Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Knižnica pre výučbu programovania v Pythone

bakalárska práca

2015

Marián Horňák

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Knižnica pre výučbu programovania v Pythone

bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatka
Školiace pracovisko:	Katedra Informatiky
Školiteľ:	RNDr. Michal Forišek PhD.

Bratislava, 2015

Marián Horňák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Marián Horňák
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Knižnica pre výučbu programovania v Pythone
A library for Python programming education

Cieľ: Cieľom práce je implementovať sadu knižníc ktorá bude fungovať ako nadstavba nad Pythonom a knižnicou PyGame. Knižnice by mali pomôcť pri výučbe programovania, konkrétne tým, že uľahčia implementáciu rôznych hier a umožnia pri tom odabstrahovať od konceptov ktoré sú pre začiatočníkov zbytočne zložité. Súčasťou práce by mala byť aj vhodná textová časť a prípadne aj prezentácia výsledného produktu na webe.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 02.10.2014

Dátum schválenia: 28.10.2014 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Chcel by som sa poďakovať RNDr. Michalovi Forišekovi PhD. za vedenie práce a užitočné poznámky, Bc. Michalovi Anderlemu za námet na prácu a množstvo zaujímavých rozhovorov, RNDr. Andrejovi Blahovi, PhD. za zlepšujúce pripomienky a nápomocnú kritiku, Majke Vajdovej za psychickú podporu a gramatickú korektúru, vedeniu Gymnázia, Párovská 1, Nitra a RNDr. Jozefovi Piroškovi za umožnenie testovania knižnice ako aj žiakom Gymnázia, Párovská 1, Nitra, ktorí sa testovania zúčastnili.

Abstrakt

Napriek veľkému množstvu softvéru zameraného na výučbu programovania je niekedy náročné vhodne si vybrať správny. Stáva sa to najmä pri výučbe mimo školského prostredia, na ktoré je tento softvér prispôsobený. Príkladom je situácia, kedy potrebujeme počas týždňového kurzu motivovať 15-ročných žiakov k programovaniu. Softvér vhodný na tento účel by mal byť dostatočne jednoduchý, aby ho žiaci stihli v krátkej pochopiť, dostatočne interaktívny, aby žiakov zaujal a dostatočne praktický, aby sa žiaci boli schopní samostatne venovať programovaniu aj po skončení kurzu. V práci sa pokúsime navrhnúť a implementovať knižnicu spĺňajúcu tieto požiadavky. Výsledný produkt potom otestujeme a prediskutujeme jeho pripravenosť na použitie v praxi.

Kľúčové slová: výučba programovania, počítačová hra, Python

Abstract

Despite the great number of existing educational programming languages it is sometimes difficult to choose the correct one. The educational software is mainly designed for the school enviroment which makes it harder to use in different conditions. Consider, for example, the situation in which you need to motivate 15 years old students to program during a week-long course. The suitable software should be simple enough to be understood in a short time, interactive enough to get the students interested and practical so the students will be able to continue working individually after the end of the course. The subject of this work is to design and implement a library that fullfills those requirements. The final product will be tested and discussed in the terms of real-life usability.

Keywords: programming education, computer game, Python

Obsah

Zoznam obrázkov

Zoznam výpisov programu

Úvod

Vzdelávanie v oblasti programovania je v dnešnej dobe neustále sa rozvíjajúcich technológií veľmi dôležité. Zodpovedá tomu aj dostupnosť softvéru špecializovaného práve na tento účel. Vo väčšine prípadov je však takýto softvér zameraný na výučbu na školách, teda predpokladá pravidelný kontakt učiteľa so žiakom počas dlhšieho obdobia. V niektorých situáciách, napríklad v letných táboroch, sa ale stretávame s iným, nárazovejším typom kontaktu. Podobná situácia sa stala motiváciou pre túto prácu.

Naším cieľom bude navrhnúť a implementovať knižnicu, ktorá dokáže sprostredkovať rýchly a dostatočne komplexný kontakt s programovaním motivujúci k ďalšiemu sebavzdelávaniu.

Súčasťou návrhu bude aj výskum v oblasti softvéru na výučbu programovania. Pozrieme sa nielen na existujúce edukatívne programy a knižnice, ale aj na faktory, ktorými tieto produkty vplývajú na výučbu programovania. Nadobudnuté poznatky spolu so skutočnosťami, ktoré motivovali k tejto práci, následne využijeme na stanovenie požiadaviek pre výslednú knižnicu ako aj na ich naplnenie a implementáciu. Výsledný produkt potom zhodnotíme, otestujeme na začiatočníkoch a posúdime možnosti jeho ďalšieho rozvoja.

Základom každého dobrého softvéru je iteratívne testovanie a vylepšovanie. V rámci tejto práce nie je možné vytvoriť dostatočné množstvo takýchto iterácií. Preto budú výsledná knižnica a jej zdrojový kód voľne šíriteľné a budú vytvorené vhodné podmienky na vylepšovanie knižnice zo strany autora ako aj zo strany iných záujemcov.

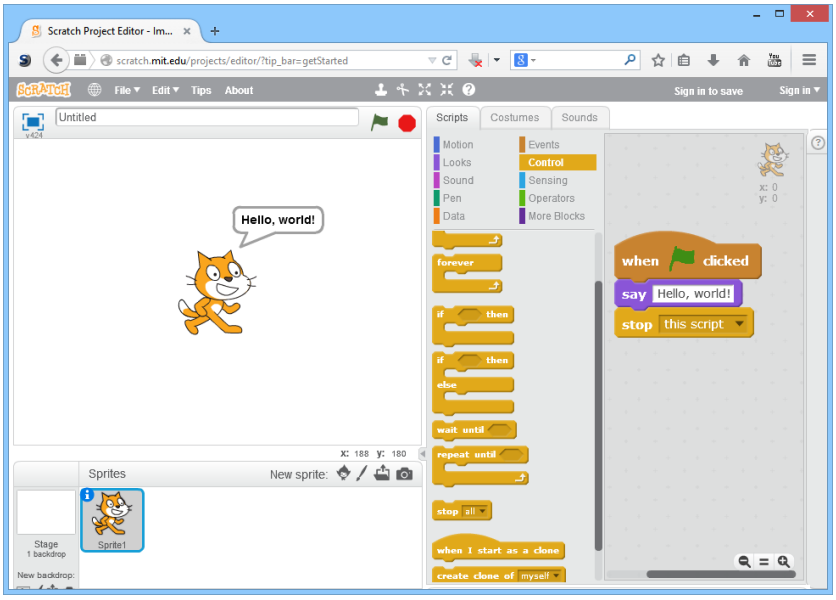
Kapitola 1

Softvér na výučbu programovania

Existuje obrovské množstvo jazykov, interpreterov a aplikácií slúžiacich na výučbu programovania. V tejto kapitole si predstavíme tie, s ktorými sa autor najčastejšie stretával alebo sú nejakým spôsobom inšpiratívne. Rozsah poskytnutý jednotlivému softvéru zodpovedá množstvu netradičných konceptov, ktoré daný softvér ponúka a ktoré využijeme v neskoršom texte.

1.1 Scratch

Scratch je interaktívne prostredie umožňujúce vytvárať programy (skripty) spájaním blokov s inštrukciami (kartičkami). Keďže celá štruktúra skriptu vzniká pomocou myši a klávesnica sa používa iba na zadávanie konštánt, programovanie si rýchlo dokážu osvojiť aj deti. Prostredie podporuje širokú škálu multimédií a z návrhu jazyka prirodzene vyplýva schopnosť paralelného spracovania skriptov. Je teda pomerne jednoduché vytvoriť aplikáciu, ktorej komponenty nezávisle interagujú s používateľom, ako aj medzi sebou. A práve tvorba takejto aplikácie môže byť atraktívna pre začínajúceho programátora. Prostredie scratch možno nájsť na jeho oficiálnej stránke [5].



Obr. 1.1: Rozhranie Scratchu [5]

1.1.1 Rozhranie

Rozhranie Scratchu (Obr. 1.1) je rozdelené na niekoľko častí. Každá časť má významnú funkciu pri vývoji a je pevnou súčasťou okna.

Paleta kartičiek obsahuje jednotlivé kartičky s inštrukciami rozdelené do ôsmich kategórií, aby sa uľahčilo ich vyhľadávanie. Kliknutím na kartičku a potiahnutím do časti určenej pre skripty začneme budovať nový skript. Ak to tvar kartičiek umožňuje, tak sa po priblížení automaticky spoja.

Časť, v ktorej programátor vidí, čo sa deje, sa nazýva scéna. Pozostáva z pozadia, po ktorom sa pohybujú a kreslia jednotlivé objekty. Scéna je jediná časť, ktorá je zobrazená pri prezentácii výsledného projektu. Na scéne je tiež možné zobraziť formuláre nastavujúce hodnoty niektorých premenných.

Pod scénou má programátor prehľad všetkých objektov a vie si zvoliť, ktoré z nich bude programovať v časti skripty. Tiež vie upravovať iné vlastnosti objektov, napríklad viditeľnosť, polohu, kostýmy (rôzne vzhľady objektu) alebo zvuky.

1.1.2 Vlastnosti

Jednou z kľúčových vlastností Scratchu je už spomínané spájanie kartičiek. Dve kartičky do seba zapadnú jedine vtedy, keď má ich spojenie syntaktický význam. Jednotlivé inštrukcie možno ukladať pod seba, čo sa dosiahne malým výbežkom na spodnej časti a identickým výsekom vo vrchnej časti kartičky. Kontrolné kartičky, napríklad cykly alebo podmienky, majú tvar písmena C, ktorým obkolesujú iné kartičky. Tvarom kartičky sa dosahuje aj typová bezpečnosť. Logické hodnoty sú šesťuholníkové, čísla oválne a reťazce obdĺžnikové. Kartička podmieneného cyklu má v sebe šesťuholníkovú dieru, do ktorej možno vložiť jedine logickú hodnotu – podmienku cyklu. Tento koncept zamedzuje syntaktickým chybám, ktoré sú častou prekážkou pre začínajúcich programátorov a umožňuje im naplno sa sústrediť na problém [9].

Ďalšou výhodou pre začiatočníkov je vysoká interaktivita prostredia. Ľubovoľný kúsok skriptu možno kedykoľvek spustiť. Počas samotného vykonávania môžeme robiť zmeny v skripte, a keď to stihneme pred vykonaním danej inštrukcie, zmeny budú zahrnuté do vykonávania. Jednotlivé objekty možno na scéne presúvať a otáčať, netreba teda zdĺhavo nastavovať súradnice objektov, keď program iba testujeme. Vďaka tomuto dostane programátor okamžitú spätnú väzbu, čo jednotlivé skripty jeho projektu robia a ľahšie sa mu hľadajú chyby.

Skripty možno spúšťať ako reakcie na rôzne udalosti. Stačí pridať zodpovedajúcu kartičku na začiatok skriptu. Medzi základné udalosti patria stlačenie klávesy alebo kliknutie na objekt. Programátor si tiež môže vytvoriť vlastné udalosti. Na jednu udalosť môže reagovať viacero skriptov, ktoré sa začnú vykonávať súčasne, takže paralelné spracovanie úloh možno dosiahnuť jednoduchým, pre jazyk prirodzeným spôsobom.

Scratch si vyvinul aj zaujímavý koncept na reprezentovanie rekurzie. Jednotlivé objekty sa vedia naklonovať, čím získajú kópie všetkých premenných pôvodného objektu, ktoré sa však následne nezávisle menia. Naklonované objekty reagujú na udalosti rovnako ako pôvodný objekt a existujú, kým sa prostredie nereštartuje alebo pokým explicitne nezmažú samé seba. Takto

si programátor môže rozdeliť úlohu na podúlohy a naživo sledovať, ako sa každá z nich vykonáva.

Zameranie prostredia na rýchle a intuitívne vytvorenie programu však výrazne redukuje možnosti jeho využitia. Práca na väčších projektoch je zdĺhavá a veľké množstvo skriptov je neprehľadné. Ak bude mať programátor následne záujem naučiť sa iný jazyk, narazí na obrovské množstvo odlišností, akými sú písanie programu klávesnicou alebo náročná paralelizácia úloh. Treba investovať nezanedbateľné množstvo času na vytvorenie návyku na nové prostredie. Túto skutočnosť si uvedomuje aj začiatočník, ktorý už videl ako vyzerá program bežného jazyka. Nemusí sa mu zdať vhodné začať programovať v niečom, čo je úplne odlišné a navyše to pôsobí detsky.

1.1.3 Zhodnotenie

Scratch je intuitívne prostredie, v ktorom sa človek naučí programovať hravo, ľahko a v princípe sám. Uľahčuje hľadanie chýb a pred mnohými nás dokonca ochráni svojím návrhom. Je vhodný pre deti, ktoré majú ešte dostatok času na učenie sa, ak chceme, aby si vybudovali intuíciu o jednotlivých programovacích konceptoch pred tým, než ich začneme učiť komplikovanú syntax. Ak však chceme v dohľadnej dobe prejsť na prakticky používaný jazyk, zo Scratchu to nebude jednoduché.

1.2 Robot Karel

Robot Karel je programovací jazyk slúžiaci na ovládanie robota v jednoduchom virtuálnom svete. Jeho základnou myšlienkou je poskytnúť priestor na naučenie sa algoritmického myslenia pomocou minimálneho počiatočného množstva inštrukcií. Programátor si ich dokáže efektívne zapamätať a rýchlo pocíti potrebu vytvárania vlastných príkazov, čo je základná myšlienka procedurálneho programovania. Existuje mnoho verzií tohto jazyka. Ja popíšem jednu z pôvodných, ktorú použila M. Synovcová vo svojej knihe Martina si

hraje z počítačem [12]. Autorka v nej vysvetľuje programovanie v Karlovi deťom. Knihou som sa inšpiroval aj pri písaní tohto textu.

Čitateľ si môže Karla vyskúšať napríklad na stránke `http://karel.oldium.net/`.

1.2.1 Rozhranie

Svet, v ktorom sa Karel pohybuje je ohraničená štvorcová mriežka. Na niektorých políčkach môže byť stena – na tie potom robot nedokáže vstúpiť. Virtuálna stena sa nachádza aj za okrajmi sveta. Na zvyšné políčka je možné pokladať značky. Na jednom políčku môže byť položený ľubovoľný počet značiek, navzájom sú však nerozlíšiteľné. Dôležitý je iba ich počet. Vo svete sa nachádza aj sám robot. Vždy stojí na niektorom políčku a je otočený do jednej zo 4 strán mriežky. Pred zadávaním príkazov si programátor nastaví svet podľa potreby.

Príkazy, ktoré programátor zadáva, sú uložené v slovníku. Na začiatku obsahuje iba 4 základné príkazy:

- `krok` – robot sa posunie o jedno políčko dopredu v smere, v ktorom je otočený
- `vlavo-vbok` – otočí robota o 90 stupňov proti smeru hodinových ručičiek
- `zdvihni` – odoberie jednu značku z aktuálneho políčka, pokiaľ sa tam nejaká nachádza
- `poloz` – pridá jednu značku na aktuálne políčko.

Slovník robota je možné rozširovať definovaním nových príkazov. Pri definícii sú dostupné už zadefinované príkazy, podmienky, cykly a aj rekurzia. Robot dokáže testovať prítomnosť steny na susedných štyroch políčkach a prítomnosť značiek na aktuálnom políčku.

1.2.2 Vlastnosti

Jednou z veľkých výhod Karla je už spomínaná potreba nových inštrukcií. Pozorný čitateľ si iste všimol, že chýba príkaz `vpravo-vbok`. Tento príkaz vieme nasimulovať použitím príkazu `vlavo-vbok` trikrát po sebe. Začínajúci programátor to zopárkrát pracne napíše a potom sám príde s myšlienkou, že by bolo oveľa príjemnejšie definovať robotovi nový príkaz.

Taktiež je unavujúce písať množstvo príkazov, ak chceme robota presunúť na druhý koniec mriežky. Takto vznikne prirodzená potreba cyklu. M. Synovcová vo svojej knihe [12] najskôr predstaví cykly vytvorené pomocou chvostovej rekurzie. Týmto spôsobom sa prvý, nekonečný program zacyklí a programátor zistí, že potrebuje podmienky.

Absenciou vnútornej pamäte núti Karel programátora využívať zásobník. Predstavme si napríklad problém, kedy chceme Karlovi počtom značiek oznámiť, o koľko políčok sa má posunúť dopredu. Najskôr ho necháme odoberať po jednej značke a vnárať sa do rekurzie, pokiaľ sú na políčku ešte nejaké značky. V tomto momente už na políčku nie je žiadna značka, ale Karel sa vnoril toľko krát, koľko značiek bolo na políčku pôvodne. Stačí teda, aby spravil krok pri každom vynáraní (dosiahneme to tak, že príkaz krok zaradíme za rekurzívne volanie).

Programátor si vie značky interpretovať rôznymi spôsobmi. Dokáže pomocou nich kresliť obrázok, používať ich ako pomocné informácie pre prechod bludiskom alebo robiť aritmetické operácie s ich počtami. Prostredie však môže pôsobiť na začiatočníka neatraktívne. Chýba interakcia s okolitým svetom, ktorá je v dnešnej dobe veľmi obľúbená.

1.2.3 Zhodnotenie

Robot Karel naučí programátora používať základné riadiace štruktúry, rekurziu a správne si deliť program na časti. Ak v ňom bude riešiť dobre zvolené úlohy, vedia mu rozvinúť algoritmické myslenie. Svojou jednoduchosťou však dokáže osloviť iba malú skupinu začiatočníkov.

1.3 Knižnica turtle v jazyku Python

Jazyk Python ponúka v rámci svojich štandardných grafických knižníc aj knižnicu turtle, ktorá implementuje základnú korytnačiu grafiku. Korytnačia grafika patrí medzi obľúbené nástroje na výučbu programovania. Množstvo jej výhod spomína napríklad Seymour Papert vo svojej knihe Mindstorms [11]. Jazyk Python je v súčasnej dobe tiež populárny. Je to interpretovaný jazyk s jednoduchou syntaxou a možnosťou interaktívneho vykonávania príkazov, čo spolu s korytnačou grafikou vytvára vhodné podmienky pre začínajúcich programátorov. Podrobnejšie sa jazyku Python budeme venovať v neskoršom texte.

Bližšie informácie o knižnici turtle možno nájsť napríklad v oficiálnej dokumentácii jazyka Python [1].

1.3.1 Rozhranie

Pri bežnej práci s knižnicou turtle vidí programátor dve okná. Prvé obsahuje ineraktívny interpreter príkazov jazyka Python a druhé grafickú plochu s korytnačkou. Korytnačku je možné pomocou sady príkazov posúvať po ploche, pričom v základnom nastavení zanecháva za sebou čiaru, a tým kreslí rôzne tvary. Programátor vidí polohu a orientáciu korytnačky na ploche.

Korytnačku je možné ovládať štyrmi základnými príkazmi. Príkazy `forward` a `backward` posúvajú korytnačku o parametrom zadaný počet pixelov v smere jej orientácie postupne vpred alebo vzad. Príkazy `left` a `right` menia orientáciu korytnačky o parametrom zadaný uhol postupne v smere alebo proti smeru hodinových ručičiek. Na kreslenie komplikovanejších tvarov je potom výhodné používať riadiace štruktúry jazyka Python, ako napríklad cykly, podmienky alebo funkcie. Knižnica turtle implementuje množstvo ďalších príkazov ovplyvňujúcich polohu korytnačky, výzor korytnačky alebo štýl čiary.

Okrem vpisovania príkazov priamo do interaktívneho interpretera je možné písať program do súboru, ktorý sa vykonáva až pri spustení. Týmto spôsobom

môžu naprogramované algoritmy existovať nezávisle od behu interpretera.

1.3.2 Vlastnosti

Ako uvádza oficiálna dokumentácia [1], knižnica turtle bola do jazyka Python zaradená za účelom motivovať deti k programovaniu v tomto jazyku. Na rozdiel od euklidovskej geometrie alebo analytickej geometrie, si koncepty v korytnačej geometrii dokáže dieťa ľahšie spojiť s reálnym svetom. Vďaka tomu sa potom dokáže viac sústrediť na samotnú implementáciu [11].

Veľká výhoda knižnice je tiež používanie natívneho prostredia rozvinutého programovacieho jazyka. Výučba môže začať jednoduchou korytnačou grafikou. S rastúcou náročnosťou úloh začne programátor potrebovať cykly, podmienky, premenné, funkcie a iné koncepty jazyka. Na konci vyučovacieho procesu už ovláda veľkú časť syntaxe jazyka a je nezávislý od korytnačej grafiky. Programátor sa teda plynule preorientoval z používania jednoduchej knižnice na znalosť syntaxe celého jazyka Python.

Napriek tomu, že kreslenie je dobrý nástroj na výučbu programovania, nie je dostatočný. Nedokáže prirodzene motivovať k používaniu pokročilejších konceptov, ako sú napríklad polia alebo triedy. Nevýhodou kreslenia je aj statický výsledok. Programátor sa môže na výsledok svojej práce iba pozerať, nedokáže s ním interagovať. Tým sa znižuje miera, ktorou si vychutnáva výsledok svojej práce.

1.3.3 Zhodnotenie

Knižnica turtle učí v praxi používaný programovací jazyk pomocou jednoduchej korytnačej grafiky. Je teda vhodná ako úvodná téma výučby jazyka Python. Na motiváciu k pokročilejším témam je však potrebné použitie inej knižnice.

1.4 Iný softvér na výučbu programovania

Pri výskume súčasného softvéru na výučbu programovania som sa stretol s viacerými zaujímavými jazykmi, ktoré síce nevyužijeme pri návrhu, ale je vhodné si ich aspoň v stručnosti predstaviť.

Jazyky rodiny Pascal sú plnohodnotné imperatívne programovacie jazyky zamerané na výučbu programovania. Ich hlavnou výhodou je jednoduchá a striktná syntax, ktorá zamedzuje začiatočníckym chybám. Ďalšie referencie a prehľadové informácie o jazyku možno nájsť v kapitole Pascal and its Successors [13], ktorú publikoval Niklaus Wirth, autor jazyka, v knihe Software pioneers.

Jazyky rodiny Logo implementujú korytnačiu grafiku so syntaxou založenou na programovacom jazyku LISP. Na rozdiel od spomínanej knižnice turtle, jazyky Logo sú primárne zamerané na výučbu a teda okrem korytnačej grafiky poskytujú aj množstvo inej funkcionality so syntaxou vhodnou pre začiatočníkov. Ďalšie informácie, publikácie a softvér možno nájsť na oficiálnych stránkach Logo Foundation [4].

Kapitola 2

Vplyv jazyka a prostredia na výučbu programovania

Programovací jazyk a vývojové prostredie sú faktory, ktoré sa musia zvoliť pred začiatkom výučby a následne sú prítomné počas celého jej priebehu. Vplývajú teda na výučbu naznedbateľným faktorom. Nami navrhovaná knižnica bude zasahovať do jazyka ako aj do vývojového prostredia. Je preto dôležité tieto vplyvy preskúmať a zohľadniť ich pri návrhu.

Treba si uvedomiť, že programovanie v sebe zahŕňa množstvo rôznych zručností a vedomostí, ktoré sa programátor musí naučiť. Medzi najdôležitejšie patrí prevádzanie problému na algoritmus, syntax príkazov, sémantika príkazov a správa programov zahŕňajúca ich vytváranie, spúšťanie, testovanie či prípadné kompilovanie [8]. Jedným z cieľov tejto kapitoly bude hľadanie spôsobov ako môžeme začiatočníka od niektorých z týchto konceptov odbremeniť.

2.1 Syntax

Syntax určuje striktné pravidlá, ktoré programátor musí od začiatku používať úplne bez chyby. Jazyk a prostredie by teda mali začiatočníka syntaxou zaťažovať čo najmenej. Programátor sa potom dokáže lepšie sústrediť na rie-

šenie problému [9].

Syntax jazyka vhodného na výučbu programovania by mala byť jednoduchá a intuitívna. Jeden z dôvodov výskytu komplikovaných konštrukcií v jazyku je snaha poskytnúť programátorovi čo najširšie možnosti využitia jazyka [8]. Túto problematiku bližšie rozoberieme v časti 2.5. Iné komplikované konštrukcie umožňujú programátorovi vyjadriť určité metainformácie o programe. Patria sem napríklad typy alebo modifikátory prístupu. V niektorých starších jazykoch (C++, Pascal) tieto informácie slúžia na optimalizáciu kompilovania. Novšie jazyky používajúce explicitné typovanie (Java, Haskell) dokážu vďaka týmto informáciám odhaliť určité chyby v programe alebo inteligentne implementovať automatické dopĺňanie. Existuje však množstvo jazykov, ktoré deklaráciu typov nepodporujú (Python, JavaScript). Programy napísané v týchto jazykoch strácajú užitočné metainformácie v prospech jednoduchšej syntaxe. A keďže pochopenie a využitie týchto metainformácií by bola ďalšia vec, ktorú by sa programátor musel naučiť, je vhodné ich z úvodnej výučby vynechať.

Asistenciu pri písaní syntakticky správneho programu môže poskytovať aj vývojové prostredie [8]. Dokáže vyhľadávať a zvýrazňovať syntaktické chyby už počas písania programu, kým má ešte programátor v aktívnej pamäti, čo daný úsek programu vykonáva. Ďalšia možnosť je ponúkať programátorovi automatické dopĺňanie textu.

Prostredia, v ktorých je program vytváraný interaktívne, dokážu syntaktické chyby vylúčiť úplne. Stačí, aby programátorovi poskytli iba toľko voľnosti, koľko jej umožnuje samotná syntax. Príkladom takéhoto prostredia je Scratch [10]. Kartičky inštrukcií je možné spojiť iba vtedy ak ich spojenie má syntaktický význam.

2.2 Interaktivita

Pod interaktivitou prostredia budeme pre potreby tohto textu rozumieť schopnosť nezávisle spúšťať jednotlivé časti programu. Vďaka interaktivite dokáže

programátor skúmať vlastnosti vstavaných funkcií jazyka a zároveň testovať funkčnosť vlastných programov. Tieto možnosti sú užitočné najmä pre začínajúceho programátora.

Interaktivitu je možné využiť aj pri odstraňovaní chýb, čo patrí k jednej z náročných a nevyhnutných činností pri programovaní [8]. Spúšťaním jednotlivých častí programu môže programátor chybu lokalizovať a testovaním správania sa chybného príkazu pochopiť jej podstate.

S určitou mierou interaktivity sa môžeme stretnúť napríklad v prostredí Scratch, v Robotovi Karlovi alebo aj v knižnici turtle.

2.3 Prístupnosť

Programátor je zároveň aj používateľom vývojového prostredia. Ak sa má plne koncentrovať na prácu, potrebuje, aby malo prostredie jednoduché a intuitívne rozhranie. Ak sa súbor s programom automaticky otvorí v editore, prostredníctvom ktorého je možné program priamo spustiť, programátor si na prostredie zvykne oveľa rýchlejšie ako v prípade spúšťania a kompilovania prostredníctvom príkazového riadku či používania externého editora. Profesionálne vývojové prostredie však nemusí byť vhodná voľba. Príliš veľa možností dokáže začiatočníka zmiasť a prístup k základným nástrojom sa môže skomplikovať.

K nezanedbateľným faktorom patrí aj náročnosť inštalácie prostredia. V prípade, že programátor používa počas výučby iné počítače ako tie, ku ktorým má prístup vo voľnom čase, prostredie je nutné na voľnočasové počítače doinštalovať. Ak by to bol technicky náročný proces, začiatočníkovi by sa ho nemuselo podariť zrealizovať a prišiel by tak o možnosť tréningu vo voľnom čase.

2.4 Výsledok práce

Dôležitým motivačným faktorom pri akejkoľvek práci je výsledok, ktorý sa touto prácou dosiahne. Ak sa žiak učí programovať, jeho hlavný cieľ je, prirodzene, vedieť programovať. Tento cieľ je však príliš vzdialený a komplexný a teda nemusí motivovať dostatočne. Oveľa zaujímavejšie sú čiastkové výsledky, teda napríklad jednotlivé programy, ktoré počas výučby vytvorí. Tieto môže žiak následne testovať, čím získa potrebu ich ďalej vyvíjať. Zároveň sa tiež presvedčí, že už aspoň niečo ovláda.

Častým úvodným príkladom v kurzoch programovania býva vypísanie textu „`Hello world`" na obrazovku. A v mnohých prípadoch prácou s textom výučba aj pokračuje. Tieto postupy boli postačujúce v minulosti, kedy bolo textové rozhranie základom práce s počítačom. V dnešnej dobe je začínajúci programátor zvyknutý na interaktívne rozhrania a používanie alebo vytváranie textovo založených programov naňho môže pôsobiť nudne či odstrašujúco. [6]

Vytváranie interaktívnych rozhraní však so sebou prináša určité problémy. Reakcia na udalosti, akými sú kliknutie myši alebo stlačenie klávesy musí nastať nezávisle na behu hlavného programu a teda je potrebné vymedziť blok inštrukcií, ktorý túto udalosť spracováva. Na vytvorenie spustiteľného bloku inštrukcií je však v mnohých jazykoch nutné predstaviť koncept funkcie, ktorý nemusí byť pre začiatočníka jednoduchý. V tomto prípade majú výhodu jazyky, ktoré sú zamerané na výučbu programovania. Napríklad v prostredí Scratch sa každý skript začína kartičkou udalosti. Problém je teda vyriešený samotným návrhom jazyka.

2.5 Zameranie jazyka

Požiadavky na jazyk sa pri výučbe a pri praktickom programovaní výrazne líšia. Kým pri výučbe obetujeme funkcionalitu v prospech jednoduchej syntaxe, v praxi sa snažíme čo najmenej obmedziť možnosti progamátora, čo

vedie k zavádzaniu komplikovaných syntaktických konštrukcií. Toto vytvára rozdiel, ktorý bude musieť začínajúci programátor niekedy prekonať. [8]

Pokiaľ predpokladáme, že programátor má dostatok času na výučbu, je vhodné sa týmto problémom zo začiatku nezaoberať a prechod k reálnemu programovaciemu jazyku riešiť až po nadobudnutí určitých skúseností. Možná voľba je teda napríklad prostredie Scratch.

Častokrát sa však stáva, že schopnosť riešiť reálne problémy je vyžadovaná v krátkej dobe po začiatku výučby. V tomto prípade je nutné použiť niektorý zo všeobecných programovacích jazykov. Komplikovanejším konštrukciám sa možno vyhnúť ich zatajovaním, je však potrebné vedieť riešiť dostatočne zaujímavé problémy. Takéto možnosti ponúka napríklad knižnica turtle pre jazyk Python.

2.6 Lokalizácia vstavaných príkazov

Množstvo programovacích jazykov používa kľúčové slová v anglickom jazyku. Pri učení sa takýchto programovacích jazykov si programátor, ktorý je menej pokročilý v angličtine, bude musieť zapamätať nielen syntax ale aj jednotlivé slová. Je to ďalšia prekážka pri učení, ktorá odvracia jeho pozornosť od samotného programovania. [7]

Avšak aj pri pokročilejšej znalosti anglického jazyka môže nastať problém. Ak je kľúčové slovo zaradené do jazyka vo význame, ktorý je menej známy alebo takmer nepoužívaný v hovorenej reči, programátora dokáže použitie zmiasť alebo narušiť jeho intuíciu. [8]

Použitie lokalizácie však môže mať negatívne následky na prenositeľnosť kódu. V určitom bode spracovania programu je potrebné jednotlivé príkazy preložiť do medzinárodnej podoby a naopak, pri zobrazení cudzieho programu je potrebné príkazy lokalizovať. V prostredí Scratch to nie je problém, nakoľko prostredie abstrahuje od súborovej reprezentácie projektu a programátor sa s ňou nestretne. Avšak jazyky, ktoré si zdrojový kód ukladajú do súboru priamo, nemajú k tomuto súboru prístup, pokiaľ nie je spustený

interpreter alebo kompilátor. O lokalizáciu takýchto súborov by sa teda programátor musel starať sám, použitím dostupných nástrojov. V praxi používané jazyky teda nepodporujú lokalizáciu kľúčových slov a príkazov.

Kapitola 3

Návrh knižnice

Obsahom tejto kapitoly bude stanovenie cieľov pre knižnicu a návrh knižnice, pri ktorom sa tieto ciele pokúsime naplniť.

3.1 Stanovenie cieľov

Najskôr popíšeme okolnosti, ktoré autora motivovali k tejto práci a následne tieto okolnosti sformalizujeme do požiadaviek kladených na výsledný produkt.

3.1.1 Motivácia

Korešpondenčný Seminár z Programovania [3] (ďalej len KSP), ako študentský spolok Fakulty Matematiky Fyziky a Informatiky Univerzity Komenského v Bratislave a súčasť občianskeho združenia Trojsten, organizuje množstvo aktivít zameraných na vzdelávanie talentovaných žiakov stredných škôl a vyšších ročníkov základných škôl v oblasti informatiky. Medzi tieto aktivity patrí aj Jesenná Škola KSP, ktorá sa zameriava na motiváciu žiakov k programovaniu a účasti na iných akciách KSP. Jesenná Škola prebieha každoročne počas jedného pracovného týždňa začiatkom školského roka. V rámci dopoludňajšieho programu absolvujú účastníci rýchly kurz programovania, kto-

rého cieľom je poskytnúť žiakom dostatočné vedomosti a motiváciu, aby boli schopní ďalej sa samostatne zdokonaľovať v tejto oblasti. Pre účely kurzu musia organizátori vybrať vhodný programovací jazyk a prostredie. Počas Jesennej Školy v roku 2014 bola navrhnutá a autorom tejto práce implementovaná knižnica Gaminator slúžiaca na jednoduchý a rýchly vývoj počítačových hier. Organizátori považovali knižnicu za úspešnú, na základe čoho sa autor rozhodol navrhnúť a implementovať novú, prepracovanejšiu verziu knižnice.

3.1.2 Požiadavky

Pokiaľ je to potrebné, k požiadavke je uvedené aj stručné odôvodnenie. Požiadavky sú nasledovné:

- Knižnica by mala byť nápomocná pri výučbe programovania.
- Knižnica by mala byť implementovaná v jazyku so všeobecným zameraním. (Žiaci by po absolvovaní kurzu mali byť schopní zapájať sa do iných aktivít KSP.)
- Knižnica by mala slúžiť na jednoduchý vývoj počítačových hier.
- Knižnica by mala byť implementovaná v jazyku s jednoduchou syntaxou. (Snaha minimalizovať zložitosť je prirodzená, obmedzený vyučovací čas však zvyšuje jej dôležitosť.)
- Knižnica by mala byť jednoduchá na inštaláciu a používanie. (Cieľom kurzu je motivovať žiakov k domácej práci.)
- Knižnica by mala byť lokalizovateľná do slovenského jazyka. (Nie sú kladené požiadavky na vedomosť anglického jazyka učastníkmi Jesennej Školy.)

3.2 Naplnenie cieľov

V predchádzajúcom texte stanovené požiadavky sa teraz pokúsime naplniť vhodným výberom technológii a návrhom konceptov. Každú technológiu alebo koncept si najskôr predstavíme a potom odôvodníme náš výber. Na ilustráciu budeme používať výpis 3.1 používajúci výslednú implementáciu knižnice

3.2.1 Python

Knižnicu budeme implementovať v jazyku Python [2]. Python je interpretovaný programovací jazyk s jednoduchou syntaxou. Ponúka možnosti procedurálneho, funkcionálneho aj objektovo orientovaného programovania. Jazyk je distribuovaný pod otvorenou licenciou, má obrovskú komunitu používateľov a množstvo užitočných knižníc.

V súčasnej dobe je Python používaný v dvoch verziách. Prechod zo staršej verzie 2 na verziu 3 spôsobil problémy v kompatibilite starších, obľúbených knižníc. Programátor sa preto musí rozhodnúť medzi väčšou podporou knižníc pri staršej verzii a modernejším návrhom jazyka pri novšej verzii. Navrhovanú knižnicu budeme implementovať kompatibilnú s oboma verziami jazyka. Treba však poznamenať, že nami používaná knižnica pygame má väčšiu podporu pre Python 2.

Syntax jazyka bola vytvorená s dôrazom na ľahkú čitateľnosť. Vďaka tomu je zároveň aj jednoduchá a intuitívna. Zaujímavým prvkom jazyka je použitie odsadzovania na vymedzenie bloku kódu určujúceho telo nejakej riadiacej entity (podmienky, cyklu, funkcie...), čím núti programátora vytvárať prehľadný kód. Python nevyžaduje deklaráciu premenných ani bodkočiarky na konci jednotlivých príkazov, nepodporuje deklaráciu typov, automaticky referencuje, dereferencuje, alokuje a dealokuje objekty. Tieto a mnohé ďalšie syntaktické vlastnosti vytvárajú vhodné prostredie pre začiatočníkov.

Programy v jazyku Python možno spúšťať priamo zo súboru so zdrojovým kódom alebo postupne zadávaním jednotlivých príkazov do interaktívneho in-

```
from gaminator import *
import random

class Potrava(Thing):

    @event("SETUP")
    def nastav_sa(self):
        self.width = 11
        self.height = 11

    def paint(self, c):
        c.color = GREEN
        c.ellipse((5, 5), 5, 5)

    @event("STEP")
    def padaj(self):
        self.y += 0.5


class Ryba(PictureThing):

    @event("SETUP")
    def nastav_sa(self):
        self.picture = open_picture("ryba.gif")

    @event("STEP")
    def hyb_sa(self):
        if game.pressed(K_UP):
            self.y -= 2
        # ...

    @collision("Potrava")
    def zjedz(self, potrava):
        potrava.world = None
        Potrava(world=self.world, x=random.randint(0, window.width), y=0)


class Akvarium(World):

    @event("SETUP")
    def nastav_sa(self):
        self.background = open_picture("voda.png")
        Ryba(world=self, x=300, y=200)
        Potrava(world=self, x=random.randint(0, window.width), y=0)


game.start(Akvarium(), True)
```

Výpis programu 3.1: Jednoduchá hra v knižnici Gaminator.

terpretera, ktorý ich okamžite vyhodnotí. Interpreter si uchováva stav medzi jednotlivými príkazmi, je teda možné pracovať s premennými z predošlých príkazov alebo dokonca importovať a používať ľubovoľnú knižnicu.

Dôvody, pre ktoré sme vybrali Python ako implementačný jazyk navrhovanej knižnice sú teda veľká popularita, jednoduchá syntax a prítomnosť interaktívneho interpretera.

3.2.2 Pygame

Pygame je knižnica poskytujúca rozhranie nízkoúrovňovej knižnice SDL pre jazyk Python. Rozhranie je mierne upravené tak, aby uľahčilo implementáciu hier. Nie je však dostatočne jednoduché pre začínajúceho programátora. Preto budeme našu knižnicu vyvíjať ako nadstavbu nad pygame.

3.2.3 Triedy ako organizačné jednotky

Základom práce s knižnicou bude vytváranie tried, ktorými popíšeme správanie jednotlivých objektov v hre. Koncept objektovo orientovaného programovania však môže byť pre začínajúceho programátora príliš komplikovaný. V nasledujúcom texte si odôvodníme, prečo je použitie tohto konceptu potrebné a nájdeme spôsob, ako tento koncept prezentovať dostatočne jednoducho.

Ako sme už spomínali v časti 2.2, pri vytváraní interaktívneho prostredia potrebujeme vedieť reagovať na udalosti. A ako reakciu na udalosť musíme vytvoriť samostatný blok kódu. Jediný spôsob ako môžeme v jazyku Python pracovať s blokom kódu je zaobaliť ho do funkcie. Ak má byť teda implementovaná hra sériou reakcií na udalosti v interaktívnom svete, bude musieť obsahovať sériu funkcií predstavujúcich tieto reakcie.

Samotné reakcie však vo väčšine prípadov možno priradiť konkrétnemu objektu na obrazovke. Toto priradenie je súčasťou porozumenia samotnej reakcii. Je teda možné a z hľadiska prehľadnosti kódu dokonca vyhovujúce, zoskupiť reakcie na udalosti podľa typu objektu, ktorého sa tieto reakcie týkajú. Úlohu zoskupovateľa však s minimálnym syntaktickým rozdielom do-

káže zastúpiť aj definícia triedy. Navyše tým každá reakcia získa implicitný kontext objektu, ku ktorému patrí. Použitie tohto konceptu možno vidieť v celom výpise 3.1.

Z pohľadu začiatočníka teda nebudeme definovať funkcie objektov určitého typu, ktoré neskôr budeme volať. Budeme definovať reakcie na rôzne udalosti a priraďovať ich k typu objektu. So stúpajúcimi skúsenosťami programátora môžeme jednotlivé výhody objektovo orientovaného programovania postupne odhaľovať.

3.2.4 Veci a Svety

Vecou budeme nazývať každú entitu, ktorá samostatne existuje na obrazovke. Svet bude entita, v ktorej sa môžu jednotlivé Veci nachádzať. Každá Vec sa môže nachádzať v najviac jednom Svete. Pokiaľ sa Vec v nejakom Svete nachádza, dokáže interagovať s inými vecami v tomto svete.

Pri základom použití zodpovedá Svet obrazovke. Má svoju výšku a šírku, ktoré zodpovedajú výške a šírke obrazovky. Existovať môže viacero Svetov, zobrazený je však práve jeden. Rôzne Svety predstavujú rôzne rozloženia obrazovky, napríklad menu, hernú obrazovku alebo obrazovku s inventárom hráča.

Vec má svoju polohu, určenú x-ovou a y-ovou súradnicou. Pokiaľ sa Vec nachádza vo Svete, ktorý je práve zobrazený, zobrazí sa tiež, a to na súradniciach určených jej polohou. Pre prípad prekryvu Vecí je možné nastaviť poradie, v ktorom sa Veci zobrazia prostredníctvom z-ovej súradnice. Každá Vec má tvar obdĺžnika so stranami rovnobežnými s okrajmi obrazovky. Rozmery tohto obdĺžnika možno určiť výškou a šírkou Veci. Zarovnanie obdĺžnika vzhľadom k polohe Veci je možné určiť pomocou reálneho čísla z intervalu $[0, 1]$ nezávisle pre vertikálnu aj horizontálnu os. Preddefinované zarovnanie je 0.5 (na stred), čo odďaľuje moment nutnosti predstavenia tohto konceptu žiakom.

Veci a Svety sú bežné koncepty, s ktorými sa stretávame pri uvažovaní nad hrou. Predpokladáme, že začiatočník dokáže s týmito konceptami pracovať

intuitívne. Výpis 3.1 obsahuje tri príklady použitia týchto konceptov.

3.2.5 Udalosti

Udalosti sú jedným zo základných spôsobov ako zabezpečiť spustenie určitého bloku kódu. Ku každej funkcii prislúchajúcej konkrétnemu typu Veci alebo Sveta bude možno priradiť ľubovoľný počet udalostí spúšťajúcich túto funkciu. Okrem klasických udalostí vyvolaných hráčom bude knižnica ponúkať aj udalosti popisujúce herný cyklus a možnosť vytvorenia vlastných udalostí.

Spracovávanie udalostí sa bude vykonávať v taktoch s nastaviteľnou frekvenciou. V každom takte sa najskôr zdetekujú udalosti týkajúce sa Vecí v aktuálne zobrazenom Svete, následne sa vyhodnotia a nakoniec sa Svet prekreslí.

Udalosti automaticky vyvolávané knižnicou budú:

- nastav – táto udalosť nastane pre každú Vec a Svet krátko po vytvorení. (výpis 3.1, riadok 6)
- krok – táto udalosť nastane práve raz v každom takte pre aktuálne zobrazený Svet a všetky Veci v ňom. (výpis 3.1, riadok 15)
- zrážka – táto udalosť pre konkrétnu Vec, pokiaľ sa prekrýva s inou Vecou zadaného typu. (výpis 3.1, riadok 32)
- stlačený kláves a uvoľnený kláves – tieto udalosti existujú vo všeobecnosti pre ľubovoľný kláves ako aj samostatne pre konkrétne klávesy. Nastanú pri stlačení alebo uvoľnení daného klávesu pre aktuálne zobrazený Svet a všetky Veci v ňom.

Knižnica bude ponúkať aj možnosť vytvoriť udalosti manuálne. Tieto udalosti nastanú pre Svet, v ktorom boli vyvolané a všetky Veci v ňom. Vykonávanie udalosti sa začne v nasledujúcom takte alebo po uplynutí zadaného času. Mapovanie vyvolanej udalosti na počúvajúcu funkciu sa bude diať na základe identifikátora udalosti – ľubovoľného objektu v jazyku Python.

Nutnosť implementovať udalosti vyplýva z potreby reagovať na aktivitu hráča. Ich koncept však môžeme použiť aj na realizáciu inej funkcionality a zmenšiť tak objem konceptov predstavených programátorovi. Z tohto dôvodu boli udalosti zaradené do knižnice.

3.2.6 Práca s grafikou

Základným grafickým prvkom v knižnici bude Obrázok. Obrázok bude možné vytvoriť tromi spôsobmi: ako prázdny Obrázok s danými rozmermi, načítaním zo súboru alebo vyrenderovaním textu. Do existujúceho Obrázku bude možné dokresliť základné objekty ako sú čiary, elipsy, mnohouholníky či iné Obrázky.

Každá Vec vlastní svoj interný Obrázok, ktorý sa automaticky prispôsobuje jej rozmerom. Kód kresliaci do tohto obrázku možno umiestniť do špeciálnej funkcie, ktorú knižnica zavolá automaticky (výpis 3.1, riadok 11). Existuje tiež špeciálny typ Veci, ktorému možno nastaviť Obrázok priamo. V tomto prípade ale stratíme kontrolu nad rozmermi Veci (výpis 3.1, riadok 24).

Každý Svet taktiež vlastní Obrázok, ktorý sa zobrazí ako pozadie (výpis 3.1, riadok 42). Na rozdiel od Veci sú však jeho rozmery nezávislé od rozmerov obrazovky. Pozadie Sveta možno využiť ako kresliacu plochu pri zoznamovaní sa s knižnicou.

3.2.7 Interaktívny režim

Knižnica bude poskytovať možnosť fungovania paralelne s interaktívnym interpreterom príkazov. Tento interpreter sa inštancuje spolu s inicializáciou knižnice a bude poskytovať prístup ku všetkým knižničným objektom, k používateľom zadefinovaným Veciam a Svetom aj k aktívnej inštancii Sveta.

3.2.8 Pokročilá práca so Svetmi

Svety budú poskytovať programátorovi niekoľko mechanizmov na správu rozloženia obrazovky: zásobník svetov a vnorené svety.

Zásobník Svetov obsahuje aktuálne aktívne Svety. Vrchný Svet zásobníka je zobrazený a vyhodnocujú sa v ňom udalosti. Zvyšné Svety v zásobníku sú pozastavené a čakajú, pokiaľ sa pozície v zásobníku nad nimi uvoľnia. Do zásobníka je možné vložiť nový Svet (a tým pozastaviť aktuálny), odstrániť aktuálny svet alebo nahradiť aktuálny svet iným.

Každý Svet je zároveň aj Vecou, ktorá sa môže nachádzať v inom Svete. Okrem spôsobu vykresľovania sa takýto Svet správa identicky ako Vec. Týmto mechanizmom možno zobraziť viacero nezávislých Svetov súčasne. Počet úrovní vnorenia nie je obmedzený. Udalosti, ktoré nastanú v nejakom Svete, nastanú aj vo všetkých vnorených Svetoch.

3.2.9 Lokalizácia knižnice

Lokalizácia kľúčových slov a štandardných knižníc jazyka Python by si vyžadovala obrovské množstvo práce a vytvorenie vlastného interpretera. Z tohto dôvodu bude knižnica poskytovať iba možnosť lokalizácie vlastného rozhrania. Základné rozhranie bude implementované v anglickom jazyku.

3.2.10 Názov a distribúcia

Knižnica bude distribuovaná pod licenciou MIT (je súčasťou elektronickej prílohy) prostredníctvom štandardnej databázy knižníc pre jazyk Python. Ako názov knižnice sa autor rozhodol ponechať pôvodný názov Gaminator.

Inštalácia knižnice zo štandardnej databázy je automatická, nemožno však zabezpečiť inštaláciu samotného jazyka Python a knižnice pygame, ktorá sa kvôli svojej závislosti na nízkoúrovňových knižniciach v štandardnej databáze nenachádza. Väčšina distribúcií Unixových systémov má v dnešnej dobe interpreter jazyka Python predinštalovaný, inštalácia Gaminatora bude teda pozostávať iba z dvoch krokov. Pre systém Windows by však bol tento proces náročnejší a teda vytvoríme inštalačný program, ktorý všetky potrebné závislosti doinštaluje.

Kapitola 4

Implementácia

V tejto kapitole sa budeme zaoberať implementáciou najnáročnejších častí knižnice, pri ktorých bolo potrebné používať pokročilé nástroje jazyka Python.

Na začiatku treba poznamenať, že všetky typy Vecí budú rozširovať základnú triedu `Thing` alebo iné typy Vecí a všetky typy Svetov budú rozširovať základnú triedu `World` alebo iné typy Svetov.

4.1 Udalosti

Jedným z najzaujímavejších problémov pri implementácii je návrh mechanizmu priraďovania udalostí k funkciám, ktoré na ne reagujú. Jednoduchým riešením tohto problému by bolo vytváranie týchto priradení počas inicializácie objektu. V tomto prípade, by sme ale oddelili definíciu funkcie od informácií, kedy sa jej telo používa, čo by mohlo byť pre začínajúceho programátora mätúce. Potrebujeme zabezpečiť, aby informácie o udalostiach boli v blízkosti definície. Nástrojom umožňujúcim spracovať funkciu v jazyku Python tesne po jej definícii sú dekorátory. Tie k funkciám pridajú užitočné informácie, ktoré neskôr spracujeme pomocou metatried. Najskôr si však tieto koncepty predstavíme.

4.1.1 Dekorátory

Dekorátory v jazyku Python sú funkcie, ktoré modifikujú iné funkcie. Modifikácia nastáva aplikovaním dekorátora na funkciu. Dekorátor je teda funkcia, ktorej jediný argument aj návratová hodnota sú funkcie. Keďže jazyk Python podporuje definíciu funkcie v tele inej funkcie a každá funkcia je objektom, definícia dekorátorov je možná bez nutnosti pridávania špeciálnej syntaxe. Zaujímavá je však syntax umožňujúca aplikovať dekorátor na práve definovanú funkciu. Stačí uviesť symbol `@` nasledovaný názvom dekorátora tesne pred definíciou funkcie. Týmto vznikne nová definícia, takže postup je možné reťaziť.

Za symbolom `@` je okrem názvu dekorátora možné uviesť aj volanie funkcie, ktorého výslednou hodnotou je dekorátor. Takýmto spôsobom dokážeme dekorátory parametrizovať.

4.1.2 Metatriedy

V jazyku Python je každá hodnota objektom a teda inštanciou nejakej triedy. Keďže trieda je tiež hodnota, aj ona je inštanciou nejakej triedy nazývanej metatrieda. Python definuje základnú metatriedu `type`, ktorú je však možné rozšíriť a tak modifikovať správanie sa triedy. Pri definícii triedy je možné uviesť aj jej metatriedu.

Metatrieda sa inštancuje po tom, čo interpreter prečíta definíciu nejakej triedy. Konštruktoru je poskytnutý zoznam všetkých atribútov vytváranej triedy. Je teda možné tieto atribúty spracovať, zmeniť, alebo dokonca vytvoriť nové.

4.1.3 Implementácia udalostí

Každá udalosť je jednoznačne identifikovaná nejakým objektom jazyka Python. Fakt, že funkcia je reakciou na nejakú udalosť, označí programátor dekorátorom `event` parametrizovaným identifikátorom tejto udalosti, ktorý

```
def event(name):
    def decorator(f):
        if not hasattr(f, "_gaminator_events"):
            f._gaminator_events = []
        f._gaminator_events.append(name)
        return f
    return decorator


class _ThingType(type):
    def __init__(cls, name, bases, dct):
        super(_ThingType, cls).__init__(name, bases, dct)
        cls._gaminator_events = defaultdict(list)
        for k in dct:
            cls._new_attribute(k, dct[k])

    def _new_attribute(cls, key, value):
        if hasattr(value, '__call__') and hasattr(value, '_gaminator_events'
    ):
            for event in value._gaminator_events:
                cls._gaminator_events[event].append(key)
            del value._gaminator_events
```

Výpis programu 4.1: Vytvorenie slovníka udalostí

aplikuje na danú funkciu. Naším cieľom bude túto informáciu spracovať a zaručiť aby sa táto funkcia zavolala pri každom vyvolaní udalosti.

Prvým krokom bude pre každý nový typ Veci (triedu rozširujúcu Vec) vytvoriť slovník, ktorého kľúče budú identifikátory udalostí a ktorého hodnoty budú zoznamy funkcií reagujúcich na danú udalosť. Dekorátor `event` modifikuje funkciu zapísaním svojho argumentu (identifikátora udalosti) do špeciálneho atribútu tejto funkcie. S využitím metatriedy `ThingType` potom pri inštancovaní triedy prejdeme zoznam jej atribútov a vyberieme z nich všetky funkcie spracované dekorátorom. S ich pomocou potom vytvoríme požadovaný slovník a nastavíme ho ako atribút práve vytváranej triede. Zdrojový kód tohto procesu možno vidieť vo výpise 4.1.

Novú udalosť nebudeme spracovávať okamžite po vyvolaní, ale až počas nasledujúceho taktu. Navyše, programátor môže vyvolať udalosť, ktorá nastane až o určitú dobu. Z tohto dôvodu si vyvolané udalosti budeme udržiavať

```
def _tick_events(self):

    while self._events_queue and self._events_queue[0][0] <= self.time:
        (_time, _id, event, args, kwargs) = self._events_queue[0]
        for cls in self._things_by_class:
            if isinstance(cls, _ThingType):
                for fname in cls._gaminator_events[event]:
                    for thing in self._things_by_class[cls]:
                        calls.append((getattr(thing, fname), args, kwargs))
        for cls in self.__class__.mro():
            if isinstance(cls, _ThingType):
                for fname in cls._gaminator_events[event]:
                    calls.append((getattr(self, fname), args, kwargs))
        heapq.heappop(self._events_queue)

    for f, args, kwargs in calls:
        f(*args, **kwargs)
```

Výpis programu 4.2: Spracovanie udalostí

v prioritnej fronte radenej podľa času, kedy udalosť nastane. V každom takte potom spracujeme všetky udalosti, ktoré mali nastať v čase skoršom alebo rovnom aktuálnemu.

Väčšina udalostí nastáva pre všetky Veci v aktuálne zobrazenom Svete. Aby sme ich boli schopní spracovať, potrebujeme pre každý Svet poznať všetky Veci daného typu, ktoré sa v ňom nachádzajú. Takéto informácie si môžeme udržiavať a meniť pri vložení alebo výbere Veci zo Sveta. Pri spracovaní udalosti teda prejdeme všetky typy Vecí v aktuálne zobrazenom Svete a pre každú ich inštanciu spustíme všetky funkcie reagujúce na aktuálnu udalosť (výpis 4.2).

Udalosti nastav a zrážka nastávajú pre špeciálnu množinu Vecí a teda ich spracovanie treba implementovať zvlášť. Myšlienkou sa však nelíšia od zvyšku udalostí.

Identifikátory udalostí sa viažu na meno funkcie, nie na jej hodnotu. Pre začiatočníka nebude tento fakt znamenať žiadny rozdiel a pokročilému programátorovi to umožní efektívne používať dedičnosť.

4.2 Interaktívny režim

Jednoduchý spôsob implementácie interaktívneho interpretera by bolo spustiť herný cyklus v samostatnom vlákne. Ak by sme potom knižnicu inicializovali v štandardnom interaktívnom interpreteri jazyka Python, knižnica by bežala nezávisle od interpretera a všetko by zdanlivo fungovalo. Tento prístup má však niekoľko problémov. Zmeny spôsobené interpreterom by mohli nastať kedykoľvek, napríklad počas vykonávania nejakej udalosti. Toto by mohlo spôsobiť neočakávané správanie programu. Príkazy z interaktívneho interpretera teda potrebujeme spúšťať v nami kontrolovanom čase.

Našťastie je interaktívny interpreter jazyka Python prístupný prostredníctvom štandardných knižníc. Vieme si ho teda prispôsobiť a spustiť podľa vlastnej logiky.

Interaktívny interpreter spustíme v samostatnom vlákne, ktoré umožní zadanie a preloženie príkazu nezávisle od behu hry. Vykonanie príkazu však bude zaobalené do synchronizovaného bloku, ktorého spustenie bude môcť nastať iba v momente, keď to v hernom cykle povolíme. Na vytvorenie takejto logiky poskytuje Python dostatočne silné nástroje. Skrátený zdrojový kód tejto logiky možno nájsť vo výpise 4.3.

4.3 Lokalizácia knižnice

Štandardné lokalizačné nástroje umožňujú jedine lokalizáciu textových konštánt v programe. Na lokalizáciu identifikátorov si budeme musieť implementovať vlastný mechanizmus. Ešte pred tým však potrebujeme rozhodnúť, v ktorej časti spracovávania kódu lokalizácia nastane.

Programy jazyka Python sú uložené v súboroch, ktoré je možné upravovať ľubovoľným textovým editorom. Podľa úvah z časti 2.6 nie je v tomto prípade vhodné lokalizovať priamo zdrojový kód hry. Lokalizovať teda budeme zdrojový kód knižnice.

Implementovať lokalizáciu knižnice počas jej inicializácie je v jazyku Pyt-

```
class _GaminatorInteractiveConsole(InteractiveConsole):
    def __init__(self, game):
        self.game = game
        locals = {}
        exec('from gaminator import *', locals)
        InteractiveConsole.__init__(self, locals)

    def runcode(self, code):
        with self.game._lock:
            InteractiveConsole.runcode(self, code)


def interact(game):
    def run():
        try:
            _GaminatorInteractiveConsole(game).interact()
        finally:
            with game._lock:
                game.end()
    Thread(target=run).start()
```

Výpis programu 4.3: Synchronizovaný interaktívny interpreter

hon možné, avšak náročné a náchylné na chybu. Oveľa jednoduchší je preklad počas inštalácie, pričom pre každú lokalizáciu nainštalujeme vlastnú verziu knižnice. Programátor sa potom bude môcť rozhodnúť, ktorú verziu použije.

Počas inštalácie je teda potrebné vytvoriť niekoľko nezávislých verzií zdrojového kódu, ktoré sa budú líšiť verejnými identifikátormi. Možným riešením tohto problému je obyčajné nahradenie pôvodných identifikátorov prekladmi. Aby sme zabránili kolíziám mien, bude každý identifikátor, ktorý je určený na preloženie, zložený zo špecifického prefixu, voliteľného oboru a anglickej lokalizácie tohto identifikátora. Takéto identifikátory bude potom navyše možné detekovať prostredníctvom regulárnych výrazov a tak automatizovane vytvárať šablóny pre lokalizačné súbory.

Kapitola 5

Testovanie a budúcnosť knižnice

Po úspešnej implementácii prebehlo testovanie knižnice na žiakoch osemročných gymnázií. V tejto kapitole zhodnotíme priebeh a výsledky tohto testovania ako aj súčasný stav knižnice Gaminator. Na základe týchto informácií potom navrhneme možné budúce vylepšenia.

5.1 Aktuálny stav knižnice

Ku dňu odovzdania práce sa knižnica Gaminator nachádza vo verzii `0.1.0`. Jej zdrojový kód je zverejnený prostredníctvom služby GitHub na adrese `http://github.com/syslo/gaminator`. Knižnica je taktiež prístupná v štandardnej databáze knižníc pre jazyk Python `http://pypi.python.org/pypi/gaminator`. Existuje webová stránka knižnice `http://people.ksp.sk/~sysel/gaminator`, na ktorej sa okrem základnej dokumentácie nachádza aj inštalačný súbor pre systém Windows. Zdrojový kód s inštalačným súborom je taktiež možné nájsť v elektronickej prílohe k tejto práci.

Knižnica implementuje kompletnú funkcionalitu popísanú v kapitole 3. Súčasťou knižnice je aj modul `starter`, ktorý umožňuje spustenie interaktívneho interpretra s aktívnym Svetom bez nutnosti písať akýkoľvek kód.

Taktiež poskytuje implementácie niekoľkých základných typov Vecí, ktoré slúžia na demonštráciu práce s objektmi a Vecami.

5.2 Testovanie knižnice

V dňoch 19.5.2015 až 22.5.2015 prebehlo v priestoroch Gymnázia na Párovskej ulici v Nitre testovanie knižnice Gaminator. Testovania sa zúčastnili žiaci osemročného gymnázia so zameraním na matematiku, z toho jedna žiačka tretieho ročníka, osem žiakov štvrtého ročníka a jedna žiačka piateho ročníka. Účelom testovania bolo zistenie schopnosti žiakov pracovať s knižnicou v časovom rozmedzí veľmi podobnom akcii Jesenná Škola KSP, ktorá je hlavnou motiváciou práce.

5.2.1 Postup testovania

Testovanie prebiehalo na neformálnej úrovni. Žiakom boli postupne vysvetlené jednotlivé koncepty knižnice a zadaná úloha, ktorej riešenie tieto koncepty vyžadovalo. Každý žiak programoval samostatne, bolo im však umožnené medzi sebou komunikovať a navzájom si pomáhať. Testovanie bolo rozdelené do štyroch dní po štyroch vyučovacích hodinách. Na konci testovania vyplnili žiaci anketu, ktorej výsledky kvôli nízkemu počtu respondentov posúdime skôr kvalitatívne ako kvantitatívne. Väčšina žiakov pred testovaním programovala iba v prostrediach Baltík alebo Imagine Logo, pričom svoje skúsenosti hodnotili ako základné.

5.2.2 Prvý deň

Deň bol zameraný na zoznámenie sa s interaktívnym interpreterom, jazykom Python a knižnicou Gaminator. Najskôr boli predstavené výrazy a premenné, potom kreslenie prostredníctvom interaktívneho interpretera a nakoniec práca s editorom, kedy sa kresliace príkazy presunuli do súboru.

Žiaci konceptom porozumeli a boli s nimi schopní pracovať. Zaujala ich interaktivita jazyka, kreslenie vnímali skôr ako prirodzený prostriedok pri výučbe. Šikovnejším žiakom bola vysvetlená práca s cyklami, čo im umožnilo kresliť pravidelné vzory.

5.2.3 Druhý deň

Deň sa sústredil na Veci a Svety. Na začiatku im boli tieto koncepty predstavené prostredníctvom modulu `starter`, kedy v interaktívnom interpreteri sledovali reakciu objektov na zmenu ich vlastností. Neskôr implementovali vlastné Veci a Svet dopisovaním kódu do predpripravenej šablóny. Šablónu je možné nájsť v elektronickej prílohe k tejto práci. Úlohou bolo vytvoriť akvárium, v ktorom hráč ovláda rybu snažiacu sa jesť padajúcu potravu a vyhýbať sa žralokovi.

Koncepty Vecí, Svetov a objektov sa vo všeobecnosti zdali žiakom jednoduché. Ich precvičenie prostredníctvom modulu `starter` však nebolo dostatočné, nakoľko pri vypĺňaní šablóny mali problémy so správnou syntaxou aj napriek jasnej predstave riešenia jednotlivých problémov. Postupom času sa však všetkým podarilo splniť základné požiadavky a mnohým sa podarilo implementovať bonusovú funkcionalitu kopírovaním a modifikovaním existujúceho kódu.

5.2.4 Tretí deň

Počas tretieho dňa boli vysvetlené Obrázky a knižnicou generované udalosti. Taktiež sa pokračovalo v rozvoji konceptu Vecí a Svetov. Žiaci pracovali so svojou hrou z predchádzajúceho dňa.

Úroveň samostatnosti žiakov sa začala výraznejšie líšiť. Šikovnejší z nich boli schopní pracovať samostatne a problémy dokázali riešiť spoluprácou. Zvyšok však potreboval výraznejšiu asistenciu, pričom ich problémy súviseli so slabým precvičením si konceptov Vecí a Svetov. Väčšina žiakov mala problémy s porozumením niektorým chybovým hláškam jazyka Python.

5.2.5 Štvrtý deň

Posledný deň boli predstavené všeobecné udalosti a práca so zásobníkom Svetov. Žiaci ďalej pracovali na svojich hrách a posilňovali si porozumenie jednotlivým konceptom. Vytvoriť pozorovanie z tohto dňa je náročné, nakoľko sa každý žiak venoval svojej práci na vlastnej úrovni.

5.2.6 Výsledky testovania

Testovanie poskytlo užitočné informácie o knižnici pozorovaním problémov žiakov ako aj anketou, ktorú žiaci na konci vyplnili.

Počas testovania sa nevyskytovali problémy s inštaláciou knižnice, používaním editora, intraktívneho interpretra alebo spúšťaním programov. Výrazy jazyka Python, jednoduché premenné alebo kreslenie tiež nespôsobovali problémy, aj keď sa vyskytli sťažnosti na súradnicový systém začínajúci v ľavom hornom rohu obrazovky.

Koncepty objektov, Vecí, Svetov a udalostí boli pre žiakov náročnejšie napriek tomu, že sa im zo začiatku zdali intuitívne. Podcenená príprava na tieto koncepty spôsobila zbytočné problémy počas ďalšieho programovania. Myšlienky Vecí, Svetov a udalostí sa však žiakom páčili.

Časté problémy sa vyskytovali najmä pri odhaľovaní chýb. Žiaci nie vždy rozumeli chybovým hláškam jazyka Python.

Vo všeobecnosti boli všetci žiaci schopní implementovať jednoduchú hru pomocou knižnice Gaminator. S knižnicou boli podľa vyjadrení v ankete spokojní. Niekoľko zaujímavých postrehov na vylepšenia z ich strany spomenieme v nasledujúcej časti.

5.3 Návrhy na ďalšiu prácu

Z aktuálneho stavu knižnice ako aj z testovania vyplývajú nasledovné možnosti na vylepšenie knižnice Gaminator:

- Rozšírenie modulu `starter` o nové objekty. Modul sa ukázal ako veľmi užitočný pri predstavovaní dôležitých konceptov knižnice.
- Pokrytie častých začiatočníckych chýb vlastnými chybovými hláškami.
- Možnosť pozastavenia generovania udalosti krok (navrhnuté žiakmi počas testovania).
- Podpora práce s myšou.
- Všeobecné rozšírenie knižnice o podporu nových typov multimédií.

Záver

Podarilo sa nám navrhnúť a implementovať knižnicu Gaminator, ktorá je pripravená uľahčiť výučbu programovania ako aj vytváranie jednoduchých hier. Produkt je propagovaný prostredníctvom webovej stránky poskytujúcej základné informácie o jeho používaní a možnostiach inštalácie.

Návrhu knižnice predchádzal výskum v oblasti sofvéru na výučbu programovania. Predstavili sme si niekoľko aktuálnych softvérových možností. Pri každej možnosti sme popísali niekoľko zaujímavých konceptov, ktoré slúžili ako inšpirácia pri ďalšej práci. Druhá časť prípravy spočívala vo výskume faktorov, ktorými bude knižnica vplývať na výučbu programovania. Ozrejmili sme dôležitosť týchto faktorov a navrhli možnosti ako ich optimalizovať.

Podnet na vytvorenie knižnice pochádza z akcie Jesenná Škola KSP, ktorá sa pokúša motivovať žiakov k programovaniu v netradičnom časovom rozmedzí. Na základe tohto podnetu a predošlého výskumu sme stanovili požiadavky, ktorých sme sa držali pri jej návrhu a implementácii. Výsledkom je knižnica v modernom skriptovacom jazyku Python zameranom na čitateľnosť kódu, ktorá umožňuje implementáciu hier prostredníctvom malého množstva intuitívnych konceptov. Taktiež poskytuje interaktívny interpreter, možnosť lokalizácie identifikátorov a jednoduchý inštalačný proces na najčastejšie používaných platformách. Môžeme teda skonštatovať, že sa nám stanovené požiadavky podarilo naplniť.

Knižnicu sa nám podarilo otestovať na žiakoch osemročných gymnázií. Výsledkom testovania bolo potvrdenie schopností začiatočníkov vytvárať hry prostredníctvom knižnice ako aj odhalenie niektorých jej nedostatkov. Dô-

ležitý výsledok testovania je potreba rozšírenia modulu `starter`, ktorý poskytuje predprogramované objekty pre knižnicu a tak umožňuje skúmať vlastnosti knižnice prostredníctvom interaktívneho interpretera bez nutnosti vlastného programovania. Pridaním ďalších objektov by sme boli schopní dosiahnuť vytváranie jednoduchých hier priamo v interpreteri.

Ako sme už spomínali v úvode, súčasný stav knižnice nie je považovaný za finálny. Okrem návrhov na vylepšenie vyplývajúcich z testovania je tiež možné rozširovať multimediálnu funkcionalitu knižnice a tak umožniť vytváranie plnohodnotnejších hier. Zdrojový kód Gaminatora je spravovaný prostredníctvom služby GitHub, ktorá umožňuje spoluprácu pri vytváraní voľne šíriteľných programov. Pri implementácii boli dodržiavané štandardy programovania v jazyku Python. Knižnica je teda pripravená na prípadný budúci rozvoj.

Elektronická príloha

Súčasťou práce je aj elektronická príloha obsahujúca nasledujúce súbory:

1. priečinok `gaminator` – kompletný zdrojový kód knižnice, verzia 0.1.0, aj s licenciou
2. `ryba_template.py` – šablóna pre hru, ktorú žiaci implementovali počas testovania
3. `setup.exe` – inštalačný súbor pre systém Windows

Literatúra

[1] Oficiálna dokumentácia knižnice turtle v jazyku Python. `https://docs.python.org/2/library/turtle.html`. Posledný prístup: 31.5.2015.

[2] Oficiálna stránka jazyka Python. `https://www.python.org/`. Posledný prístup: 31.5.2015.

[3] Oficiálna stránka Korešpondenčného Seminára z Programovania. `https://www.ksp.sk/`. Posledný prístup: 31.5.2015.

[4] Oficiálna stránka Logo Foundation. `http://el.media.mit.edu/logo-foundation/`. Posledný prístup: 31.5.2015.

[5] Oficiálna stránka prostredia Scratch. `https://scratch.mit.edu/`. Posledný prístup:31.5.2015.

[6] Esteban Walter Gonzalez Clua. A game oriented approach for teaching computer science. In *Anais do XXVIII Congresso da SBC*, pages 10–19, 2008.

[7] Arman Kamal, Md Nuruddin Monsur, Syed Tanveer Jishan, and Nova Ahmed. Chascript: Breaking language barrier using a bengali programming system. In *Electrical and Computer Engineering (ICECE), 2014 International Conference on*, pages 441–444. IEEE, 2014.

[8] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for

novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.

[9] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: a sneak preview [education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*, pages 104–109. IEEE, 2004.

[10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[11] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

[12] M. Synovcová, J. Kofránek, and V. Houf. *Martina si hraje s počítačem: 107 programů pro robota Karla : pro děti od 8 let*. Albatros, 1989.

[13] Niklaus Wirth. Pascal and its successors. In *Software pioneers*, pages 108–119. Springer, 2002.