástroje, které dokážou simulovat fotorealistický obraz, existují a není jich málo. Dokonce i v *Processingu* existují podobná rozšíření, která přímo zpracovávají a zajišťují obrazový výstup.

Tento příklad je trochu extrémní, ale naprosto pravdivý. Věc, kterou se snažím ilustrovat, je, že i s minimálním počtem primitivních geometric-

kých tvarů lze docílit téměř nekonečného (spočitatelně obrovského) množství obrazů, což by nám mělo dlouho stačit.

Nyní zpět k hodnotám. Máme-li již jakékoli hodnoty v paměti programu, můžeme kterékoli z nich použít na jakoukoli kreslicí funkci. Zde začíná naše experimentální část, často se totiž stává, že výsledek kreslení neumíme plně předpovědět a teprve zobrazením nám kresba vzhledem k hodnotě začne dávat smysl.

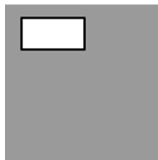
6.4 Kresba

Jak zobrazit hodnoty, které uchovává program v paměti? Jakoukoli hodnotu lze použít například jako jeden z vstupů (tzv. argumentů) v kreslicích funkcích.

Již zmiňovaný *rect()*, tedy rectangle – obdélník, vyžaduje k svému úspěšnému vykreslení čtyři parametry, čtyři číselné hodnoty[3]. Tyto parametry, hodnoty, můžeme buď zadat jako přímé číselné hodnoty, nebo do těchto parametrů můžeme zadat naši proměnnou, která obsahuje číslo.

Názorně:

```
// prvni zpusob vykresleni obdeniku
rect(10, 8, 40, 20);

// druhy zpusob vykresleni obdelniku
int prvniCislo = 10;
int druheCislo = 8;

rect(prvniCislo, druheCislo, 40, 20);
```



[3] Zde nezáleží, jestli se jedná o celé číslo *(int)* nebo číslo s desetinnou čárkou *(float)*.

Nyní je nám zřejmé, jak hodnoty mohou ovlivnit zobrazení. Místo hodnot zadaných číselně se zde v druhém způsobu vykreslení obdélníku objevují naše proměnné, které již mají zadaný obsah. Výsledek zobrazení bude v obou případech stejný, ovšem z hlediska struktury programu je druhý způsob daleko flexibilnější.

V *Processingu* je předdefinovaných jen několik základních tvarů, se kterými lze bohatě vystačit. Kromě již zmíněného *rect()* jsou zde také základní tvary:

point() bod (X, Y)

line() čára (X_1, Y_1, X_2, Y_2)

ellipse() elipsa $(X, Y$, šířka, výška)

point() trojúhelník $(X_1, Y_1, X_2, Y_2, X_3, Y_3)$

quad() čtyřúhelník $(X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4, Y_4)$

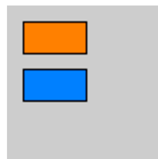
Nakreslete jeden bod, čáru a elipsu tak, aby se vzájemně nepřekrývaly. Zkuste změnit jejich pořadí, sledujte změny po spuštění.

6.4.1 Výplň a obrys

Některé tvary mají výplň a jiné (*point()* a *line()*) pouze konturu. Potřebujeme-li změnit barvu kresby objektů, můžeme použít vědomosti z předchozí kapitoly *(viz Barva, str. 48)*.

Výplň je definována jednou barevnou hodnotou přidanou do funkce *fill()*. V kódu bude zápis vypadat následovně:

```
fill(255,127,0);
rect(10, 10, 40, 20);

fill(0,127,255);
rect(10, 40, 40, 20);
```



Opak příkazu *fill()* je *noFill()*. Potřebujeme-li vykreslovat u objektů s výplní pouze jejich konturu, můžeme použít právě příkaz *noFill()*.

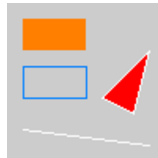
K ovládání barvy kontury použijeme příkaz *stroke()*, ten opět přijímá hodnotu barvy. Jeho opakem je *noStroke()*.

Ukažme si názorně zápisy:

```
fill(255, 127, 0);
noStroke();
rect(10, 10, 40, 20);

noFill();
stroke(0,127,255);
rect(10, 40, 40, 20);

noFill();
stroke(0,127,255);
triangle(80, 70, 60, 60, 90, 30);


stroke(255);
line(10, 80, 90, 90);
```

Všimněte si zde pořadí, v jakém příkazy určují výsledný obraz. Jeden příkaz na změnu barvy bude aplikován postupně na všechny objekty, které poté budou vykresleny, dokud opět nezměníte nastavení barev pro kresbu.

Výsledek kódu vypadá následovně:



Představme si nyní, že hodnota zadaná prvním způsobem zápisu se již nikdy v průběhu programu nemůže proměnit. V případě druhého způsobu zápisu získáváme díky možnosti změny jednoho parametru kontrolu nad kresebným výstupem.

Veškeré příkazy, které jsme si zatím ukázali, byly spuštěny pouze jednou. To znamená, že obdélník byl vykreslen a program, který již neměl žádné další příkazy, jednoduše skončil.

V další kapitole si předvedeme, jak vdechnout procesům pohyb a stálost, následující kapitola je věnována animaci.

6.5 Pohyb

6.5.1 Proměnlivost

Potřebujeme-li programu vdechnout život, musíme nejprve pochopit, jakým způsobem se program může proměňovat.

Proměnlivost se dotýká základní myšlenky programu trvajícího v čase. K tomu, abychom takovou proměnlivost mohli okem pozorovat, program může komunikovat pomocí různých výstupů. *Processing*, který byl navržen s důrazem na vizuální výstup, bude nejčastěji komunikovat právě obrazem. Obecně řečeno, proměnlivost je jakákoli změna hodnot v délce běhu programu, lze si ji představit například jako rozdíl mezi dvěma okénky filmu.

U filmových okének je rozdíl sice velmi těžko definovatelný, ilustruje ovšem dobře celý problém. Film nám zprostředkovává iluzi pohybu sledem několika okének za vteřinu. Stejně se chová i vizuální program. Rozdíl mezi filmem a programem je zhruba ten, že program může mít teoreticky dva a více různých konců, nebo nemusí končit vůbec. Zde by se dalo namítnout, že program je také nakonec nositelem kauzálního sledu operací. Vtip je v tom, že stejný program spuštěný dvakrát nemusí produkovat stejný výsledek.

Změny v chodu programu se dá docílit několika způsoby. Základní změna v běhu programu je lidská intervence, která je z pohledu programu, respektive tvůrce programu, jen velmi těžko předpovídatelná. Změna v chování programu bez takového vnějšího zásahu je pak vždy umělá *(viz Náhoda, str. 95)*.

Vraťme se ovšem k proměnlivosti. Proměnlivost je nezbytnou součástí animace. Pohyb je proměna stavu v čase. Měli bychom mít vždy na paměti, že program nerozumí tomu, co zrovna zobrazuje, program rozumí pouze hodnotám. Rozpohybovat lze tyto hodnoty, a tím posléze například jevy, které se v programu zobrazují a člověku dávají nějaký smysl.

Pro příklad si můžeme ukázat, jak proměna jedné hodnoty ovlivňuje obraz. K tomu, abychom mohli hodnotu proměňovat, ji musíme nejdříve pojmenovat.

```
int y = 0;
```

Tímto jsme pojmenovali svoji proměnnou. Jednou z možných variant zápisu je okamžité přiřazení hodnoty, tedy *y* je nyní nula. Pro animaci hodnoty použijeme přírůstek, tedy při kresbě jednotlivého okénka přičteme k hodnotě 1. Abychom viděli nějaký výsledek, můžeme opět kreslit objekt, který bude využívat proměňující se hodnotu v čase.

```
void setup(){
    size(640,480);
}

void draw(){
    background(255);

    // prirustek o jedna
    // lze zapsat i zkracene y++;
    y += 1;

    // kresba
    line(0 ,y ,width ,y);

}
```

Po spuštění tohoto programu uvidíte animaci trvající 480 okének. Animace bude pokračovat i nadále, hodnota *y* se bude stále proměňovat. Kresba bude teoreticky probíhat mimo plátno. K tomu, abychom kresbu udrželi v mezích plátna, můžeme použít jednoduchou podmínku, která bude vracet hodnotu *y* zpět na nulu.

```
if(y > height){
    y = 0;
}
```

Celá tato procedura jen ilustruje logiku programu. Hodnota *y* je zde přímo zobrazena, ale stejně tak se dá použít kdekoli jinde, tedy ne nezbytně na kresbu. Pro lepší představu jiného použití uvedu následující příklad.

Pod hodnotou si lze například představit pevně stanovenou hodnotu, dejme tomu výšku domu. Zadání výšky nepostaví dům, ale stanoví hod-

notu, se kterou můžeme dál počítat. Známe-li výšku plánované budovy, můžeme kupříkladu přidávat patra, dokud se této výšky nedosáhne. Tuto proceduru již lze považovat za logickou konstrukci a jedním ze základních postupů, jak takové logiky v programu docílit, je podmínka *(viz Podmínka, str. 68)*.

Vyzkoušejte přidat podmínku na konec programu nebo ji vynechat, ujistěte se, že rozumíte tomu, co v animaci způsobuje, zaměňte číslo v podmínce za jiné, s nižší hodnotou a pozorujte výsledek.

6.5.2 Animace

Statická kresba na plátno je jen jeden z možných výstupů. Animace v *Processingu* znamená, jak jsme si ukázali v předchozím příkladě, překreslovat plátno pokaždé jinými obrazy. Nyní si ukážeme základní pojetí tvorby pohyblivého obrazu.

Jak jsme si již uvedli dříve *(viz Základní struktura programu, str. 55)* funkce *draw()* je pouze prostředek pro změnu, nepřináší změnu sama o sobě. Funkce *draw()* zajišťuje sousledné vyvolání příkazů v neustálém trvání[4]. V *Processingu* si v podstatě uvědomíme, že psaním příkazů do smyčky píšeme pravidla jakoby pro jedno okénko filmu. Pohyb je vždy zajištěn proměnou našich datatypů *(viz Základní datatypy, str. 46)*, které v tomto okénku figurují.

Tedy například napíšeme-li takto samotnou smyčku:

```
void draw(){
    // 60x za vterinu spustena operace
}
```

vše uzavřené do složených závorek bude spuštěno šedesátkrát za vteřinu. Bude-li obsah základní funkce *draw()* neměnný, tj. budeme-li vyvolávat šedesátkrát za vteřinu totéž, animace tím bude v případě vizuálního výstupu jen abstraktním pojmem.

V tomto příkladu již dochází k spuštění smyčky. Výsledkem bude pouze šedivá plocha o rozměrech sto krát sto pixelů, jelikož na plátno nebylo zatím nic vykresleno. Následujícími příkazy *background()* a *rect()* nakreslíme nejprve plné pozadí a následně obdélník.

[4] Po dobu běhu programu.

Zápis bude vypadat takto:

```
void draw(){
    background(255);
    rect(10 , 10 , 30 , 30);
}
```

Takový zápis již provádí animaci a kreslí na plátno. Animace ovšem v tomto případě nebude patrná, jelikož zatím se v obraze nic neproměňuje. Veškeré hodnoty jsou statické, tudíž se vykresluje jeden čtverec na bílém pozadí na stále stejném místě.

Animace je proměna obrazu v čase, jednotlivá okénka se v čase musí lišit. Přemýšlejte, jakým způsobem k proměně může dojít.

K rozhýbání objektů jednoduše potřebujeme proměnit jeden z parametrů v čase. Abychom docílili animace, zkusme například vložit místo parametru pro kresbu čtverce v ose X počitadlo okének. *Processing* má již nastavenou proměnnou s údajem o počtu uběhlých okének pod názvem *frameCount*.

Proměnnou *frameCount* lze využít následovně:

```
void setup(){
    size(640,480);
}

void draw(){
    background(255);
    rect( frameCount % width , 10 , 30 , 30 );
}
```

Výsledkem tohoto zápisu bude opět animace. Čtverec se bude pohybovat rychlostí šedesáti pixelů za vteřinu zleva doprava. Tato hodnota v podstatě udává počet vykreslených cyklů funkce *draw()*, ta je v tomto případě pouze využita k pohybu vykreslovaného čtverce.

Počet okének vykreslovaných za vteřinu můžeme ovlivnit pomocí funkce *frameRate()* vložené do funkce *setup()*.

Tento pohyb je velmi názorný, ukazuje, že hodnotu počtu vykreslených okének lze použít například i jako hodnotu pro animaci. Stejně tak si můžeme založit i vlastní proměnnou, která bude mít identickou funkci.[5]

[5] V tomto konkrétním příkladu to sice nedává smysl, dá se použít stejně tak standardní proměnná *frameCount*, ukázka je zde jen pro ilustraci možností využití proměnných.

```
// nase promenna nazvana priznacne: pocitadlo
int pocitadlo = 0;


void setup(){
    size(640,480);
    frameRate(30);
}

void draw(){
    background(0);
    rect( pocitadlo, 10 , 30 , 30 );

    // zde jiz pricitame hodnotu, lze napsat i zkracenym
        zapisem:
    // pocitadlo++;

    pocitadlo += 1;

    // zde muzeme pouzit modulo na omezeni pocitadla na sirku
         platna

    pocitadlo = pocitadlo % width;

}
```

Definicí této hodnoty lze například časovat animace. Další metodou je animaci časovat pomocí funkce *millis()*. Tato funkce je nezávislá na počtu vykreslených okének za vteřinu, měří počet uběhnutých milisekund od startu programu. Může se nám také hodit pro precizní časování událostí v programu.

6.5.3 Dynamika pohybu

K docílení jednoduché interakce není zapotřebí mnoho. Animace je ovšem složitější problém, který budeme řešit v *Processingu* nejvíce matematicky. Zdá se, že dosažení dynamického pohybu je jeden z průvodních jevů pokročilého programování.

Nyní si můžeme ukázat, jak lze zkombinovat dosavadní znalosti k vytvoření nelineárního pohybu:

```
float x;
float y;
float rychlost;

void setup(){
    size(640,480);
     x = width / 2;
     y = height / 2;
     rychlost = 0.1;
}

void draw(){
    background(255);
    x += (mouseX - x) * rychlost;
    y += (mouseY - y) * rychlost;
    rect(x,y,30,30);
}
```

Čtverec bude nyní následovat kurzor nelineárním pohybem. Dynamika pohybu je dána prostou formulí:

```
x += (mouseX - x) * rychlost;
y += (mouseY - y) * rychlost;
```

Opište formuli a zkuste změnit rychlost, pozorujte změny, pozměníte hodnoty tak, aby se objekt pohyboval po křivce.

K proměnným *x* a *y* přiřazujeme zlomek rozdílu mezi pozicí kurzoru a proměnnou v obou osách. Tuto formuli lze využít téměř kdekoli, například při jemném přechodu barev a vyhlazování hodnot obecně.

Druhá proměnná, *rychlost*, by se k docílení podobného efektu měla pohybovat v rozmezí od 0 do 1.

6.6 Podmínka

Po seznámení s proměnlivostí procesů se můžeme dostat k prvnímu opravdovému strukturování programu. Zkusme nyní nastínit, jak taková struktura vypadá. Patrně nejpřímější řízení dějů v programu je podmínka. Podmínka říká: Jestliže je něco pravda, spusť následné příkazy. Podmínka funguje v podstatě jako výhybka, která odklání program do jeho různých cest. Uvedu zde krátký příklad vytváření podobné struktury v programu:

```
boolean prepinac = false;
int hodnota;

if(prepinac == true){
    hodnota = 1;
}else{
    hodnota = 0;
}

println(hodnota);
```

Velmi prostá struktura je v tomto příkladu vytvořena jednou podmínkou. Podmínka je v *Processingu* (a řadě jiných jazyků) značena slovem *if*, „jestli“. „Jestli“ potřebuje dostat odpověď na otázku v kulatých závorkách (ta může být jen pravda nebo nepravda), k tomu se hodí nejlépe již zmíněný *boolean*. Výsledkem *booleanu* může být porovnávání dvou čísel, znaků nebo řetězců znaků, tj. operace, které nám vrátí *true* nebo *false*.

Definujeme-li tedy náš *boolean* na začátku jako nepravdu, podmínka se v tomto případě nenaplní a program spustí kód uvozený složenými závorkami následujícími až slovo *else*.

V našem příkladu se jedná o druhou větev podmínky; ta sice není povinná, ale pro ukázku jsem ji zde rovnou zmínil. Tedy za ukončením *if* a vložením složených závorek můžeme dále za slovem *else*, „jestliže ne“, říci, co se stane v případě nenaplnění naší podmínky.

V tomto konkrétním případě se k proměnné *hodnota* přiřadí číslo 0.

To program následně ověřuje tiskem do konzole, kde se objeví 0.

6.6.1 If

Jak jsme si již uvedli, podmínka je jedním ze základních stavebních kamenů programu. Podmínku lze použít ve všech případech, kdy předpokládáme jednoznačnou odpověď *ano* nebo *ne*. Podmínka v *Processingu*, stejně jako v řadě ostatních jazyků, je vytvořena příkazem *if*.

Konstrukce podmínky vypadá pak následovně:

```
boolean pravda = true;

if(pravda == true){

    // spust nasledujici blok

}
```

Za povšimnutí zde stojí dvojité rovnítko. Tento speciální symbol se používá při již zmíněném porovnávání dvou stran. Proměnná nazvaná *pravda* získala hodnotu *true*. V případě podmínky nezáleží na počtu proměnných v kulatých závorkách, otázka může být i více kombinována. Záleží na výsledku, který vždy musí být pravda nebo nepravda, tedy *true* anebo *false*. Pro zkonstruování takové věty lze použít následující znaménka:

Vyzkoušejte i jiné logické operace, ujistěte se, že správně rozumíte, jak fungují, použijte vykřičník jako zápor, změňte logické operace.

```
boolean pravda = true;
int cislo = 3;
String text = "Lorem ipsum dolor et amet";

if(cislo == 3){
    // tato podminka je spustena
}

if(cislo != 3){
    // tato podminka neni spustena, znamenko nerovna se je
        opakem rovnitka
}

if(cislo > 3){
    // tato podminka neni spustena, cislo neni vetsi nez 3
}

if(cislo < 3){
    // tato podminka neni spustena, cislo neni mensi nez 3
}

if(cislo >= 3){
    // tato podminka je spustena, znamenko vetsi nebo rovna
        se, cislo rovna se 3
}
```

```
if(cislo <= 3){
    // tato podminka je spustena, znamenko mensi nebo rovna
        se, cislo rovna se 3
}

if(text.equals("Lorem ipsum dolor et amet")){
    // tato podminka je spustena, porovnavani textu je take
        mozne, ovsem pozor
    // k porovnavani celych retezcu textu se pouziva funkce
        equals("String")
}
```

Takovým výčtem operací jsme si pokryli základní konstrukci podmínky. Další znaménka slouží k skládání jednotlivých otázek. Znaménko pro logické „A“ je „&&“. Používá se následovně:

```
boolean a = true;
boolean b = false;

if(a == true && b == false){
    // tento blok bude spusten, a je pravda A b je pravda
}

if(a == true && b == true){
    // tento blok nebude spusten, b totiz neni pravda, b ==
        false
}
```

„Ampersand“ se na české klávesnici nevyskytuje, na anglické klávesnici jej naleznete většinou pod SHIFT - 7, pro logické „nebo“ se používá dvojitá svislá čára „||“, tzv. pipe:

```
boolean a = true;
boolean b = false;

if(a == true || b == false){

    // tento blok bude spusten, k tomu staci pouze, ze a je
        pravda NEBO b je pravda

}


if(a == true || b == true){
```

```
    // tento blok bude tedy opet spusten
}

if(a == false || b == true){
    // tento blok jiz spusten nebude, a ani b neni pravda
}
```

Pomocí skládání otázek můžeme zkonstruovat celé věty a složitější podmínky. V kombinaci s kulatými závorkami si vystačíme pouze s logickým *AND* a *OR*. Pokusme se ilustrovat situaci následovně:

```
boolean a = true;
boolean b = true;
boolean c = false;

if( (a && b) || c ){
    // tento blok bude spusten
}

if( !(a && b) || c ){
    // tento blok nebude spusten
}
```

Jaký je rozdíl mezi prvním a druhým blokem? Poslední specialita booleanovských operací je znaménko vykřičník. Takové znaménko před proměnnou typu *boolean* znamená opak tvrzení, tedy, je-li *pravda* pravdou, *!pravda* je nepravdou, výsledkem je *false*.

Zkuste si v duchu představit otázky, na které lze odpovědět „ano“ nebo „ne“, zkuste je poté přepsat do tvaru podmínky, zkuste k tomu využít další proměnné, porovnávejte je

```
boolean pravda = true;

println( pravda ); // tiskne true
println( !pravda ); // tiskne false
```

Dále si všimněte, že v podmínce chybí jakékoli porovnávací znaménko. Ve skutečném programování se setkáte spíše s tímto zápisem, je to zápis zkrácený. Místo neustálého rozepisování *něco rovná se rovná se true* lze

napsat pouze v kulatých závorkách. Je-li naše *něco* datového typu *boolean*, *Processing* bude takovému zápisu rozumět. Tedy nejjednodušší zápis může vypadat i takto:

```
boolean a = true;

if(a)
    ;//pouze tento jediny radek bude spusten


if(a){
    // cely tento blok
    // bude spusten
}
```

K úplnému zkrácení se dají vypustit i složené závorky, to vede opět k rychlejšímu zápisu. Rozdílem takového zápisu je fakt, že taková podmínka spouští pouze jeden následující řádek. Přestože je to naprosto správný způsob programování, někteří programátoři takový zápis neradi používají kvůli zhoršení přehlednosti v kódu. Opět se s takovým zápisem často setkáte, člověk je jednoduše líný živočišný druh a *if* se v programování používá opravdu často.

6.6.2 Else

K dalšímu rozšíření podmínek slouží již zmíněný příkaz *else* a dá se přeložit takto: jestliže je něco pravda, udělej toto, jestliže ne, proveď něco jiného. Zápis vypadá následovně:

```
boolean a = false;

if(a){

    // kdyby a bylo true, tak by se spustil tento blok kodu

}else{

    // ale protoze je nase a false, spusti se nyni kod
    // v tomto bloku

}
```

6.6.3 ? :

Vyzkoušejte si zkrácené zápisy, ověřujte jejich funkčnost tiskem do konzole nebo například proměnou barev objektů.

Abychom podmínky vyčerpali nadobro, ukážeme si poslední možný zápis, který může být ještě úspornější.

```
boolean a = true;

if(a)
background(0);


background( a ? 0 : 255 );
```

6.7 Interakce

Interakce se dá obecně definovat jako vzájemné působení. V počítačové terminologii se má nejčastěji na mysli vzájemné působení člověka a stroje. Přímá interakce, která znamená vzájemné působení dvou prvků, předpokládá rozměr času[6]. Takový rozměr trvání programu v čase umožňuje uživateli přímý vstup do proměnných, tedy do obsahu našich hodnot.

Je dobré si uvědomit, že **interakce** je do jisté míry při práci s počítačem pozorovatelná neustále. Pouhý pohyb myši lze považovat za interakci člověka se strojem. Pohyb uživatele přímo proměňuje hodnoty vykreslující pozici kurzoru, stiskem klávesy v textovém editoru píšeme text atp.

Při interakci je nutné do určité míry předvídat chování uživatele. Interakce primárně vychází z fyzické zkušenosti s předměty kolem nás, je jí myšleno vše, co může uživatel přímo ovlivnit.

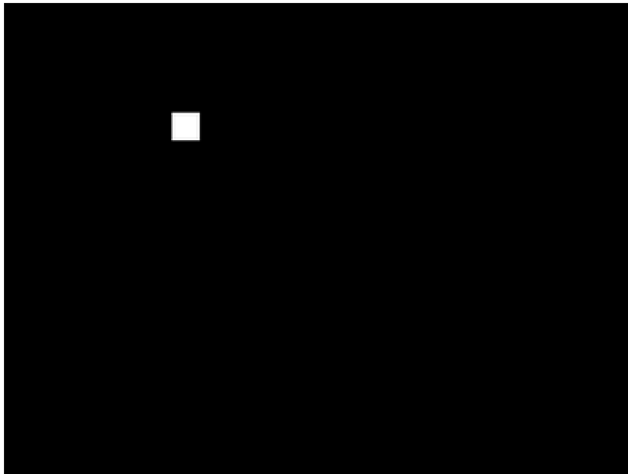
Interakce se dá rozdělit do dvou základních oblastí – na interakci přímou a nepřímou nebo, chcete-li, na interakci vědomou a nevědomou. Mezi těmito kategoriemi neexistuje jednoznačná hranice.

Obecně je toto rozlišení spíše stupnicí od jednoduchých, přímo a okamžitě viditelných jevů po uživatelově vstupu ke komplikovanějším procedurám spouštěným jen s částečným (nebo žádným) vědomím uživatele.

[6] Tedy program trvající v čase.

Základní, tedy přímou interakci si můžeme ukázat v následujícím příkladu:

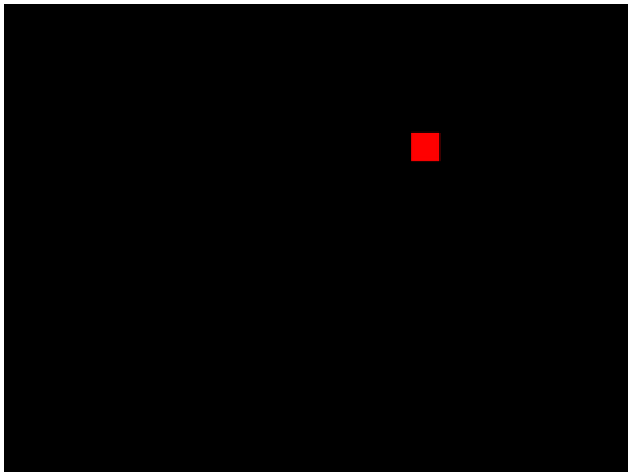
```
void setup(){
    size(640,480);
}

void draw(){
    background(0);
    // vsimnete si parametru mouseX a mouseY
    rect(mouseX,mouseY,30,30);
}
```



Toto je jedna z nejzákladnějších možných interakcí. Vstupem pro pozici kresleného obdélníka se nám stávají dva parametry *mouseX* a *mouseY*.

Čtení pozice myši je jedním z nejsnazších možných způsobů interakce stroje s člověkem. Nyní si ukážeme, jak lze detekovat stisknutí tlačítka myši. Pro tento účel můžeme využít předdefinovaných metod *Processingu*. K detekci kliku lze použít funkci *mousePressed()*.

```
boolean stisknuto = false;

void setup(){
    size(640,480);
}

void draw(){
    background(0);

    if(stisknuto){
        fill(255);
    }else{
        fill(255,0,0);
    }

    rect(mouseX,mouseY,30,30);
}

void mousePressed(){
    stisknuto = !stisknuto;
}
```



Všimněte si poslední funkce *(viz* *Funkce, str. 83)*. Ta je programem spuštěna jen tehdy, stiskne-li uživatel tlačítko myši. Abychom uchovali informaci

o stisku, přepneme jím stav jedné z našich proměnných nazvané *stisknuto*. Jednořádkový zápis s vykřičníkem je zkrácení pro přepnutí stavu *(viz Podmínka, str. 68)*.

K tomu, abychom detekovali stisk tlačítka myši, použijeme funkci *mousePressed()*. Další možností interakce je detekovat, které tlačítko myši jsme stiskli, popřípadě detekovat stisk klávesy. Výčet dalších možností je následující:

```
// detekce stisku tri ruznych tlacitek mysi
void mousePressed(){
    if(mouseButton==LEFT){

    }else if(mouseButton==RIGHT){

    }else if(mouseButton==CENTER){

    }
}


// detekce tazeni stisknute mysi
void mouseDragged(){
    // muzeme zde opet detekovat tlacitko
}

// timto detekujeme moment
// uvolneni stisku tlacitka
void mouseReleased(){
    // opet zde muzeme detekovat, ktere
    // tlacitko bylo stisknuto
}

// stisk klavesy s detekci typu
void keyPressed(){

    if(key=='a'){

    }else if(keyCode=='ENTER'){

    }
}
```

```
// detekce uvolneni klavesy
void keyReleased(){

}
```

Tímto výčtem jsme v podstatě vyčerpali možnosti interakce se standardními vstupy uživatele v *Processingu*. Veškeré tyto definice musí stát mimo základní funkce *Processingu*, tedy mimo *setup()* a *draw()*.

Proměnná *mouseButton* nám říká údaj o naposledy stisknutém tlačítku myši, stejně tak funguje proměnná *key* a *keyCode* ve funkci *keyPressed()* a *keyReleased()*.

Obě proměnné *key* a *keyCode* jsou buď vyjádřeny číslem, které lze vyhledat například v mapě znaků, nebo znakem ve tvaru *char*. Hodnotu *key* lze také porovnávat přímo se znakem uvozeným apostrofy (například *'k'*). V případě *keyCode* lze porovnávat hodnotu kódovaných kláves.

Jestli je klávesa kódovaná, zjistíme dále pomocí proměnné *CODED*. Asi takto:

```
void keyPressed(){

    if(key == CODED){
        if(keyCode==ENTER){
            // detekce klavesy ENTER
        }
    }else{
        if(key==' '){
            // detekce mezerniku
            // porovnano s prazdnym znakem
        }
    }
}
```

Hrubý výčet pojmenovaných kláves, které můžeme porovnat s proměnnou *keyCode*, je následující:

```
// spiky
UP, DOWN, LEFT, RIGHT

// specialni klavesy
ALT, CONTROL, SHIFT

// dalsi dulezite klavesy
BACKSPACE, TAB, ENTER, RETURN, ESC, DELETE
```

Nevíme-li si rady, jak určitou klávesu detekovat, můžeme si jednoduše vytisknout do konzole její číslo pomocí příkazu *println()*.

```
void keyPressed(){
    println("cislo klavesy: "+key);
}
```

Vyzkoušejte detekovat různé klávesy, zkonstruujte jednoduché přepínače, které mohou reagovat na podněty.

6.8 Pole

Jedna proměnná může uchovat pouze jednu hodnotu. V programování se brzy dostaneme do situace, kdy budeme potřebovat operovat s vícero hodnotami najednou. Tak například, chceme-li použít deset čísel z Fibonacciho řady, můžeme si je definovat postupně.

```
int hodnota0 = 2;
int hodnota1 = 3;
int hodnota2 = 5;
int hodnota3 = 8;
int hodnota4 = 13;
int hodnota5 = 21;
int hodnota6 = 33;
int hodnota7 = 54;
int hodnota8 = 87;
int hodnota9 = 141;
```

K zkrácení zápisu místo neustálého přepisování téhož je zde takzvané pole. Pole neboli *Array* se v programování vyznačuje hranatými závorkami.

```
int mojePoleHodnot[];
```

Všimněte si zde pouze hranatých závorek na konci názvu proměnné. Druhý možný zápis s naprosto identickým významem je vložit hranaté závorky před název definované proměnné.

```
int [] mojePoleHodnot;
```

Oba způsoby jsou naprosto identické. Pouhá definice pole ovšem pole nevytváří. K tomu, abychom nyní do pole něco mohli začít ukládat, ho musíme nejprve inicializovat na pevný počet hodnot. V tomto případě lze

hodnoty zadat přímo v definici pole. Pro takovou operaci je možné využít zápis pomocí složených závorek:

```
int mojePoleHodnot[] = {0,1,1,2,3,5,8,13,21,33};
```

Tento zápis inicializace je v tomto konkrétním případě zřejmě nejrychlejší. Hodnoty z pole, které je nyní celé uloženo pod jednou proměnnou, můžeme získat následujícím způsobem:

```
int mojePoleHodnot[] = {0,1,1,2,3,5,8,13,21,33};

println(mojePoleHodnot[0]);
println(mojePoleHodnot[1]);
println(mojePoleHodnot[5]);
//atd.
```

Další způsob zápisu je ovšem obvyklejší, a jak se dozvíme v příští kapitole *(viz Smyčka, str. 81)*, bude pro nás nejvýhodnější.

```
int mojePoleHodnot[];

void setup(){
    mojePoleHodnot = new int[10];

    mojePoleHodnot[0] = 0;
    mojePoleHodnot[1] = 1;
    mojePoleHodnot[9] = 33;
    // atp.
}
```

Všimněte si, že se v zápisu objevilo slovo *new*. Toto slovo jsme zatím nepoužili a jeho význam rozvedeme později *(viz Třída a objekt, str. 89)*. K tomu, abychom přiřadili hodnotu jednotlivých míst v poli, musíme nejdříve uvést do hranatých závorek, kam budeme hodnotu zapisovat. Stejně tak se později k číslu, které jsme uložili, můžeme dostat.

Tímto způsobem jsme si definovali pole o rozměru deseti hodnot. Stále je ovšem patrné, že v kódu se zbytečně opakujeme rozepisováním názvu jedné proměnné.

Hodnot v polích nebude vždy jen deset. K lepší práci s poli potřebujeme rychlejší způsob. Jak na to, zjistíte v následující kapitole věnované smyčkám.

Pole jsou abstraktní koncepce. Ujistěte se, že dobře rozumíte, jakým způsobem pracují. Najděte si pro pole vhodnou paralelu, například

6.9 Smyčka

Smyčka je mocný nástroj a dobrý pomocník při práci s opakovanými operacemi. V minulé kapitole jsme si definovali pole. Definici nyní zopakujeme a zkusíme ji více automatizovat pomocí smyčky.

```
int mojePoleHodnot[];

void setup(){
    mojePoleHodnot = new int[10];

    for(int i = 0; i < 10; i += 1){
        mojePoleHodnot[i] = 0;
    }
}
```

A první smyčka je na světě. Zápis, který na první pohled vypadá trochu odpudivě, může být váš dobrý přítel. Trochu si jej rozebereme.

Formule začíná slovem *for*. Následují kulaté závorky. V nich je v podstatě rozsah, který potřebujeme zopakovat. Hodnoty jsou odděleny středníkem.

Vyzkoušejte si rozdílné smyčky s různým rozsahem, opakujte si složitější zápis „for“, dokud jej nebudete umět sami bezchybně zkonstruovat, zkuste smyčku využít s měnící se proměnnou uvnitř, kreslete ze smyčky objekt.

počátek int i = 0

konec i < 10

přírůstek i += 1

Hodnota, se kterou pracujeme, se jmenuje *i*, ale může se jmenovat dle vaší libosti například:

```
    for(int pocitadlo = 0; pocitadlo < 10; pocitadlo += 1){
        mojePoleHodnot[pocitadlo] = 0;
    }
```

Podle potřeby lze také definovat rozsah smyčky. Kupříkladu počítání v opačném pořadí:

```
    for(int pocitadlo = 9; pocitadlo >= 0; pocitadlo -= 1){
        mojePoleHodnot[pocitadlo] = 0;
    }
```

Název pro jednoduchou smyčku *i* se používá velmi často, může například značit slovo *index*. Smyčky vykonávají v programu tvrdou práci, využívají se pro hromadné operace, z nichž se často stávají náročné výpočty.

Vraťme se ovšem k definici Fibonacciho řady z předchozí kapitoly *(viz Pole, str. 79)*. Pokusíme se nyní skloubit to, co jsme se naučili o smyčkách a polích. Zkusíme si napsat jednoduchý program, který nám vygeneruje tolik čísel z Fibonacciho řady, kolik budeme potřebovat.

```
int delka = 18;
int fibonacci[];

void setup(){
    fibonacci = new int[delka];
    fibonacci[0] = 0;
    fibonacci[1] = 1;


    for(int i = 2; i < delka; i += 1){
        fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
    }

    println(fibonacci);
}
```

V konzoli se nám objeví:

```
[0] 0
[1] 1
[2] 1
[3] 2
[4] 3
[5] 5
[6] 8
[7] 13
[8] 21
[9] 34
[10] 55
[11] 89
[12] 144
[13] 233
[14] 377
[15] 610
[16] 987
[17] 1597
```

Čísel samozřejmě může být nepočitatelně. Smyčky mají tu výhodu, že dokážou opakovat jednu operaci. Můžeme využít jazyka k definici operace a zapojením dalších proměnných ji povýšit na princip. Máme-li takto zpracovaný princip, můžeme jej například zobrazit nebo s ním dále pracovat. K lepšímu uspořádání se dostaneme v další kapitole *(viz Funkce, str. 83)*.

Pro ilustraci Fibonacciho řady můžeme využít druhou smyčku ke kresbě:

```
for(int i = 0; i < delka; i += 1){
    line(fibonacci[i],0,fibonacci[i],height);
}
```

a výsledkem kresby bude:

6.10 Uspořádání struktury programu

Jednotlivé bloky kódu lze dále strukturovat. Strukturování v kódu se provádí zpravidla tam, kde potřebujeme opakovaně spouštět identický blok kódu nebo si zpřehlednit probíhající operace.

6.10.1 Funkce

Nejjednodušším řešením problému je jeho rozdělení na menší části. Problém, jak dojít k jednomu výsledku, se dá rozložit na jednotlivé po sobě jdoucí události a ty postupně popsat.

Dílčí operace si pro přehlednost můžeme definovat do bloku takzvané funkce. Základní funkcí je *void*. *Void* v podstatě tvoří blok kódu, který uceluje nějakou proceduru, ta může být posléze jednoduše spuštěna jedním příkazem. Jestliže by mohly jednotlivé proměnné ve větě označovat podstatná jména, funkce se dají například chápat jako slovesa. S funkcemi jsme se setkali již dříve při definici základních smyček programu.

```
void setup(){

}

void draw(){
    nakresliObdelnik();
}

void nakresliObdelnik(){
    rect(10,10,20,20);
}
```

Vedle standardních funkcí *Processingu* *setup()* a *draw()* jsme si nyní definovali třetí funkci, nazvanou *nakresliObdelnik()*. Tato funkce je pak spuštěna uvnitř kreslicí smyčky *draw()*.

V tomto konkrétním případě samotná funkce nedává mnoho smyslu. Příkaz *rect()* v podstatě není nic jiného nežli funkce *Processingu* „nakresli obdélník“. Všimněme si ovšem, že tato funkce *rect()* má v kulatých závorkách parametry, které označují, kde má být obdélník vykreslen a jaké má mít rozměry. Tyto parametry se v programování obecně nazývají *arguments* nebo také vstupní hodnoty funkce.

Chceme-li vytvořit funkci a potřebujeme-li do ní poslat vstupní parametry, musíme při její definici vepsat do kulatých závorek, jaké parametry bude funkce očekávat.

```
void nakresliObdelnik(int x, int y){
    rect( x , y , 20 , 20 );
}
```

Jednotlivé parametry se v kulatých závorkách oddělují čárkou a může jich být teoreticky neomezené množství. Parametry vždy musí dodržovat dané datatypy, v opačném případě *Processing* funkci nedokáže najít.

Důležité je upozornit, že vstupní hodnoty jsou vytvořeny pokaždé, spustíme-li funkci, a zanikají, je-li funkce ukončena. Takovým proměnným se říká dočasné nebo lokální. V programátorské praxi se můžeme setkat s jejich uvozováním podtržítkem.

```
void nakresliObdelnik( int _x, int _y ){
    rect( _x , _y , 20 , 20 );
}
```

Podtržítka nemají žádný faktický význam, jsou jen součástí názvu proměnné a programátoři je používají pro lepší organizaci.

V *Processingu* lze dále definovat funkce, které se jmenují stejně, ty pak ale musí mít jinou skladbu vstupních hodnot. Představme si, že budeme potřebovat dvě funkce *nakresliObdelnik()*, jednu můžeme nechat bez vstupních parametrů a druhou budeme definovat parametry *x* a *y*. Tak můžeme odlišit dvě různé funkce pouze způsobem spuštění.

```
void nakresliObdelnik( int _x, int _y ){
    rect( _x , _y , 20 , 20 );
}

void nakresliObdelnik(){
    rect( 10 , 10 , 20 , 20 );
}

void draw(){
    nakresliObdelnik();
    nakresliObdelnik( 10 , 20 );
}
```

Tento příklad je správně a vykreslí dva různé obdélníky. Máme nyní na výběr, potřebujeme-li zadat parametry, nebo ne, v druhém případě se nakreslí obdélník v předem nadefinované pozici.

Void je pouze jedna z možností definic funkcí. Další možné funkce můžeme definovat podle datatypů, které budeme očekávat jako výsledek funkce. Tedy můžeme si definovat funkci, která nám zjistí, zdali se kurzor nachází v daném obdélníku. Místo prázdné funkce *void* budeme očekávat odpověď ve tvaru *boolean* pravda nebo nepravda. Funkce uvozená datatypem bude vždy očekávat odpověď ve stejném tvaru takového datatypu. Pomocí slova *return* můžeme funkci přiřknout výsledek, v tomto případě *true* nebo *false*.

```
Void setup(){

    size(640,480);

}

void draw(){
    if( jeKurzorVObdelniku(10,20,20,20) ){
        nakresliObdelnik(10,20);
    }

}

boolean jeKurzorVObdelniku(int _x, int _y, int _sirka, int
    _vyska){

    if(mouseX >= _x && mouseX <= _x +_sirka &&
    mouseY >= _y && mouseY <= _y +_vyska){

        return true;

    }else{

        return false;

    }
}

void nakresliObdelnik(){
    rect( 10 , 10 , 20 , 20 );
}

void nakresliObdelnik(int _x, int _y){
    rect( _x , _y , 20 , 20 );
}

void nakresliObdelnik(int _x, int _y, int _sirka, int _vyska)
    {
    rect( _x , _y , _sirka, _vyska );
}
```

Bude-li se kurzor nacházet v prostoru vymezeném v otázce, nakreslí se obdélník o stejné velikosti. Náš program již začíná být strukturovaný.

Tímto způsobem můžeme dále větvit podmínky a rozdělovat kód do jednotlivých spustitelných bloků příkazů.

Funkce stejně jako proměnné doporučuji pojmenovávat stručně a výstižně. U funkcí dále platí, že by měly dělat pouze jednu věc a měly by ji provádět dobře. Je lepší problém rozdělit na více funkcí, a tím strukturovat program. Strukturování podobným způsobem je velmi výhodné, budete-li se ke kódu vracet s odstupem času nebo bude-li jej číst někdo jiný než vy.

Budete-li kód chtít sdílet, je dobré jej učinit co nejvíce čitelným pro ostatní. Zde je velmi dobrá tradice napsat ke každé funkci jeden řádek nebo odstavec komentáře o tom, co funkce přesně dělá (pokud to není na první pohled zřejmé).

Co se délky funkcí týče, dobré pravidlo zní: Přesáhne-li počet vnitřních proměnných tucet, měli byste už přemýšlet o rozdělení do dvou funkcí.

6.10.2 Funkce a jejich datatypy

Funkce *void* je jen jednou možností z celé škály definic funkcí. Prázdná funkce *void* nic nevrací zpět, jednoduše řečeno, nemá žádný výsledek. Funkci s výsledkem můžeme definovat pomocí příkazu *return* zevnitř funkce.

Příkaz *return* musí navracet hodnotu stejného datatypu, jako je naše funkce. Pro pochopení budu lépe ilustrovat zápis plodné funkce. Řekněme, že potřebujeme funkci, která k vstupní proměnné přičte polovinu její původní hodnoty. Taková funkce by mohla vypadat následovně:

```
float prictiPolovinuHodnoty(float _vstup){

    _vstup += _vstup / 2.0;
    return _vstup;

}
```

Za pozornost zde stojí místo slova *void* nám již dobře známé slovo *float*. Zapojení takové funkce do chodu našeho programu by v praxi mohlo vypadat následovně:

```
float a = 5;

println(a);
// vytiskne 5 do konzole
```

```
a = prictiPolovinuHodnoty(a);

println(a);
// vytiskne 7.5

a = prictipolovinuHodnoty(a);

println(a);
// vytiskne 11.25

//atd.
```

Jak můžeme vidět, definice funkce zůstala stejná a výsledky se liší podle zaslaných argumentů, vstupních hodnot do funkce. Naše vlastní definice funkcí mohou mít dále jakékoli datatypy, které již známe.

```
String pozdrav(){
    return "ahoj!";
}

int inverze(int _vstup){
    return vstup*(-1);
}

// vice moznosti podminkou

String pozdrav(int _vstup){
    if(_vstup==1){
        return "ahoj vracim jedna!";
    }else if(_vstup==2){
        return "zdravim tu mate dva!";
    }else{
        return "dobry den, nevim co chcete ...";
    }
}
// atd.
```

Zkuste si zvykat na strukturování pomocí funkcí, využívejte je všude, kde vznikne ucelená operace, zkuste přemýšlet a defino-

V další kapitole se přesuneme de facto již k definici vlastních datatypů, budeme hovořit o třídách a objektech.

6.10.3 Třída a objekt

Funkce jsou jedním ze způsobů, jak strukturovat program. Veškerý kód, který jsme doposud psali, je vlastněn takzvanou třídou. Třída vlastní veškeré funkce i proměnné, které v *Processingu* běžně píšeme. Můžeme to nazvat mateřským objektem. V rámci tohoto objektu můžeme definovat své vlastní objekty, které mohou mít různé vlastnosti.

V úvodu jsme se dozvěděli, že *Processing* je objektově orientovaný jazyk. Nad tím je na čase se pozastavit. Co to vlastně znamená objekt? Ilustrace koncepce programování je prostá, stačí se rozhlédnout kolem sebe a zaměřit se na jeden konkrétní objekt (hrnek, propiska, kniha). Tento objekt má nějaké vlastnosti. Tak třeba psací pero má svoji hmotnost, rozměry ve třech osách, barvu inkoustu, obsah inkoustu a vlastní barvu pera, pravděpodobně také nějaké logo jako potisk. Propiska dále může být „otevřená“ nebo „zavřená“, mohli bychom takto pokračovat dále. Všechny tyto vlastnosti si můžeme představit jako naše proměnné. Třída pak bude to, co mají psací pera společného, jednotlivými parametry budeme odlišovat jedno pero od druhého.

To, co s psacím perem můžeme dělat, si představme jako funkci. Tak například uzávěr pera lze „otevřít“ a „zavřít“. Perem pak můžeme psát atp. Tyto činnosti ovlivňují naše parametry konkrétního objektu.

Třída je tedy souhrn definic, které popisují nějaký typ objektu, pero, tužku, hrnek. Objekt je pak konkrétní propiska, hrnek atp.

V kódu naše propiska vypadá takto:

```
class Pero{

    // promenne hodnoty propisky
    float rozmery[];
    float pozice[];
    color barvaNaplne;
    boolean otevrene;
    String napis;

    // konstruktor misto pro vstupni hodnoty
    Pero(float _x, float _y, color _barva){
        pozice = new float[2];
        barvaNaplne = _barva;
```

```
        pozice[0] = 30;
        pozice[1] = 40;
    }

    // funkce pera
    void otevri(){
        otevrene = true;
    }

    void zavri(){
        otevrene = false;
    }

    void napisNeco(){

        otevri();
        stroke(barvaNaplne);
        line(pozice[0],pozice[1],10,10);
        zavri();
        // a tak dale
    }
}
```

Tímto jsme si definovali naši propisku. Než si ukážeme, jak ji použít, vysvětleme si některé nejasnosti. Třída se definuje pomocí slova *class*, za ním následuje název třídy, v našem případě *Pero*.

Uvnitř složených závorek, které vymezují definici, nyní můžeme nastavit potřebné vlastnosti, proměnné. Následuje takzvaný konstruktor. Konstruktor není v definici povinný, ale pro ilustraci jej zde uvádím. Konstruktor se nám bude hodit později, kdy budeme chtít vytvořit naši propisku připravenou k práci.

Konstruktorů v definici jedné třídy může být i více. Rozlišujeme je počtem, pořadím a typem zasílaných argumentů. Tímto způsobem si můžeme popsat všechny způsoby vytváření objektu. Například podle stupně známých proměnných, které jsme schopni v dané chvíli nově vznikajícímu objektu definovat.

Konstruktor je zkrátka místo, kde můžeme do nově vznikajícího objektu zaslat počáteční informace. Budu-li například nyní potřebovat dvě propisky, jednu červenou a druhou modrou, mohu je vytvořit následujícím způsobem:

```
Pero modrePero;
Pero cervenePero;

void setup(){

    size(640,480);

    // vytvareni dvou objektu zde
    modrePero = new Pero(30,30,color(0,0,255));
    cervenePero = new Pero(50,80,color(255,0,0));

}


void draw(){


    modrePero.otevri();
    modrePero.napisNeco();
    modrePero.zavri();

    cervenePero.otevri();
    // atp.
}
```

Podobně nyní můžeme vytvářet propisek, kolik budeme chtít. Definice třídy pomocí slova *class* je jako připravená šablona. Vytvoření samotného objektu slovem *new* vytváří jednu instanci.

Příkaz *new* se používá vždy, žádáme-li třídu, tj. šablonu, o manipulovatelný objekt. Tento příkaz se dá přirovnat k výrobě objektu. Slovem *new* žádáme šablonu o produkt, se kterým budeme moci manipulovat.

V předchozím příkladu se vyskytl ještě jeden důležitý znak, který není na první pohled dobře patrný. Novým znakem je tečka. Tečka je velmi důležitá v objektově orientovaném přemýšlení. Skrze tečku za objektem, perem, se v *Processingu* můžeme dostat ke všem definovaným vnitřním proměnným, stejně tak jako můžeme spouštět veškeré funkce objektu.

Pro použití tečky se v programátorském žargonu vyskytuje výraz „tečková syntaxe“. V případě naší třídy ji můžeme ilustrativně využít pro změnu vnitřních hodnot objektu, změňme například barvu náplně našeho modrého pera na černou.

```
modrePero.barvaNaplne = color(0);
```

Jak je z příkladu patrné, modrá propiska je objekt, který má pojmenovanou hodnotu *barvaNaplne*. K němu se dostaneme pomocí tečkové syntaxe a *Processing* nám jej dovolí změnit.

Příklad s propiskou je ryze abstraktní a jistě naleznete sami řadu lepších.

Rozhlédněte se kolem sebe a zkuste se zamyslet o objektech a jejich možných vlastnostech.

6.10.4 Práce s objekty

Nyní již víme, jak definovat a vytvořit objekt. Tyto schopnosti nám nyní velmi ulehčují práci se složitějšími vztahy mezi objekty.

Zkusme si pro ukázku naplnit nám známé pole jednotlivými objekty. Takové uvažování se nyní může jevit trochu abstraktně, zkusme ho proto nejdříve uvést nějakým příkladem.

Místo propisek si vytvoříme *entitu*, která bude reagovat na náš podnět. Bude mít následující vlastnosti. Za prvé každá z *entit* bude mít své vlastní číslo, dále bude mít údaje o své pozici a každá z *entit* bude následovat kurzor jinou rychlostí, naposledy se jednotlivé *entity* budou odlišovat barvou. Při přemýšlení nad interakcí objektů s uživatelem si můžeme vzpomenout na kapitoly o pohybu *(viz Interakce, str. 74)* a jeho dynamice *(viz Dynamika pohybu, str. 67)*.

Můžeme zde pro ukázku definovat více konstruktorů, tedy více možných způsobů vytvoření objektu.

Nejprve se pusťme do obecné definice naší *entity*:

```
class Entita {
  int id;
  // vice promennych na jednom radku
  float x, y;
  float rychlost;
  color c;

  // prvni konstruktor
  Entita(int _id, int _x, int _y, int _rychlost, color _c) {
    id = _id;
    x = _x;
    y = _y;
    rychlost = _rychlost;
    c = _c;
  }
```

```
  // druhy konstruktor
  Entita(int _id) {
    id = _id;
    x = random(width);
    y = random(height);
    rychlost = random(0.01, 0.3);
    c = color(random(255));
  }

  void kresli() {
    noStroke();
    fill(c);
    ellipse(x, y, 10, 10);
  }

  void posunSeKeKurzoru(){
    x += (mouseX - x) * rychlost;
    y += (mouseY - y) * rychlost;
  }
}
```

Definice je až na pár výjimek opakováním známého. První novinka je možnost definovat více proměnných stejného typu na jeden řádek. Je to pouze úspornější.

Dále přichází ony dva konstruktory. Oba se jmenují stejně, mají ovšem rozdílný počet vstupních argumentů, a tím se odlišují. Při tvorbě naší nové *entity* jako živého objektu ji můžeme vytvořit oběma způsoby. V druhém konstruktoru jsem použil příkaz *random()*, ten zatím neznáme, více se o něm můžete dočíst v další kapitole o náhodě. V podstatě nám tento příkaz umožňuje vygenerovat náhodné číslo[7]. To se nám nyní bude hodit: chceme, aby každá z entit, kterou vytvoříme, měla vlastní způsob chování.

Funkce *kresli()* pouze zobrazí na konkrétních souřadnicích každé *entity* elipsu. Stejně tak bychom zde mohli kreslit jakoukoli jinou obrazovou reprezentaci našeho objektu. Tvar ponechám na vaší fantazii.

K funkci *posunSeKeKurzoru()* není třeba příliš dodávat, dělá přesně to, co byste od ní očekávali. Připomínáme si zde formuli z kapitoly o pohybu *(viz* *Dynamika pohybu, str. 67)*.

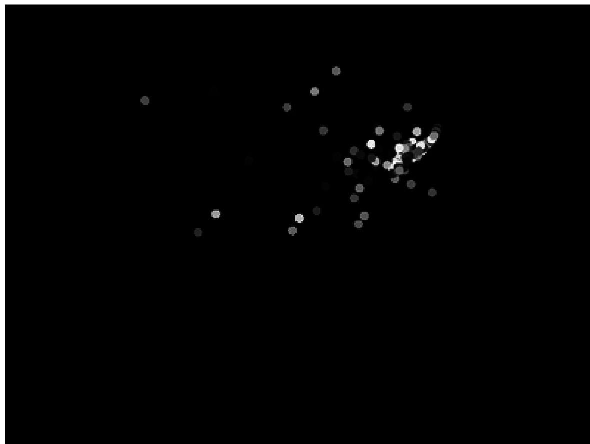
---

[7] Správně tedy pseudonáhodné číslo.

Nyní nám nic nebrání zaplnit rozličnými instancemi objektu celé pole:

```
int pocet = 300;
Entita[] naseEntity;

void setup() {
  size(640, 480);

  naseEntity = new Entita[pocet];

  for (int i = 0 ; i < naseEntity.length; i++) {
    naseEntity[i] = new Entita(i);
  }
}

void draw() {
  background(0);

  for (int i = 0 ; i < naseEntity.length; i++) {
    naseEntity[i].posunSeKeKurzoru();
    naseEntity[i].kresli();
  }
}
```

Výsledkem kódu bude celý had tří set entit různých odstínů šedi lační- cích po vašem kurzoru.



Rozeberme si jen letmo, k čemu vlastně došlo. Nejprve jsme definovali prázdné pole určené pro naše *entity*. Posléze jsme mu stanovili velikost a celé pole jsme naplnili novými objekty *entit* (odborněji *instancemi*). V poli se tedy nachází na každém místě jedna *entita*. Každou *entitu* poté ve funkci *draw()* k životu probouzíme prostřednictvím vnitřní funkce objektu *posunSeKeKurzoru()*. Všimněte si spuštění příkazu přes *tečkovou syntaxi*. Tím dochází k drobnému posunu každé okénko animace směrem ke kurzoru a následnému vykreslení tvaru na plochu funkcí *kresli()*.

6.11 Náhoda

V minulé kapitole jsme hovořili o výrazu *random()*, nyní se pokusíme poodhalit tajemství náhody v programování.

Proč má smysl v případě strojů vůbec hovořit o náhodě? Náhodu je z pohledu výpočetní techniky třeba důsledně vnímat jako vnější vliv. V rámci počítačové logiky náhoda v podstatě neexistuje (není přípustná). Každý jev má svoji příčinu a může mít svůj následek. Počítačové jazyky znají jen pojem takzvané pseudonáhodnosti.

Předložka pseudo- již nastiňuje určitou náhražku. Pseudonáhodnost je ve své podstatě strojová simulace náhody v čistě logickém prostředí. Náhodou pojmenováváme zpravidla jev, který nedokážeme pro jeho složitost předpovědět. Reálný svět vnímaný člověkem obsahuje takzvanou pravou náhodu, tedy to, co skutečně nelze vypozorovat.

Situace přenesení náhody do světa logického uvažování je velmi problematická proto, že člověk velmi těžko stroji popisuje fakt srozumitelnosti. Některá jednoduchá pravidla srozumitelná jsou, a jiná, z pohledu stroje obdobně jednoduchá, se lidskou optikou jeví již jako nesrozumitelná.

Pseudonáhodou člověk definuje takový výpočet, který samozřejmě vychází z logického výpočtu, člověk už ale nedokáže předpovědět jeho výsledek.

Konkrétní příklad, jak lze v *Processingu* vygenerovat pseudonáhodné číslo pomocí příkazu *random()*, je:

```
float nahodna = random(10);
println(nahoda);
```

Proměnná nazvaná *nahoda* nyní přijímá pseudonáhodnou hodnotu v rozmezí od nuly do deseti, již následně vytiskne do konzole. Druhý možný zápis je zadání dvou čísel, tedy rozsahu výsledné pseudonáhodné hodnoty.

```
float nahodna = random(-10,20);
println(nahoda);
```

Jak je zřejmě pochopitelné, náhoda se nyní bude pohybovat v rozmezí čísla od minus deseti do dvaceti. To, co se vám konkrétně objeví v konzoli, nedokážu předpovědět, jedná se totiž o vaše pseudonáhodné číslo.

Tisk do konzole je samozřejmě jeden z možných výstupů. Náhodné číslo můžeme zobrazit graficky. Můžeme například animovat pozici elipsy na ploše:

```
void setup(){
    size(640,480);
}


void draw(){
    background(255);
    fill(255);
    ellipse(width/2 + random(-5,5), height/2 + random(-5,5),
        20, 20);
}
```

Spustíme-li tento program, uvidíme kmitající elipsu ve středu kreslicího plátna. Kmitání probíhá v rozmezí od minus pěti do plus pěti pixelů od středu v obou osách. Pro každé jednotlivé okénko se obměňuje nová pseudonáhodná hodnota, a tím dochází k animaci.

Příkazy *width* a *height* označují šířku a výšku plátna.

6.11.1 Perlinův šum

Dalším možným strůjcem pseudonáhodně generované hodnoty je takzvaný *Perlinův šum* (*Perlin noise*). *Perlinův šum* je pojmenovaný po svém vynálezci Kenu Perlinovi a je velmi často využíván při generování přirozeně vypadajícího náhodného pohybu. Oproti příkazu *random()* se *Perlinův šum* *noise()* odvíjí plynule. Pro ilustraci si zde můžeme ukázat rozdíl mezi *Perlinovým šumem* a holým pseudonáhodným číslem.

```
void setup(){
    size(640,480);
}


void draw(){
    background(255);
    fill(255);
    ellipse(width/2 + noise( frameCount/30.0 ), height/2 +
        noise( frameCount/10.0 ), 20, 20);
}
```

Pohyb obrazce bude nyní probíhat plynule ve zdánlivě nepředpovídatelných směrech. Příkaz *noise()* předpokládá vstup, který určuje pozici v šumu. *Perlinův šum* si můžeme představit jako neměnný graf, kterým můžeme procházet tam i zpět v závislosti na zaslané hodnotě. Jelikož do příkazu *noise()* posíláme proměnnou hodnotu *frameCount*, procházíme variacemi *Perlinova šumu*, výstupem bude vždy číslo v rozmezí nula až jedna. Budeme-li potřebovat rozmezí jiné, můžeme šum vynásobit požadovaným rozsahem (jak je tomu v předchozím příkladu).

Vyzkoušejte použití náhody a šumu ke kresbě různých prvků v obraze, ovládejte rozsah pomocí myši.

Kapitola 7 Metody zobrazení

7.1 Renderovací mody

Doposud jsme pracovali pouze s jednou metodou zobrazení. Standardně je *Processing* nastaven do dvourozměrného modu vykreslování, který se nazývá *JAVA2D*. Tento mod bude nastaven vždy, nezvolíte-li jiný mód. Další varianty jsou následující[1]:

JAVA2D Základní renderovací engine, který jsme používali doposud. Tento render podporuje dvoudimenzionální kresbu a vyznačuje se dobrou kvalitou a precizností obrazu. Je obecně pomalejší než akcelerované rendery.

P2D (*Processing 2D*) – Akcelerovaný 2D render pro dvoudimenzionální kresbu, velmi výkonný pro operaci s jednotlivými pixely, postrádá ovšem přesnost *JAVA2D*.

P3D (*Processing 3D*) – Akcelerovaný 3D render, používaný zejména pro webové aplikace. Přesnost je opět menší než u standardního renderu, dominuje rychlost.

OPENGL Rychlý 3D render využívající systém *OpenGL* pro grafické karty. *OpenGL* se hodí k vykreslování velkého množství objektů v relativně dobré kvalitě. Kvalita a hlavně rychlost velmi závisí na hardwaru.[2]

---

[1] *Processing 2.0* přichází se značně pozměněným renderovacím enginem, varianty renderů *P2D* a *P3D* vzhledem ke zpětné kompatibilitě fungují, vnitřně je ovšem kompletně nahrazuje render *OPENGL*.

[2] Ve verzi 1.0 render vyžaduje *import* knihovny *(viz Knihovny, str. 127)*.

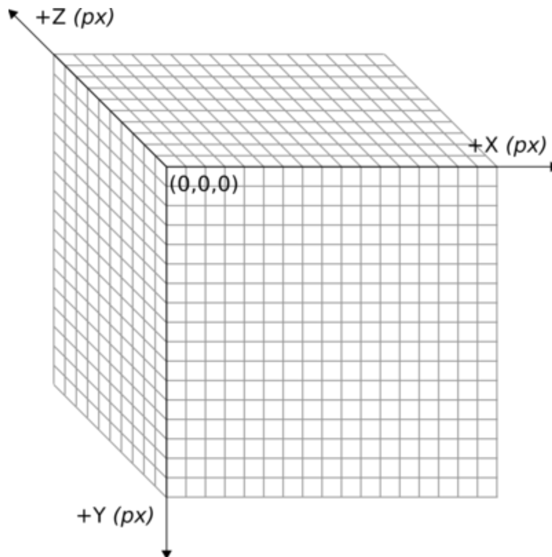
Použití jednotlivých vykreslovacích prostředí lze provést ve funkci *setup()* pomocí příkazu *size()*:

```
void setup(){

    //nastaveni renderu
    size(640, 480, P3D);

}

void draw(){

}
```

7.2 3D zobrazení

V případě renderů, které operují ve třetí dimenzi, v *Processingu* funguje řada příkazů podobně, jako jsme byli zvyklí v dvoudimenzionálním zobrazení. Největší změny se budou samozřejmě týkat zobrazení. V případě kresby primitivních obrazců v řadě případů přibude právě třetí rozměr.

Velmi ilustrativně je vidět rozdíl na příkazech *point()* a *line()*. U těchto příkazů jsme byli zvyklí zadávat dva parametry pro každou osu, tedy X, Y pro bod, popřípadě čtyři parametry X_1, X_2, Y_1, Y_2 pro linku. K těmto parametrům v případě trojdimenzionálních zobrazení přibývá pro každý bod logicky ještě jeden parametr pro osu Z.



Osa Z jakoby míří směrem k nám, negativní hodnoty se pak zobrazují ve větší vzdálenosti. Měrnou jednotkou je opět jeden pixel a hodnota nula pro všechny tři osy se nachází v levém horním rohu.

7.2.1 Kamera

Zobrazení je zde pro ilustraci, ve skutečnosti probíhá v perspektivě, tedy se sbíhajícími se úběžníky. V prostoru se lze pohybovat pomocí nastavení pomyslné kamery funkcí *camera()*.

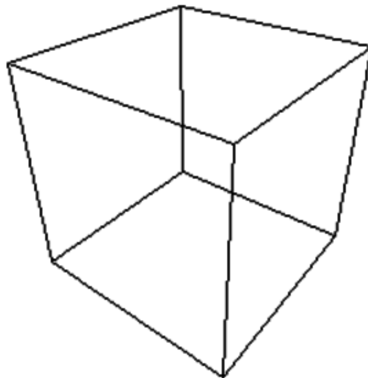
```
void setup() {

  // nastaveni 3D renderu
  size(300, 300, P3D);
  stroke(0);
  noFill();

    // prikaz pro nastaveni kamery
    camera(-170.0, -135.0, 320.0, 50.0, 50.0, 0.0,
    0.0, 1.0, 0.0);
```

```
}

//kresba
void draw() {

  background(255);

  box(120);
}
```

Příkaz *camera()* nám nastaví pohled v trojrozměrné scéně. Příkaz přijímá poměrně hodně parametrů, rozeberme si, co všechno tyto parametry znamenají.

První tři parametry značí pozici kamery (nebo oka, chcete-li) ve třech rozměrech. Další tři parametry určují pozici bodu v prostoru, na který se kamera soustředí. Zbylé tři parametry udávají pozici nahoru ve tvaru vektoru.

Chceme-li se orientovat v prostoru tak, jak jsme byli zvyklí doposud, poslední tři hodnoty nastavíme jako $0, 1, 0$.



Kamera byla v předchozím příkladu nastavena ve funkci *setup()*. Stejně tak lze nastavení obnovovat a měnit pro každé okénko. Shodně, jako jsme již animovali pohybující se objekt *(viz Animace, str. 65)*, můžeme animovat i kameru proměnou jednotlivých parametrů.

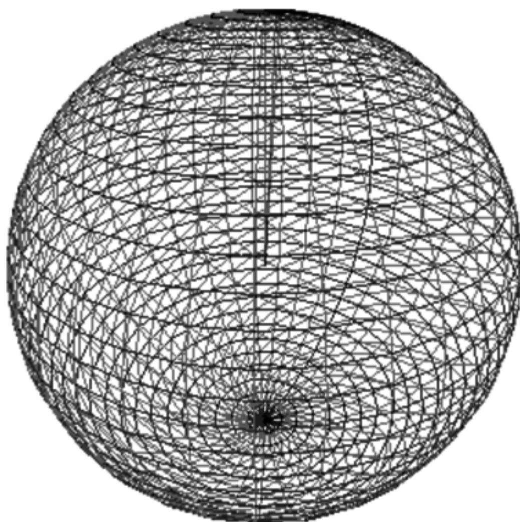
7.2.2 3D objekty

V kontextu 3D zobrazení jsme zatím hovořili o bodu, lince a v předchozím příkladu jsme si zobrazili krychli příkazem *box()*. Stejným příkazem *box()* lze zkonstruovat i kvádr. Ten lze vytvořit pomocí tří parametrů pro rozměry ve třech osách X, Y, Z.

Mezi další jednoduché tvary s možností zobrazení ve třech dimenzích patří dále koule – *sphere()* a úsečka ve třech dimenzích *line()*. Její definice vyžaduje jediný parametr, ten určuje poloměr zobrazované koule. Nastavení počtu segmentů lze dále definovat příkazem *sphereDetail()*. Pokud rozlišení nenastavíme, je standardně nastaveno na hodnotu 30, která klade nový segment každých 12 stupňů podle prostého výpočtu $360/X = 12°$. X je hodnota, kterou přijímá příkaz *sphereDetail()*.

Zkuste proměňovat hodnoty proměnné camera(), pozorujte jak se objekt proměňuje, ujistěte se, že rozumíte všem hodnotám zadávaným do funkce camera().

```
void setup() {

  // nastaveni 3D renderu
  size(300, 300, P3D);
  sphereDetail(36);
  stroke(0,100);
  noFill();

    // prikaz pro nastaveni kamery
    camera(-170.0, -135.0, 130.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0);
}

//kresba
void draw() {
  background(255);
  sphere(120);

}
```



Na veškeré objekty se vztahuje definice výplně a kontury, tedy funkce *fill()* a *stroke()* *(viz Výplň a obrys, str. 61)*.

Tím výčet trojdimenzionálních objektů pomalu končí. Nic nám nebrání vytvářet kompozice pomocí bodů, krychlí a čar v prostoru. Složitější tvary lze stejně jako u dvoudimenzionálního zobrazení zkonstruovat pomocí vlastní definice tvaru, za použití takzvaných *vertexů* – sadou příkazů *vertex()* uzavřených do bloku *beginShape()* a *endShape()*.

Zde si musíme uvědomit, že konstrukce objektu po nás bude vyžadovat veliké množství informací. Pro přirovnání si vezměme konstrukci koule, v případě vaší vlastní konstrukce byste museli každý z bodů definovat buď ručně, nebo lépe matematicky.

Metod, jak konstruovat 3D objekt, je mnoho a rozhodně by toto téma vystačilo na celého průvodce. Nyní nám postačí vědět, že je možné konstruovat i složitější tvary.

Další možností k tvorbě modelů je využít jiných programů (jako například otevřený software *Blender*) a následně pomocí rozšíření importovat externí modely přímo do *Processingu*[3] *(viz* Knihovny*, str. 127)*.

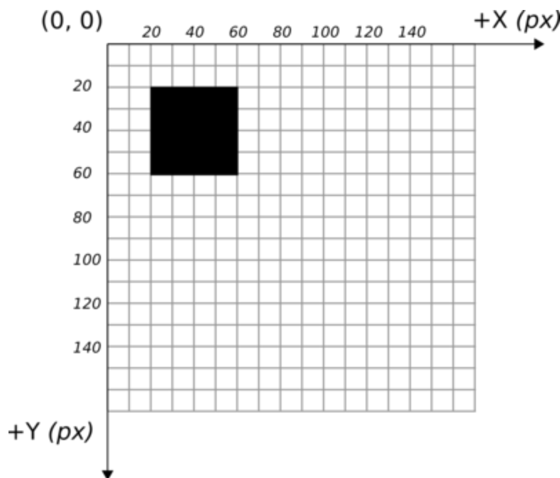
7.3 Transformace

Vraťme se nyní zpět k dvěma rozměrům. *Processing* disponuje příkazy, které usnadňují práci s prostorem. Souhrnně se takové funkce dají nazvat transformace.

Uvedeme si zde základní koncepci práce s objekty v prostoru a jejich zobrazením. Nové funkce se jmenují *translate()*, *rotate()* a *scale()*.

Představme si situaci, kdy potřebujeme pohybovat s jedním objektem. Jak jsme si ukázali již dříve *(viz* Animace*, str. 65)*, můžeme objektem pohybovat zadáváním jeho pozice.

Druhý způsob je v *Processingu* běžně využívaný a je o poznání dynamičtější, dovoluje nám totiž s objekty nejen pohybovat, ale měnit jejich rotaci nebo velikost.



[3] K tomu lze použít externí knihovnu *ObjLoader* v *Processingu 1.x* anebo vestavěnou proměnnou *PShape* v *Processingu 2.x*.

Představme si nyní náš objekt jako čtverec na pozici $X = 20, Y = 20$ o šířce a výšce dvaceti pixelů.

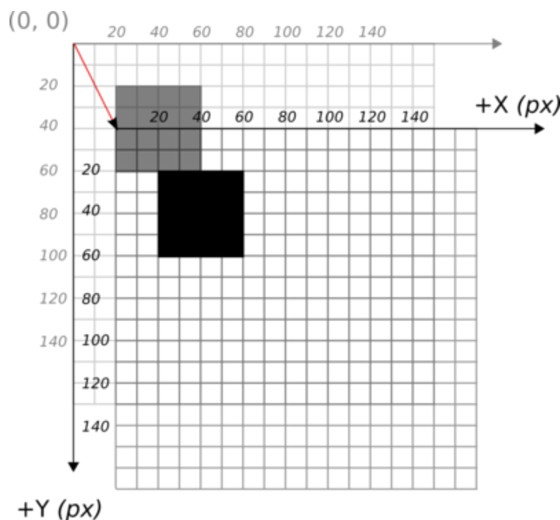
Jestliže budeme chtít přesunout čtverec o dvacet pixelů doprava a čtyřicet dolů, můžeme přičíst ke koordinátům tyto hodnoty:

```
rect(20 + 20, 20 + 40, 40, 40);
```

Nebo můžeme použít transformaci posunu. Ta nám posune jakoby celou čtvrtkou papíru, na který kreslíme. Posun je vyjádřen funkcí *translate()*.

```
translate(20, 40);
rect(20, 20, 40, 40);
```

Čtverec se tak zobrazí na stejném místě, jako bychom docílili ručním posunem. Způsob je ale odlišný, ve skutečnosti jsme posunuli celým kreslicím plátnem.



Veškeré tvary, které nyní vykreslíme, budou posunuty o náš posun, tedy nemusíme posouvat všemi jednotlivými tvary zvlášť. Další operací, kterou můžeme provést, je rotace. Rotace vždy probíhá kolem bodu $X = 0$, $Y = 0$. Pomocí transformace můžeme přesunout objekt na požadované

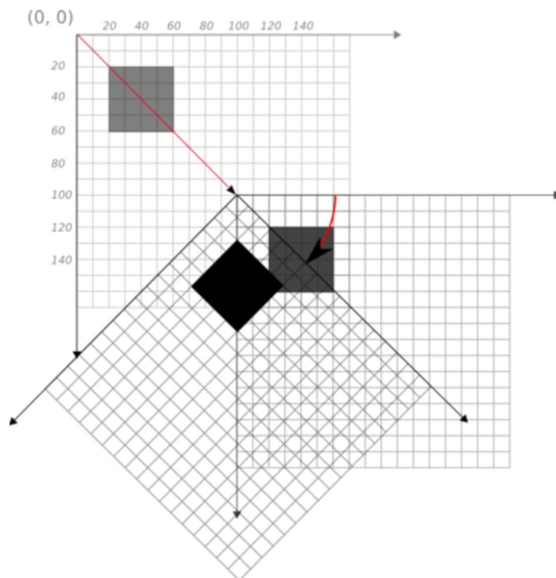
místo. K dalším operacím se nám budou hodit dva příkazy: *pushMatrix()*, funkce, která započíná transformace, a *popMatrix()*, která navrací plátno na původní místo. Při vícenásobných transformacích se nám může běžně stát, že budeme tyto bloky transformací vrstvit jednu do druhé. Exemplární rotaci si můžeme ukázat u podobného příkladu se čtvercem.

Proměňte hodnoty ve funkci translate(), můžete použít například pozici myši mouseX a mouseY a pozorovat změny v kresbě.

```
void setup() {
  size(500, 500);
  fill(0);
  rectMode(CENTER);
}

void draw() {

  background(255);

  pushMatrix();
  translate(150, 150);

  rotate( radians(45) );
  rect(20, 20, 100, 100);
  popMatrix();
}
```

Výsledkem bude čtverec otočený o 45°. Příkaz *rotate()* přijímá úhel v radiánech, chceme-li otáčet o 45°, musíme nejprve provést konverzi mezi úhly a radiány. Můžeme to provést snadno pomocí příkazu *radians()*. Opačnou konverzi lze provádět pomocí příkazu *degrees()*.

Naše rotace v diagramu bude vypadat asi takto:



První šipka znázorňuje posun, druhá šipka rotaci plátna. Osvojení si koncepce transformací je otázka zvyku, výhody toho, co se nyní jeví jako nepraktické, si uvědomíte možná později. Velikou výhodou transformací je například to, že si můžete definovat nový střed kresby, ke kterému se můžete vztahovat v relativních vzdálenostech.

Bez transformací se téměř vůbec neobejdete při kreslení ve třech rozměrech. Podobně jako u jiných příkazů i příkazy pro transformace zůstávají stejné. K funkci *translate()* přibude třetí rozměr a funkce *rotate()* se obohatí o rotace ve třech osách *rotateX()*, *rotateY()*, *rotateZ()*.

7.3.1 Použití transformací

Jako souhrn všeho, co jsme nastínili v úvodu do transformací, se nyní pokusme analyzovat jeden z příkladů ze stránek *openprocessing.org* vytvořený Keithem Petersem.

```
float[] x = new float[20];
float[] y = new float[20];
```

```
float segLength = 9;

void setup() {
  size(500, 500);
  smooth();
  strokeWeight(5);
  stroke(0, 100);
}

void draw() {
  background(225);
  dragSegment(0, mouseX, mouseY);
  for(int i=0; i<x.length-1; i++) {
    dragSegment(i+1, x[i], y[i]);
  }
}

void dragSegment(int i, float xin, float yin) {
  float dx = xin - x[i];
  float dy = yin - y[i];
  float angle = atan2(dy, dx);
  x[i] = xin - cos(angle) * segLength;
  y[i] = yin - sin(angle) * segLength;
  segment(x[i], y[i], angle);
}

void segment(float x, float y, float a) {
  pushMatrix();
  translate(x, y);
  rotate(a);
  line(0, 0, segLength, 0);
  popMatrix();
}
```

V úvodu se nám objevují dvě pole, která představují soupis souřadnic. Další proměnná je definice délky jednoho segmentu. Ve funkci *setup()* si můžeme povšimnout nového výrazu *smooth()*.

Funkce *smooth()* zajišťuje vyhlazení renderování, její protějšek je *noSmooth()*. Vykonává jemnější obrazový výstup na úkor rychlosti.

Další funkcí, kterou jsme doposud neuvedli, je *strokeWeight()*, ten definuje tloušťku čar a potažmo všech objektů. Šířka čáry je nastavena na 1.0. Funkce pracuje stejně jako výraz *stroke()*, nastavuje tloušťku čáry pro všechny objekty níže.

Nakonec tu máme dvě definované funkce. První funkce *dragSegment()* přijímá hodnotu i ze smyčky a provádí trigonometrické matematické operace. Funkce začíná načtením minulé hodnoty souřadnic, kterou porovná s hodnotami současnými a uloží výsledek. Zajímavá je funkce *atan2()*. Ta zjišťuje úhel v radiánech podle dvou hodnot v pořadí Y a X. Zde jako vstupní hodnoty pro výpočet úhlu zadáváme rozdíl mezi současnými souřadnicemi segmentu pozicí minulého segmentu.

Podle zjištěného úhlu se dále modifikuje současná pozice za pomoci zpětného převedení na souřadnice přes *cos()* a *sin()*. Trigonometrii zde příliš rozvádět nebudu. Sinusovou křivku lze zkonstruovat lineárním přírůstkem ve funkci *sin()*. Další goniometrické funkce jsou v *Processingu* *asin()* a *acos()*, šikovné pro práci s funkcemi jsou předdefinované hodnoty zlomku čísla π, *PI*, *HALFPI*, *QUARTERPI*.

A konečně přicházejí na řadu naše transformace. Ve funkci *segment* dochází ke kresbě jednotlivých segmentů hada. Funkce přijímá tři argumenty, dva pro pozici X, Y a jeden pro úhel α. Sled transformací pomocí posunu a rotace již známe.

Transformace mohou být náročné na prostorovou představivost, nezoufejte při prvních nezdarech, po nějaké době a cviku s dvěma rozměry se můžete pokusit o transformace ve třetím rozměru.

Kapitola 8 Práce s daty

Jak jsem v úvodu knihy *(viz Sketch, str. 39)* zmínil, adresář pro externí soubory se nazývá DATA. Tento adresář je umístěn přímo v adresáři projektu. Veškeré soubory, které do tohoto adresáře umístíme, budou programu dostupné.

V případě ukládání bude *Processing* zapisovat soubory do adresáře projektu. Všechny cesty k souborům budou tedy relativní.

Ukažme si nyní, jakým způsobem můžeme uložit a načíst vnější data do našeho programu.

8.1 Ukládání informací

Kresba do okna programu je jen jeden způsob výstupu programu. V určitém momentu budeme potřebovat zaznamenat obraz nebo jiná data pro další práci nebo archivaci. *Processing* umí ukládat data do rozmanitých formátů, při práci s obrazem se jedná o základní druhy bitmap a vektorového výstupu ve standardizovaném formátu *PDF*.

Při ukládání holých dat můžeme například využít prosté textové soubory, které se hodí při dalším zpracování pomocí *Processingu*. Velmi častý formát takového zápisu je takzvaný **CSV**, což je v podstatě textový soubor obsahující data oddělená čárkou. Jeho velikost závisí na objemu dat. V případě hieraticky strukturovaných dat je možné použít k uložení kupříkladu formát *XML*.

8.1.1 Ukládání obrazových dat

Doposud jsme pracovali s obrazem, který je dočasný. Po zavření okna *Processingu* obraz nenávratně zmizí. K tomu, abychom z *Processingu* dostali obrazový výstup, potřebujeme kresbu uložit do souboru.

Processing umí ukládat různé formáty souborů. Nejběžněji budeme potřebovat uložit obrazová data jako rastrový obrázek. Formát obrázku si můžeme zvolit.

Processing umí ukládat rastrová data do následujících formátů:

TIFF nekomprimovaný obraz s příponou **.tif*, **.tiff*

JPEG ztrátově komprimovaný obraz s příponou **.jpg*, **.jpeg*

TARGA bezztrátově komprimovaný obraz (RLE) s příponou **.tga*

PNG bezztrátově komprimovaný obraz (DEFLATE) s příponou **.png*

S některými formáty jste se již mohli setkat v minulosti. Jedná se o standardní a rozšířené formáty pro uchovávání obrazových dat.

Uložení obrázku ze současného plátna se provádí příkazem *save()*. Příkaz potřebuje znát cestu a koncovku souboru. Nenapíšeme-li plnou cestu, *Processing* data uloží přímo do adresáře naší sketche.

Nevíte-li, jakou koncovku souboru zvolit, doporučil bych vám vřele formát PNG. Takzvaný Portable Network Graphics je standard, který vyniká relativně malou velikostí a bezztrátovou kompresí dat. Vaše data tak budou uložena jedna ku jedné a můžete s nimi beze ztráty dále operovat.

Samotné ukládání se provádí následujícím způsobem:

```
void setup(){
    size(640,480);
}

void draw(){
    background(255);
    line(mouseX,mouseY,pmouseX,pmouseY);

}
```

```
void mousePressed(){
    save("obrazek.png");
}
```

Příkaz *save()* provede uložení obrazových dat okamžitě. Kdybychom vložili příkaz do kreslicí smyčky, *Processing* by se pokusil ukládat obrázek šedesátkrát za vteřinu, což by se mu z důvodu relativní náročnosti operace zřejmě nepodařilo a byli bychom svědky zpomalení rychlosti ve vykreslování.

V příkladu tedy budeme ukládat obrázek jen po kliknutí myší.

Nyní si představme, že z nějakého důvodu potřebujeme zachytit plnou sekvenci po sobě jdoucích obrázků. Problém příkazu *save()* je ten, že v případě pevně zadané cesty bude stále přepisovat obrazovými daty tentýž soubor. Abychom uložili vícero obrázků, potřebujeme proměňovat cestu, kam *Processing* obrázky ukládá. Můžeme k tomu využít počitadlo nebo zvláštní příkaz *Processingu saveFrame()*:

```
void setup(){
    size(640,480);
}

void draw(){
    // kresba probehne zde

    save("obrazek" + nf( frameCount , 5 ) + ".png");
}
```

Využití počitadla je možné, ale trochu zdlouhavé. Všimněte si zde pouze způsobu, jakým číslujeme názvy souborů. Hodnota *frameCount* se mění podle počtu vykreslených okének. Příkaz *nf()* je zde z kosmetických důvodů, přidává pouze daný počet nul před naše počitadlo. Celý tento zápis se dá ovšem elegantně zkrátit příkazem *saveFrame()*.

```
void setup(){
    size(640,480);
}

void draw(){
    // kresba probehne zde

    saveFrame("obrazek#####.png");
}
```

Příkaz *saveFrame()* vyžaduje stejně jako příkaz *save()* cestu k ukládanému souboru. Příkaz má ovšem navíc tu vlastnost, že vložíme-li do názvu souboru znaky křížku, *Processing* je rozpozná jako počet nul v počitadle okének.

Vygenerujte si jednoduchou animaci za použití metody saveFrame(), zkuste ukládat jen na stisknutí tlačítka.

8.1.2 Ukládání holých dat

Ukládání informací do obrazu je jedna z možností. Někdy se nám může hodit uložit informace v jiném formátu. Řekněme, že potřebujeme uložit pozice myši pro pozdější animaci. Jistě bychom mohli uložit tuto informaci do série obrázků, z hlediska programování by to ovšem nebylo prozřetelné.

Konkrétní údaje o pohybu kurzoru známe. Můžeme je tedy zaznamenat například do nám již známého pole souřadnic. Pro tento účel si vytvoříme program, který bude umět zaznamenávat údaje o pohybu kurzoru do textového souboru. Předtím, než se pustíme do práce, je dobré si uvědomit, že text neobsahuje číselné informace. Vzpomínáte na kapitolu o základních datatypech *(viz Základní datatypy, str. 46)*? S textem, tedy ani textovým souborem, nelze manipulovat jako s čísly.

Při ukládání dat do textové podoby tudíž musíme mít již na mysli, jak je posléze budeme číst, a proto zvolíme vhodný formát pro ukládání. Vhodným formátem pro uchovávání informací v textové podobě je **CSV**, Comma Separated Values. Jak z anglického názvu vyplývá, jedná se o hodnoty, které jsou odděleny čárkou. Pro naše potřeby budou stačit hodnoty dvě, pro osu X a osu Y našeho kurzoru.

Program si můžeme navrhnout následovně:

```
float osaX[];
float osaY[];
int pocetHodnot = 300;

void setup(){

    size(640,480);
    osaX = new float[pocetHodnot];
    osaY = new float[pocetHodnot];
}
```

```
void draw(){

    osaX[frameCount] = mouseX;
    osaY[frameCount] = mouseY;

    if(frameCount >= pocetHodnot-1){
        ulozDoSouboru();
        exit();
    }
}

void ulozDoSouboru(){

    String[] holyText = new String[pocetHodnot];
    for(int i = 0 ; i < pocetHodnot; i++){
        holyText[i] = osaX[i]+", "+osaY[i];
    }

    saveStrings("soubor.csv",holyText);
}
```

Program je hotový. Rozeberme si doposud nevídané věci. Pole by nám měla být již srozumitelná, v tomto případě pouze používáme datatyp *float*. Vytvořili jsme si dvě rozdílná pole pro dvě osy.

Ve funkci *draw()* nyní dochází k samotnému zápisu, a to pomocí již známé proměnné *frameCount*. Hodnota je pokaždé zapsána na novou pozici. Následuje podmínka, která dříve novým příkazem *exit()* ukončuje program. Podmínka je spuštěna pouze tehdy, máme-li již dostatečný počet zaznamenaných hodnot.

Podmínka navíc, dříve než ukončí celý program, spustí námi definovanou funkci pro uložení nashromážděných dat.

Tato funkce prostřednictvím smyčky převede číselné informace do textové podoby. Proměnné *osaX* a *osaY* funkce spojí dohromady pomocí znaménka plus spolu s čárkou a mezerou. Dále již následuje funkce *Processingu* *saveStrings()*, která zajišťuje zápis pole textu do souboru. Koncovku jsem zde zvolil **.csv*, ale je možné použít jakoukoli jinou, například **.txt*.

Podívejme se nyní do adresáře projektu (pomocí klávesy CTRL + k). Soubor s názvem *soubor.csv* by měl být plný zaznamenaných hodnot.

Vyzkoušejte si ukládání různých proměňujících se hodnot, ověřte si po uložení v textovém editoru jak výsledný text vypadá.

8.1.3 Tisk do PDF, ukládání vektorů

Ukládat obrazové informace lze v rozmanitých formátech. Představme si, že jsme v *Processingu* navrhli grafiku, kterou bychom rádi tiskli v dobré kvalitě. Standardními postupy, tj. uložit obrazová data jako rastrový obrázek, docílíme kvality výsledného souboru jen v rozměrech zadaných v příkazu *size()*. Sketch je primárně připravena pro zobrazení na obrazovce, a rozměry výsledného obrazu tedy většinou nepřesahuje rozměry naší obrazovky.

Nic nám sice nebrání nastavit velikost sketche na několikanásobně větší plochu, než jakou naše obrazovka fyzicky disponuje, důmyslnějším řešením ovšem je uložit obrazová data jako vektorová ve formátu *PDF*. *PDF* je standardní formát pro práci s grafikou a předtiskovou přípravou a kromě rastrových obrazových dat může obsahovat i data vektorová.

Největší výhodou vektorových dat je obecně škálovatelnost výstupních rozměrů. Jelikož jsou tvary vyjádřeny matematicky, celý obraz lze teoreticky tisknout v jakýchkoli rozměrech ve stejné kvalitě.

Exportovat vektorová data lze v rozdílných formátech, dvourozměrná data budeme nejčastěji exportovat právě ve formátu *PDF* nebo *SVG*. Trojrozměrná data lze exportovat nejjednodušeji pomocí formátu *DXF*.

Processing má funkci exportu do *PDF* již v sobě, pro jeho aktivaci ovšem musíme *Processingu* sdělit, že ji hodláme využít v naší sketchi.

Postup při použití vykreslování do *PDF* je následující:

```
import processing.pdf.*;

void setup() {
  size(400, 400, PDF, "jmenoSouboru.pdf");
}

void draw() {

  //kreslit budeme zde
  line(0, 0, width/2, height);

  //a zde ukoncime program
  println("hotovo");
  exit();
}
```

V úvodu se objevil neznámý příkaz. Takzvaný *import* značí načtení některých zvláštních funkcí rozšiřující funkce *Processingu*. Takové rozšíření se provádí pomocí takzvaných knihoven, o kterých se dočtete dále *(viz Knihovny, str. 127)*.

Pro náš tisk je nyní důležité, že za příkazem *import* se nachází právě cesta ke knihovně s *PDF* manipulacemi. Tuto cestu si nemusíme pamatovat, stejného výsledku docílíme i pomocí rozhraní *Processingu*. V liště *Sketch* → *Import Library* → *PDF Export*.

V kódu je dále důležitá nová definice funkce *size()*. Víme už, že tato funkce definuje rozměry sketche, v našem případě bude ovšem definovat rozměry výsledného *PDF* souboru. Za rozměry je dále nutné definovat vykreslovací prostředí *PDF* a jméno souboru, do kterého chceme zapisovat. Cesta k souboru je relativní, to znamená, že výsledný soubor se zobrazí přímo v adresáři naší sketche.

Předchozí příklad vykreslil první okénko programu přímo do souboru, aniž by cokoli zobrazil a ukončil sám sebe. V dalším příkladu si ukážeme, jak vykreslovat v průběhu programu jinou fázi animace. K podobné manipulaci potřebujeme spustit a ukončit vykreslování do souboru na vyžádání.

```
import processing.pdf.*;

boolean zaznam;

void setup() {
  size(400, 400);
}

void draw() {
  if (zaznam) {
    // vsimnete si ze #### bude nahrazeno cislem okenka
    beginRecord(PDF, "frame-####.pdf");
  }

  // kresleme zde
  background(255);
  line(mouseX, mouseY, width/2, height/2);

  if (zaznam) {
    endRecord();
    zaznam = false;
  }
}

// Use a keypress so thousands of files aren't created
void mousePressed() {
  zaznam = true;
}
```

Jak je na příkladu dobře vidět, zapisování do souboru spouštíme pomocí proměnné *zaznam*. Nahrávání vektoru do souboru začíná příkazem *beginRecord()* a končí příkazem *endRecord()*. Celou proceduru spouštíme prostřednictvím příkazu *mousePressed()*.

Pomocí předchozího způsobu uvidíme animaci vykreslovanou na plochu, kterou lze v daný moment uložit do jednotlivých souborů pojmenovaných podle čísla okénka, kdy kresba proběhla.

Podobným způsobem, jakým lze zapsat vektorový obraz, je i metoda postupné kresby na jedno plátno, tedy do jednoho souboru.

```
import processing.pdf.*;

void setup() {
  size(400, 400);
  beginRecord(PDF, "platno.pdf");
}

void draw() {
  // nebudeme vykreslovat pozadi, abychom neprepsali nasi
      predchozi kresbu

  // kresba pomoci mysi
  line(mouseX, mouseY, width/2, height/2);
}

// ukoncime kresbu a program
void keyPressed() {
  if (key == 'q') {
    endRecord();
    exit();
  }
}
```

Takovým postupem lze kreslit opakovaně do jednoho souboru. Kresbu a celý program ukončíme pomocí funkce *keyPressed()*, tedy klávesy Q.

Můžete vytvářet neomezeně složité tvary, zkuste jiné barvy výplně a obrysů, otevřete vždy výsledný soubor a ověřujete si výslednou podobu vektorové kresby. Zkuste kresbu načíst a upravit v jiném vektorovém editoru (například otevřený Inkscape).

8.2 Načítání informací

Za vnější informace lze v *Processingu* považovat vše, co není zdrojovým kódem *Processingu*. Takové informace můžeme souhrnně nazvat data. *Processing* umí nakládat s širokou škálou dat. Mezi ty nejčastěji používané bude patřit obrazová informace – ať už v podobě rastrového, nebo vektorového obrazu. Mezi další vektorová data patří 3D modely nebo například záznamy z databází ve formátu *XML*, *CSV*, naprosto holé textové soubory atp.

8.2.1 Načítání obrázků

Processing ukládá obrázky do vnitřní paměti. Využívá k tomu speciální datatyp nazvaný *PImage*. Pomocí tohoto datatypu si můžeme do našeho programu načíst externí obrazová data, se kterými můžeme dále pracovat. Načíst obrázek můžeme prostřednictvím příkazu *loadImage()*. Příkaz bude očekávat jediný parametr, cestu k obrázku.

Zde je namístě připomenout *(viz Sketch, str. 39)*, že budeme-li potřebovat externí data, je dobré si je umístit přímo do adresáře naší sketche do složky s názvem *DATA*. Veškeré cesty, které budeme v kódu psát, budou k našim souborům relativní, to znamená, že přemístěním externích dat do našeho adresáře můžeme místo:

```
PImage obrazek = loadImage("/obrazek/kdesi/v/hlubinach/
    adresaru/obrazek.jpg");
```

nyní napsat jednoduše:

```
PImage obrazek = loadImage("obrazek.jpg");
```

Načítání obrázků provádíme zpravidla ve funkci *setup()*. Tedy obrázek potřebujeme načíst pouze jednou za celý běh programu, a jelikož se jedná o poměrně náročnou operaci, je dobré ji v průběhu neopakovat příliš často.[1]

Správné načítání obrázku do paměti a následné vykreslení bude vypadat takto:

```
PImage obrazek;

void setup(){
  size(512,512);
  obrazek = loadImage("obrazek.jpg");
}

void draw(){
  image(obrazek,0,0);
}
```

Zde stojí za povšimnutí funkce *image()* – touto funkcí můžeme zobrazovat jednotlivé obrázky. Druhé dva parametry udávají pozici levého horního rohu obrázku v obou osách. Pro změnu, například vycentrování obrázku, buď můžeme odečíst polovinu ze šířky obrázku:

[1] Jako například každé okénko animace, což je velmi obvyklá začátečnická chyba.

```
image(obrazek, -obrazek.width/2, -obrazek.height/2);
```

anebo použít funkci *imageMode()*. Funkce dokáže proměnit chování příkazu *image()*, možné hodnoty jsou *CORNER*, *CORNERS* a *CENTER*. *CORNER* je standardní chování příkazu *image()*, místo odčítání poloviny šířky a výšky obrázku můžeme použít právě mod *CENTER*. Modus *CORNERS* se nám může hodit, potřebujeme-li definovat dva rohy místo šířky a výšky vykreslovaného obrázku.

Dejte si pozor na velikost obrázků, pracujte vždy raději s menšími.

8.2.2 Načítání textových souborů

Textové soubory jsou jednou z nejjednodušších cest, jak ukládat textové informace. To, že potřebujeme textový soubor, poznáme, jakmile budeme chtít pracovat s větším objemem textu, který se bude hromadit u definic proměnných.

Pokusme se nyní vrátit k předchozímu příkladu pro ukládání textových souborů *(viz Ukládání holých dat, str. 114)*. Zde jsem názorně ukázal, jak si uložit data generovaná *Processingem* pro další použití. Zkusme je tedy načíst. Pro připomínku, data vypadají následovně:

Na každém řádku jsou dvě hodnoty, které představují určitý záznam. Víme, že to je záznam pohybujícího se kurzoru, ale data mohou být opravdu cokoli. Co je pro nás důležité, je to, že jsme si uložili data v jednom tvaru, tzv. *CSV*. Vždy se jedná o dvě hodnoty s desetinnou čárkou na jednom řádku oddělené čárkou a mezerou.

K samotnému načtení textových informací do paměti programu tak, abychom s textem mohli dále pracovat, použijeme stejně jako při ukládání pole řetězcových hodnot. To, co zní odpudivě, je ve skutečnosti docela prosté:

```
String celyText[] = loadStrings("soubor.csv");
```

Příkazem *loadStrings()* načteme veškeré textové informace ze souboru do obsahu naší proměnné. Proměnná musí být pole, pole typu *String*.

Co příkaz *loadStrings()* přesně udělá? Naplní celé pole hodnotami, každý jeden řádek přiřadí jedné přihrádce. Pro prosté ověření si nyní můžeme vytisknout obsah naší proměnné do konzole.

```
println(celyText);

//////////// vysledek:
//[0] 0.0, 0.0
//[1] 0.0, 0.0
//[2] 0.0, 0.0
//[3] 0.0, 0.0
//[4] 0.0, 0.0
//[5] 0.0, 0.0
//[6] 0.0, 0.0
//[7] 39.0, 34.0
//[8] 46.0, 37.0
//[9] 50.0, 39.0
//... atd.
```

Tisk do konzole pomocí *println()* je nastaven tak, že jestliže tiskne pole, zobrazí také hodnoty o místech, na kterých jsou informace v poli uložené.

8.3 Operace s textem

V minulé kapitole jsme úspěšně načetli hodnoty do paměti programu. Budeme-li je nyní potřebovat k něčemu použít, musíme si rozmyslet, v jakém tvaru se nacházejí.

Navážeme-li na příklad z minulé kapitoly, naše proměnná má v sobě stále tytéž hodnoty a je ve tvaru pole *String[]*.

Kdyby se jednalo například o souvislý text – báseň, patrně bychom ji chtěli zachovat v textové podobě. Jelikož jsou to ale čísla – hodnoty, budou nám více platné pod datovým typem, se kterým budeme moci zacházet jako s číslem. Potřebujeme je tedy převést do čísel.

Převod z textových údajů do hodnot jiných typů se nazývá obecně *parsing*. Je to velmi častá procedura ve styku s daty třetí strany nebo i s daty, která jsme sami uložili.

8.3.1 Parsing, získávání hodnot z externích dat

K převodu textu do užitelných hodnot můžeme využít následující postup. Budeme potřebovat zkonstruovat smyčku *(viz Smyčka, str. 81)*, která projde veškerý obsah načteného textového souboru a řádek po řádku naplní

pole číselných hodnot odpovídajících hodnotám textovým. Celou tuto proceduru si můžeme uzavřít do jedné funkce.

Zopakujeme si načtení dat do pole. Program bude vypadat následovně:

```
String celyText[];
float hodnotyX[];
float hodnotyY[];

void setup(){

    nactiTextDoHodnot();

    println(hodnotyX);
}


void nactiTextDoHodnot(String nazevSouboru){

    celyText = loadStrings(nazevSouboru);

    hodnotyX = new float[celyText.length];
    hodnotyY = new float[celyText.length];

    for(int i = 0 ; i < celyText.length ; i++){
        String radek = celyText[i];
        String[] hodnoty = splitTokens(radek,", ");

        hodnotyX[i] = parseFloat(hodnoty[0]);
        hodnotyY[i] = parseFloat(hodnoty[1]);
    }

}

//////////// vysledek:
//[0] 0.0
//[1] 0.0
//[2] 0.0
//[3] 0.0
//[4] 0.0
//[5] 0.0
//[6] 0.0
//[7] 39.0
//[8] 46.0
//... atd.
```

Projděme si, co se v programu stalo. Celá procedura zde spočívá ve funkci nazvané *nactiTextDoHodnot()*. Jako argument do této funkce posíláme název souboru, který chceme načíst.

Jako první se spustí funkce, kterou již známe, *loadStrings()*, ta naplní naše textové pole textem. Nyní je zapotřebí nastavit velikost číselných polí, pro přehlednost jsem zde vytvořil dvě pro každou osu. Jsou nazvány *hodnotyX* a *hodnotyY*. Jejich budoucí velikost můžeme zjistit, bude stejná jako velikost textového pole. Přes příkaz *length* se můžeme dostat k délce jakéhokoli pole.

Nyní začneme s procházením textového pole, smyčka bude mít opět stejnou délku jako textové pole. Dočasná proměnná *radek* je zde spíše pro ilustraci, přiřazuje se k ní text pokaždé nového řádku.

Speciální textovou funkcí *splitTokens()* rozdělíme řádek podle rozdělujících znaků. Víme, že náš zápis v souboru odděluje jednotlivé hodnoty čárkou a mezerou. Funkce *splitTokens()* očekává první argument, text, který rozdělujeme, a jako druhý – výčet znaků, které jej rozdělují, výsledně vrací nové pole rozděleného textu těmito znaky.

Poslední nevídaná funkce je *parseFloat()*, ačkoli se nejedná přímo o funkci *Processingu*, ale přímo *Javy*, poslouží nám zde velmi dobře. Funkce *parseFloat()* očekává text a převádí jej do čísla s desetinnou čárkou. Kdybychom měli čísla celá, mohli bychom použít také funkci *parseInt()*, která dělá naprosto totéž s celými čísly.

Po načtení všech hodnot se program vrací do funkce *setup()*, odkud byl spuštěn. Zde si můžeme pro ověření vytisknout hodnoty z číselného pole *hodnotyX*.

Ujistěte se, že dobře rozumíte rozdílu mezi textovou informací a číselnou hodnotou v programu. Zkuste vytvořit jinou strukturu dat a poté ji sami přečíst pomocí funkce parseInt() nebo parseFloat().

8.4 Vizualizace hodnot

Vyjdeme-li z předchozího příkladu, máme již hodnoty zpracovány ve dvou číselných polích. Cesta k jejich následné vizualizaci je již prostá. Potřebujeme smyčku, která bude procházet celé pole a nakreslí podle hodnot patřičný tvar.

```
String celyText[];
float hodnotyX[];
```

```
float hodnotyY[];

void setup(){

      size(640,480);

    nactiTextDoHodnot("soubor.csv");

}

// kresba probehne zde
void draw(){
  background(0);
  stroke(255, 127, 0);

  for(int i = 1 ; i < hodnotyY.length ; i++){

   line(hodnotyX[i], hodnotyY[i],
        hodnotyX[i-1], hodnotyY[i-1]);

  }
}

void nactiTextDoHodnot(String nazevSouboru){

    celyText = loadStrings(nazevSouboru);

    hodnotyX = new float[celyText.length];
    hodnotyY = new float[celyText.length];

    for(int i = 0 ; i < celyText.length ; i++){
        String radek = celyText[i];
        String[] hodnoty = splitTokens(radek,", ");

        hodnotyX[i] = parseFloat(hodnoty[0]);
        hodnotyY[i] = parseFloat(hodnoty[1]);
    }

}
```

Výsledkem bude záznam pohybu myši, který může vypadat následovně:

Zkuste jiné hodnoty než pohyb myši, experimentujte s jinými způsoby zobrazení.

Za povšimnutí zde stojí konstrukce linky *line()*, která vždy spojuje koordináty z předchozího místa v našem poli s hodnotami současnými pomocí šikovného posunu o jednu pozici vzad *[i-1]*.

Kapitola 9 Rozšíření Processingu

9.1 Knihovny

9.1.1 Vestavěné knihovny

Ve *sketchbooku* se dále nacházejí[1] veškerá rozšíření pomocí takzvaných knihoven neboli libraries. Ty jsou umístěny v adresáři *sketchbooku* ve složce *libraries*. Knihovny jsou silným rozšířením celého jazyka a pokrývají mnoho možností nakládání s externími daty, zajišťují komunikaci se zařízeními, práci s videem, zvukem nebo s trojrozměrnými daty či vektorovým obrazem.

Knihovny jsou v podstatě moduly, které se zaměřují většinou pouze na jednu problematiku. Některé moduly umí například komunikovat se zapojenou kamerou nebo videem, jiné dovedou přenášet informace přes síť anebo komunikovat se zařízeními napojenými přes sériový port. Knihovna je sada příkazů, jež rozšiřuje samotný jazyk o nové možnosti.

Knihovnami se také dá dobře ilustrovat, k čemu uživatelé *Processingu* převážně programovací prostředí využívají. To sice nemusí být v žádném případě směrodatné pro vaši práci, může to být ovšem velmi inspirativní. Knihovny vám pomohou získat jistý přehled o kontextu užívání *Processingu*.

Knihovny se nejobecněji dělí na dvě základní skupiny. Jednu tvoří interní knihovny *Processingu*, které přichází již nainstalované se softwarem, a knihovny vnější, vytvořené komunitou uživatelů. Veškeré nainstalované knihovny, které má Processing k dispozici, lze nalézt v liště pod tlačítkem *Sketch* → *Import Library*.

[1] Od verze 1.0.

Výčet komunitních knihoven je již poměrně obsáhlý, pro ilustraci zde budeme hovořit jen o knihovnách vestavěných. Vestavěných knihoven je nepoměrně méně, jejich výčet se proměňuje relativně pomalu a jejich dokumentace je kvalitně zpracovaná:

Video
: Základní rozhraní mezi *Apple Quicktime* a *Processingem*. U platformy Linux tato knihovna dlouhodobě nefunguje kvůli závislosti na propietárním softwaru, musíme tedy využít alternativy – například implementaci nativní knihovny *GStreamer*.

Network
: Zajišťuje základní síťovou + internetovou komunikaci.

Serial
: Podpora pro komunikaci se sériovými porty, externí zařízení typu *(RS-232)*.

PDF Export
: Knihovna pro export do *PDF*.

OpenGL
: Technologie pro podporu *Java* implementace akcelerované grafiky prostřednictvím *JOGL*.

Minim
: Využívá *JavaSound API* k snadné obsluze zvukového výstupu.

DXF Export
: Knihovna pro exportování vektorových dat ve formátu *DXF* (Blender, Autocad, 3D).

Arduino
: Knihovna určená pro komunikaci s *Arduinem*.

JavaScript
: Nese metody pro komunikaci mezi *JavaScriptem* a processingovým appletem ve své webové podobě.

SVG Import
: Knihovna pro načítání vektorové grafiky ve formátu *SVG* (*Inkscape*, *Adobe Illustrator*).

XML Import
: Podpora načítání dat v *XML* formátu.

Neříkají-li vám tyto zkratky nic, nevadí, jedná se většinou o datové standardy a typy zařízení, se kterými můžete pomocí knihoven pracovat. Každá z těchto knihoven má svoji dobrou dokumentaci: na stránkách projektu nebo přímo v tzv. *Examples*, příkladech pro každou knihovnu, lze tímto způsobem zobrazit základní nápovědu.

Projděte si stránky processing.org a přečtěte si výčet knihoven, získáte lepší představu o možnostech rozšíření.

9.1.2 Komunitní knihovny

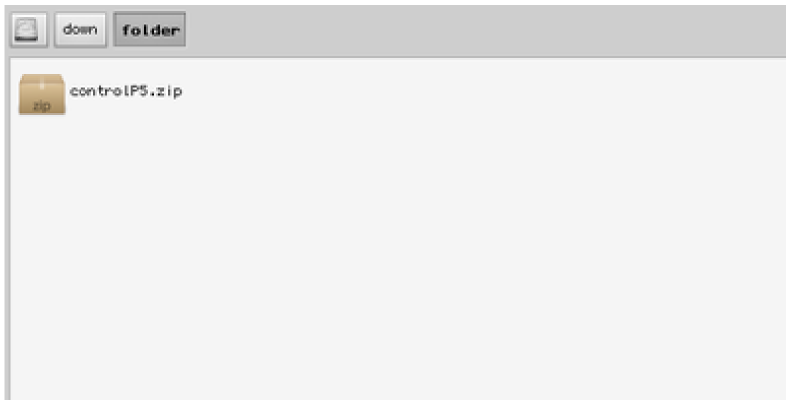
Instalace nových knihoven se zpravidla provádí umístěním adresáře s knihovnou do adresáře v našem *sketchbooku* nazvaném *libraries*. Pro kompletní výčet a popis všech komunitních knihoven zde není místo, je jich opravdu mnoho a další stále přibývají. Na aktuální stav se vždy můžete podívat na stránkách projektu *processing.org*.

9.1.3 Práce s knihovnou

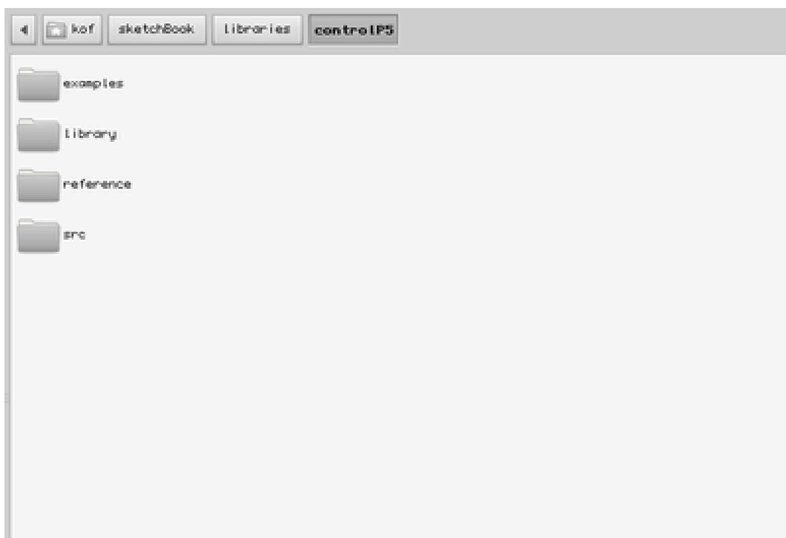
Jako poslední příklad si zde uvedeme práci s jednou externí knihovnou. S jednou z interních jsme již pracovali dříve *(viz Tisk do PDF, ukládání vektorů, str. 116)*.

Dobrým příkladem funkční a použitelné knihovny pro *Processing* je knihovna *ControlP5* od Andrease Schlegela. Tato knihovna nám dovoluje snadno vytvářet interaktivní prvky jako tlačítka a různé typy táhel (sliderů) i vstupních kolonek.

Knihovnu si nejprve musíme nainstalovat. Najdeme ji na webové adrese: *http://www.sojamo.de/libraries/controlP5*, anebo přímo na stránkách *processing.org*.



Soubor rozbalíme do adresáře *libraries* v našem *sketchbooku*, tj. adresáře, kam ukládáme naše sketche. Vnitřní struktura knihovny vypadá následovně:



Nejdůležitější složkou je v každé knihovně adresář *library*, zde se také nacházejí veškeré funkční prvky knihovny. Dále si můžeme všimnout dvou adresářů, *examples* a *reference*. Zde můžeme najít příklady použití i obsáhlou dokumentaci.

Máme-li spuštěný *Processing*, je nutné jej nyní restartovat.

Po spuštění je knihovna připravena k použití, přidat potřebné komponenty do kódu nyní můžeme buď ručně, nebo pohodlněji vybráním z nabídky: *Sketch* → *Import Library* → *controlP5*.

V kódu se nám zobrazí potřebný příkaz *import* s náležitými komponenty z knihovny. Nyní si můžeme přečíst nápovědu nebo si projít jednotlivé příklady skrze nabídky: *File* → *Examples* → *Contributed Libraries* → *controlP5* → *ControlP5controlEvent*. Zobrazí se nám sketch:

```
/**
 * ControlP5 ControlEvent
 * by andreas schlegel, 2009
 */
import controlP5.*;

ControlP5 controlP5;

int myColorRect = 200;
int myColorBackground = 100;

void setup() {
  size(400,400);
  frameRate(25);
  controlP5 = new ControlP5(this);
  controlP5.addNumberbox("n1",myColorRect,100,160,100,14).
      setId(1);
  controlP5.addNumberbox("n2",myColorBackground
      ,100,200,100,14).setId(2);
  controlP5.addTextfield("n3",100,240,100,20).setId(3);
}

void draw() {
  background(myColorBackground);
  fill(myColorRect);
  rect(0,0,width,100);
}
```

```
void controlEvent(ControlEvent theEvent) {
  println("got a control event from controller with id "+
      theEvent.controller().id());
  if(theEvent.controller().id() == 1) {
    myColorRect = (int)(theEvent.controller().value());
  }else if(theEvent.controller().id() == 2){
    myColorBackground = (int)(theEvent.controller().value());
  }else if(theEvent.controller().id() == 3){
    println(theEvent.controller().stringValue());
  }
}
```

Veškeré příkazy by nám měly být vcelku povědomé, samozřejmě až na ty, které ovládají funkce knihovny *ControlP5*.

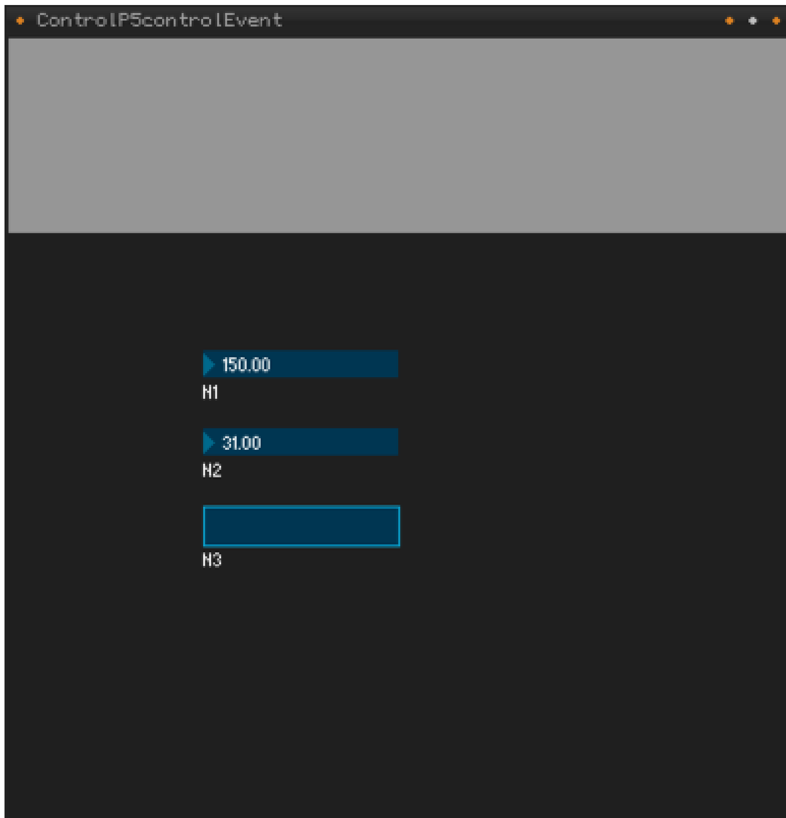
Všimněme si okamžitě objektu *ControlP5*, to je ústřední objekt při komunikaci s knihovnou. Objekt je vlastně třída, kterou jsme již rozebírali dříve *(viz Třída a objekt, str. 89)*. Vytvořením instance objektu z knihovny příkazem *new* provážeme funkce knihovny s naším programem. Slovo *this* referuje k našemu programu a říká knihovně, s čím bude své funkce propojovat. V podstatě ukazuje na nadřazenou (tzv. *super*) třídu samotné processingové aplikace (typu PApplet).

Nyní již můžeme pomocí tečkové syntaxe vytvářet objekty z knihovny. V knihovně *ControlP5* je jich opravdu mnoho, zde si ukážeme pouze dva typy. Příkazem *controlP5.addNumberbox* přidáváme objekt s hodnotou v číselném formátu.

Objekt požaduje několik parametrů pro vytvoření: vlastní jméno, jméno proměnné, kterou bude ovládat, na konci definice proměnné také můžeme vidět speciální funkci *setId(1)*, která nám bude identifikovat aktivitu daného prvku.

Tu můžeme detekovat pomocí nové funkce z knihovny nazvané *void controlEvent(ControlEvent theEvent)*. Ta zachytává veškeré dění s prvky a posílá identifikační hodnotu, kterou posléze můžeme přečíst a napsat podle ní podmínku.

O kresbu prvků do smyčky *draw()* se knihovna již postará sama.



Výsledkem bude jednoduché *GUI*, grafické uživatelské rozhraní, pomocí kterého můžeme ovládat hodnoty v našem programu.

Vyzkoušejte další příklady z knihovny a zkuste i knihovny další. Všech rozšíření je nyní přes 100 kusů, možností aplikace a kombinací je neomezené.

9.2 Poznávejte, experimentujte!

Toto je konec úvodu do *Processingu*. Tento skromný průvodce vám chtěl pootevřít dveře do světa generované grafiky a programování jako takového. Další cesta bude již na vás.

Na závěr bych vám chtěl popřát hodně trpělivosti a odhodlání, obojího budete potřebovat nemálo. Výsledek své snahy ovšem můžete několikanásobně zúročit v další práci.

Hlavně se nebojte experimentovat, pocity frustrace nad nefungujícím kódem určitě vyváží uspokojení z vlastnoručně vytvořeného fungujícího programu. Odložte veškeré průvodce a zkoušejte stále nové postupy, i po několika letech zjistíte, že stojíte stále na začátku a můžete se učit novému. Čtěte kódy ostatních lidí a vyměňujte si programy, jejich zdroje – vědomosti dále sdílejte s druhými lidmi.

Učte se vždy novému, nic není krásnějšího než na věci přicházet. A když věcem rozumíte, nevlastněte je, předávejte vědomosti trpělivě dále.

Aktualizovanou podobu tohoto průvodce můžete najít v elektronické verzi repozitáře *GIT* na serveru *GitHub* na adrese:
https://github.com/K0F/processing_1

Slovník

acos() Inverzní funkce *cos()*, arkuskosinus. Funkce převádí hodnoty rozmezí od -1,0 do 1,0 na hodnoty v rozmezí od 0 do π. … 110

asin() Inverzní funkce *sin()*, arkussinus. Funkce převádí hodnoty rozmezí od 1,0 do -1,0 na hodnoty v rozmezí od 0 do π *(3,1415927)*. … 110

atan2() Vypočítává úhel z jednoho bodu do bodu druhého. K výpočtu můžeme použít zápis *atan2*($Y_2 - Y_1, X_2 - X_1$). Výsledný úhel je udáván v radiánech, tedy v rozmezí od -π do π. *Atan2()* je funkce, která se hojně využívá při určení směru od objektu ke kurzoru nebo jinému bodu. Pozor při zadávání parametrů. Všimněte si obráceného pořadí parametrů. Oproti zvyklosti jsou zadávány v pořadí Y, X. … 109

background() Funkce nastavuje barvu pozadí na plátně. Standardní barva je světle šedá. Spuštěním této funkce s definicí barvy v kulatých závorkách bude vyplněna celá plocha jednolitou barvou. … 65

beginRecord() Otevírá nový soubor pro zápis do souboru ve formátu *PDF* nebo *DXF*. Kresba tak probíhá souběžně na plátno i do souboru. Pro ukončení kresby musíme spustit funkci *endRecord()*. … 118, 138

beginShape() Pomocí funkcí *beginShape()* a *endShape()* můžeme definovat složitější tvary. Funkce *beginShape()* otevírá definici tvaru. Následuje funkce *vertex()*, která určuje jednotlivé body tvaru. Tvar je nakonec uzavřen příkazem *endShape()*. Kresba tvarů může probíhat ve dvou i třech rozměrech. V závislosti na renderu můžeme definovat *vertex()* dvěma nebo třemi hodnotami. Doplňujícím parametrem v kulatých závorkách funkce *beginShape()* dále může být

uveden způsob, jakým budou body spojovány. Přípustné varianty jsou: POINTS, LINES, TRIANGLES, TRIANGLEFAN, TRIANGLESTRIP, QUADS a QUADSTRIP. … 104, 138, 148

boolean Datatyp, který může mít jen dva stavy: *true* a *false*. … 12, 69, 72, 85

box() Krychle nebo kvádr. Krychli můžeme definovat jedním parametrem, který určuje délku jedné hrany. Kvádr je možno definovat stejným příkazem zadáním délky tří hran: X, Y, Z. … 103

Built with Processing Doslova znamená: „Postaveno s *Processingem*“. Jedná se o zvláštní komunitní frázi, která se objevuje u projektů využívajících *Processing*. Fráze vyjadřuje určitý vděk všem participantům a tvůrcům *Processingu* za tvorbu tohoto nástroje. … 33

camera() Nastavuje pozici kamery skrze parametry: pozice oka, střed zobrazení a osa označující směr nahoru. Funkce zavolaná bez jakýchkoli parametrů nastaví kameru na výchozí pozici. Výchozí pozice lze vyjádřit následovně: *camera(width/2.0, height/2.0, (height/2.0) / tan(PI*60.0 / 360.0), width/2.0, height/2.0, 0, 0, 1, 0)*. … 101, 102

class Takzvaná třída. Slovo označující deklaraci nové třídy, definici objektu. Třída je souhrn funkcí a proměnných a představuje šablonu pro objekt. Jména tříd jsou většinou označována počátečním velkým písmenem, instance objektů (produkt ze šablony) pak písmenem malým. Třída je základní stavební kámen objektově orientovaného programování. … 90, 91

CODED Používá se v porovnání se systémovou proměnnou *key* k zjištění, zdali poslední stisknutá klávesa byla speciálním znakem. Klávesa se posléze detekuje proměnnou *keyCode*. … 78

color() Vytváří barvu ve speciálním datatypu *color*. Vstupní hodnoty jsou interpretovány buď jako hodnoty RGB (červená, zelená, modrá), nebo HSB (odstín, sytost, jas). Jednotlivé mody vytváření barev lze přepínat funkcí *colorMode()*. Ve výchozím nastavení mají barvy rozmezí 256 odstínů šedi, tedy konkrétně od 0 do 255. … 48

`colorMode()` Funkce měnící způsob, jakým *Processing* interpretuje barvu. Možné argumenty jsou RGB a HSB. Nepovinný je druhý argument, určující rozsah hodnot. Výchozí nastavení této hodnoty je 255. Příkaz ovlivňuje nastavení barev pro kresbu, tj. například pro funkce *fill()* a *stroke()*. … 48, 136

`cos()` Počítá hodnotu kosinu z úhlu. Tato funkce očekává parametr vyjadřující úhel, parametr může být v rozsahu od 0 do $\pi * 2$. Hodnoty se pak pohybují mezi -1 do 1. … 110, 135

`CSV` Neboli „Comma Separated Values“. Jedná se o uznávaný standard uložených dat. S tímto standardem zachází mnoho programů. Jde v podstatě o textový soubor s informacemi oddělenými specifickým znakem. Nejčastěji je tímto specifickým znakem čárka. Rozlišovacím znakem může ovšem být například středník nebo teoreticky jakýkoli jiný znak. … 111, 114, 121

`DATA` Adresář uvnitř **sketche**, obsahuje další externí soubory (obrázky, textové soubory atd.). … 39

`degrees()` Obrácená funkce *radians()*. Převádí stupně v radiánech do úhlových stupňů. … 107

`draw()` Základní kreslicí smyčka programu. Kód uvozený ve složených závorkách za touto funkcí bude spouštěn několikrát ve vteřině. Využívá se k animaci a interakci uživatele s programem. … 55, 65, 66, 78, 84, 88, 94, 115, 132, 138

`ellipse()` Kreslí elipsu nebo kruh. Elipsa se stejnou šířkou i výškou představuje kruh. První dva parametry udávají pozici, třetí uvádí šířku a čtvrtý výšku elipsy. Nastavení způsobu vytváření elips lze provést pomocí funkce *ellipseMode()*. … 61

`ellipseMode()` Ovlivňuje způsob, jakým je vykreslována elipsa. Možné mody jsou *RADIUS*, *CENTER*, *CORNER*, *CORNERS*. Výchozí modem je *CENTER*. První dva parametry nastavují pozici elipsy, druhé dva pak šířku a výšku elipsy. V případě modu *RADIUS* druhé dva

parametry určují poloměry elipsy. Modus *CORNER* definuje pozici levého horního bodu, následně šířku a výšku. Poslední modus *CORNERS* definuje dva body, mezi které bude elipsa vepsána. … 137

else Doplňuje podmínku *if* – „jestliže“ o „jestliže ne“. Blok kódu za tímto příkazem je spuštěn, jestliže není předchozí *if()* podmínka splněna. Struktura kódu tím dovoluje spustit dva různé bloky kódu v případě naplnění podmínky anebo jejího nenaplnění. … 69, 73, 139

endRecord() Zastavuje nahrávání do souboru spuštěné pomocí funkce *beginRecord()*, dále uzavírá zápis do souboru *(EOF)*. … 118, 135

endShape() Párová funkce s *beginShape()*, uzavírá kreslení tvaru. Funkce může být doplněna o výraz *CLOSE* (psáno dovnitř kulatých závorek), který uzavře tvar a pokusí se vytvořit jeho výplň. … 104, 135, 148

exit() Ukončí program. Program bez funkce *draw()* bude ukončen automaticky poslední řádkou kódu. Programy obsahující funkci *draw()* běží, dokud není program zvenčí ukončen nebo není zavolán příkaz *exit()*. … 115

false Nepravda neboli 0. … 69, 70, 72, 85, 136

fill() Funkce nastavující barvu výplně tvarů. Například spustíme-li *fill(204, 102, 0)*, veškeré objekty, kreslené níže v kódu, budou mít oranžovou barvu výplně. … 61, 104, 136, 144

float Datatyp schopný udržet proměnnou ve tvaru čísla s desetinnou čárkou. … 87, 115, 147

flow Neboli plynutí, tok, je zvláštním stavem mysli popisovaným programátory. Jedná se o stav, kdy je člověk plně zanořen do práce. Jakékoli vyrušení z tohoto stavu si vyžaduje dlouhou koncentraci k opětovnému nastolení bdělosti. Podle zkušených programátorů vyvolání takového stavu trvá několik desítek minut práce s kódem. … 43

for Definuje smyčku, tedy opakující se sekvenci příkazů. Struktura *for* se zadává pomocí tří parametrů: inicializace pozice (kde smyčka začíná), ověření (konečná podmínka, která smyčku ukončuje) a přírůstku (jeden krok rozdílu při opakování smyčky). Tyto části jsou vždy odděleny středníkem. Smyčka pokračuje, dokud je naplněn výsledek prostřední podmínky, tedy *true*. … 81

frameCount Proměnná uchovávající údaj o počtu vykreslených okének od startu programu. … 66, 97, 113, 115, 141

frameRate() Nastavuje počet okének, které mají být vykresleny za jednu vteřinu. Výchozí nastavení je šedesát okének za vteřinu. Pokud je údaj vyšší, než *Processing* může v danou chvíli vykreslovat, počet okének za vteřinu se sníží na maximální možnou hodnotu. … 66

GNU / GPL Jedna z licencí otevřeného softwaru zaručující volně dostupný zdrojový kód. Základním konceptem licence je podmínečná volná dostupnost zdrojového kódu takto licencovaného programu. A dále nutnost použití stejné licence pro veškeré projekty takový kód využívající. Licencí zajišťujících otevřenost softwaru je nepočitatelně, mezi oblíbené například patří i méně restriktivní *BSD / MIT* licence, pod kterou je vyvíjen například i programovací jazyk – prostředí *Processing*. … 12, 37, 141

GNU / Linux Zkratka „Gnu Is not Unix". *GNU* je projekt původně založený Richardem Stallmanem, jedná se o operační systém a rodinu programů s otevřeným zdrojovým kódem. … 37, 143

HALFPI Konstanta určující polovinu čísla π, 3,14159265 . … 110

if Příkaz značící podmínku. Vyžaduje následné uvedení dotazu v kulatých závorkách, pokud je podmínka naplněna, spouští následující kód uzavřený ve složených závorkách. Komplementárním příkazem je *else*, který může spustit další blok kódu, není-li naplněna předchozí podmínka.. … 69, 73, 138, 139

if() Viz *if*. … 138

image() Zobrazuje obrázek v datatypu *PImage*. K jejich načtení používáme funkci *loadImage()*. Barvu a průhlednost obrázku lze dále definovat pomocí funkce *tint()*. Funkce *image()* jako první přijímá parametr s názvem proměnné obrázku, který bude zobrazovat. Zbylé dva parametry jsou pozice vykreslení obrázku v ose X a Y. Ve výchozím modu souřadnice určují levý horní roh obrázku. Třetí a čtvrtý parametr je nepovinný, může dále nastavit šířku a výšku zobrazovaného obrázku. Mody pro alternativní zobrazování lze přepínat pomocí funkce *imageMode()*. … 120, 121

imageMode() Ovlivňuje způsob zobrazování obrázků. Dovoluje použít konstanty *CORNER*, *CORNERS* a *CENTER*. Ve výchozím nastavení *CORNER* bude obrázek umístěn levým horním rohem na definované souřadnice X a Y. Modus *CORNERS* mění chování druhých dvou parametrů, které místo relativní šířky a výšky nastavují absolutní hodnoty v osách X a Y. Poslední modus *CENTER* umístí obrázek na střed definovaných souřadnic, zbylé dva parametry opět udávají relativní šířku a výšku obrázku. … 121, 139

import Výraz načítá externí součásti do *Processingu*. Využívá se zejména při práci s knihovnami. … 99, 117, 131

indenting Neboli zarovnávání kódu do úhledných paragrafů. Zarovnávání slouží k rychlejší orientaci programátora ve struktuře kódu. … 41

interakce *(lat. interactio od inter-agere, jednat mezi sebou)* Znamená vzájemné působení, jednání, ovlivňování všude tam, kde se klade důraz na vzájemnost a oboustrannou aktivitu na rozdíl od jednostranného, například kauzálního působení. … 74

key Systémová proměnná *Processingu* obsahující hodnotu naposledy stisknuté klávesy, velmi často se používá v kombinaci s další systémovou funkcí *keyPressed()* k detekci stisknuté klávesy. … 78, 136, 140

keyCode Systémová proměnná *Processingu*, podobně jako *key* zančí, která z netextových kláves byla stisknuta jako poslední (*ENTER*,

BACKSPACE, LEFT atp.). Velmi často se používá v kombinaci s další systémovou funkcí *keyPressed()* k detekci naposledy stisknuté klávesy. … 78, 136

keyPressed() Funkce je spuštěna pokaždé, je-li stisknuta libovolná klávesa na klávesnici. … 78, 119, 140

keyReleased() Funkce spuštěná tehdy, uvolníme-li libovolnou stisknutou klávesu. … 78

line() Funkce pro kresbu úsečky. Ve 2D funkce přijímá čtyři parametry: X_1, Y_1,X_2, Y_2. Ve 3D zobrazení funkce přijímá šest parametrů pro určení koordinátů každého bodu ve třech dimenzích, tedy: X_1, Y_1,X_2, Y_2, X_3, Y_3. Barvu čáry je možné změnit předchozím nastavením barvy obrysů funkcí *stroke()*. … 61, 100, 103, 126

loadImage() Načítá obrázek do proměnné ve formátu *PImage*. V kulatých závorkách se definuje cesta a název obrázku (včetně přípony) ve formátu *String*. Obrázky by měly být umístěny v adresáři *DATA*. *Processing* podporuje grafické formáty *PNG, GIF, JPEG a TGA*. Průhlednost obrázků je podporována pouze ve formátech *GIF, TGA* nebo *PNG*. … 119, 139, 143

loadStrings() Funkce načítající externí textová data do formátu pole řetězců: *String[]*. V takto vytvořeném poli každý slot obsahuje jeden řádek textu. … 121, 124

millis() Funkce udávající počet uplynulých milisekund od startu programu. Na rozdíl od proměnné *frameCount* tato funkce udává precizní časový údaj od startu programu. Výsledná hodnota nezávisí na počtu vykreslených okének za vteřinu. Výsledný údaj tak lze využít například pro přesné časování animace, a to bez ohledu na počet okének za vteřinu. V případě ukládání animací do videa je pro časování animace naopak téměř nepoužitelný, především kvůli zpomalení animace při ukládání okének. … 67

MIT licence Je licencí kompatibilní s licencí *GNU / GPL*, jedná se o zvláštní licenci modifikovanou na půdě Univerzity MIT – *Massachusetts Institute of Technology* k možnému využití svobodného softwaru i v komerčním prostředí. … 33

mouseButton Pomocí této funkce *Processing* automaticky registruje, které tlačítko myši bylo stisknuto. Jestliže bylo stisknuto levé tlačítko, tato systémová proměnná bude mít hodnotu *LEFT*. V případě pravého tlačítka bude mít proměnná hodnotu *RIGHT*. U prostředního tlačítka pak hodnotu *CENTER*. … 78, 142

mousePressed() Funkce je spuštěna pokaždé je-li stisknuto tlačítko myši. K následné detekci tlačítka můžeme použít proměnnou *mouseButton*. … 75, 77, 118

mouseX Proměnná, která udává pozici kurzoru na plátně processingového okna v ose X. … 75

mouseY Proměnná, jež udává pozici kurzoru na plátně processingového okna v ose Y. … 75

new Výraz uvozující vytvoření instance z objektu. … 80, 91, 132

nf() Nástroj na formátování čísel do patřičného tvaru v textu. Přijímá parametr celého čísla a předřazuje před něj počet nul z druhého parametru. Tato funkce se velmi často používá k zarovnávání řad čísel. … 113

noFill() Zakazuje výplň všech následujících objektů. Jestliže je příkaz použit dohromady s příkazem *noStroke()*, *Processing* teoreticky nebude nic vykreslovat na plátno. … 61

noise() Udává hodnotu *Perlinova šumu* na specifické pozici. *Perlinův šum* je generátor pseudonáhodných čísel vytvářející plynule přecházející pseudonáhodnou hodnotu. Algoritmus byl vyvinut Kenem Perlinem v osmdesátých letech 20. století a je velmi často používán právě v grafických aplikacích. Používá se například při tvorbě procedurálních textur, pohybu objektů nebo proměně geometrie v čase. Rozsah

Perlinova šumu se pohybuje vždy v rozmezí hodnot 0,0 a 1,0. Funkce *noise()* přijímá jednu, dvě nebo tři hodnoty. Výsledná hodnota se pohybuje v jedno- až trojrozměrném poli. Šum může být například snadno animován přírůstkem v parametru. Hodnoty mají tu vlastnost, že plynule přecházejí jedna v druhou. … 96, 97

noSmooth() Opak funkce *smooth()*. Vypíná vyhlazování hran. … 109

noStroke() Funkce zabraňující vykreslování kontur, jedná se o protějšek funkce *stroke()*. … 62, 142

noTint() Resetuje barvu a průhlednost vykreslených obrázků. … 147

otevřený software Také nazývaný svobodný. V zásadě software s veřejně dostupným zdrojovým kódem (viz *GNU / Linux*). … 33

parseFloat() Funkce pro čtení čísel ve formátu (*String*) a získávání číselné hodnoty s desetinnou čárkou ve tvaru *float*. … 124

parseInt() Funkce pro čtení čísel ve formátu (*String*) a získávání celých čísel ve tvaru *int*. … 124

PI Konstanta určující číslo π, 3,14159265. … 110

PImage Proměnná schopná v sobě udržet obrazová data a obrázky obecně. Pro načtení dat do *Processingu* použijte *loadImage()*. … 119, 139, 141

point() Kreslí bod v dvourozměrném zobrazení. Funkce vyžaduje parametry X, Y. Při trojrozměrném zobrazení bodu funkce vyžaduje tři parametry pro X, Y a Z. Barvu bodu můžeme kontrolovat pomocí příkazu *stroke()*. … 61, 100

popMatrix() Funkce načítá z uloženého koordinačního systému a uzavírá blok transformací. Vyžaduje uložení koordinátů předchozím vyvoláním příkazu *pushMatrix()*. … 106, 144

print() Funkce pro tisk do konzole. Vstupní hodnotou může být jakákoli proměnná, holý text, nebo kombinace obojího. Slouží jako základní nástroj k sledování proměn v programu. Využívá se zejména při vylaďování programu a při kontrole výsledků. … 49, 144

println() Funkce pro tisk do konzole. Funguje stejně jako příkaz *print()*, s tím rozdílem, že každou tisknutou hodnotu zakončuje novým řádkem. … 79, 122

pushMatrix() Ukládá nastavení koordinačního systému (neboli matrix). Funkce dovoluje vrstvení jednotlivých transformačních operací, jako je *rotate()*, *translate()* nebo *scale()*. Funkce vyžaduje následné ukončení transformace příkazem *popMatrix()*. … 106, 143

quad() Funkce kreslící čtyřúhelník. Nejvíce je použitelná pro lichoběžníky. Funkce vyžaduje parametry X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4 a Y_4. Barvu čtyřúhelníku můžeme kontrolovat pomocí příkazu *stroke()*. Barvu výplně definujeme funkcí *fill()*. … 61

QUARTERPI Konstanta určující čtvrtinu čísla π, 3,14159265 . … 110

radians() Převádí úhlové stupně do stupňů v radiánech. Tedy například hodnotu $0 - 360°$ převede na 0 až $2 * \pi$. … 107, 137

random() Funkce, která vrací pseudonáhodné číslo. Není-li zadána žádná hodnota do kulatých závorek, výsledek funkce se pohybuje v rozmezí mezi $0 - 1$. Je-li zadána jedna vstupní hodnota, číslo se pohybuje mezi nulou a zadanou hodnotou. Jsou-li zadány dvě hodnoty oddělené čárkou, výsledek se bude pohybovat v rozmezí těchto dvou hodnot. … 93, 95, 96

rect() Funkce vykresluje obdélník, přičemž přijímá čtyři parametry: počátek X, počátek Y, šířku a výšku. Možné způsoby vykreslování obdélníka se dají nastavit funkcí *rectMode()*. … 60, 61, 65, 84, 144

rectMode() Funkce ovládá modus vykreslování obdélníka. Možné vstupní parametry jsou pouze čtyři: *CORNER* je základní chování funkce

rect(). Tedy prvními dvěma parametry při kresbě umístíme nejdříve první roh v ose X a Y. Poté definujeme relativní šířku a výšku obdélníka. Druhou možností je *CORNERS*. Tento modus definuje dva rohy zvlášť, tedy v absolutních číslech na ploše. Dalším modem je *CENTER*. V tomto modu jsou obdélníky vystředěny kolem bodů X a Y. Pomocí dvou dalších parametrů opět funkce *rect()* definuje šířku a výšku tohoto obdélníka. Posledním modem je *RADIUS*. Ten nastavuje opět středové body X a Y v prvních dvou parametrech. Druhé dva parametry udávají polovinu šířky a polovinu výšky obdélníka. … 144

`return` Speciální příkaz značící návrat hodnoty z funkce. Příkaz zároveň ukončuje spuštěnou funkci. … 85, 87

`rotate()` Provádí rotaci plátna v dvourozměrném zobrazení. Střed rotace je vždy aktuální souřadnice $X = 0$ a $Y = 0$. … 105, 107, 108, 144

`rotateX()` Funkce provádí rotaci celým prostorem kolem osy X. Vyžaduje hodnotu udanou ve stupních – radiánech. Hodnota se pohybuje v rozmezí 0 až $2 * \pi$. … 108

`rotateY()` Funkce provádí rotaci celým prostorem kolem osy Y. Vyžaduje hodnotu udanou ve stupních – radiánech. Hodnota se pohybuje v rozmezí 0 až $2 * \pi$. … 108

`rotateZ()` Funkce provádí rotaci celým prostorem kolem osy Z. Vyžaduje hodnotu udanou ve stupních – radiánech. Hodnota se pohybuje v rozmezí 0 až $2 * \pi$. … 108

`save()` Pomocí příkazu save() ukládáme plátno programu do obrázku. Příkaz vyžaduje cestu k souboru. Cestu lze zadat buď úplnou (absolutní), nebo jen pouhý název souboru. Zadáním pouhého názvu se soubor uloží přímo do adresáře *sketche*. Příkaz dále rozlišuje mezi příponami automaticky. Obrázek tak uloží v patřičných formátech: TIF, JPG, PNG nebo TGA. … 112–114, 145

`saveFrame()` Příkaz ukládá sérii obrázků. Stejně jako příkaz *save()*. Funkce vyžaduje cestu k souboru. Speciální vlastností příkazu je možnost

vložit za vstupní hodnotu *String* symbol křížku, budou posléze detekovány a nahrazeny číslováním sekvence okének. Každé číslo bude označovat číslo okénka podle proměnné. … 113, 114

saveStrings() Pomocí příkazu *saveStrings()* ukládáme pole ve formátu *String*. Tento příkaz přijme dva parametry: název souboru a název našeho pole s textovou informací. Příkaz ukládá textový soubor naplněný hodnotami z pole. Jednu hodnotu zapisuje vždy na nový řádek. … 115

scale() Mění velikost geometrie vůči vykreslovacímu plátnu. Výchozí hodnota je 1.0. … 105, 144

setup() Základní funkce pro nastavení výchozích parametrů programu. Tato funkce je spuštěna pouze jednou, vždy na začátku běhu programu. … 55, 66, 78, 84, 99, 102, 109, 120, 124

sin() Vypočítává sinus z daného úhlu. Tato funkce očekává údaj úhlu zadaný ve stupních radiánu, tedy hodnoty od 0 do 6.28. Zpětně funkce vrací hodnotu v rozmezí od -1 do 1. … 110, 135

size() Nastavuje rozměry okna programu. Hodnoty jsou udávány v pixelech pro osu X a Y. Tento příkaz by měl být spuštěn jako první ve funkci *setup()*. Jestliže není příkaz spuštěn, okno programu má výchozí velikost $100*100$ pixelů. Ve funkci *size()* je dále přípustný třetí parametr, ten je vyjádřen slovy: JAVA2D (výchozí render), *P2D* (akcelerovaný 2D render), *P3D* (akcelerovaný 3D render), *OPENGL* (3D akcelerovaný render). … 99, 116, 117

sketch Neboli *náčrt*, je koncept v prostředí *Processing* pro uchovávání jednotlivých projektů v adresářích. Jedna *sketch* je adresář, kam *Processing* ukládá data uživatele (tj. především kód a externí soubory). … 137

smooth() Zapíná takzvaný anti-aliasing, tedy vyhlazování hran kresby. Funkce nastavuje vyhlazování pro celý program. Zapnutí vyhlazování má za následek poměrně výrazné zpomalení kresby. Výstup kresby je ovšem o poznání jemnější. … 109, 143

sphere() Funkce kreslící kouli. Požaduje jediný argument: poloměr koule. Rozlišení koule je možné kontrolovat funkcí *sphereDetail()*. … 103

sphereDetail() Funkce nastavuje úroveň detailu při vykreslování objektu koule. Parametr zadaný ve formátu čísla *float* ovlivňuje vykreslovaný počet stran koule. … 103, 146

splitTokens() Funkce pro práci s textem přijímající dva parametry: samotný řetězec a dále takzvaný delimiter – znak, kterým řetězec chceme rozdělit. Funkce vrací pole řetězců, *String[]*. … 124

String Datatyp pro řetězec textu. Jedná se o speciální datatyp schopný uchovat více znaků pod jednou proměnnou. S tímto datatypem se pojí řada speciálních funkcí pro nakládání s textem. Představit si jej můžeme jako jednorozměrné pole znaků. … 121, 141, 143, 145, 146

stroj Stroj je technické zařízení, které přeměňuje jeden druh energie nebo síly v jiný – ať už kvalitativně, nebo kvantitativně. Původně byly stroje jen mechanické, ale dnes se tak označují i zařízení pracující na jiných fyzikálních či technických principech – například elektrický transformátor. Strojem je v této knize téměř výhradně myšlen počítač. Počítač je programovatelný typ stroje, který přijímá vstup, ukládá a zpracovává data a umožňuje výstup v požadovaném formátu. … 27

stroke() Funkce nastavující barvu obrysů tvarů. Například – spustíme-li *stroke(255, 0, 0)*, veškeré další kontury objektů budou mít červenou barvu. … 62, 104, 109, 136, 141, 143, 144

strokeWeight() Funkce nastavuje šířku vykreslovaných obrysů v pixelech. … 109

syntax highlighting Barevné odlišování slov podle jejich významů. Odlišování slouží k lepší orientaci programátora v kódu. … 41

this Speciální výraz vztahující se k objektu, ve kterém se zrovna pohybujeme. Slouží k identifikaci zejména pro práci s knihovnami nebo námi definovanými třídami. … 132

tint() Ovlivňuje barvu a průhlednost vykreslených obrázků. Protějškem je vyraz *noTint()*. … 139

translate() Provádí pohyb celého plátna na zadané souřadnice. V trojrozměrném modu vykreslování požaduje třetí rozměr pro osu Z. … 105, 106, 108, 144

true Pravda neboli 1. … 69, 70, 72, 85, 136, 138

vertex() Funkcí *vertex()* definujeme body pro tvorbu složitějších tvarů. Funkci musí předcházet příkaz *beginShape()* a musí ji následovat příkaz *endShape()*, přičemž záleží na pořadí, ve kterém jsou jednotlivé body definovány. Body mohou být dvou- i trojrozměrné. *Vertex* tedy můžeme definovat pomocí souřadnic X, Y nebo X, Y, Z. … 104, 135

void Neboli prázdná funkce. Z anglického „prázdno“. Funkce, která zpět nevrací žádný výsledek, pouze spouští sérii zadaných příkazů. Ty jsou uvozeny složenými závorkami. … 55, 83, 85, 87

Rejstřík

3D objekty, 101
3D zobrazení, 98

? : , 72

Abstrakt, 4
acos(), 108
Animace, 63
Ascioglu, Sinan, 21
asin(), 108
atan2(), 108

background(), 63
Barva, 46
beginRecord(), 116, 136
beginShape(), 102, 136, 146
Bomb, The, 24
boolean, 12, 52, 67, 70, 71, 83
box(), 101
Built with Processing, 31

camera(), 99, 100
chybové hlášení, 48
Cinder, 13
class, 88, 89
CODED, 76
color(), 46
colorMode(), 46, 134
cos(), 108, 133
CSV, 109, 112, 119
Cíle příručky, 10

DATA, 38
datatypy, 45
degrees(), 105
Dokonalost jazyka, 26
draw(), 53, 54, 63, 64, 76, 82, 86, 93, 113, 130, 136
Dynamika pohybu, 65

Editor, 39
ellipse(), 59
ellipseMode(), 135
Else, 71
else, 67, 71, 137
Empirický přístup k programování, 32
endRecord(), 116, 133
endShape(), 102, 133, 146
Examples, příklady přímo v Processingu, 21
exit(), 113
Experiment, 22

false, 67, 68, 70, 83, 134
fill(), 59, 60, 102, 135, 142
flight404, 14
float, 85, 113, 144
flow, 41
for, 79
frameCount, 64, 95, 111, 113, 139
frameRate(), 64
Fry, Ben, 22
Funkce, 81
Funkce a jejich datatypy, 85
funkce setup() a draw(), 53

GNU / GPL, 12, 35, 139
GNU / Linux, 35, 141

HALFPI, 108
Hodgin, Robert, 14
Hodnota a její zobrazení, 57

If, 67
if, 67, 71, 136, 137
if(), 136
image(), 118, 119
imageMode(), 119, 138
import, 97, 115, 129
indenting, 39
Ingac, Marcin, 14
Interakce, 72
interakce, 72
IRC, 21

Jednoduchý programovací jazyk, 25

Kamera, 99
key, 76, 134, 138
keyCode, 76, 134
keyPressed(), 76, 117, 138, 139
keyReleased(), 76
Klávesové zkratky, 40
Knihovny, 125
Knihy a průvodci, 22
Komentář, 43
komentáře v kódu, 44
Komunitní knihovny, 127
konzole, 46
Krajiny a světy, 13
Kresba, 58

line(), 59, 98, 101, 124
loadImage(), 118, 138, 141
loadStrings(), 119, 122
Local Bilateral Symmetry, 17
Logika programování, 43

McCabe, Johnatan, 17, 18
millis(), 65
MIT licence, 31
mouseButton, 76, 140
mousePressed(), 73, 75, 116
mouseX, 73
mouseY, 73

Na úvod, 9
Načítání informací, 117
Načítání obrázků, 118
Načítání textových souborů, 119
nelineární pohyb, 66
new, 78, 89, 130
nf(), 111
noFill(), 60

noise(), 94, 95
noSmooth(), 107
noStroke(), 60, 140
noTint(), 145
Náhoda, 93

objektově orientované jazyky, 28
obsah, 4
Operace s textem, 120
Orientace v prostoru, 54
Otevřenost softwaru, 31
otevřený software, 31
ovládání, 37

parseFloat(), 122
parseInt(), 122
Parsing, získávání hodnot z externích dat, 120
PDE, 36, 37
Perlinův šum, 94
PI, 108
PImage, 118, 138, 139
Podmínka, 66
Poděkování, 3
Pohyb, 61
point(), 59, 98
Pole, 77
popMatrix(), 105, 142
Použití transformací, 106
Počátky programovacího jazyka, 23
Pravidla formátování v knize, 12
print(), 47, 142
println(), 77, 120
Processing Examples, 22
programovací jazyky, 28
Proměnlivost, 61
prostor, 54
Proč zrovna Processing?, 28
Práce s knihovnou, 127
Práce s objekty, 90
práce s textem, 51
pushMatrix(), 105, 141

quad(), 59
QUARTERPI, 108

radians(), 105, 135
random(), 91, 93, 94
Reas, Casey, 22
rect(), 58, 59, 63, 82, 142, 143
rectMode(), 142
Renderovací mody, 97
return, 83, 85
rotate(), 103, 105, 106, 142
rotateX(), 106
rotateY(), 106
rotateZ(), 106

save(), 110–112, 143
saveFrame(), 111, 112
saveStrings(), 113
scale(), 103, 142
Schmidt, Karsten, 16, 17
setup(), 53, 64, 76, 82, 98, 101, 107, 118, 122
Seznam základních datatypů, 45
Shiffman, Daniel, 22
sin(), 108, 133
size(), 98, 114, 115
Sketch, 37
sketch, 135

Sketchbook, 38
smooth(), 107, 141
Smyčka, 79
Soustředěná činnost, 40
sphere(), 101
sphereDetail(), 101, 144
splitTokens(), 122
String, 119, 139, 141, 143
stroj, 25
stroke(), 60, 102, 107, 135, 139, 141, 142
strokeWeight(), 107
svobodný software, 31
syntax highlighting, 39
Syntetický obraz, 32
Síť, 21

textový editor, 39
this, 130
tint(), 138
Tisk do konzole, 46
Tisk do PDF, ukládání vektorů, 114
Toxilibs, 16
toxilibs, 16
Transformace, 103
translate(), 103, 104, 106, 142
triangle(), 59
true, 67, 68, 70, 83, 134, 137
Turing, Alan, 18, 24
tvary 2D, 59
Tvorba softwaru, 30
Třída a objekt, 87

Ukládání holých dat, 112
Ukládání informací, 109
Ukládání obrazových dat, 110
Uspořádání informací, 11
Uspořádání struktury programu, 81

vertex(), 102, 133
Vestavěné knihovny, 125
vestavěné knihovny, 125
Vizualizace hodnot, 122
void, 54, 81, 83, 85
Volba vhodného jazyka, 27
Výplň a obrys, 59

Zobrazení, 54
Základní datatypy, 44
Základní diskové operace, 37
Základní operace s datatypy, 48
Základní pravidla a zvyklosti, 41
Základní prostředí, 36
Základní struktura programu, 53

Kryštof Pešek
Processing Beta

Vydalo Nakladatelství Akademie múzických umění v Praze

Redakce: Tomáš Pospiszyl, Andrea Slováková, Zdeňek Holý
Jazyková redakce: Jana Křížová, Olga Stehlíková
Tisková příprava: Jan Omasta
Obálka, grafická úprava a sazba: Kryštof Pešek

Tisk NAMU
1. vydání
Praha 2013

ISBN: 978-80-7331-224-4
www.namu.cz, www.amu.cz