Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA

Peter Bašista

Vizualizace algoritmů

Katedra aplikované matematiky

Vedúci bakalárskej práce: Mgr. Martin Mareš, Ph.D.
Študijný program: Informatika (B1801), obecná informatika

2009

Chcel by som sa poďakovať Mgr. Martinovi Marešovi, Ph.D. za cenné pripomienky pri zostavovaní práce a za námet a pomoc pri vytváraní programu.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a s jej zverejňovaním.

V Prahe dňa 6. augusta 2009 Peter Bašista

Obsah

Abstrakt **7**

Úvod **9**

1 Porovnanie vizualizačných systémov **11**
1.1 Výber vhodného vizualizačného systému . . . 11

2 Algorithm Animation Library **13**
2.1 Predstavenie . . . 13
2.2 Princíp vytvárania animácie . . . 13
2.3 Objekty a ich atribúty . . . 14
2.4 Identifikácia objektov . . . 15
2.5 Závislosti . . . 15
2.6 Formáty výstupu . . . 17
2.7 Postup pri používaní . . . 17

3 Vizualizačné komponenty **19**
3.1 Label . . . 19
3.2 Vertex . . . 20
3.3 Node . . . 21
3.4 Edge . . . 22
3.5 Array . . . 23
3.6 Forest . . . 23
3.7 Graph . . . 25
3.8 Vlastné vizualizačné komponenty . . . 26

4 Príklady použitia **28**
4.1 Animácia triedenia . . . 28
4.2 Animácia vkladania prvkov do AVL stromu . . . 29
4.3 Animácia prehľadávania grafu do šírky . . . 30
4.4 Animácia prehľadávania grafu do hĺbky . . . 30

5 Inštalácia a používanie **32**
5.1 Inštalácia v OS Linux . . . 32
5.2 Inštalácia v OS Windows . . . 33
5.3 Demonštračný program . . . 33
5.4 Jednoduchý príklad použitia . . . 34
5.5 Vytvorené obrázky a videá . . . 38

Literatúra **39**

Názov práce: Vizualizace algoritmů
Autor: Peter Bašista
Katedra (ústav): Katedra aplikované matematiky
Vedúci bakalárskej práce: Mgr. Martin Mareš, Ph.D.
E-mail vedúceho: *mj@ucw.cz*

Abstrakt: Cieľom práce je navrhnúť program pre vizualizáciu činnosti algoritmov pri výuke. Užívateľ vytvorí algoritmus a pripojí k jeho datovým štruktúram pripravené vizualizačné komponenty. Program potom zaznamená priebeh výpočtu algoritmu a zobrazí ho ako interaktívnu animáciu, prípadne vytvorí súbor PDF zachytávajúci kľúčové body výpočtu.

V predloženej práci sa zameriavame na predstavenie knižnice Algorithm Animation Library. Táto knižnica bola navrhnutá tak, aby umožnila užívateľovi pokiaľ možno čo najväčšiu flexibilitu pri tvorbe animácií, a zároveň tak, aby jej ovládanie bolo používateľsky čo najprijateľnejšie. V práci vysvetlíme základné princípy jej fungovania a používania, v krátkosti ju porovnáme s alternatívami a vyslovíme závery o tom, kedy je vhodné knižnicu Algorithm Animation Library použiť.

Kľúčové slová: vizualizácia algoritmov, objektový systém, grafická knižnica Cairo

Title: Visualisation of algorithms
Author: Peter Bašista
Department: Department of Applied Mathematics
Supervisor: Mgr. Martin Mareš, Ph.D.
Supervisor's e-mail address: *mj@ucw.cz*

Abstract: The goal of this work is to design a computer program for visualisation of algorithms and their execution in a classroom. The user writes the algorithm and attaches the prepared visualisation components to its data structures. The program will then record the progress of the computation and displays it as an interactive animation, alternatively it creates a PDF document with the key moments of the animation.

In the presented work we introduce the Algorithm Animation Library. This library has been designed to be as flexible as possible, and at the same time as easy and friendly to use as possible. We present our approach towards animation of algorithms, briefly compare the developed library with the alternatives and conclude on when it is convenient to use the Algorithm Animation Library.

Keywords: visualization of algorithms, object framework, Cairo graphics library

Úvod

Hlavnou inšpiráciou pri tvorbe knižnice Algorithm Animation Library bola potreba čo najjednoduchšie, ale pritom s možnosťou čo najväčšej flexibility vytvárať animácie jednoduchých algoritmov pri výuke. V súčasnosti existujú už predpripravené animácie pre najrôznejšie algoritmy. Získať ich je veľmi jednoduché, pretože sa častokrát nachádzajú na webe ako applety do webového prehliadača. Požiadavka vytvoriť animačný systém, ktorý by umožňoval animovať takmer úplne ľubovoľné algoritmy sa teda môže zdať z tohoto pohľadu neopodstatnená.

Užívateľ si však veľmi často potrebuje animáciu algoritmu prispôsobiť na mieru. Chce napríklad poukázať na nejaký špeciálny prípad či zdôrazniť bežne nesledované vlastnosti. Napokon nemusí byť spokojný ani s dostupným grafickým vyhotovením animácie algoritmu, ktorý potrebuje. Taktiež mu nemusí vyhovovať spôsob, akým je možné animáciu ovládať a predvádzať. Máloktorý animačný systém alebo program pre animáciu konkrétneho algoritmu umožňuje užívateľovi až takú veľkú flexibilitu pri tvorbe a prezentácii animácie. A práve toto prázdne miesto sa snaží zaplniť animačný systém Algorithm Animation Library.

Jeho veľmi veľkou výhodou je výborná flexibilita. Je totiž postavený na jednoduchom objktovom modeli, ktorý užívateľovi umožňuje obmieňať a nastavovať si veľmi detailne všetky časti animácie. Je to docielené používaním jednoduchých vizualizačných komponentov, ktoré sa dajú veľmi dobre prispôsobovať na najrôznejšie použitie. Každý vizualizačný komponent obsahuje totiž atribúty, ktorých nastavenie ovplyvňuje jeho vzhľad a správanie sa. Naviac je možné nadefinovať si rôzne závislosti medzi atribútmi jednotlivých objektov, a tak automaticky uchovávať celú animovanú scénu v „konzistentnom stave“. Užívateľ je teda od tejto činnosti oslobodený a môže sa naplno venovať vytváraniu zaujímavých častí animácie.

Prispôsobovať sa dá nielen grafický vzhľad animovanej scény, ale aj spôsob prezentácie výstupu. Knižnica Algorithm Animation Library umožňuje animáciu predvádzať priamo na monitore, alebo vygenerovať dokument s kľúčovými bodmi animácie. Taktiež umožňuje vygenerovať sériu obrázkov, ktoré sa následne môžu enkódovať na video.

Veľká flexibilita a široké možnosti pri vytváraní animácie však môžu znamenať aj veľkú náročnosť jej prípravy. Knižnica Algorithm Animation Library sa snaží túto náročnosť čo najviac znížiť. Užívateľovi ponúka štyri základné vizualizačné komponenty: Vertex (vrchol), Edge (hrana medzi vrcholmi), Node (samostatne stojaci typ vrchola) a Label (popisný text). Pomocou týchto komponentov sa dajú dobre vizualizovať najmä algoritmy s grafmi, stromami, poľom a podobné.

Táto knižnica obsahuje naviac aj tri pomocné, kontajnerové typy vizualizačných komponentov. Jedná sa o Array (pole), Forest (les, množina stromov) a Graph (všeobecný orientovaný graf). Tieto kontajnerové vizualizačné komponenty sú implementované pomocou základných vizualizačných komponentov. Ich úlohou je zjednodušiť užívateľom často vyžadované operácie pri vytváraní animácií algoritmov s poľom, stromami či grafmi. Takýmito operáciami sú napríklad: zmena poradia prvkov v poli, umiestnenie vrcholu na

správne miesto v strome či získanie zoznamu všetkých susedov istého vrchola v grafe.

Užívateľ nie je obmedzený iba na vizualizačné komponenty implementované priamo knižnicou Algorithm Animation Library. Pre implementáciu vlastných, či už základných alebo kontajnerových, typov vizualizačných komponentov sú vytvorené veľmi dobré podmienky.

Táto knižnica používa na reprezentáciu týchto komponentov triedy. V ich hierarchii sa nachádza okrem iného základný abstraktný typ *DrawableObject* (vykresliteľný objekt), ktorého potomkami sú všetky vizualizačné komponenty. Vytvorením novej triedy, ktorá bude potomkom triedy DrawableObject, si teda užívateľ môže vytvoriť vlastný vizualizačný komponent, ktorý bude presne odpovedať jeho potrebám a predstavám. Vďaka tomu ho potom nemusí prácne vykresľovať pomocou štandardných vizualizačných komponentov.

V tejto práci predstavíme knižnicu Algorithm Animation Library. Rozoberieme hlavné princípy jej fungovania a objasníme základný návrh a štruktúru tejto knižnice. Vysvetlíme postup pri vytváraní animácie a jej následnom prezentovaní v rôznych formách výstupu. Na záver uvedieme inštrukcie pre nainštalovanie tejto knižnice a ukážeme niektoré príklady jej použitia.

Kapitola 1

Porovnanie vizualizačných systémov

V tejto kapitole by sme sa radi venovali porovnaniu knižnice Algorithm Animation Library s inými systémami na vytváranie animácii algoritmov. Medzi nimi by sme v prvom rade chceli spomenúť animačný systém Algovision [1], ktorého autorom je prof. RNDr. Luděk Kučera, DrSc.

Ide o komplexný systém, ktorý umožňuje animovať veľké množstvo algoritmov a detailne sledovať ich priebeh. V porovnaní s knižnicou Algorithm Animation Library ide o veľmi rozsiahly program. Obsahuje veľmi veľké množstvo predpripravených vizualizácií pre veľa najznámejších algoritmov. Jeho silnou stránkou je hlavne prispôsobenosť na použitie pri výuke. Algoritmy, ktoré predvádza sú dobre komentované a postup pri ich prezentácii sleduje najmä pochopenie princípu ich fungovania. Slabšou stránkou tohoto systému je však to, že užívateľovi nedáva k dispozícii také rozmanité možnosti na prispôsobenie vizualizácie, aké by niekedy potreboval.

To je jednou zo spoločných čŕt mnohých vizualizačných systémov. Ponúkajú animácie, ktoré sú pevne nastavené a dajú sa v nich meniť iba dopredu zvolené vlastnosti. Spôsob prezentácie majú tiež veľmi často obmedzený napríklad pevným rozlíšením či nemožnosťou spätnej animácie. Užívateľ je niekedy samozrejme spokojný s tým, čo je dostupné a jednoducho to použije.

Avšak nie je to tak stále. Veľmi náročný užívateľ si môže pripraviť a naprogramovať celú vizualizáciu algoritmu úplne sám a na mieru si ju prispôsobiť. To je však nie príliš častý prípad. Nedá sa očakávať, že by niekto takto postupoval stále, keď bude potrebovať vizualizovať nejaký algoritmus.

Veľmi výhodné je mať k dispozícii taký animačný systém, ktorý ponúka širokú škálu možností nastavenia animácie a zároveň je jeho ovládanie primerane jednoduché. Animačná knižnica Algorithm Animation Library sa snaží ponúknuť istý medzistupeň medzi predpripravenými animáciami, ktoré je možné len veľmi málo prispôsobovať a medzi vytvorením a naprogramovaním si vlastnej animácie. V zásade sa síce pri použití knižnice Algorithm Animation Library musí tiež vytvoriť program, avšak väčšina vizualizačnej práce už nie je na starosti užívateľa.

Vďaka vizualizačným komponentom je užívateľ od samotnej vizualizácie pri používaní tejto knižnice takmer úplne oslobodený. A keďže je možné definovať si závislosti, tak sa typicky musí starať naozaj iba o samotný priebeh animácie.

1.1 Výber vhodného vizualizačného systému

Keďže užívatelia majú typicky rozdielne požiadavky na vzhľad, formu výstupu či možnosti ovládania animácie, je ťažké odporúčať jeden animačný systém, ktorý by sa hodil na

všetko a bol by v každej situácii tým najlepším riešením. Je však možné urobiť aspoň akúsi klasifikáciu týchto systémov, na základe ktorej už bude možné niektoré z nich pre konkrétneho užívateľa odporučiť.

Najčastejšie sa vyskytujúcim typom animačných systémov sú jednoúčelové programy, typicky ponúkajúce animáciu jediného algoritum. Spravidla sú veľmi jednoduché na používanie a od užívateľa nevyžadujú až na vstupné dáta takmer žiadne nastavenie. Preto sú veľmi vhodné na rýchle zoznámenie sa s prezentovaným algoritmom. Ich vizualizačné schopnosti sú však dopredu pevne nastavené a v prevažnej väčšine prípadov ich nie je možné meniť. To však pre užívateľov, ktorí hľadajú okamžite prístupnú vizualizáciu, nie je typicky rozhodujúce, a teda je pre nich takýto typ animačného systému úplne prijateľný.

Ak je teda požiadavkou rýchlo prístupná a okamžitá animácia, vtedy nie je účelné, aby užívateľ siahal po zložitejších systémoch. V takomto prípade sa nedá použitie knižnice Algorithm Animation Library odporúčať.

Ďalším typom vizualizačných systémov sú väčšie animačné knižnice a programy. Tie typicky umožňujú animáciu viacerých algoritmov, často z jednej tematickej oblasti. Existujú však aj také, ktoré obsahujú celú škálu rôznych algoritmov, napríklad už spomínaný program Algovision [1].

Takéto systémy často používajú podobný štýl vizualizácie pre všetky algoritmy, ktorých animáciu ponúkajú. Nie je to však problémom, pretože tieto algoritmy často spolu súvisia a tým pádom ich spoločný štýl prispieva k vytváraniu väzieb medzi nimi. Často obsahujú aj názorný popis či vysvetlenie fungovania príslušných algoritmov. Sú teda v značnej miere prispôsobené na výuku.

Väčšinou sa však takéto systémy nezaoberajú vizuálnou stránkou až tak podrobne a neumožňujú jej detailnejšie prispôsobenie. To môže síce niektorým užívateľom vadiť, ale veľká väčšina z nich je s takýmito systémami spokojná, pretože im ide skôr o prezentáciu algoritmu ako o jeho vizualizáciu.

Ani v tomto prípade teda nie je vhodné odporúčať použitie knižnice Algorithm Animation Library. Užívatelia, ktorí hľadajú takýto typ vizualizácie algoritmov si totiž typicky nechcú alebo nepotrebujú animáciu príliš prispôsobovať a až tak ich nezaujíma vyhotovenie samotnej vizualizácie.

Extrémnym prípadom z opačnej strany sú komplexné systémy na vytváranie animácií a grafiky, ktoré umožňujú veľmi pokročilé grafické stvárnenie nielen algoritmov. Medzi takéto systémy patrí napríklad známy Adobe Flash [2] a iné. Ich nevýhodou je však to, že nie sú uspôsobené na vytváranie vizualizácií algoritmov. Postupovať sa teda musí od úplných základov.

Pre užívateľov, ktorí potrebujú vytvoriť veľmi podrobne nastavenú animáciu nejakého algoritmu sú takéto systémy jedným z riešení. Ak však títo užívatelia nepožadujú až takú veľkú škálu možností a vedia sa uspokojiť s predpripravenými vizualizačnými komponentami, vtedy je na mieste odporučiť im použitie knižnice Algorithm Animation Library. Vďaka závislostiam je práca s ňou menej náročná ako práca s komplexnými animačnými systémami, a to najmä pri automatickom vytváraní väčšieho počtu objektov a závislostí.

Kapitola 2

Algorithm Animation Library

V tejto kapitole bližšie predstavíme knižnicu Algorithm Animation Library, vysvetlíme jej základnú štruktúru a ozrejmíme princípy jej fungovania.

2.1 Predstavenie

Knižnica Algorithm Animation Library je objektovým systémom, ktorý slúži na vytváranie animácií. Pozostáva z objektov — *vizualizačných komponentov* (kapitola 3). Tieto objekty sú hlavným nástrojom tvorby *scény*, alebo jedného dejstva animácie.

Každé dejstvo animácie sa z pohľadu knižnice Algorithm Animation Library skladá zo série *snímok*. Tieto snímky predstavujú presné kópie stavov scény z času, kedy boli vytvorené. Pri vytváraní animácie sa dá nastaviť, koľko snímok sa má za jednu sekundu animácie vytvoriť. Prednastavená hodnota je 30 snímok za sekundu.

Niektoré snímky majú špeciálny význam, pretože začínajú nový úsek animácie. Takéto snímky sú považované za *kľúčové*. Pri generovaní série snímok sa stále prvá snímka označí za kľúčovú a ostatné snímky ostanú obyčajnými. Dodatočne je však tiež možné určiť, ktoré z vygenerovaných snímok budú kľúčové.

Z užívateľského pohľadu prebieha vytváranie snímok nasledovne: Najskôr sa scéna uvedie do takého stavu, v ktorom má byť na začiatku ďalšej časti animácie. Ďalej sa nastavia prípadné *závislosti* (kapitola 2.5), ktoré majú byť počas tejto časti animácie v platnosti. Zjednodušene môžeme povedať, že tieto závislosti sú vlastne funkcie, ktoré sa vykonávajú pri každom posune času scény. Záverečným krokom je spustenie *generovania* snímok pre požadovane dlhý časový úsek animácie.

Celý priebeh animácie je viazaný na *čas*. Každý konkrétny čas je ďalej prostredníctvom aktuálneho nastavenia počtu snímok za sekundu viazaný na konkrétne číslo snímky. Animácia každej scény sa začína v čase $t = 0$. Aktuálny čas scény sa potom zvyšuje s každou vygenerovanou snímkou.

2.2 Princíp vytvárania animácie

Pri generovaní animácie pomocou knižnice Algorithm Animation Library sa aktuálny stav scény priebežne kopíruje a vytvárajú sa snímky. Tie sú viazané na konkrétny čas animácie.

Aby bolo vôbec možné generovať animácie, ktoré nepozostávajú iba zo série staticky pripravených snímok, knižnica Algorithm Animation Library umožňuje užívateľovi pristupovať k aktuálnemu času scény a následne aj na základe jeho hodnoty stav scény meniť. To znamená, že užívateľ môže vytvoriť závislosť, ktorá bude na základe aktuálneho času

meniť vlastnosti vybraných objektov. Takto sa môže napríklad vytvoriť animácia pohybu, zmena veľkosti či farby, ktorá bude závislá na čase. Následne sa iba spustí generovanie snímok a predtým nastavené závislosti sa postarajú o to, aby sa požadované vlastnosti korektne menili počas celého času animácie.

Najčastejším druhom zmeny stavu scény počas jedného úseku animácie je *lineárna* zmena niektorých vlastností vizualizačných komponentov. Pre zjednodušené vytváranie takýchto zmien stavov obsahuje táto knižnica funkciu, ktorá závislosti pre lineárne zmeny vygeneruje automaticky.

Najprv sa označí stav scény, ktorý bude počiatočný pre ďalší úsek animácie. Potom sa vykonajú požadované zmeny scény a nakoniec sa spustí lineárna interpolácia a generovanie snímok. Automaticky vytvorené lineárne závislosti sa pritom postarajú o to, aby sa stav scény lineárne zmenil z počiatočného na koncový.

Súčasná implementácia uchováva každú snímku ako úplnú kópiu všetkých vizualizačných komponentov, ktoré sa v scéne nachádzali v čase jej vzniku. Tieto snímky sú potom pri prezeraní animácie na monitore načítavané z pamäte a vykresľované. Vďaka tomu je okrem iného možné prehrávať animáciu aj v opačnom smere.

Knižnica Algorithm Animation Library umožňuje, aby sa pri generovaní snímok zároveň vytváral výstup aj v iných formátoch. Je možné vytvárať obrázky vo formáte PNG alebo SVG z každej snímky animácie, alebo iba z kľúčových snímok. Tiež je možné vygenerovať dokument vo formáte PDF alebo PostScript obsahujúci všetky kľúčové snímky animácie.

2.3 Objekty a ich atribúty

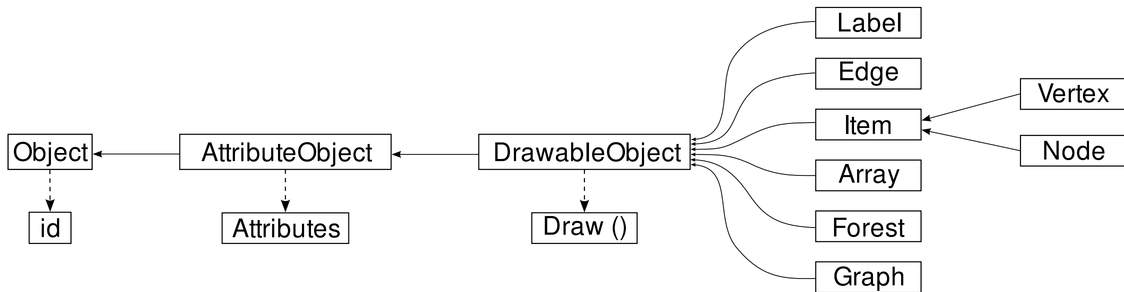
Každý typ vizualizačného komponentu, ktorý sa používa v tejto knižnici má svoje špecifické vlastnosti, a preto je reprezentovaný vlastnou triedou. Reprezentantom jedného konkrétneho vizualizačného komponentu v animácii je inštancia takejto triedy.

Každá takáto inštancia sa vyznačuje svojimi osobitnými vlastnosťami. Sú to napríklad poloha, veľkosť, tvar, farba či popisný text. Tieto vlastnosti sa v rámci knižnice Algorithm Animation Library uchovávajú v špeciálnych objektoch, takzvaných *atribútoch*.

Atribút je objekt, ktorý pre jednu inštanciu triedy nejakého vizualizačného komponentu uchováva jednu dátovú položku. Oproti bežným dátovým položkám tried majú atribúty tú výhodu, že nemusí byť už pri kompilácii pevne dané, ktoré atribúty existujú a aký majú typ. Súčasná implementácia poskytuje atribúty, ktoré sú schopné uchovávať štandardné dátové typy `int`, `double` a `string`. Tiež sú k dispozícii atribúty uchovávajúce vlastné dátové typy `RGBColor` a `RGBAColor`, ktoré slúžia na uchovávanie nepriehľadných, respektíve priehľadných farieb.

Triedy reprezentujúce vizualizačné komponenty sú odvodené z triedy DrawableObject, teda vykresliteľný objekt. Táto trieda je odvodená z triedy *AttributeObject*, čiže atribútový objekt, alebo objekt, ktorý má atribúty. To znamená, že každý vykresliteľný objekt v tejto knižnici obsahuje aj atribúty. Tie sú v rámci jedného atribútového objektu jednoznačne identifikované podľa názvu.

Pre lepšiu názornosť je na obrázku 2.1 zachytená časť hierarchie objektov používaných v knižnici Algorithm Animation Library.



Obrázok 2.1: Zjednodušená hierarchia vizualizačných komponentov

2.4 Identifikácia objektov

Vizualizačné komponenty sú v rámci jednej scény jednoznačne identifikované číslom určujúcim poradie pridania ku objektom scény. Toto číslo sa chápe ako jednoznačný identifikátor, ID každého objektu v scéne. Na jeho základe je možné získať ukazovateľ na ľubovoľný objekt, ktorý sa predtým pridal medzi objekty scény. Nie je teda nutné si pri používaní pamätať ukazovatele na všetky objekty, s ktorými potrebujeme pracovať. Stačí poznať ich ID.

Prístup k objektom scény je riešený cez medzivrstvu identifikátorov hlavne preto, aby bolo možné jednoznačne rozlišovať objekty aj po ich skopírovaní do vytvorených snímok a asociovať ich s objektami v pôvodnej scéne. Pri prezeraní animácie sa totiž pracuje už iba s vygenerovanými snímkami, a tie obsahujú kópie pôvodných objektov. Užívateľ by čelil veľkému problému, ak by sa pokúšal asociovať vytvorené kópie objektov s pôvodnými objektami v scéne, najmä ak by ich tam bolo veľa rovnakého typu. Vďaka jednoznačnej identifikácii každého objektu, ktorá sa skopírovaním nezmení, je takáto asociácia veľmi jednoduchá. Na jej základe môže napríklad užívateľ skontrolovať, či sa vo vytvorených snímkach animácie nachádzajú naozaj objekty, ktoré tam očakáva a či majú tie správne hodnoty atribútov.

2.5 Závislosti

Pri animácii algoritmov sa často stáva, že zatiaľ čo nejaký objekt počas animácie mení svoje atribúty, niektoré iné objekty musia na túto zmenu reagovať zmenou svojich atribútov. Napríklad pri animácii pohybu vrchola v grafe musí každej zmene jeho polohy odpovedať aj príslušná zmena polôh všetkých hrán, ktoré sú s daným vrcholom incidentné. Bolo by veľmi nepríjemné a prácne, ak by sa užívateľ musel starať o všetky takéto zmeny sám. Preto knižnica Algorithm Animation Library prináša *závislosti*.

Sú to tiež objekty, odvodené z triedy Dependence, teda závislosť. Ich hlavným obsahom je ukazovateľ na funkciu, ktorá je vykonávateľom danej závislosti. Takáto funkcia môže napríklad načítať aktuálnu polohu vrchola v grafe a podľa nej upraviť polohu konca s ním incidentnej hrany.

Takéto závislosti môžu veľmi zjednodušiť vytváranie animácií. Pomocou nich je možné automaticky udržiavať scénu v „konzistentnom stave“. Užívateľ teda môže vytvárať animáciu tak, že bude vykonávať iba kľúčové zmeny v scéne a nechá závislosti, aby dokončili všetko ostatné.

Vytvorené závislosti sa používajú pri každej zmene scény, až kým nie sú zo zoznamu závislostí odstránené. V prípade, že je závislostí viac, tak sú vykonávané v topologicky správnom poradí.

Toto korektné topologické usporiadanie závislostí sa vytvorí automaticky. Využíva sa pri tom precedenčný graf, ktorý sa medzi závislosťami udržiava na základe toho, ako sa ich funkcie správajú, teda ktoré atribúty čítajú a ktoré zapisujú.

Vďaka atribútom implementovaným pomocou objektov je možné prístup k nim kontrolovať. Funkcie uložené v závislostiach sa zavolajú najskôr „naprázdno" (teda všetky zmeny, ktoré vykonajú budú hneď zrušené), a tým sa získa zoznam atribútov, ktoré jednotlivé závislosti čítali a ktoré zapisovali.

Na základe týchto informácií sa následne vytvorí už spomínaný precedenčný graf. Pri jeho vytváraní sa sledujú atribúty, ku ktorým pristupuje viacero závislostí.

Každý takýto atribút je potenciálnym zdrojom konfliktu. Mohlo by sa totiž stať, že sa hodnota atribútu najprv prečíta jednou závislosťou a až potom sa pozmení inou. To znamená, že by mohli vznikať nekonzistentné stavy, kedy by závislosť, ktorá čítala atribút ako prvá, pracovala s jeho neaktuálnou hodnotou. Jedinou výnimkou je, ak jeden atribút číta viacero závislostí, ale žiadna ho nezapisuje. V tomto prípade konflikt nenastáva a takýto atribút nepridá do výsledného precedenčného grafu medzi závislosťami žiadnu hranu.

Vo všetkých ostatných prípadoch sa do tohoto grafu pridá hrana alebo hrany zaručujúce, že závislosť, ktorá sledovaný atribút zapisuje sa vykoná skôr ako všetky závislosti, ktoré tento atribút čítajú.

Pre lepšiu názornosť uvedieme príklad. Predpokladajme, že užívateľ vytvoril scénu pozostávajúcu z dvoch objektov — vrchola a hrany. Jeden koniec hrany pritom naviazal na vrchol. Ďalej definoval závislosť, ktorá na základe času nejakým spôsobom mení súradnice vrchola.

Precedenčný graf sa v tomto prípade bude vytvárať nasledovne: V scéne sú prítomné dve závislosti — prvou z nich je závislosť definovaná užívateľom a druhou je závislosť, ktorá vznikla pri naviazaní hrany na vrchol. Tieto dve závislosti sa teda najskôr „naprázdno" použijú. Týmto použitím sa zistí, že prvá závislosť nečíta žiaden atribút a zapisuje súradnice vrchola. Druhá závislosť zase číta súradnice vrchola a jeho polomer a zapisuje súradnice naviazaného konca hrany. Teda potenciálny konflikt nastáva v dvoch atribútoch — horizontálnej a vertikálnej súradnici vrchola.

Napĺňanie precedenčného grafu bude prebiehať nasledovne: Na základe prvého z týchto atribútov sa tam pridá hrana, ktorá vynúti, aby sa závislosť pohybujúca vrcholom vykonala skôr ako závislosť nastavujúca polohu hrany. Na základe druhého atribútu by sa mala do precedenčného grafu pridať tá istá hrana, čo znamená, že sa graf už nezmení.

Následne sa precedenčný graf topologicky zotriedi, čo je v tomto prípade triviálne, pretože obsahuje iba dva vrcholy a jednu hranu. Teda výsledné topologické poradie závislostí bude nasledovné: Najskôr sa má vykonať závislosť, ktorá mení polohu vrchola a po nej závislosť, ktorá nastavuje polohu naviazaného konca hrany.

Závislosti sa teda budú v tomto poradí vykonávať počas celej nasledujúcej časti animácie.

V prípade, že jeden atribút zapisuje viacero závislostí nastáva situácia, ktorá sa nedá automaticky korektne vyriešiť, pretože na to nie je dostatočné množstvo informácií.

Avšak už samotná existencia takýchto závislostí znamená logickú chybu pri ich definícii. Ak je jeden atribút zapisovaný dvoma rôznymi závislosťami, tak namiesto nich môže užívateľ vytvoriť jedinú, ktorá bude sama zapisovať pôvodný atribút. Nie stále je to jednoduché, ale všeobecným pravidlom pri definovaní závislostí by malo byť, že každý atribút môže zapisovať nanajvýš jedna závislosť. Tým sa jednoducho dá zabezpečiť korektnosť pri určovaní poradia ich vzájomného vykonávania.

Ak sa napriek všetkému v precedenčnom grafe nájde cyklus, knižnica Algorithm Animation Library to nijako nerieši. Vypíše sa akurát varovanie, ale precedencia závislostí v cykle ostane neurčená. Nie je zaručené žiadne konkrétne poradie použitia týchto závislostí. Užívateľ má teda hlavnú zodpovednosť za to, že naraz nebudú v scéne existovať závislosti, ktoré by boli vo vzájomnom konflikte. V opačnom prípade musí počítať s prípadnou nekonzistentnosťou pri ich aplikovaní.

Použitím takýchto závislostí sa teda dajú docieliť ľubovoľné animácie, ktoré sú popísateľné funkciami. Je však nutné poznamenať, že nie je prípustné, aby sa tieto funkcie správali úplne ľubovoľne.

Je to kvôli správnemu určeniu precedencií medzi závislosťami. Musí totiž platiť, že ak sa vykonajú dve rôzne volania funkcie na úplne rovnaký stav scény a v rovnaký čas, tak každé takéto zavolanie zanechá scénu v úplne rovnakom stave. Ak by to nebolo zaručené, tak by súčasná implementácia určovania precedencií medzi závislosťami nefungovala správne.

2.6 Formáty výstupu

Vykresľovanie je v knižnici Algorithm Animation Library riešené pomocou grafickej knižnice Cairo [3]. Táto knižnica umožňuje veľmi jednoduché kreslenie dvojrozmerných scén, pričom používa unifikované rozhranie pre kreslenie na všetky typy výstupov. Je teda rovnako jednoduché výstup vykresliť na obrazovku ako vytvoriť sériu obrázkov alebo dokument s kľúčovými snímkami animácie.

V súčasnosti umožňuje knižnica Algorithm Animation Library zobraziť vygenerovanú animáciu priamo na monitore, vygenerovať Portable Document Format (PDF), prípadne PostScript dokument s kľúčovými snímkami animácie alebo vygenerovať PNG či SVG obrázky s kľúčovými alebo so všetkými snímkami animácie.

2.7 Postup pri používaní

Knižnica Algorithm Animation Library sa primárne používa s jazykom C++, v ktorom je napísaná. Pre jej užívateľa je podstatné, že v zdrojovom kóde programu, ktorý túto knižnicu bude používať je potrebné použiť direktívu:

```
#include <aal.h>
```

ktorá zabezpečí vloženie deklarácií všetkých potrebných objektov a funkcií. Pri kompilácii je tiež potrebné odovzdať kompilátoru informáciu o umiestnení hlavičkového súboru `aal.h`. Linker zase musí obdržať informáciu o umiestnení knižnice `aal`, a to pre účely samotného linkovania, ako aj následného spúšťania.

Všetky tieto kroky sú vo forme prepínačov pre kompilátor GCC [7] súčasťou Makefile, ktorý je obsiahnutý spolu so zdrojovými kódmi ukážkovej aplikácie v distribúcii knižnice Algorithm Animation Library.

V zásade je možné postup pri použití tejto knižnice zhrnúť v nasledujúcich bodoch:

- Vytvorenie vizualizačných komponentov, ktoré sa budú používať.
- Nadefinovanie stále platných, invariantných závislostí medzi týmito objektami.
- Postupné uvádzanie scény do všetkých požadovaných stavov, s prípadným doplňaním či odstraňovaním závislosti alebo aj objektov.

Veľmi často je postačujúce, ak sa zmena medzi jednotlivými kľúčovými stavmi animácie vykoná jednoduchou lineárnou interpoláciou medzi hodnotami všetkých atribútov všetkých objektov v týchto kľúčových stavoch. Ako už bolo spomenuté, knižnica Algorithm Animation Library ponúka možnosť automatického vytvorenia závislostí pre takúto zmenu.

Najskôr sa zavolaním funkcie `int Scene::SetLinearInterpolationInitialState (void)` na objekte aktuálnej scény nastaví súčasný stav scény ako východzí pre lineárnu interpoláciu. Následne sa môžu vykonať ľubovoľné zmeny atribútov ľubovoľných vizualizačných komponentov tak, aby sa scéna uviedla do stavu, v ktorom má byť po skončení lineárnej interpolácie. Na záver sa jednoduchým zavolaním funkcie `int Scene:: LinearInterpolation (time dt)` vygenerujú snímky pre požadovanú lineárnu interpoláciu z nastaveného východzieho stavu do aktuálneho stavu za požadovaný čas.

Ak sa východzí stav lineárnej interpolácie dopredu nenastaví, tak sa ním rozumie aktuálny stav scény. Konečný stav pre lineárnu interpoláciu sa v tom prípade získa jedným použitím súčasných závislostí v aktuálnom čase. Následne sa vykoná lineárna interpolácia medzi týmito dvoma stavmi.

Takéto chovanie je výhodné napríklad vtedy, keď sa do scény pridávajú ďalšie objekty. Ak sa týmto objektom nastavujú závislosti, ktoré hneď pri prvom použití výrazne zmenia niektoré ich atribúty, znamenalo by to v animácii výrazný skok. Použitie lineárnej interpolácie je jednou z možností, ako takúto zmenu zmierniť, prideliť jej istý časový úsek a spraviť z nej súčasť animácie.

Kapitola 3

Vizualizačné komponenty

Hlavnými a najčastejšie používanými objektami knižnice Algorithm Animation Library sú vizualizačné komponenty. V tejto kapitole si ich bližšie predstavíme, uvedieme zoznam ich atribútov či prípadných špeciálnych funkcií a typické oblasti ich použitia.

Pri používaní týchto vizualizačných komponentov nie je nutné nastavovať všetky ich atribúty. Postačí, ak sa nastavia iba tie atribúty, ktoré užívateľ využije. V tom prípade však hodnoty všetkých nenastavených atribútov ostanú také, na aké boli v jednotlivých objektoch inicializované. To znamená typicky nulové čísla, čierne a úplne priehľadné farby a prázdne reťazce.

Užívateľ má tiež možnosť nastaviť vizualizačným komponentom aj ľubovoľné iné atribúty, nielen tie, ktoré sú uvedené v nasledovnom popise. Je teda možné si v týchto objektoch uchovávať aj nejakú dodatočnú informáciu, ktorá sa bude môcť neskôr využiť. Jedinou podmienkou je, aby názvy doplnených atribútov neboli rovnaké ako názvy už používaných atribútov.

Všetky súradnice, vzdialenosti a veľkosti používané nasledujúcimi vizualizačnými komponentami sa udávajú v *bodoch*[1]. Nie je však nutné pracovať iba s celočíselnými hodnotami, a preto sa ukladajú v atribútoch typu `double`. Pre veľkosť písma platí, že sa udáva ako vzdialenosť medzi dvoma susednými *referenčnými čiarami*[2] textu. Všetky používané reťazce vyžadujú kódovanie UTF-8.

3.1 Label

Ide o vizualizačný komponent predstavujúci jednoduchý popisný text. Pri animácii algoritmov sa asi najčastejšie využije pre zobrazenie názvov, popisov jednotlivých objektov, informácie o aktuálnom stave algoritmu či vysvetliviek.

Okrem polohy a samotného textu umožňuje tento komponent nastaviť aj základné atribúty fontu, ako je štýl, veľkosť, farba, sklon či váha písma. Pre nastavenie týchto vlastností obsahuje tento vizualizačný komponent nasledujúce atribúty:

[1]Konkrétna podoba týchto bodov sa líši použitým výstupným zariadením. Na monitore sú to napríklad pixely. Nemusí však nutne platiť, že jeden užívateľom používaný bod odpovedá jednému bodu na výstupnom zariadení. Predvolené nastavenie je však presne takéto. Zmeniť túto transformáciu je možné pomocou funkcií grafickej knižnice Cairo.

[2]Je to pokus o preklad názvu *baseline*. Ide o myslenú čiaru, na ktorej sú v riadku umiestnené písmená.

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`x`	horizontálna súradnica stredu textu
double	`y`	vertikálna súradnica stredu textu
double	`fontsize`	veľkosť písma
string	`text`	zobrazený text
int	`font_slant`	sklon písma (prípustné hodnoty: 0 — normálne písmo, 1 — sklonené písmo, 2 — kurzíva)
int	`font_weight`	váha písma (prípustné hodnoty: 0 — normálna váha, 1 — tučné písmo)
RGBAColor	`fontcolor`	farba písma

Vizualizačný komponent Label neobsahuje žiadne špecifické funkcie. Pri použití sa zadávajú súradnice stredu textu, čo je výhodné najmä pre malé nápisy, ktoré sa často požadujú vycentrované, pretože popisujú nejaký objekt. To ale na druhej strane znamená, že tento vizualizačný komponent nie je vhodný na zobrazovanie dlhších textov. Knižnica Algorithm Animation Library využíva iba takéto jednoduché zobrazovanie textu. Pre potreby animácie algoritmov to však vo väčšine prípadov postačuje.

3.2 Vertex

Tento vizualizačný komponent je navrhnutý tak, aby poskytoval pohodlnú, jednoduchú a dobre vyzerajúcu vizualizáciu vrchola v grafe. Jeho použitie nie je obmedzené iba na vizualizáciu grafov, ale hodí sa prakticky na každý účel, ktorý vyžaduje zobrazenie guľôčky s prípadným popisom. Veľkou výhodou je, že na tento vizualizačný komponent môžu byť automaticky napojené *hrany* (kapitola 3.4). To umožňuje jednoduchú vizualizáciu grafov, stromov, prípadne vývojových diagramov či iných podobných štruktúr.

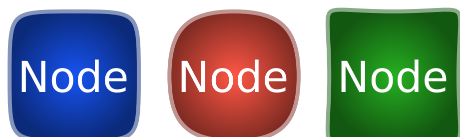
Tento vizualizačný komponent umožňuje nastavenie svojej polohy, veľkosti, farby a popisného textu. Pre nastavenie týchto vlastností obsahuje nasledujúce atribúty:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`x`	horizontálna súradnica stredu
double	`y`	vertikálna súradnica stredu
double	`r`	polomer
RGBColor	`cl`	farba pozadia
double	`gr`	koeficient zmeny farby pri radiálnom gradiente $[0-1]$
RGBAColor	`ocl`	farba okrajovej čiary
double	`owidth`	hrúbka okrajovej čiary
RGBColor	`lcl`	farba popisného textu
double	`lsize`	veľkosť písma popisného textu
string	`label`	popisný text

Zaujímavým atribútom je `gr`. Určuje koeficient, ktorým sa prenásobí každá zložka farby pozadia, pričom sa takto získaná farba použije ako koncová pre radiálny gradient, ktorý sa na pozadí vytvorí. Nastavenie tohoto koeficientu na 1 teda farbu nijako nezme-

ní. Vykresľovanie pozadia, ktoré obsahuje farebný prechod, je však veľmi odporúčané, pretože prispieva k lepšiemu vzhľadu tohoto vizualizačného komponentu.

3.3 Node



V porovnaní s komponentom Vertex ide o graficky pokročilejší vizualizačný komponent. Predstavuje tiež istý typ vrchola. Je vhodný najmä ako prvok poľa alebo inej štruktúry, ktorá nevyžaduje aby sa na tento komponent napájala nejaká hrana. Možnosť mať na tento vizualizačný komponent napojené hrany však v súčasnej implementácii chýba.

Dôvodom je tvar tohoto vizualizačného komponentu, ktorý sťažuje identifikáciu jeho okraja, a teda aj zistenie korektných súradníc pre koniec napojenej hrany. V nasledujúcej aktualizácii knižnice Algorithm Animation Library sa však určite počíta s pridaním implementácie tohoto rozpoznávania. Bude teda možné, aby aj vizualizačný komponent Node mohol mať na seba napojené hrany.

Pre nastavenie svojho vzhľadu disponuje vizualizačný komponent Node nasledujúcimi atribútmi:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`x`	horizontálna súradnica stredu
double	`y`	vertikálna súradnica stredu
double	`width`	šírka
double	`height`	výška
double	`roundness`	miera zaoblenia vrchola (vyššie číslo znamená viac zaoblený tvar)
RGBColor	`cl`	farba pozadia
double	`gr`	koeficient zmeny farby pri radiálnom gradiente
RGBAColor	`ocl`	farba okrajovej čiary
double	`owidth`	hrúbka okrajovej čiary
RGBColor	`lcl`	farba popisného textu
double	`lsize`	veľkosť písma popisného textu
string	`label`	popisný text

Tento vizualizačný komponent teda disponuje takmer rovnakými atribútmi ako vizualizačný komponent Vertex. Obsahuje naviac iba jediný atribút — `roundness`. Ten určuje výsledný tvar tohoto vrchola. Udáva sa v bodoch a predstavuje vzdialenosť, o ktorú sa zmení poloha kontrolných bodov kriviek definujúcich okrajovú čiaru.

Tieto referenčné body sú štyri a sú umiestnené na každej strane vrchola presne v strede. Ich kontrolné body sú umiestnené v príslušných susedných rohoch pomyselného štvorca. Nastavením kladnej hodnoty sa tieto kontrolné body posúvajú späť k svojím príslušným referenčným bodom, čím sa vrchol stáva zaoblenejším. Predvolená hodnota je nulová a vrchol sa pri nej zobrazí tak, ako prvý vrchol zľava na ilustračnom obrázku. Nastavovať sa dajú aj záporné hodnoty, pri ktorých sa vrchol stáva hranatejším až začína byť v rohoch natiahnutý.

3.4 Edge

Tento vizualizačný komponent predstavuje hranu medzi dvoma vrcholmi. Táto hrana môže mať svoje konce *naviazané* na nejaké vrcholy. V takom prípade sa konce hrany prispôsobujú polohám vrcholov, na ktoré sú naviazané. Využíva sa na to funkcia `int Edge::Bind (...)`, ktorá automaticky vytvorí príslušnú závislosť. Užívateľ si teda môže jej jednoduchým zavolaním zaistiť, aby sa hrana stále nachádzala na správnom mieste.

Vizualizačný komponent Edge poskytuje užívateľovi tieto atribúty:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`x1`	horizontálna súradnica prvého konca hrany
double	`y1`	vertikálna súradnica prvého konca hrany
double	`x2`	horizontálna súradnica druhého konca hrany
double	`y2`	vertikálna súradnica druhého konca hrany
RGBAColor	`a1cl`	farba šípky na prvom konci hrany
RGBAColor	`a2cl`	farba šípky na druhom konci hrany
RGBAColor	`lcl`	farba samotnej hrany
double	`lwidth`	hrúbka hrany
double	`a1size`	výška
double	`a1size`	dĺžka šípky na prvom konci hrany
double	`a1base`	dĺžka základne šípky na prvom konci hrany
double	`a2size`	dĺžka šípky na druhom konci hrany
double	`a2base`	dĺžka základne šípky na druhom konci hrany
int	`arrow1present`	prítomnosť šípky na prvom konci hrany (prípustné hodnoty: 0 — šípka sa nezobrazuje, 1 — šípka sa zobrazuje)
int	`arrow2present`	prítomnosť šípky na druhom konci hrany (prípustné hodnoty: 0 — šípka sa nezobrazuje, 1 — šípka sa zobrazuje)
double	`size_displacement`	veľkosť postranného posunu hrany v smere od jej prvého konca k druhému

Špeciálnu pozornosť si zaslúži atribút `size_displacement`, ktorý slúži na posunutie koncových bodov hrany v smere kolmom na smer hrany. Kladná hodnota udáva posun doprava v smere od prvého ku druhému koncu hrany. Toto posunutie zároveň zohľadňuje tvar vrchola, na ktorý je hrana naviazaná a pri posune do strany upraví polohu koncov hrany aj pozdĺžne. To sa využíva napríklad vtedy, ak v grafe potrebujeme kresliť hrany medzi jednotlivými vrcholmi aj obojsmerne a nechceme, aby sa pritom prekrývali. Jednoduchým nastavením tohoto atribútu sa hrana automaticky posunie nabok a vytvorí sa miesto pre spätnú hranu.

3.5 Array

Pole, vizualizačný komponent, ktorý je prvým z *kontajnerových* vizualizačných komponentov. Tento typ vizualizačných komponentov sa vyznačuje tým, že sa samé o sebe nevykresľujú, ale slúžia iba ako medzivrstva prístupu ku základným vizualizačným komponentom. Aby bola takáto medzivrstva užitočná, poskytuje typicky zjednodušenie niektorých operácií, ktoré by sa inak museli vykonávať zložitejšie. Umožňuje napríklad volaním jednej funkcie vykonať operáciu, ktorú by užívateľ musel inak vykonávať pomocou viacerých iných funkcií. Kontajnerový vizualizačný komponent Array nie je výnimkou.

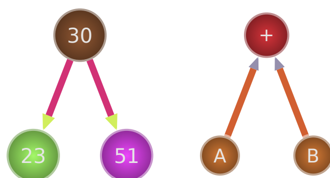
Užívateľovi prináša zjednodušenie práce s poľom objektov. Umožňuje pracovať s vrcholmi typu Node alebo Vertex. Obsahuje dve pomocné funkcie. Prvou z nich je `int Array::Add (ObjectID newitemid)`, ktorá pridá vrchol s daným identifikačným číslom na koniec poľa. Taktiež je k dispozícii funkcia `int Array::Swap (int i, int j)`, ktorá vymení vrcholy na miestach s indexami i a j. Užívateľ môže pristupovať k jednotlivým prvkom v tomto kontajnerovom objekte aj priamo, na základe indexu. Využíva sa na to verejná dátová položka `ItemVector Array::items`.

Využitie kontajnerového vizualizačného komponentu Array spočíva v tom, že automaticky, na základe aktuálneho poradia v poli, umiestňuje vrcholy, ktoré do neho boli pridané. Tým sa zjednodušuje vytváranie animácie používajúcej pole, pretože sa užívateľ nemusí starať o správne umiestnenie jednotlivých jeho prvkov a môže sa venovať iným, zaujímavejším častiam animácie. Tento objekt poskytuje, tak ako všetky ostatné vizualizačné komponenty, niekoľko atribútov, na základe ktorých sa dá jeho správanie prispôsobiť požadovanému účelu. Jedná sa o tieto atribúty:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`x`	horizontálna súradnica stredu poľa
double	`y`	vertikálna súradnica stredu poľa
double	`d`	horizontálna vzdialenosť medzi susednými prvkami

Týchto atribútov nie je veľa a poskytujú iba základné nastavenie polohy poľa a rozostupu jeho prvkov. Ostatné vlastnosti, ako napríklad popis poľa či spôsob animácie výmeny, sa v súčasnej implementácii nedajú nastaviť priamo. Užívateľ si teda musí vytvárať závislosti pre animáciu výmeny alebo iných operácií s poľom sám.

3.6 Forest



Tento vizualizačný komponent predstavuje les, teda graf, ktorý nie je nutne súvislý, ale každý jeho komponent súvislosti je strom. Prináša zjednodušenie práce s takýmito grafmi tým, že po zakorenení stromu ponúka možnosť správy závislostí medzi otcom a synom. Umožňuje určiť, ktoré z vrcholov budú koreňovými. Na to slúžia fun-

kcie `int Forest::AddRoot (...)`, `int Forest::RemoveRoot (...)` a `int Forest::ReplaceRoot (...)`.

Ďalej je možné pridávať vrcholy na voľné miesta potomkov ľubovoľného vrchola v grafe. Na to je určená funkcia `int Forest::AddChild(...)`. Zámena súčasného potomka za iného sa vykoná pomocou funkcie `int Forest::ReplaceChild(...)`.

Zrušiť vzťah medzi rodičom a potomkom je možné dvoma spôsobmi. Prvý z nich využíva funkciu `int Forest::DetachChild(...)`, ktorá iba zaručí, že príslušný potomok už nebude potomkom žiadneho vrchola v grafe. Druhý spôsob predstavuje funkcia `int Forest::RemoveChild(...)`, ktorá tento vrchol naviac vymaže z aktuálnej scény.

Vizualizačný komponent Forest umožňuje tiež overiť, či sú konkrétne vrcholy koreňovými alebo nie. Používajú sa na to funkcie `IDList Forest::GetRootIDs (void)` a `bool IsRoot (ObjectID objectid)`. Samozrejmosťou je možnosť získania zoznamu synov konkrétneho vrchola. Na to sú určené funkcie `IDList Forest::GetChildrenIDsOf (...)` a `ObjectID Forest::GetChildIDByPosition (...)`. Tiež je možné zistiť aktuálneho rodiča požadovaného vrchola, na čo slúži funckia `ObjectID GetParentOf (ObjectID childid)`.

Zoznam pomocných funkcií, ktoré ponúka vizualizačný komponent Forest je teda značne obsiahly. Tieto funkcie umožňujú jednoducho pridávať a odoberať vrcholy z konkrétneho miesta v strome, a tak značne zjednodušujú používanie stromov pri animácii. Vrcholy sú automaticky vizuálne udržiavané na správnych miestach, ktoré odpovedajú ich polohe v strome. Naviac sa medzi nimi automaticky vytvoria hrany, ktoré symbolizujú ich vzájomné vzťahy. Konkrétna podoba tohoto vyobrazenia sa dá nastaviť pomocou atribútov.

Je možné nastaviť si výšku jednej úrovne v strome, ďalej minimálny rozostup susedných vrcholov a koreňov a samotné umiestnenie celého lesa. Taktiež sa dajú nastavovať predvolené parametre hrán, ktoré sa automaticky pri pridávaní vrcholov do lesa vytvárajú. Je samozrejme možné pridané hrany dodatočne upravovať alebo úplne odstrániť.

Každému vrcholu, do ktorého vedie hrana z jeho rodiča, sa nastaví atribút typu `int` s názvom `parents_edge_id`. Hodnota tohoto atribútu predstavuje identifikátor hrany, ktorá do neho vedie z jeho rodiča. Tým je užívateľovi umožnené jednoducho k tejto hrane pristupovať a podľa potreby ju upravovať.

Nasleduje zoznam atribútov, ktoré využíva vizualizačný komponent Forest:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`levelheight`	výška jednej úrovne lesa
double	`spacing`	najnižšia povolená vzdialenosť dvoch susedných vrcholov v lese
double	`roots_x`	horizontálna súradnica stredu oblasti s koreňmi lesa
double	`roots_y`	vertikálna súradnica stredu oblasti s koreňmi lesa
RGBAColor	`edge_a1cl`	predvolená farba šípky na hornom (otcovskom) konci hrany
RGBAColor	`edge_a2cl`	predvolená farba šípky na dolnom (synovskom) konci hrany
int	`edge_arrow1present`	predvolené nastavenie prítomnosti šípky na hornom konci hrany

pokračovanie na ďalšej strane

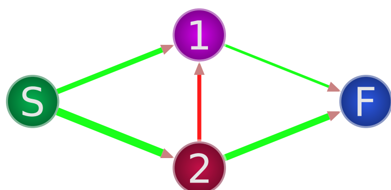
Typ atribútu	*Názov atribútu*	*Popis atribútu*
int	`edge_arrow2present`	predvolené nastavenie prítomnosti šípky na dolnom konci hrany
double	`edge_a1size`	predvolená dĺžka šípky na hornom konci hrany
double	`edge_a1base`	predvolená šírka šípky na hornom konci hrany
double	`edge_a2size`	predvolená dĺžka šípky na dolnom konci hrany
double	`edge_a2base`	predvolená šírka šípky na dolnom konci hrany
RGBAColor	`edge_lcl`	predvolená farba hrany
double	`edge_lwidth`	predvolená šírka hrany
double	`edge_side_displacement`	predvolená veľkosť stranového posunutia hrany

Pomocou atribútov `roots_x` a `roots_y` sa nastavuje pozícia stredu oblasti s koreňmi lesa. Táto oblasť je pozdĺžny pás, v ktorom sú umiestnené všetky korene daného lesa. Umiestnené sú vedľa seba tak, aby bola zachovaná minimálna vzdialenosť medzi každými dvoma susednými vrcholmi a tiež medzi koreňmi. S narastajúcou šírkou svojich podstromov sa teda postupne posúvajú ďalej od seba.

Ak je pre užívateľa nežiadúce, aby sa korene automaticky umiestňovali v koreňovej oblasti, môže túto vlastnosť zavolaním funkcie `int Forest::UnSetRootDependencies (void)` deaktivovať.

Podstromy sa vytvárajú smerom nadol, alebo v smere rastu vertikálnych súradníc. Ak užívateľ potrebuje les, v ktorom by stromy košateli smerom nahor, dá sa to docieliť nastavením zápornej hodnoty atribútu `levelheight`.

3.7 Graph



Ide o vizualizačný komponent reprezentujúci orientovaný graf. Poskytuje rozhranie pre jednoduché narábanie s typickou grafovou reláciou, ktorá určuje, ktoré dva vrcholy sú spojené hranou.

Aj tento vizualizačný komponent dáva užívateľovi k dispozícii možnosť nastavenia predvoleného vzhľadu automaticky vytváraných hrán. Tie je samozrejme možné podľa potreby neskôr upravovať alebo úplne odstrániť. Každú ďalšiu vlastnosť grafu si už však musí užívateľ spravovať sám. Teda tento vizualizačný komponent sa nijako nestará o umiestnenie vrcholov ani o ich vzhľad. Poskytuje iba rozhranie pre narábanie s grafovou reláciou, pričom automaticky vytvára a maže hrany medzi príslušnými vrcholmi.

Pridávanie hrán pritom zohľadňuje existenciu hrany v opačnom smere. V takomto prípade sa hrany automaticky nastavia tak, aby sa nevykresľovali cez seba. Užívateľ je

teda oslobodený od vykonávania tejto činnosti. Postačí ak si nastaví požadovanú vzájomnú vzdialenosť opačných hrán.

V tomto vizualizačnom komponente sa dajú nastavovať tieto atribúty:

Typ atribútu	*Názov atribútu*	*Popis atribútu*
double	`twoway_edge_displacement`	veľkosť postranného posunutia hrán, ktoré vedú opačnými smermi medzi rovnakými vrcholmi
RGBAColor	`edge_a1cl`	predvolená farba šípky na výstupnom konci hrany
RGBAColor	`edge_a2cl`	predvolená farba šípky na vstupnom konci hrany
int	`edge_arrow1present`	predvolené nastavenie prítomnosti šípky na výstupnom konci hrany
int	`edge_arrow2present`	predvolené nastavenie prítomnosti šípky na vstupnom konci hrany
double	`edge_a1size`	predvolená dĺžka šípky na výstupnom konci hrany
double	`edge_a1base`	predvolená šírka šípky na výstupnom konci hrany
double	`edge_a2size`	predvolená dĺžka šípky na vstupnom konci hrany
double	`edge_a2base`	predvolená šírka šípky na vstupnom konci hrany
RGBAColor	`edge_lcl`	predvolená farba hrany
double	`edge_lwidth`	predvolená šírka hrany
double	`edge_side_displacement`	predvolená veľkosť stranového posunutia hrany

Väčšina z týchto atribútov sa týka predvoleného vzhľadu vytváraných hrán. Odlišný je jedine atribút `twoway_edge_displacement`. Pomocou neho sa dá nastaviť, ako budú od seba vzdialené dve protichodné hrany v grafe. Je treba rozlišovať medzi týmto atribútom, ktorý nastavuje postranný posun hrany iba v prípade existencie hrán v oboch smeroch a atribútom `edge_side_displacement`, ktorý daný posun nastaví pre všetky vytvárané hrany.

3.8 Vlastné vizualizačné komponenty

Objekt typu Graph teda uzatvára zoznam dostupných predpripravených vizualizačných komponentov. Užívateľovi je však umožnené, aby si definoval aj vlastný vizualizačný komponent.

Postupuje sa pri tom tak, že sa vytvorí nová trieda, ktorá bude odvodená z triedy DrawableObject, teda vykresliteľný objekt. V tejto triede musí užívateľ implementovať virtuálnu funkciu `DrawableObject::Draw (...)`, ktorá slúži na samotné vykresľovanie konkrétneho vizualizačného komponentu. Vykresľovanie sa vykonáva prostriedkami grafickej knižnice Cairo [3].

Zjednodušene sa dá povedať, že táto knižnica poskytuje mnohé bežné vykresľovacie funkcie typu `moveto`, `lineto`, `rectangle`, `arc` a podobné. Tiež je pomocou nej možné vykresľovať krivky, robiť jednoduché, aj maticové, transformácie či na základnej úrovni pracovať s textom. Ponúka aj zaujímavé nástroje na prácu s farbami, akými sú napríklad vytváranie rôznych farebných gradientov. Viac informácií o postupe kreslenia pomocou tejto grafickej knižnice je k dispozícii na jej domovskej stránke [3] v sekcii dokumentácia.

Kapitola 4

Príklady použitia

V tejto kapitole si bližšie predstavíme niektoré príklady použitia knižnice Algorithm Animation Library. Pôjde o algoritmy, ktoré sa bežne vyučujú, a tým pádom je ich vizualizácia často požadovaná.

4.1 Animácia triedenia

Pre triedenie sme zvolili algoritmus Quicksort. Vytvorili sme referenčný objekt typu Node (kapitola 3.3), ktorý predstavuje jeden prvok v poli. Nastavili sme mu požadovaný vzhľad a všetky ostatné prvky vizualizovaného poľa sme vytvorili ako jeho kópiu. Text v tomto objekte reprezentuje číslo, ktoré sa na danej pozícii v poli nachádza.

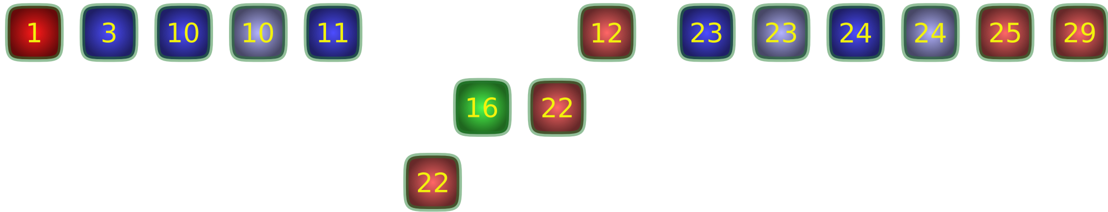
Naprogramovali sme funkciu, ktorá vykonáva jednoduché výmeny prvkov v poli na základe ich vzdialenosti za presne určený čas. Ďalej sme zabezpečili, aby sa vyznačovali aktuálne triedené partície. Počas triedenia sa taktiež zvýrazňujú už zoradené prvky a práve aktuálny pivot.

Postup vytvorenia animácie spočíval v tom, že sme si najprv vytvorili funkciu, ktorá pomocou algoritmu Quicksort triedi vstupné pole čísel. Toto vstupné pole sme asociovali s objektom typu Array tak, že sme sledovali každú zmenu poľa v pôvodnej funkcii a pridali sme k nej aj odpovedajúcu zmenu vo vizualizačnom komponente Array. Teda napríklad na miesto, kde sa v pôvodnej funkcii nachádzali príkazy, ktoré vymieňali dva prvky v poli na miestach i a j sme pridali volanie funkcie `Swap(i, j)` na príslušnom objekte typu Array. Toto volanie vymenilo prvky na mieste i a j aj vo vizualizačnom komponente.

Aby sa tieto výmeny nediali iba okamžitou zmenou zo snímky do snímky, pridali sme zároveň volanie funkcie `ObjectID WorkingContext::InsertDependence (...)`, ktorá pridala do aktuálnej scény nami predtým vytvorenú závislosť. Následne sme spustili generovanie snímok na príslušne dlhý čas, podľa vzdialenosti vymieňaných prvkov poľa. Po vygenerovaní sme predtým vytvorenú závislosť zo zoznamu závislostí odstránili.

Takto sme postupovali na každom mieste nášho pôvodného kódu, ktorý nejako upravoval obsah pôvodného poľa čísel. Tým sme docielili to, že pole čísel bolo stále s vizualizačným komponentom Array aktualizované.

Po ukončení celého triedenia boli teda všetky snímky vygenerované a animácia pripravená. Ukážka je zachytená na nasledujúcom obrázku.



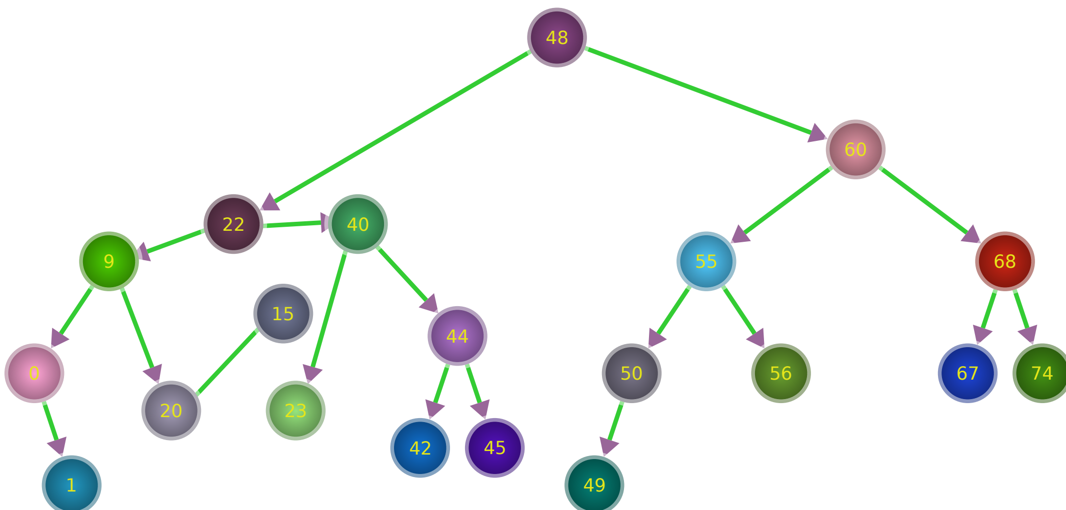
Obrázok 4.1: Ukážka z animácie triedenia

4.2 Animácia vkladania prvkov do AVL stromu

Pri vytváraní tejto animácie sme postupovali tak, že sme najskôr implementovali vkladanie do AVL stromu s použitím vlastnej dátovej štruktúry typu uzol. Keď sme mali túto implementáciu hotovú, začali sme do nej pridávať vizualizačné komponenty knižnice Algorithm Animation Library.

Použili sme vizualizačný komponent Forest, ktorý uchovával informácie o vzťahoch medzi rodičom a dieťaťom v triediacom strome. Nastavili sme predvolené atribúty automaticky vytváraných hrán a postupne sme prechádzali kód našej implementácie. Na každom mieste, kde sa stav stromu zmenil sme ho zmenili aj vo vizualizačnom komponente Forest.

Aby sme vytvorili aj animáciu a nie iba statické zmeny stavov stromu, použili sme knižničnú funkciu `int Scene::LinearInterpolation (...)`, pomocou ktorej sme si vygenerovali lineárne prechody medzi susednými stavmi stromu. Nastavili sme, aby každý takýto prechod trval dve sekundy. No a výsledkom bola animácia, ktorej ukážka sa nachádza na obrázku 4.2



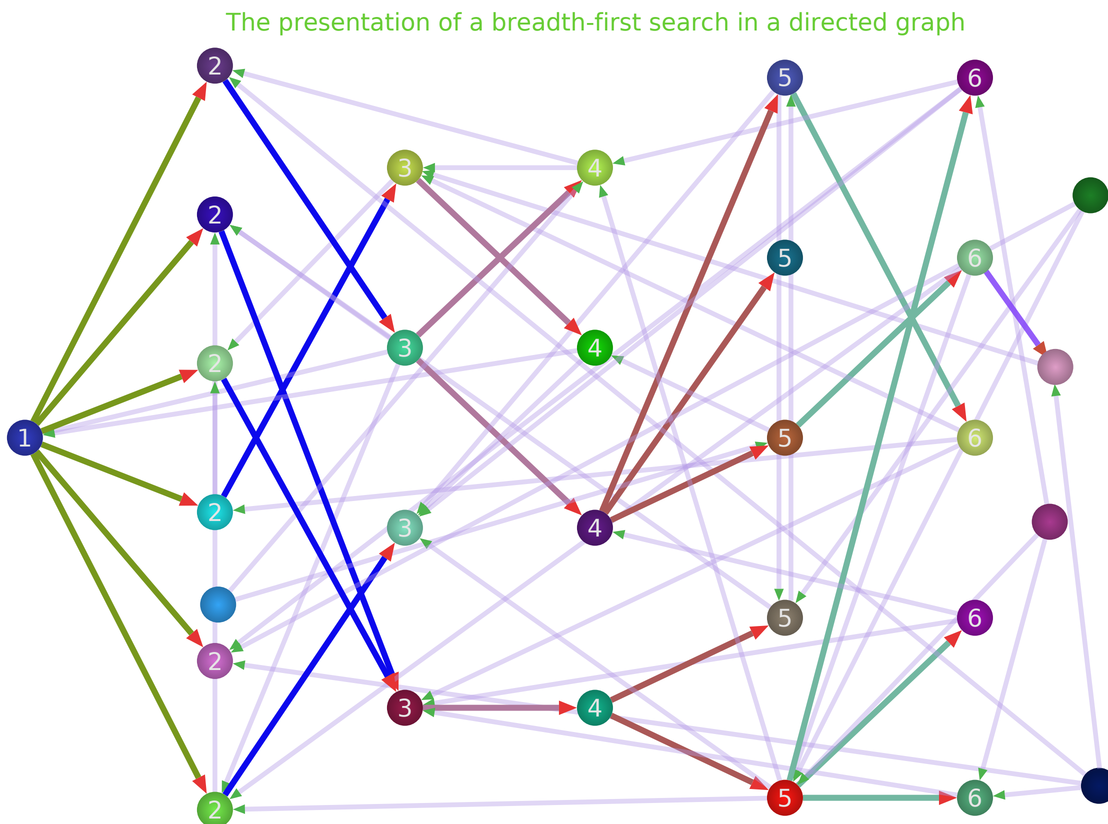
Obrázok 4.2: Ukážka z animácie vkladania do AVL stromu

4.3 Animácia prehľadávania grafu do šírky

Pri vytváraní animácie prehľadávania grafu do šírky sme postupovali tak, že sme priamo použili ponúkané schopnosti vizualizačného komponentu Graph, ktoré nám umožnili uchovávať si informácie o hranách v grafe priamo v tomto vizualizačnom komponente a jednoduchým spôsobom ich meniť. Na začiatku sme si taktiež nastavili predvolený vzhľad hrán a rozostup navzájom opačných hrán.

Na vyhľadávanie susedných vrcholov sme použili priamo funkciu `IDSet Graph::GetNeighbours (...)`. Takto sme postupne získali všetkých susedov všetkých vrcholov v aktuálnej vrstve. Vyfiltrovali sme tie, ktoré ešte neboli pridané do žiadnej predchádzajúcej vrstvy a vzápätí sme ich presunuli do pozície novej vrstvy využitím lineárnej interpolácie. Na záver presunu sme nastavili týmto vrcholom popisný text, ktorý označuje vrstvu, v ktorej sa nachádzajú.

Ukážka z takto získanej animácie je na nasledujúcom obrázku:



Obrázok 4.3: Ukážka z animácie prehľadávania grafu do šírky

4.4 Animácia prehľadávania grafu do hĺbky

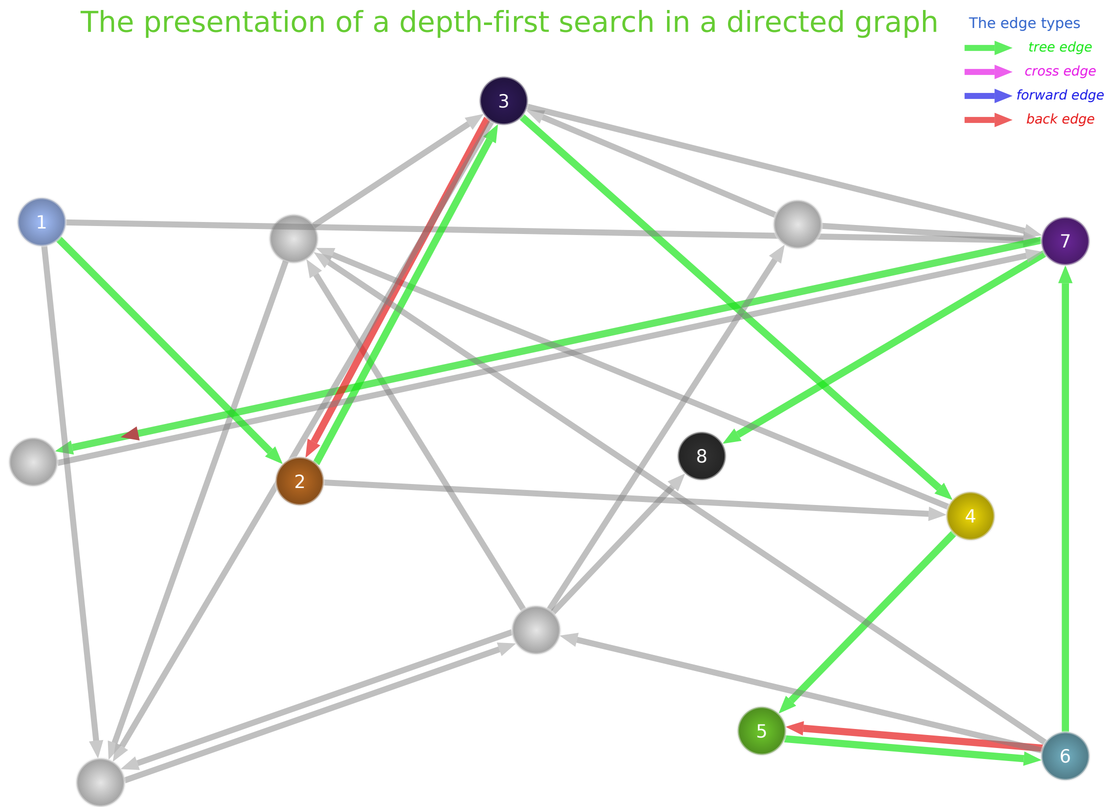
Postup pri vytváraní tejto animácie bol veľmi podobný ako pri vytváraní animácie prehľadávania do šírky. Taktiež sme si na úvod nastavili predvolené parametre hrán a ich rozostup v prípade existencie hrán oboma smermi. Následne sme spustili z jedného vrchola prehľadávanie do hĺbky.

Využívali sme pritom zásobník, do ktorého sme si ukladali ešte nespracované, ale už objavené vrcholy. Postupne sme teda vrcholy zo zásobníka vyberali a vykonávali príslušnú

animáciu. Farebne sme v grafe označovali a číslovali prejdené vrcholy. Taktiež sme farebne rozlišovali typy hrán, ktorými sme prešli. Zároveň sme maličkou šípkou stále naznačovali, kadiaľ v grafe postupujeme.

Túto maličkú šípku sme vytvorili tak, že sme definovali hranu, ktorá mala nulovú šírku čiary a bola v nej viditeľná iba jedna koncová šípka. Touto hranou sme pohybovali a znázorňovali sme tým pohyb po grafe.

Ani v tomto prípade sme nepoužili vlastné závislosti a na všetko sme vystačili s lineárnou interpoláciou. Ukážka z výslednej animácie je zachytená na obrázku 4.4.



Obrázok 4.4: Ukážka z animácie prehľadávania grafu do hĺbky

Kapitola 5

Inštalácia a používanie

V tejto kapitole popíšeme postup pri inštalácii knižnice Algorithm Animation Library na operačnom systéme Linux a naznačíme postup inštalácie aj na operačnom systéme Windows. Zároveň uvedieme aj jednoduchý príklad použitia tejto knižnice s využitím vlastných závislostí.

5.1 Inštalácia v OS Linux

Pre úspešnú kompiláciu tejto knižnice v operačnom systéme Linux je potrebné mať nainštalované nasledujúce programy a knižnice:

- Kompilátor C++, najlepšie GCC [7]
- Utilita Make, napríklad GNU Make [8]
- nástroj pkg-config[1] [9]
- vývojová distribúcia knižnice gtkmm [4] verzie aspoň 2.16
- vývojová distribúcia knižnice cairomm [5] verzie aspoň 1.6

Je potrebné poznamenať, že minimálne verzie knižníc gtkmm a cairomm uvedené vyššie predstavujú iba verzie, s ktorými bola knižnica Algorithm Animation Library otestovaná. Je pravdepodobné, že funkčná bude aj s mierne staršími verziami týchto knižníc.

Najnovšiu verziu knižnice Algorithm Animation Library je možné získať z jej domovskej stránky [15]. Po stiahnutí súboru `aal.tar` sa príkazom `tar -xvf aal.tar` tento súbor rozbalí a vypíše sa vzniknutá adresárová štruktúra. Vytvorí sa adresár `aal`, kde sa všetky súbory z tohoto archívu umiestnia.

V tomto adresári sa nachádza podadresár `lib`, v ktorom sú všetky zdrojové kódy samotnej knižnice. Ďalej sa tam nachádza podadresár `src`, kde sú zdrojové kódy pripraveného demonštračného programu.

Pre skompilovanie tohoto demonštračného programu vrátane celej knižnice stačí z adresára `aal` vyvolať utilitu Make jednoduchým príkazom `make`. Ak užívateľ potrebuje skompilovať iba knižnicu Algorithm Animation Library bez demonštračného programu, učiní tak príkazom `make lib`.

Po skompilovaní knižnice sa v adresári lib vytvorí súbor `libaal.so`. Ide o zdieľanú knižnicu, s ktorou sa budú linkovať programy, ktoré knižnicu Algorithm Animation Library používajú.

[1] Ide o nástroj, ktorý generuje prepínače potrebné pre kompiláciu a linkovanie programov využívajúcich nainštalované knižnice kompilátorom GCC.

5.2 Inštalácia v OS Windows

Pre skompilovanie knižnice Algorithm Animation Library v operačnom systéme Windows je potrebné mať nainštalované nasledujúce programy a knižnice:

- Kompilátor C++, napríklad Microsoft Visual C++ [11] alebo MinGW [12]
- vývojové verzie knižníc gtkmm a cairomm pre OS Windows[2]

Pri kompilácii pomocou systému MinGW[3] je postup inštalácie veľmi podobný postupu inštalácie na operačnom systéme Linux. Rozdielny je však samozrejme postup pri inštalácii potrebných knižníc.

Pri použití kompilátora Microsoft Visual C++ je postup v zásade odlišný, a to kvôli rozdielnym kompilačným príkazom a inému prístupu ku riadeniu kompilácie. Najčastejšie sa pri používaní vývojového prostredia Microsoft Visual C++ totiž nepoužíva utilita Make alebo nejaké jej varianty, ale priebeh kompilácie sa riadi nastavením projektu. Preto si v krátkosti uvedieme, ako je potrebné projekt, ktorý bude využívať knižnicu Algorithm Animation Library, nastaviť.

Najskôr je potrebné vytvoriť si nový, prázdny projekt, ktorý bude predstavovať samotnú knižnicu Algorithm Animation Library. Medzi hlavičkové súbory je potrebné pridať všetky súbory s príponou `h` z adresára `aal/lib/src` a medzi zdrojové súbory zase všetky súbory s príponou `cpp` z toho istého adresára.

Ďalej je potrebné zmeniť typ projektu zo spustiteľného súboru na dynamickú, respektíve statickú knižnicu, podľa požiadaviek užívateľa. No a nakoniec je nevyhnutné pridať do projektu takzvané *property sheets*[4] obsahujúce správne nastavenia pre gtkmm a cairomm. Tie sú napríklad súčasťou distribúcie gtkmm on Microsoft Windows [14]. K dispozícii sú pre Debug aj Release typy projektov a pre rôzne verzie vývojových prostredí Microsoft Visual C++ či Microsoft Visual Studio. Ich pridaním by malo byť všetko pripravené na kompiláciu knižnice Algorithm Animation Library.

Aby sa projekty, ktoré využívajú túto knižnicu kompilovali a linkovali správne, je potrebné vytvoriť závislosti medzi takýmito projektmi a projektom predstavujúcim samostatnú knižnicu. Vývojové prostredie sa potom samo postará o nastavenie vhodných parametrov pre úspešné kompilovanie a linkovanie.

5.3 Demonštračný program

Súčasťou distribúcie knižnice Algorithm Animation Library je aj demonštračný program. Po skompilovaní ho užívateľ nájde ako spustiteľný súbor s názvom `aal_demo`, respektíve `aal_demo.exe`. Po jeho spustení bez parametrov sa vypíše krátka informácia o tom, ako ho používať.

Tento program v sebe implementuje vizualizáciu už predstavených štyroch algoritmov:

- Triedenie pomocou Quicksortu
- Vkladanie prvkov do AVL stromu
- Prehľadávanie do šírky

[2]Dostupné sú napríklad v rámci distribúcie gtkmm on Microsoft Windows [14].

[3]Skratka pre „Minimalist GNU for Windows“ — je to distribúcia kompilátora GCC a aplikácií predstavujúcich GNU Binutils [13] pre použitie vo vývoji natívnych aplikácií pre operačný systém Windows.

[4]Ide o súbor nastavení obsahujúcich najmä korektné cesty ku hlavičkovým súborom a knižniciam.

- Prehľadávanie do hĺbky

Umožňuje každý z nich spustiť na náhodných dátach, ale zároveň poskytuje aj možnosť načítania vstupných dát zo súboru. Podobne je možné pre každý príklad vytvoriť výstupný súbor, ktorý obsahuje naposledy použité vstupné dáta.

Tieto súbory sú v jednoduchom textovom formáte. Pre animácie triedenia a vkladania prvkov do AVL stromu sa používajú rovnaké vstupné súbory, a teda je možné ich zamieňať. V týchto súboroch sú uložené čísla, ktoré sa nachádzajú v poli pred začiatkom triedenia alebo čísla, ktoré sa majú vkladať do AVL stromu.

Na ich prvom riadku je za znakom `#` uvedený komentár, ktorý popisuje, o aké dáta ide. Na druhom riadku sa nachádza číslo n, ktoré určuje počet zoraďovaných prvkov v poli alebo počet pridávaných prvkov do AVL stromu. Napokon na treťom riadku sa nachádza presne n čísel, ktoré predstavujú počiatočné prvky poľa alebo prvky, ktoré sa budú postupne vkladať do AVL stromu.

Podobne sa pre animáciu prehľadávania grafu do šírky a prehľadávania do hĺbky používajú rovnaké vstupné súbory, takže je ich tiež možné medzi sebou zamieňať. Tieto súbory reprezentujú graf a jeho vizuálny vzhľad.

Ich formát je nasledovný: Na prvom riadku je za znakom `#` taktiež komentár, ktorý popisuje, o aké dáta ide. Na druhom riadku je číslo n označujúce počet vrcholov v použitom grafe. Na ďalších n riadkoch sa nachádzajú súradnice stredov týchto vrcholov. Poradie, v ktorom sa konkrétny vrchol nachádza v tomto zozname určuje jeho identifikačné číslo. Na zvyšných riadkoch až do konca súboru sú uvedené dvojice čísel i a j, ktoré symbolizujú, že v grafe existuje hrana, ktorá vedie z vrchola s identifikačným číslom i do vrchola s identifikačným číslom j.

5.4 Jednoduchý príklad použitia

Pre názornosť uvedieme teraz jednoduchý príklad použitia knižnice Algorithm Animation Library. Základným krokom pri písaní programu, ktorý používa túto knižnicu je vloženie už spomínaného hlavičkového súboru `aal.h`. Teda v úvodnej časti zdrojového kódu by sa mala vyskytnúť direktíva:

```
#include <aal.h>
```

Keďže všetky objekty tejto knižnice sa nachádzajú v *priestore mien*[5] `aa`, tak všetky objekty, ktoré budeme používať musia mať pred svojim typom túto klasifikáciu.

V tomto jednoduchom príklade vytvoríme animáciu, ktorá bude predstavovať dva vrcholy spojené hranou, ktoré si navzájom vymenia pozície. Keďže nechceme, aby sa pri animácii tieto vrcholy prekrývali, musí mať každý z nich vlastnú trajektóriu výmeny. Prvý vrchol teda necháme opísať polkružnicu a zaujať miesto druhého vrchola. Ten zase lineárne presunieme na pozíciu prvého vrchola.

Najprv si musíme vytvoriť okno a následne aj *pracovný kontext*, teda objekt `wc` typu `aa::WorkingContext`, ktorý budeme pri animácii neustále používať. Tento objekt je miestom, kde sa v okne výsledná animácia zobrazí. Je teda potrebné umiestniť ho tak, aby bol pokiaľ možno v okne čo najlepšie viditeľný a mal dostatok miesta. V tomto príklade sme tento objekt umiestnili ako jediný viditeľný komponent do celého okna.

[5] Je to pokus o preklad názvu *namespace*.

```
Gtk::Main kit(argc, argv);
Gtk::Window w;
w.set_title(Glib::ustring("An exchange of two vertices"));
w.set_default_size(300, 200);
aa::WorkingContext wc;
w.add(wc);
```

V zdrojovom kóde si ďalej nastavíme veľkosť scény, s ktorou budeme pracovať a deklarujeme potrebné objekty:

```
wc.SetActiveSceneSize(300, 200);
aa::Vertex * v1 = new aa::Vertex();
aa::Vertex * v2 = NULL;
aa::Edge * e = new aa::Edge();
```

Ukazovateľ na vrchol `v2` inicializujeme iba na `NULL`, pretože ho neskôr vytvoríme ako kópiu vrchola `v1`.

Ďalším krokom je nastavenie požadovaného vzhľadu prvého vrchola `v1` a hrany `e`. To docielime nastavením ich atribútov:

```
v1->SetDouble("x", 50);
v1->SetDouble("y", 150);
v1->SetDouble("r", 30);
v1->SetDouble("gr", 0.4);
v1->SetRGB("cl", 0.3, 0.3, 1);
v1->SetRGBA("ocl", 0.9, 0.9, 0.9, 0.6);
v1->SetDouble("owidth", 7);
v1->SetRGB("lcl", 1, 1, 1);
v1->SetDouble("lsize", 35);
v1->SetString("label", "1");
e->SetDouble("lwidth", 10);
e->SetDouble("a1size", 20);
e->SetDouble("a1base", 10);
e->SetDouble("a2size", 20);
e->SetDouble("a2base", 10);
e->SetRGBA("a1cl", 0.1, 0.1, 0.9, 0.8);
e->SetRGBA("a2cl", 0.9, 0.1, 0.1, 0.8);
e->SetRGBA("lcl", 0.1, 0.9, 0.1, 0.8);
e->SetInt("arrow1present", 1);
e->SetInt("arrow2present", 1);
```

Dôležité je uvedomiť si, že nie je potrebné nastavovať polohy koncov hrany `e`. Je to kvôli tomu, že túto hranu chceme automaticky naviazať na vrcholy `v1` a `v2`, a teda o umiestnenie jej koncov sa bude starať automaticky vytvorená závislosť.

Po nastavení vzhľadu teda môžeme vytvoriť vrchol `v2` ako kópiu vrchola `v1`. Samozrejme zmeníme vrcholu `v2` nejaké atribúty, aby sme ho od vrchola `v1` odlíšili. Následne každý z týchto vizualizačných komponentov pridáme do aktuálneho pracovného kontextu a naviažeme hranu `e` na vytvorené vrcholy:

```
v2 = new aa::Vertex(* v1);
v2->SetDouble("x", 250);
v2->SetRGB("cl", 1, 0.3, 0.3);
```

```
v2->SetString("label", "2");
wc.Add(* v1);
wc.Add(* v2);
wc.Add(* e);
e->Bind((* v1), (* v2));
```

Je dôležité volať špecifické funkcie vizualizačných komponentov, ako napríklad funkciu `int aa::Edge::Bind (...)`, až potom, keď sme tieto objekty pridali do aktuálneho pracovného kontextu. Dôvodom je to, že tieto funkcie už využívajú prítomnosť aktuálnej scény. Napríklad funkcia, ktorá naväzuje vrcholy na hranu používa atribúty vrcholov, na ktoré získava ukazovatele na základe ich ID z aktuálnej scény.

Ďalším krokom bude vytvorenie objektu, ktorý bude uchovávať premenné pre funkciu určenú na vykonávanie závislosti. Deklarácia a definícia tohoto objektu vyzerá nasledovne:

```
class AnimationObject {
public:
        int move_vertices (aa::Scene * scene) {
                aa::time t(scene->GetTime());
                double progress(t / duration);
                double x(midx - cos(progress * pi) * radius);
                double y(midy - sin(progress * pi) * radius);
                v1->SetDouble("x", x);
                v1->SetDouble("y", y);
                v2->SetDouble("x", v2startxpos - 2 * progress * radius);
                return (0);
        }

        static const double pi = M_PI;
        aa::time duration;
        aa::Vertex * v1;
        aa::Vertex * v2;
        double v2startxpos;
        double radius;
        double midx;
        double midy;
};
```

Táto trieda obsahuje jedinú funkciu a niekoľko dátových položiek. Tie slúžia na uchovanie ukazovateľov na vrcholy, ktorých pohyb má byť touto funkciou animovaný. Taktiež sa pomocou nich nastavujú rôzne parametre animácie. Pri použití tohoto objektu teda najskôr vytvoríme jeho inštanciu a nastavíme požadované hodnoty jej dátových položiek:

```
AnimationObject ao;
ao.duration = 10;
ao.v1 = v1;
ao.v2 = v2;
ao.v2startxpos = 250;
ao.radius = 100;
ao.midx = 150;
ao.midy = 150;
```

Teraz už môžeme vytvoriť závislosť, ktorá využíva funkciu `int AnimationObject::move_vertices (aa::Scene * scene)`.

```
wc.InsertDependence(new aa::AdaptiveDependence(wc.GetActiveScene(),
        aa::AdaptiveDependenceInitiationFunctor(),
        sigc::mem_fun(ao, &AnimationObject::move_vertices)));
```

Signatúra tejto funkcie je veľmi dôležitá. Knižnica Algorithm Animation Library podporuje v súčasnosti dva typy závislostí. Typ, ktorý používame v tomto príklade, `aa::AdaptiveDependence`, vyžaduje od použitej funkcie, aby jej návratovou hodnotou bol `int` a jej jediným parametrom bol ukazovateľ na objekt typu `aa::Scene`. Parameter, ktorý táto funkcia pri zavolaní dostane, predstavuje ukazovateľ na aktuálnu scénu. Užívateľ ho teda môže využiť na zistenie aktuálneho času scény, získanie ukazovateľov na iné objekty podľa ich ID alebo aj na iné veci.

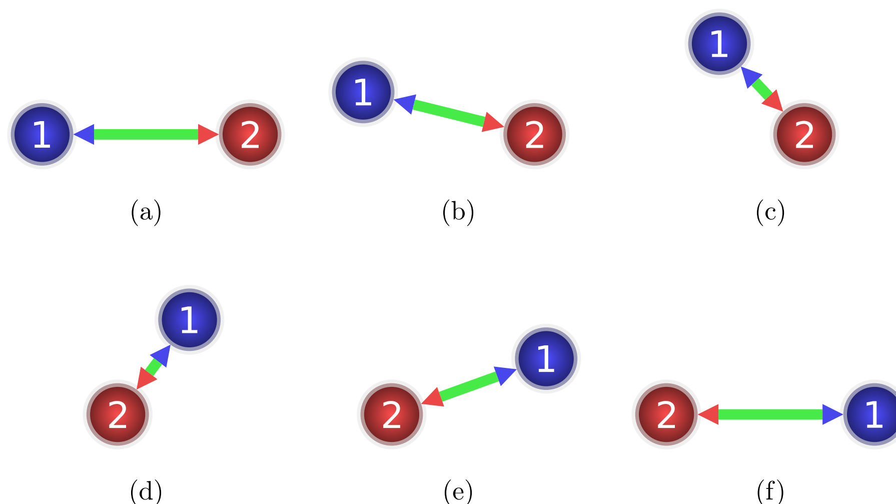
Pomocou knižnice sigc++ [6] je možné použiť aj funkcie s inou signatúrou, aká je predpísaná pre funkcie, ktoré vykonávajú závislosti. Táto knižnica totiž dokáže okrem iného pridávať funkciám fiktívne parametre či zafixovať skutočné parametre na konkrétne hodnoty. To vlastne umožňuje, aby sa ako funkcia, ktorá vykonáva závislosti, použila takmer ľubovoľná funkcia, ktorú má užívateľ k dispozícii.

Teda program, ktorý bude animovať výmenu dvoch vrcholov je už takmer hotový. Ostáva už len zavolať funkciu, ktorá vygeneruje snímky animácie:

```
wc.AdvanceTime(10);
```

Jej jediným parametrom je požadovaná dĺžka animácie v sekundách. V našom prípade sme výmenu nastavili tak, že bude trvať 10 sekúnd, a teda na rovnako dlhý čas necháme vygenerovať aj snímky.

Spustením animácie sa môžeme presvedčiť, že vytvorená závislosť naozaj funguje a že výmena prebehne v poriadku, tak ako má. V tomto konkrétnom prípade by sme dostali animáciu, ktorá je zhruba zachytená na nasledujúcej sérii obrázkov:



Obrázok 5.1: Zachytenie niektorých častí jednoduchej animácie výmeny vrcholov

5.5 Vytvorené obrázky a videá

Zdrojový kód príkladu, ktorý sme popísali v tejto kapitole, rovnako ako aj samotnej knižnice Algorithm Animation Library a skôr predstavených demonštračných príkladov sa nachádza na priloženom CD. Sú na ňom tiež umiestnené ukážky animácií vo forme vygenerovaných PDF a PostScript dokumentov, PNG a SVG obrázkov a videí. Nachádza sa tam taktiež elektronická verzia tejto práce vo formátoch PDF a PostScript.

Ďalším zdrojom ukážok animácií vytvorených pomocou tejto knižnice je jej domovská stránka [15]. Okrem aktuálnej verzie knižnice sú tam k dispozícii obrázky, PDF dokumenty a videá z animácií, ktoré boli v tejto práci predstavené.

Literatúra

[1] Kučera Luděk: *Algovision*,
http://kam.mff.cuni.cz/~ludek/Algovision/Algovision.html (6.8.2009)

[2] *Adobe Flash*,
http://www.adobe.com/products/flash/ (6.8.2009)

[3] *The Cairo Graphics Library*,
http://cairographics.org/ (6.8.2009)

[4] *gtkmm — C++ Interfaces for GTK+ and GNOME*,
http://www.gtkmm.org/ (6.8.2009)

[5] *cairomm — A C++ API for cairo*,
http://cairographics.org/cairomm/ (6.8.2009)

[6] *libsigc++ — The Typesafe Callback Framework for C++*,
http://libsigc.sourceforge.net/ (6.8.2009)

[7] *GCC, the GNU Compiler Collection*,
http://gcc.gnu.org/ (6.8.2009)

[8] *GNU Make*,
http://www.gnu.org/software/make/ (6.8.2009)

[9] *pkg-config Wiki*,
http://pkg-config.freedesktop.org/wiki/ (6.8.2009)

[10] *Exuberant Ctags*,
http://ctags.sourceforge.net/ (6.8.2009)

[11] *Microsoft Visual C++*,
http://msdn.microsoft.com/en-us/visualc/ (6.8.2009)

[12] *MinGW — Minimalist GNU for Windows*,
http://www.mingw.org/ (6.8.2009)

[13] *GNU Binutils*,
http://www.gnu.org/software/binutils/ (6.8.2009)

[14] *gtkmm on Microsoft Windows*,
http://live.gnome.org/gtkmm/MSWindows/ (6.8.2009)

[15] *Homepage of the Algorithm Animation Library*,
http://basip6am.matfyz.cz/aal/ (6.8.2009)