Programatorska dokumentacia
Radoslav Glinsky

Tento text sluzi pre potreby blizsieho oboznamenia sa s programom "Graficky editor", ktory sluzi ako zapoctovy program z predmetu Objektovo orientovane programovanie v akademickom roku 2007/2008. Je to tzv. programatorska dokumentacia, ktora ma za ciel objasnit objektovy navrh programu, popisat postupy pri rieseni problemov v programe, ako aj vysvetlit niektore zvolene algoritmy. Tato dokumentacia je velmi vhodna pre tych, ktori sa budu blizsie zaujimat o zdrojovy kod editoru (vratane mna samotneho, ked sa k nemu budem vraciat s odstupom casu).

Zakladna trieda programu, ktora obsahuje metody na obsluhu "vsetkeho", je trieda view. Program takisto zacina volanim jej metody init() - ta vykonava vsetky potrebne nastavenia pre OpenGl. Tieto nastavenia dalej nebudem rozoberat. Ak sa init() podari, je nasledna volana metoda update(), ktora v sebe obsahuje "nekonecnu" while smycku. V tejto smycke osetrujem stlacenie klaves Esc (vyvolava zrusenie nekonecnej while smycky -> koniec programu) a Tab (vyvolava prerusenie programu -> minimalizuje program). Ak dojde k preruseniu smycky (stlacenie Esc), pred samotnym ukoncenim programu sa este zavola metoda shutdown(), ktora obsluzi spravne ukoncenie pouzivania SDL knihovny. Ak teda nie je aplikacia ani ukoncena ani minimalizovana, vola sa metoda render().

Metoda render() je urcena na graficke vykreslovanie. Kresli vytvorene objekty, cudliky (vratane ich tooltipov-popiskov), kurzor. Na jej zaciatku a konci su nejake OpenGL nastavenia (tie su ale v tejto dokumentacii nie velmi podstatne). Okrem toho sleduje, ze ci nedoslo ku kliknutiu mysou (a ze ci po kliknuti je tlacitko mysi stale stlacene – kvoli obsluhe drag&drop) alebo ze ci nebolo rolovane koleckom na mysi. Ak napr. bolo kolecko rolovane, nastavuje sa premenna zoom, ktora urcuje priblizenie "kamery". Inak aktualnu polohu kurzoru na obrazovke urcuje premenna cursor. Ked sa ale zoom-uje, vytvorene objekty vsak nemenia svoje suradnice, takisto kurzor na obrazovke ostava na svojom mieste, preto je potrebne vzdy v kazdom priechode cyklom upravit "obrazovkove" suradnice kurzoru na suradnice, ktorymi su vnutorne reprezentovane objekty, cudliky, apod. Podobne ako zoom sluzi premenna camera_position, ta urcuje vertikalne a horizontalne posunutie, ktore vznika pri pohybe kurzoru niekde cez okraj obrazovky.

Ak dojde ku kliknutiu, nastavia sa premenne from, from_pom, to na suradnice kurzoru v momente, kedy bolo kliknute. Tieto suradnice si musime pamatat z dovodu neskorsieho mozneho vyuzitia pri drag&drop. Vyvola sa metoda view::mouse_click(). Ta vo svojom tele testuje, ze kde bolo kliknute. Najprv testuje, ci to nebolo na nejaky z ovladacich cudlikov (funkciou toolbar.check_toolbar(). Na margo, toolbar je objekt (datovy clen triedy view), ktory v sebe zdruzuje vsetky ovladacie cudliky, a ma metody na spolupracu s nimi. Ak teda bolo kliknute na nejaky cudlik, zmeni sa prislusne prave aktivny cudlik, to indikuje premenna toolbar.active_button. Ak nebolo kliknute ne ziadny cudlik, suradnice kliknutia sa poznamenaju (pripisu na koniec) do vector<complex> action. To je potrebne, ak napr. vytvarame novy objekt, ktory potrebuje viac kliknuti (napr. ciara potrebuje aspon dve kliknutia na obrazovke).

Ak je este stale drzane tlacitko mysi, overime, ze ci sa aktualne suradnice (ne)zhoduju so suradnicami, kde bolo naposledy kliknute. To overime porovnanim if(from_pom!=to). Ak je podmienka splnena (tj. muselo dojst k potiahnutiu mysi za staleho drzania tlacitka), vykona sa metoda mouse_drag(). Ked si rozoberieme pripady, mouse_drag() nastava v 3 pripadoch:

1) ked chceme oznacit objekty, vtedy "nakreslime" akysi obdlznik, kedy sa oznacia vsetky objekty, ktore v nom lezia
2) oznacene objekty jednoducho "chytime" a posunieme
3) oznacene objekty maju na stranach tzv. posuvatka a rotovatka. Ked za nich "chytime" a potiahneme, objekty sa zmensuju/zvacsuju resp. rotuju.

Metoda mouse_drag() najprv overi, ze ci nastal 2) alebo 3) pripad. Pripad 2) nastava vtedy, ked sa kurzor aktualne nachadza nad nejakym objektom. Vtedy len zavolame selection.move(to-from_pom). Objekt selection (ktory v sebe udrziava pointre na prave oznacene

objekty) je potomok triedy “objekt”, ma preto takisto definovanu metodu move(). Pripad 3) musime osetrit tak, ze ihned po kliknuti na obrazovke zistime, ci bolo kliknute na nejake posuvatko/rotovatko. Objekt selection ma metodu test_arrow_hit(). Ta vrati cislo posuvatka/rolovatka, ak bolo na neho kliknute. Inak vrati hodnotu -1. Tuto hodnotu si teda ihned po kliknuti uchovame v premennej arrow_persist_scaling (prikazom arrow_persist_scaling=selection.test_arrow_hit(screen_pos)). Teraz este musime zistit, ci selection budeme zmensovat/zvacsovat alebo rotovat. To nam napovie bool premenna selection.rotating (selection.rotating=true, ak budeme rotovat, inak selection.rotating=false). Ak budeme rotovat, vypocitame stred rotacie a uhol, a jednoducho volame selection.rotate(middle,angle). Ak budeme zmensovat/zvacsovat (jednym slovom “scale”-ovat), zistime si stred scale-ovania a multiplikator scale-ovania. V zavislosti na umiestneni posuvatka/rolovatka dalej scale-ujeme rozne. Musime tu rozlisovat 8 pripadov (lebo je 8 posuvatok/rolovatok). Selection ma aj metody scale_x a scale_y (zdedene po triede “objekt”), ktore pri tomto scale-ovani budeme vyuzivat. Takymto sposobom osetrime vyssie spominany pripad 3). Ak nastane pripad 1) tj. chceme len oznacovat objekty, ziskame si zoznam vsetkych vykreslovanych objektov, a zistujeme, ci ich bounding_box-y (minimalne obdlzniky, v ktorych sa nachadzaju) sa nachadzaju kompletne vnutri nasho “oznacovacieho” obdlznika. Ak ano, pridame ich do pomocneho selection – do selection_pom. Tato pomocna premenna sluzi prave na tieto ucely, lebo pri oznacovani takymto “obdlznikom” zaroven testujeme, ze ktore objekty, ktore uz su v selection, a nenachadzaju sa v oznacovacom obdlzniku, mozeme zo selection odstranit. Takto by sme odstranili aj objekty, ktore by sme nemali. Preto pouzijeme pomocny selection_pom objekt, do ktoreho mozeme objekty pridavat a smelo aj odoberat. Nakoniec (po skonceni tejto metody mouse_drag()) oba selection aj selection_pom zlucime dohromady, a to prikazom: selection.make_union(&selection_pom).

Funkcia render() dalej pokracuje vykreslenim objektov: canvas.draw(). Canvas je objekt, ktory obsahuje ako datovu polozku pole objekt_group-ov a takto implementuje undo (vid dalej). Vzdycky je aktivny prave jeden objekt_group z tohto pola, a ten je aj vykreslovany.

Nasleduje volanie funkcie toolbar.do_action(action,this). Toolbar obsahuje v sebe vsetky ovladacie cudliky, a jedine co urobi je, ze zavola prislusnu do_action() metodu prave aktivneho cudlika. V tomto kroku moze nastat jedna vynimka, a to prave ked sa stane aktivny cudlik “save image in png format”. V tejto chvili nemozeme len tak zavolat funkciu na spravenie snapshot-u, pretoze obdlznik, ktory ma zachytavat co sa ulozi do obrazku, moze byt nasledkom zoom-u alebo posunutia do stran uplne zmeneny (zmenseny/zvacseny alebo uplne posunuty (aj mimo zobrazovanu obrazovku). Preto musime spravit nasledujuci trik: Len si poznamename, ze je vytvaranie obrazku aktivny cudlik, ale nezavolame jeho metodu do_action(). Avsak v dalsom priebehu cyklu (kedy uz vieme dopredu, ze budeme robit snapshot) OpenGl kameru nijak nenastavime na platny zoom a posunutie do stran (tj. obrazovkove suradnice budu totozne s vnutornymi), a teraz uz mozeme kludne spravit zelany obrazok. Akonahle sa obrazok urobi, aktivny cudlik nastavujeme na select button. Takze uz nasledujuce volanie render() prebehne “normalne” - OpenGl kamera sa patricne zazoomuje resp. posunie.

Zvysok funkcie render() je uz samovysvetlujuci:
draw_cursor() - vykresli kurzor, navyse kontroluje, ci sme neopustili s kurzorom dovolene hranice na obrazovke; ak ano, nastavuje premennu camera_position (posunutie kamery v rovine)
toolbar.draw_toolbar() - vykresli vsetky cudliky
color_toolbar.draw_toolbar() - vykresli vsetky farebne cudliky, na ktore ked klikneme, tak sa selection prislusne zafarbi (zmeni farbu)
toolbar.check_button_tooltips(cursor) – overi, ze ci sa nahodou kurzor nenachadza nad nejakym ovladacim cudlikom; ak ano, vykresli pri kurzore popisok (tooltip) co robi tento cudlik.

Undo

Zvlast narocna uloha bola navrhnut co najlepsi sposob, ako vyriesit tzv. undo. Pre tento ucel som vytvoril triedu canvas_group (a z nej instanciu – konkretny objekt canvas). Canvas obsahuje polozku pole objekt_group-ov, kde kazdy ma vlastne objekty (objekt_group* can). Vzdycky, ked sa vykona nejaka zmena na zobrazovanych objektoch, tie objekty, ktorych sa zmena tyka, sa zduplikuju. Ziska sa prave aktivny zoznam objektov a menene objekty sa v nom nahradia (objekty, ktore sa nijak nemenili sa neduplikuju). Takto upraveny zoznam objektov canvas ulozi do svojho pola na index o jedna vyssi, nez ne prave zobrazovany. Nasledne inkremenuje zobrazovany index (eventualne ked narazime na najvyssi mozny index, vratime sa opat k nule). Ked to zhrnieme, canvas pri hocijakej zmene duplikuje len objekty, ktore sa menia. Ostatne len okopiruje (teda pointre na nich, nie cele objekty). Tento postup ma tu vyhodu, ze sa duplikacia nerobi vzdy na vsetkych objektoch. Avsak, prinasa to aj nevyhodu. Dan za usetreny cas a pamat zaplatime neskor. Ked sme uz presli "jedno cele kolo", tj. velkost pola (=pocet undo), musime novu zmenu zapisovat na index, kde uz nieco je. To nieco mozu byt:

1) objekty, ktore su v zoznamoch v celom poli jedinecne
2) objekty, ktore sa este nachadzaju v zozname objektov ineho indexu pola

V pripade 1) pred prepisanim objekty jednoducho odalokujeme – uz ich nebudeme potrebovat. V pripade 2) odalokovavat este nemozeme, lebo co ak by uzivatel chcel v dany moment pouzit undo, a teda by v takomto pripade siahal uz na skor dealokovany objekt -> segmentation fault. Nastastie recept na tento problem je jednoduchy. Staci, ked pred prepisanim zoznamu na prepisovanom indexe pola, si prejdeme cely tento zoznam objektov a pre kazdy objekt postupujeme tak, ze prehladame zvysne zoznamy na zvysnych indexoch. Ak sa objekt nachadza aspon v jednom z nich, tak objekt neodalokujeme. Ak sa nenachadza v ziadnom, objekt mozme bez obav odalokovat. Teraz uz nie je sanca, aby k nemu mohol este niekto pristupit (lebo ziaden iny odkaz uz neexistuje, a ten co prave skumame ideme prepisat). Opisany postup vykonava funkcia canvas_group::overwrite().

Podobne pristupujeme k rieseniu problemu aj pri ukoncovani programu. Vtedy sa automaticky vola destruktor canvas_group::~canvas_group(). On musi odalokovat vsetky objekty, ktore este v poli so zoznamami objektov zostali. Niektore objekty sa vsak nachadzaju na viacerych miestach, no my mozeme kazdy odalokovavat len raz. Koname nasledovne: prechadzame pole od zaciatku po koniec, z kazdeho indexu si vytiahneme zoznam objektov, ktore su v nom uschovane. Kazdy z tychto objektov nechame odstranit zo vsetkych nasledujucich zoznamov (a to prikazom for(j=i+1;j<size;++j) can[j].rem_obj(*it);

kde i je aktualny index, size je velkost pola, *it je odstranovany objekt, can je to nase pole a je typu objekt_group. Preto mozeme pouzit metodu rem_obj(), ktora zo svojho zoznamu odstranuje objekt zadany v parametre.

Na konci destruktoru ~canvas_group() mame istotu, ze kazdy objekt sa nachadza v poli prave raz, a preti uz len volame prikaz delete [] can; ten postupne vola destruktory na jednotlive objekty typu objekt_group. A tie uz volaju delete na vsetky objekty v svojom zozname.

Co sa tyka samotneho postupu pri undo (krok naspat) alebo opacne krok dopredu, nie je uz nic lahsieho. Vykonavaju to metody canvas_group::backward() a canvas_group::forward(). V principe v metode backward() len dekrementujeme active (--active) a v metode forward() prave naopak ++active, kde active urcuje index pola, ktory je prave aktivny (vykreslovany). Tieto metody okrem toho este pracuju s premennou diff, ktorej ulohou je zabezpecit, aby backward() nebolo volane viackrat ako je size, a na druhej strane aby forward() nebolo volane viackrat po sebe, ako bolo predtym volane backward().

Toolbar

V dalsej casti sa zamerame na cudliky, konkretne na ich virtualne metody do_action().

- select_button::do_action(vector<complex>& p, view* v)
  vector<complex>& p urcuje body, kde bolo kliknute na obrazovke.
  Najprv otestujeme, ze ci nahodou nebolo kliknute na farebny panel.
  v->color_toolbar.check_toolbar((p[0]+v->camera_position)*v->zoom,r,g,b)
  kliknuty bod p[0] vsak musime este upravit, lebo ten kliknuty bod je prave systemovy, neodpoveda obrazovkovemu (to znamena, musime pripocitat posunutie a vynasobit zoom-om). Ak napokon bolo kliknute na farebny cudlik, tak prislusne farby sa nam uz vratia v podobe parametrov r,g,b, ktore uz volana funkcia vyplnila. Teraz musime zabezpecit to, aby sa objekty v selection zduplikovali, vymenili sa za tie ich "stare kopie" a ulozili do canvas ako dalsia zmena. Musime este navyse objekty zo selection vymenit za nove kopie. Ak to budeme mat spravnene, volame v->canvas.change(ob_uk_list2), kde ob_uk_list2 je novy zoznam objektov. Dalej volame v->selection.change_color_absolutely(r,g,b). Toto volanie zmeni farbu uz novym duplikatom (a stare nezmeni).
  Ak nebolo kliknute na ziadny farebny cudlik, skusime, ci sme klikli na nejaky objekt. Ak ano, osetrujeme niekolko moznych pripadov. Ak drzime shift, tak sa kliknuty objekt prida do selection (ak v nom este nie je) alebo sa kliknuty objekt zo selection odstrani (ak v nom uz je). Ked nedrzime shift a klikneme na objekt zo selection, tak sa posuvatko pri selection zmeni na rotovatko alebo opacne. Ked nedrzime shift a klikneme na objekt, ktory este nie je v selection, tak sa selection vymaze a prida sa do neho len ten prave kliknuty objekt.
  Dalsia uloha select_button je aj kontrola na niektore stlacene klavesy. Klavesa 'r' maze oznacene objekty. Vtedy ziskame aktualny zoznam objektov, vymazeme z neho tie, ktore su zaroven aj v selection. Takto upraveny zoznam nechame ulozit do canvas. Dalej klavesy-sipky pohynaju selection v danom smere. Klavesy PgUp a PgDn rotuju selection okolo vlastneho stredu. Klavesy Home a End zvacsuju/zmensuju selection. Klavesy 1, 2, 3 umoznuju menit farbu selection-u, a to tak, ze pri ich drzani pridavaju (v tomto poradi) cervenu, zelenu, modru zlozku farby v rgb modely. Pri drzani Shift klavesy navyse sa spravaju opacne – odoberaju.
  Upozornenie: predosle klavesy (sipky, PgUp, PgDn, Home, End, 1, 2 ,3) sa neprejavia v undo. Ked sa zamyslime, tak to ani nie je mozne. Nemozeme pri kazdom drzani takejto klavesy duplikovat menene objekty, lebo by sa nam v momente zaplnil cely canvas (cely undo), lebo tieto zmeny su velmi malicke a pritom velmi caste. Preto by sme pri naslednom pouziti undo ani "nevideli rozdiel".

- makegroup_button::do_action(vector<complex>& p, view* v)
  Ziska si zoznam objektov zo selection. Ak ma tento zoznam 0 alebo 1 prvok konci. Dalej pracuje takto: vytvori novy objekt typu objekt_group. Objekty zo selection postupne odobera z uplneho zoznamu vsetkych prave vykreslovanych objektov a pridava ich do novovytvoreneho objekt_group. Do uplneho noveho zoznamu objektov prida nakoniec tento objekt_group. A este objekty zo selection nahradi jednym prave vzniknutym objekt_group.

- destroygroup_button::do_action(vector<complex>& p, view* v)
  Pracuje presne naopak ako predchadzajuci makegroup_button. Akurat ze na zaciatku este musim overovat (pomocou dynamic_cast), ze ci sa skutocne jedna o objekt_group, aby sme ho mohli rozdelit na jednotlive objekty.

- copy_button::do_action(vector<complex>& p, view* v)
- remove_button::do_action(vector<complex>& p, view* v)
  Funguju podobne ako makegroup_button alebo destroygroup_button az na male rozdiely.

- polygon_button::do_action(vector<complex>& p, view* v)
  Pre nakreslenie mnohouholnika je potrebne drzat stlacenu klavesu Shift, aby sme poznali, kedy

mame prestat so zadavanim bodov. Funguje to asi tak, ze ked na obrazovke naklikame aspon 3 body a pustime Shift, tak sa do canvas prida novy objekt – polygon. Dalsia vymozenost je taka, ze aj ked drzime Shift, tak sa nam vykresluje polygon pomocou niekolkych spojenych ciar na zaklade postupnosti naklikanych bodov.

- line_button::do_action(vector<complex>& p, view* v)
  Pracuje na rovnakom principe ako polygon_button

- rectangle_button::do_action(vector<complex>& p, view* v)
  Obdlznik vznika stylom drag&drop. Samotne kliknutie nestaci, musime preto otestovat, ci tlacitko mysi bolo pustene na inom mieste, nez na tom, kde bolo poslednykrat kliknute. To nam otestuje podmienka if(p[0]==v->cursor && !exaIsMouseButtonDown(1)). Podmienka znamena to, ze suradnica kliknutia (p[0]) je totozna s aktualnom suradnicou polohy kurzoru(v->cursor) a sucasne bolo tlacitko mysi pustene. V tomto pripade koncime. Ak podmienka nie je splnena pokracujeme a to dvomi sposobmi:
  1) tlacitko mysi je este stale stlacene – vtedy len vytvorime docasny objekt rectangle a nechame ho vykreslit
  2) tlacitko mysi bolo pustene – alokujeme novy rectangle a ten pridame do canvas-u

- ellipse_button::do_action(vector<complex>& p, view* v)
- ellipse_circle_button::do_action(vector<complex>& p, view* v)
- star_button::do_action(vector<complex>& p, view* v)
  Vsetky tieto cudliky pracuju podobne ako rectangle_button, lisia sa len v tom, ze tymto objektom sa inak vyratavaju suradnice, ktorymi su inicializovane

- raise_step_button::do_action(vector<complex>& p, view* v)
  Novy zoznam vsetkych objektov vznikne tak, ze postupne prechadzame vsetky objekty zo selection a volanim metody objekt_group raise_step() este na starom zozname docielime zelany vysledok – spravny novy zoznam. Ten napokon zavedieme do canvas.

- raise_top_button::do_action(vector<complex>& p, view* v)
- lower_step_button::do_action(vector<complex>& p, view* v)
- lower_bottom_button::do_action(vector<complex>& p, view* v)
  Velmi podobne ako raise_step_button

- backward_button::do_action(vector<complex>& p, view* v)
  Metoda len vola v->canvas.backward();
- forward_button::do_action(vector<complex>& p, view* v)
  Metoda len vola v->canvas.forward();

- screen_shot_button::do_action(vector<complex>& p, view* v)
  Na zaciatku si vytvorime nazov suboru, pod akym chceme ulozit obrazok. Ja som zvolil, ze sa nazov bude samostatne odvodzovat od datumu a casu. Bude sa ukladat v adresari “snapshots”. Funkcia dalej zavola inu metodu screen_shot_button::exaScreenshot(), ktora nam cez parametre vrati data – bitmapu pixelov (to zaobstara funkcia glReadPixels()) rozmedzia, ktore sme zadali. Nakoniec uz len volame exaImageWritePNG() so specifikovanym nazvom suboru (obrazku) a pixelovou bitmapou.

- move_points_button::do_action(vector<complex>& p, view* v)
  Trieda move_points_button obsahuje navyse tieto premenne:
  1)poly_objekt* po - ukazatel na objekt, ktoreho bod budeme presuvat, vsimnime si, ze tento objekt musi byt typu poly_objekt (resp. objekt z tejto triedy odvodeny), lebo len taketo objekty sa skladaju z bodov, ktore je mozne presuvat
  2)complex* point - ukazuje na konkretny bod, ktory je posuvany

3)bool persist_moving - indikuje stav, kedy mame nadalej pokracovat s presuvanim bodu
Takze rozoberieme si mozne pripady, ako postupuje nasa do_action() metoda:
Ak je splnena podmienka if(p.size()>0), to znamena ze bolo niekde kliknute, musime zistit, ze kde. Ak dalej plati, ze uz mame oznaceny nejaky poly_objekt, ktoreho bod chceme presuvat (plati podmienka if(po)), musime testovat, ci sme klikli na nejaky z jeho bodov. To otestujeme podmienkou if(po->test_hit_drag_points(p[0],&point,v->zoom)), kde p[0] su suradnice miesta, kde bolo kliknute, point je bod, ktory ak podmienka uspeje, sa automaticky nastavi na prave oznaceny bod, zoom zadavame preto, ze pri kliknuti na bod sa v podstate nikdy netrafime presne na suradnice, ale len do nejakeho okolia bodu. Preto v tolerancii okolia musime brat do uvahy aj zoom (cim vacsi zoom, tym mensia tolerancia a naopak). Ak napokon posledne zmienena podmienka plati, nastavime priznak persist_moving=true. Ak nie je splnena, testujeme, ci sme neklikli na nejaky uplne iny objekt typu poly_objekt. A to tymito prikazmi:
objekt* ob;
ob=v->canvas.get_pointed_object_concrete(p[0]);
po=dynamic_cast<poly_objekt*> (ob);
Ak bol nejaky objekt kliknuty, musime este dodatocne overit, ci to je naozaj poly_objekt, preto tam mame dynamic_cast.
Dalej, ak je teda nejaky poly_objekt oznaceny (if(po)) nechame ho, nech vykresli svoje body: po->draw_resizeable_points(). Nastanu opat 2 pripady, bud je tlacitko mysi este stale kliknute (vtedy mozeme nejaky oznaceny bod presuvat) alebo uz je pustene (vtedy uz nebudeme nicim posuvat, a nastavime priznak persist_moving=false). Ak je tlacitko este stale stlacene a priznak persist_moving==true, vtedy posuvame bod. Volame metodu oznaceneho objektu: po->move_point(point,(v->to – v->from_pom)), kde point je referencia na posuvany bod, a druhy parameter urcuje posunutie.
To by bolo v skratke vsetko, nezabudnime vsak este na spravne fungovanie undo. Objekty duplikujeme vsak len vtedy, ked doslo k prvemu posunutiu (teda, ak je splnena podmienka if(v->from==v->from_pom)).Objekty sa zduplikuju, ale pozor, odkaz na posuvany objekt aj jeho bod sme duplikaciou stratili (uz je to fyzicky iny objekt). Musime este previest tieto kroky, a to: pri duplikacii volame pretazenu metodu duplicate(), ktora ak narazi na objekt specifikovany v parametre, a zduplikuje ho, tak pointer na zduplikovanu instanciu spatne parametru priradi (temp.duplicate(&po)). Tak poriesime, ze aj po duplikacii budeme mat odkaz na “spravny” objekt. Avsak duplikaciou sme stratili aj odkaz na posuvany bod. Nevadi, ak mame presuvany objekt, a vieme suradnice bodu, ktory sa ma presuvat, tak uz staci este raz otestovat, ci “nahodou” nebolo kliknute na nejaky z bodov novozduplikovaneho objektu. Mame istotu, ze “bolo”: po->test_hit_drag_points(*point,&point,v->zoom).

Metody objektov

Doposial sme opisovali funkcie z pohladu celkovej funkcnosti programu. Vzdy sme sa ale opierali o konkretne metody definovane pre rozne objekty. Tieto metody sme este podrobnejsie nerozoberali, preto sa pokusim popisat a objasnit niektore z nich, ktore na prvy pohlad nie su uplne zrejme.

Opisal by som najprv, ako funguju metody triedy objekt move(), scale() a rotate(). Vsetky tieto metody su spolocne pre vsetky odvodene triedy. Su zalozene na nasobeni matic (aj ked samotne nasobenie je velmi "skryte"). Ak sa pohybujeme v dvojrozmernom priestore, matica, kotoru nasobime nejaky bod z tohto priestoru, ma rozmery 3x3. Avsak 3 z tychto 9 rozmerov su stale rovnake, preto nam vystaci pole velkosti 6. V metodach move(), scale(), scale_x(), scale_y(), rotate() na zaklade parametrov sa vytvori "matica" so 6 hodnotami. Ked touto maticou vynasobime nejaky bod, ten bod zmeni patricne (podla toho, aka je to metoda) svoje suradnice, ako si zelame. Samotne vynasobenie matice a bod je spracovane v (takisto spolocnej metode pre vsetky odvodene objekty) objekt::mult_matrix_one(float matrix[6],complex& c), kde matrix je matica so 6 prvkami a c predstavuje suradnice bodu v 2D priestore.

Potom, co sa vypocita prislusna nasobiaca matica, zavola sa (v kazdej modifikujucej metode) mult_matrix(matrix), kde parameter matrix je prave ta vypocitana modifikujuca matica. Metoda mult_matrix(matrix) je uz prave virtualna, a kazda trieda si ju uz predefinuje podla vlastnych poziadaviek. Ale obecne ide len o to, ze v tele tejto metody mult_matrix(matrix) sa vola funkcia mult_matrix_one() pre kazdy jeden bod, ktory volajuca trieda chce modifikovat.

Rozobral by som aj metody, ktore umoznuju presuvat nejaky objekt blizsie do popredia resp. dalej do pozadia (ak sa, samozrejme, pred nim resp. za nim nejaky objekt nachadza – ak sa objekty prekryvaju). Najprv si ale uvedomme, ze ako vlastne prebieha vykreslovanie objektov. Tie sa vsetky nachadzaju v jednom list-e, a vykresluju sa od prveho az po posledny. Celkom navrchu je ten, ktory sa vykresluje neskor (tj. prekresli objekty, co su na rovnakej poziicii vykreslene skor). Ulohou preto je to, aby pri "premiestnovani" objektu sme ho len presunuli vramci list-u, v ktorom sa nachadza, na nejake ine (spravne) miesto. Z metod

objekt_group::raise_step(objekt* p_o, bool ukoncit)
objekt_group::raise_top(objekt* p_o)
objekt_group::lower_step(objekt* p_o,bool ukoncit)
objekt_group::lower_bottom(objekt* p_o)

opiseme len prvu dvojicu (raise_step() a raise_top()). Druha dvojica sa principom velmi podoba na prvu. V metode objekt_group::raise_step() postupne prechadzame list objektov od zaciatku po koniec. Pokial nenarazime na presuvany objekt, nic nerobime. Od momentu, kedy narazime na presuvany objekt, budeme kazdy objekt v list-e testovat, ze ci sa nahodou neprekryva s tym presuvanym objektom. Ak sa prekryva (co znamena, ze presuvany objekt je pod prave testovanym objektom), odstranime presuvany objekt z list-u: o.remove(p_o) a hned ho naspat vlozime, avsak tentokrat az za prave testovany objekt: o.insert(it,p_o), kde it je iterator ukazujuci na miesto za prave testovanym objektom.

Po jednom takomto premiestneni mozeme skoncit (metoda raise_step()) alebo mozeme s takymito premiestneniami pokracovat az do konca list-u s objektami (metoda raise_top()). Zavadzat dve metody, ktore robia az na jednu vynimku to iste, je vsak zbytocne. Preto nasa metoda raise_step() zavadza implicitny parameter bool ukoncit=true, ktory urcuje, ci sa ma premiestnovat az do konca, alebo sa ma ihned po prvom premiestneni ukoncit. Toto ma vyhodu, ze v metode raise_top() staci zavolat raise_step(p_o,false), kde p_o je ukazatel na premiestnovany objekt.