Електротехнички факултет Универзитета у Београду
Катедра за Рачунарску технику и информатику

SeeGL – Софтверски алат за учење графичке библиотеке OpenGL

Завршни рад мастер академских студија

Ментор
др ИгорТартаља

Студент
Марко Првуловић 3123/2010

Апстракт

У раду је представљен софтверски алат *SeeGL*. Алат је образовног карактера. Корисницима нуди алтернативни, мање напоран и атрактивнији начин проучавања рада библиотеке *OpenGL*, у односу на конвенционално читање и анализирање доступне документације. Интерактивно, путем графичког интерфејса, корисник може да добије на увид стање свих елемената коначног аутомата који се налази у основи библиотеке. Корисник може да састави програм који за цртање користи библиотеку *OpenGL*, да посматра промене стања коначног аутомата након извршења сваке наредбе и види резултат цртања. У случају неочекиваног резултата цртања, корисник може брзо и једноставно, визуелном инспекцијом, пратећи приказан ток којим ова библиотека обрађује податке, да установи да ли се извор проблема налази у погрешним вредностима параметара неког позива, у погрешном редоследу позива, у изостављеном или сувишном позиву.

Садржај

1. Увод

Интензиван развој технологија производње микрочипова током претходне две деценије омогућио је да брз графички оријентисан хардвер постане широко доступан, по релативно ниским ценама. У том периоду, могућности хардвера и технике формирања слика значајно су напредовале, пружајући бољи квалитет нацртаних слика, уз краће време цртања. Пратећи развој хардвера, сложеност графичких библиотека, попут *OpenGL* [1] ili *DirectX* [2], порасла је до те мере да је данас потребно уложити значајан труд у разумевање њиховог функционисања, пре писања програма за цртање једноставне сцене. Графичка библиотека *OpenGL* у основи се може посматрати као коначан аутомат. Програмер на располагању има стотинак наредби за контролу стања овог аутомата. Начин на који се обрађују параметри наредби цртања, па самим тим и резултујућа слика, зависе од стања аутомата у тренутку позива наредбе цртања. Начин употреба ових наредби може бити врло неинтуитиван, јер је интерфејс за програмирање апликација (енг. *Application Programming Interface* – API) моделиран са идејом да програмер има максималну слободу приликом подешавања стања коначног аутомата. Било каква логичка грешка у стању, коју *OpenGL* не може да открије зато што представља валидно стање аутомата, може довести до неочекиваног приказа. Тада програмер, анализом програма, а често и низом проба, мора да установи порекло грешке и да је отклони.

Софтверски алат са богатим интерактивним графичким интерфејсом, једноставним механизмом задавања наредби и визуелизацијом тока обраде података кроз делове коначног аутомата, био би корисно образовно средство, које би могли да примењују како инструктори, тако и обучавана лица. И поред тога што графичка библиотека *OpenGL* већ две деценије представља *de facto* индустријски стандард за развијање платформски независних 3D графичких апликација, постоји тек мали број јавно доступних софтверских алата који поседују могућност визуелизације и интеракције са коначним аутоматом ове библиотеке. Због тога, такав софтвер би представљао веома напредно јавно доступно средство за интерактивно упознавање рада коначног аутомата ове графичке библиотеке популарне и у академским и у индустријским круговима.

У овом раду се презентира развој софтверског алата *SeeGL* осмишљеног тако да одговори наведеним изазовима. Овај алат представља образовни софтвер намењен за упознавање рада коначног аутомата графичке библиотеке *OpenGL*. *SeeGL*, путем графичког корисничког интерфејса, омогућава кориснику да састави *OpenGL* програм, изврши га до обележене позиције између две наредбе и очита жељене вредности стања коначног аутомата на обележеној позицији. Додатно, кориснику је приказана шема тока обраде података са назнакама група модалних атрибута коначног аутомата, као и назнакама наредби које се користе за контролу тих атрибута. Тиме корисник може једноставно да уочи распоред модалних атрибута коначног аутомата, као и редослед операција приликом извршавања *OpenGL* програма, јер се линије које повезују елементе на шеми боје у зависности од покренуте наредбе.

Остатак рада организован је на следећи начин. У другом поглављу изложен је детаљан опис постављеног задатка и значај његовог решавања. У том поглављу објашњени су и најзначајнији термини коришћени у раду. Потом, у трећем поглављу дат је сажет преглед постојећих решења. Функционални аспекти софтверског алата *SeeGL* изложени су у четвртом поглављу. Неки од кључних и интересантних проблема у имплементацији софтвера, као и њихова алтернативна решења, разматрани су у петом поглављу. У том поглављу су описане и техничке карактеристике имплементације. У закључку изнете су најважније карактеристике развијеног алата, као и планови за даљи развој и могућа унапређења.

2. Поставка проблема

У овом поглављу биће објашњени коришћена терминологија, мотивација за развој са основним захтевима, као и специфични проблеми везани за визуелизацију коначног аутомата. Коначно, биће објашњен планиран начин коришћења алата *SeeGL* код предавања и самосталног учења.

2.1. Терминологија

У овом одељку дефинисани су значајни термини који се користе у остатку рада. За све термине дат је и назив који је уобичајен у литератури на енглеском језику.

- **Пиксел** (енг. ***pixel***): најмањи елемент слике 2D растерских приказивача.
- **Теме** (енг. ***vertex***): структура података која описује тачку у 2D или 3D простору. Темену се, поред позиције у 2D или 3D простору, обично придружују следећи подаци: боја, нормала (за теме у 3D простору) и координате у простору текстура.
- **Исцртавање** (енг. ***rendering***): процес стварања 2D слике од модела сцене, које обавља неки рачунарски програм.
- **Растеризација** (енг. ***rasterization***): техника исцртавања која итерира кроз скуп примитива, и за сваку примитиву рачуна боју одговарајућих пиксела екрана који припадају унутрашњости и ивици примитиве.
- **Програм за сенчење** (енг. ***shader program***): програм који контролише начин на који се врши обрада геометрије (темена) или врши избор боја сваког пиксела слике. Програмима за сенчење се омогућава флексибилна контрола једног дела проточне обраде података графичког хардвера.
- **Проширена стварност** (енг. ***augmented reality***): приказ стварног окружења проширен са звучним или графичким додацима који су генерисани од стране рачунарског програма обично на основу неког улазног параметра (нпр. видео, звук, GPS - енг. Global Positioning System и други).
- **Компонента графичког интерфејса** (енг. **GUI w*idget***): основни елемент графичког интерфејса који може да се прикаже. Приказује податке које корисник може да модификује.

2.2. Мотивација и основни захтеви за развој

Графичка библиотека *OpenGL* има релативно једноставан процедуралан интерфејс за програмирање апликација (енг. *Application Programming Interface – API*), чије основе могу да се савладају без великог напора. Међутим, било каква напредна употреба, осим солидног знања из геометрије у 2D и 3D простору, од програмера захтева познавање поступка којим *OpenGL* обрађује геометријске податке и бира боју за сваки пиксел који нацрта на основу обрађене геометрије. Пример, чест у пракси, када је неопходно познавање овог поступка, јесте комбиновање неколико текстура за бојење површи неког геометријског тела, уз примену динамичког осветљења. У овом примеру, програмер може нехотице да направи грешку изостављања неког неопходног позива или грешку задавања неодговарајуће вредности параметра приликом конфигурисања јединица за текстурирање, што може произвести неочекиван резултат на основу којег је тешко прецизно утврдити узрок грешке. Узрок може бити и неодговарајући редослед позивања наредби. У сваком од случајева, библиотека *OpenGL* не пријављује никакву грешку, јер је аутомат у исправном стању.

У циљу решавања описаног проблема, појавила се потреба развоја софтверског алата који ће кориснику, који учи коришћење библитеке *OpenGL*, обезбедити информативан и једноставан начин проучавања њеног рада. Један од првих проблема који поменути алат треба да реши јесте приказ шеме тока обраде података и вредности релевантних атрибута коначног аутомата кроз функционалне блокове при извршавању наредби *OpenGL* програма, како би корисник могао да визуелизује редослед операција приликом извршавања наредби ове библиотеке. Пошто ток обраде података зависи од вредности атрибута (стања) коначног аутомата библиотеке, битно је да се кориснику прикаже на које делове тока обраде података утиче који атрибут аутомата, и која наредба може изменити вредност ког атрибута. Кориснику је потребно обезбедити брзо претраживање и проналажење одабраног модалног атрибута или наредбе на шеми тока обраде. Други значајан проблем је састављање *OpenGL* програма. Кориснику је потребно обезбедити неки напреднији начин састављања програма, у односу на писање изворног кôда у текстуалном облику. Корисно је да при писању програма буде ослобођен бриге око алокације и деалокације меморије и да једноставно може да поставља вредности параметара наредби, односно функција које програм позива. Потребно је да корисник може да сваки састављен програм сачува у датотеци и да га учита из ње. Посебно, у циљу шире доступности и једноставније употребе, очекује се да алат буде развијен у стандардном језику, како би се могао превести и користити на шире заступљеним платформама, као и да корисник кроз алат може да састави и извршава програм без превођења. Коначно, због непрестаног развоја нових верзија библиотеке *OpenGL*, као и механизма екстензија кроз који је произвођачима хардвера омогућено да развој библиотеке делимично прилагоде својим циљевима, потребно је кроз алат обезбедити конзистентан начин проширивања подршке за нове могућности библиотеке.

2.3. Специфични проблеми

Основни проблем, који треба решити приликом израде предметног алата, је визуелизација шеме тока обраде података ове библиотеке. Подаци су модални атрибути који одређују стање коначног аутомата *OpenGL*-а, док њихову обраду покрећу позиви функција (наредби) ове библиотеке. Позивањем било које наредбе библиотеке покреће се ток операција над атрибутима аутомата. Свако следеће позивање наредбе покреће ток операција који зависи од затечених вредности атрибута, које су оставили претходни позиви. У наставку су наведени проблеми на које се наилази приликом анализе на који начин је кориснику најбоље приказати покретање позива, операције над атрибутима и вредности атрибута коначног аутомата.

Први проблем визуелизације дијаграма тока обраде је одређивање скупа информација које су значајне кориснику, а које је могуће дохватити или израчунати. Наиме, имплементације *OpenGL* проточне обраде података се могу разликовати између произвођача хардвера. Због тога делови тока обраде података у библиотеци *OpenGL*, који нису условљени самом спецификацијом, могу да се разликују између имплементација. Додатни проблем је тај што од имплементације зависи у ком тренутку ће која операција започети, односно да ли ће бити одлагана до последњег тренутка када ју је могуће извршити или ће њено извршење почети пре тога. Дакле, кориснику је врло тешко приказати симулацију стварног тока обраде података, јер је он сакривен самом имплементацијом. Међутим, кориснику је могуће приказати редослед операција који је дефинисан расположивом спецификацијом библиотеке *OpenGL*. Та информација је кључна за разумевање рада ове библиотеке и она представља срж предметног алата. Може се сматрати да су сви недовољно прецизирани елементи спецификације заправо места на којима је произвођачима хардвера намерно остављен простор да уведу одређене оптимизације. Стога, с обзиром на то да су потенцијалне оптимизације платформски зависне, њихово изучавање није од великог интереса за оне који уче стандардну библитеку, те стога није од интереса ни да такве специфичности буду подржане у оквиру предметног алата. Постоји више начина како да се прикаже редослед операција који није дефинисан спецификацијом. Нека од тих решења укратко су изложена у наставку, а елаборирана у одељку 5.3:

- операције чији редослед није дефинисан документацијом приказују су као да се извршавају секвенцијално, у редоследу који је (произвољно) изабрао аутор
- операције чији редослед није дефинисан документацијом приказују су као да се извршавају паралелно
- операције чији редослед није дефинисан документацијом приказују се као једна сложена операција.

Други проблем је био како визуелизовати случајеве у току обраде података када библиотека, на основу тога да ли је атрибут омогућен или не, одлучује којим током ће се наставити обрада података. На сликама 1, 2 и 3:

- Пуне линије представљају контролне токове.
- Испрекидане (тачкасте) линије представљају токове података.
- Правоугаоници заобљених ивица представљају функционалне блокове – функције које за неке улазне податке могу имати излазне податке.
- Правоугаоници представљају информационе блокове – податке, атрибуте коначног аутомата.
- Кружни елементи представљају елементе гранања где на основу неког услова контролни ток може наставити неким излазним гранама.

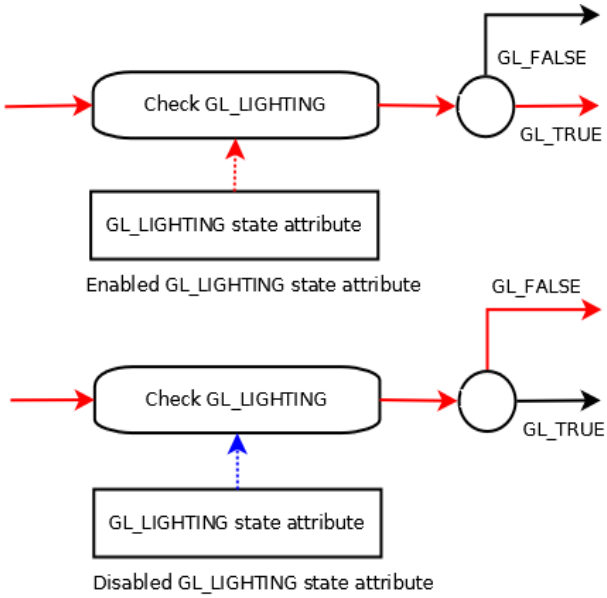
У овом раду, разматрана су следећа два решења проблема приказа гранања контролних токова у зависности од вредности атрибута, тако да шема буде што прегледнија и интуитивнија:

- Користити елемент гранања, код кога се један излазни ток активира када је омогућен, а други када је онемогућен атрибут коначног аутомата. Илустрација овог решења дата је на слици 1. Овакав начин приказивања је уочљив кориснику, али оптерећује приказ у случају када постоји више елемената истог типа. Пример таквог случаја јесу извори светла, којих може бити више. У том случају, овакво решење би захтевало да се прикаже по један

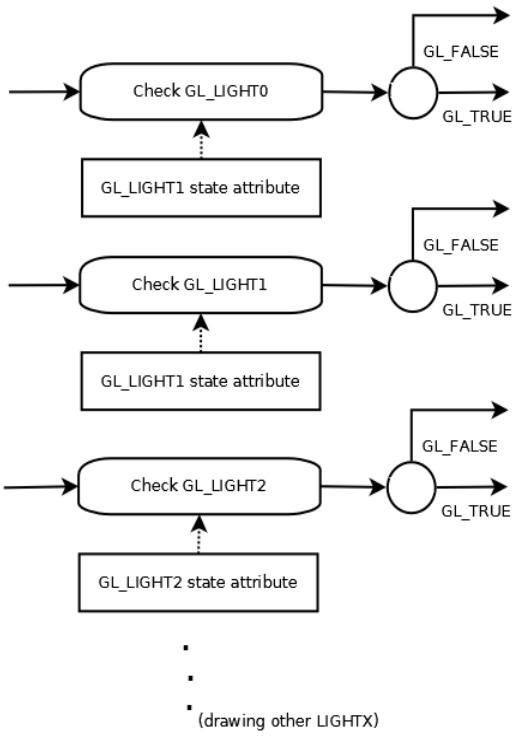
атрибут за сваку појаву елемента истог типа (у претходном примеру – по један атрибут за сваки извор светла). На слици 2 је приказано ово решење, за три извора светла (GL_LIGHT0 до GL_LIGHT2). Библиотека подржава до 8 извора светала.

- Не користити елемент гранања, него бојити активиран излазни ток података у једну боју када је омогућен, односно у другу када је онемогућен. Илустрација овог решења дата је на сликама 3а и 3б. У случају када постоји више елемената истог типа, овакво решење омогућава да се уштеди на простору потребном за приказивање тако што би се више контролних токова визуелно објединило у један. Мана овог приступа је што би обједињени контролни ток на излазу функције био активиран увек када је активиран контролни ток на улазу функције, чак и када су сви атрибути који утичу на функцију онемогућени. С обзиром на то да неки од обједињених токова података (односно атрибути коначног аутомата у једном блоку података) могу бити омогућени, а остали онемогућени, потребно је увести трећу боју којом се означава да нису сви токови података у датом тренутку омогућени или онемогућени (мешовито стање обједињеног тока), што је приказано на слици 3в.

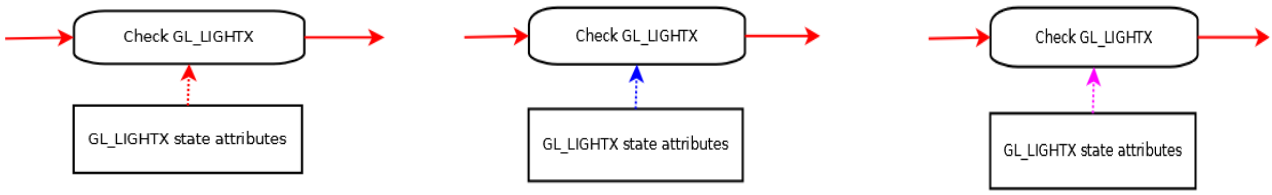
И у једном и у другом решењу контролни ток се боји црвеном бојом ако је активан, иначе је црне боје. Токови података се боје плавом бојом ако су одгварајући атрибути онемогућени, а црвеном ако су атрибути омогућени. Само у другом решењу токови података (у случају да су визуелно обједињени) се боје љубичастом бојом ако су неки атрибути омогућени, а неки не.



Слика 1– Употреба елемента гранања са два излазна тока где један представља ток обраде када је атрибут омогућен, а други када је онемогућен



Слика 2 – Употреба једног елемента гранања по појави извора светла.



а) б) в)

Слика 3 – Употреба бојења тока података, где боја даје информацију о стању одговарајућих сродних атрибута. а) црвена боја означава ток података када су сви `GL_LIGHTX` атрибути омогућени. б) плава боја означава ток података када су сви `GL_LIGHTX` атрибути онемогућени. в) љубичаста боја означава ток података када је су неки `GL_LIGHTX` атрибути омогућени, а неки нису.

У одељку 5.3 биће детаљно објашњен начин на који је решен сваки од наведених проблема.

Поред проблематике визуелизације дијаграма тока обраде података, уочавају се следећи

практични проблеми везани за развој програма који користи библиотеку *OpenGL*, и који треба да функционалности ове библиотеке учини доступним кориснику кроз графички интерфејс:

- проблем интерактивног уноса и извршавања програма састављеног од позива *OpenGL* наредби. Начин на који је овај проблем решен у овом раду изложен је у одељку 4
- проблем пројектовања објектно-оријентисаног модела софтвера који ће омогућити проверу грешака приликом састављања наредби ове библиотеке, њихово безбедно позивање и дохватање вредности атрибута коначног аутомата, заједно са вредностима на одговарајућем стеку. Начин на који је овај проблем решен у овом раду изложен је у пододељку 5.2.2
- проблем дохватања података из коначног аутомата приликом задавања геометријских примитива захтева симулацију вредности атрибута коначног аутомата. Овај проблем се јавља зато што, кад се коначни аутомат библиотеке пребаци у стање за задавање (дефинисање) примитива, библиотека не дозвољава дохватање вредности ниједног атрибута. Начин на који је овај проблем решен у овом раду изложен је у одељку 5.3.
- проблем прегледања садржаја стека за атрибуте аутомата. Овај проблем није једноставан за решавање, јер библиотека не дозвољава преглед свих вредности на стеку неког атрибута, па се мора прибећи или симулацији стека приликом извршавања или позивању наредби за скидање елемената са стека да би се очитале његове вредности, што захтева рестаурацију стека враћањем претходно скинутих елемената са стека. Начин на који је овај проблем решен у овом раду изложен је у одељку 5.3.

2.4. Употреба програма и циљни корисници

С обзиром на то да је предметни софтверски алат образовног карактера, очекује се да нађе примену у одржавању наставе из предмета који се баве проучавањем библиотеке OpenGL, како од стране наставника, тако од стране обучаваних лица, што је објашњено у наставку.

Приликом предавања, очекиван је сценарио у којем предавач, употребом алата, учита постојећи или интерактивно састави неки *OpenGL* програм, и демонстрира његов рад. Затим, предавач уноси измене параметара наредби или мења секвенцу наредби, тако што, на пример, измени редослед наредби, дода или избаци неке наредбе. На месту сваке измене, предавач зауставља извршавање програма и указује на промене на шеми тока обраде података коначног аутомата и промене у резултату (нацртаној слици). Алат треба да омогући предавачу да одржи овакво предавање независно од платформе, конфигурације рачунара или доступног преводиоца.

Код самосталног изучавања принципа функционисања библиотеке *OpenGL*, алат омогућава кориснику брзо и једноставно анализирање рада неког *OpenGL* програма. Очекује се да корисник приликом анализе има могућност да дохвати све оне информације које би иначе морао да тражи по расположивој документацији, уколико му овај алат не би био доступан. Додатно, кориснику би, прегледном анимираном шемом тока обраде података, било омогућено интуитивно разумевање ефеката наредби, а посебно осмишљеним едитором и интерпретером програма експериментисање са редоследом извршења наредби и вредностима њихових параметара. Уколико би приликом састављања програма направио грешку која коначан аутомат доводи у неисправно стање, овај алат би му приликом извршавања доставио информацију где је погрешио. У случају да корисник састави програм, чије извршавање ни у једном тренутку не пребацује коначни аутомат у неисправно стање, а добије неочекивани резултат, он има могућност да, пратећи обојени ток на шеми, установи где је погрешио.

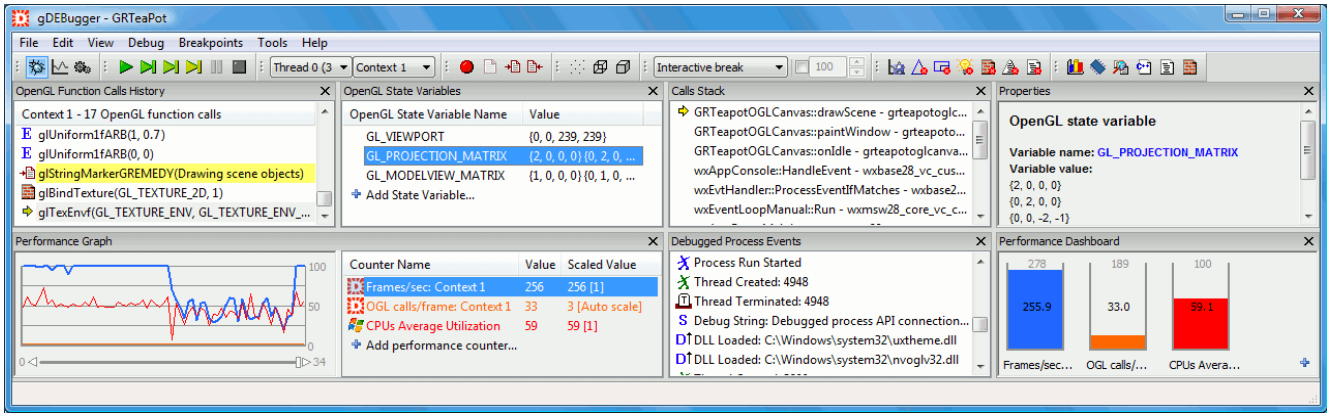
3. Постојећа решења

И поред тога што графичка библиотека *OpenGL* још од 1992. године представља најпопуларније платформски независно решење за развијање 3D графичких апликација, аутор је успео да пронађе релативно мали број софтверских алата који поседују могућност визуелизације и интеракције са коначним аутоматом који се налази у основи ове графичке библиотеке. У овом раду, биће издвојени следећи алати: *gDEBugger* [3], *BuGLe* [4] и *glslDevil* [5]. Поменута софтверска решења имају, у мањој или већој мери, развијене могућности анализе оних елемената ове графичке библиотеке, који су подржани и имплементирани на употребљеном хардверу. Наиме, неки елементи библиотеке нису обавезни и њихова подршка, односно имплементација, је опциона. Стога, ови алати пружају могућност анализе само оних елемената који јесу имплементирани. Корисник би могао да користи ове алате за проучавање и разумевање рада ове библиотеке, иако су ови алати генерално намењени за напредније кориснике. Сва три алата не захтевају накнадно превођење *OpenGL* програма и не захтевају изворни код апликације која се анализира, већ врше пресретање позива библиотеци у време извршења постојећег извршног програма. У наставку, дат је кратак преглед издвојених постојећих решења, са акцентом на следеће особине, које би требало да поседује напредно софтверско решење за изучавање и демонстрацију саме библиотеке:

- визуелизација тока података кроз коначни аутомат библиотеке *OpenGL* и стања елемената аутомата након извршења сваке инструкције програма, што омогућава кориснику да види у којој фази обраде је неки атрибут коначног аутомата у омогућеном или онемогућеном стању,
- напредно интерактивно састављање *OpenGL* програма, што подразумева низ могућности за корисника, попут додавања, избацивања и измене наредбе са провером да ли је вредност параметра дозвољена за ту наредбу, груписање наредби у блокове, онемогућавање наредбе или блока итд,
- могућности дохватања вредности атрибута коначног аутомата,
- могућности дохватања израчунатих података из неке фазе проточне обраде података (нпр. вредности координата сваког темена),
- могућност извршавања до задате наредбе.

3.1. Алат *gDEBugger*

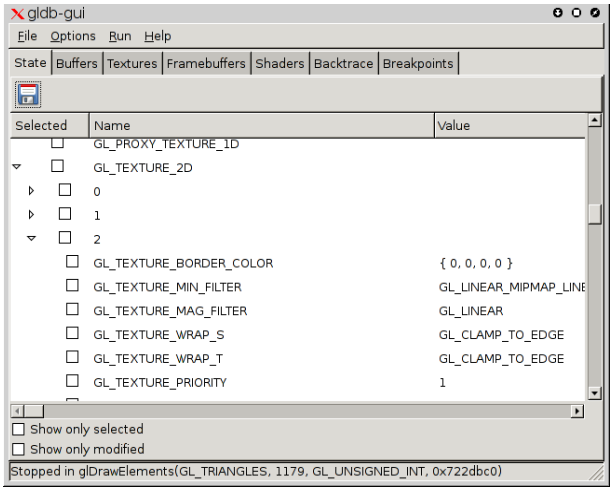
Софтверски алат *gDEBugger* [3] се употребљава за проналажење грешака, оптимизацију перформанси и потрошње меморије код апликација које користе *OpenGL* и *OpenCL* [6]. Изглед графичког интерфејса овог алата је приказан на слици 4, где се могу уочити 8 прозора. Горњи ред прозора са лева на десно чине: прозор историје *OpenGL* позива, прозор са вредностима атрибута коначног аутомата, прозор стека *OpenGL* позива и прозор са информацијама о садржају меморије графичке карте. Садржај меморије графичке карте је организован у објекте (нпр, текстуре, бафери цртања, програми за сенчење...). Доњи ред чине прозори на којима су приказани графикони и подаци корисни приликом оптимизације *OpenGL* програма. Овај алат пружа могућност заустављања рада коначног аутомата графичке библиотеке *OpenGL* у произвољном тренутку и очитавања текућих вредности атрибута аутомата, због чега се овај алат може користити за упознавање рада самог коначног аутомата, иако му то није примарна намена. Наиме, овај алат, поред тога што нуди кориснику богат скуп података добијених из библиотеке (преглед текстурних јединица, садржаја бафера, вредности атрибута коначног аутомата итд), такође у реалном времену пружа кориснику и скуп додатних аналитичких података, као што су искоришћеност меморије и остварене перформансе. Ови подаци су корисни *OpenGL* програмеру приликом оптимизације рада апликације која се анализира. Додатно, овај алат се може користити и за анализу програма за сенчење (енг. *shader*). Овај алат нема могућност интерактивног састављања *OpenGL* програма, већ служи једино за анализирање постојећих (извршних) програма.



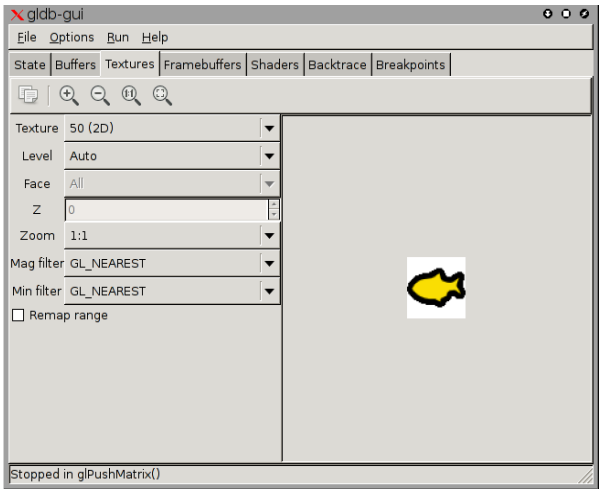
Слика 4 – Изглед графичког интерфејса алата *gDEBugger*

3.2. Алат *BugLE*

Софтверски алат *BuGLe* [4] је још један пример софтверског алата за проналажење грешака у програмима који користе библиотеку *OpenGL*. Овај алат се, као и претходни, истиче могућношћу заустављања коначног аутомата и приказивања његових атрибута, као и приказивања садржаја текстура и бафера. На сликама 5а и 5б се могу видети неке од могућности овог алата, конкретно преглед вредности атрибута коначног аутомата и приказ атрибута текстура. Овај алат има могућност вођења евиденције (енг. *logging*) о позивима наредби ове библиотеке. По сваком позиву, овај алат проверава да ли је коначни аутомат прешао у невалидно стање. Као и претходни алат, *BuGLe* нема могућност састављања *OpenGL* програма, већ служи једино за анализирање постојећих (извршних) програма.



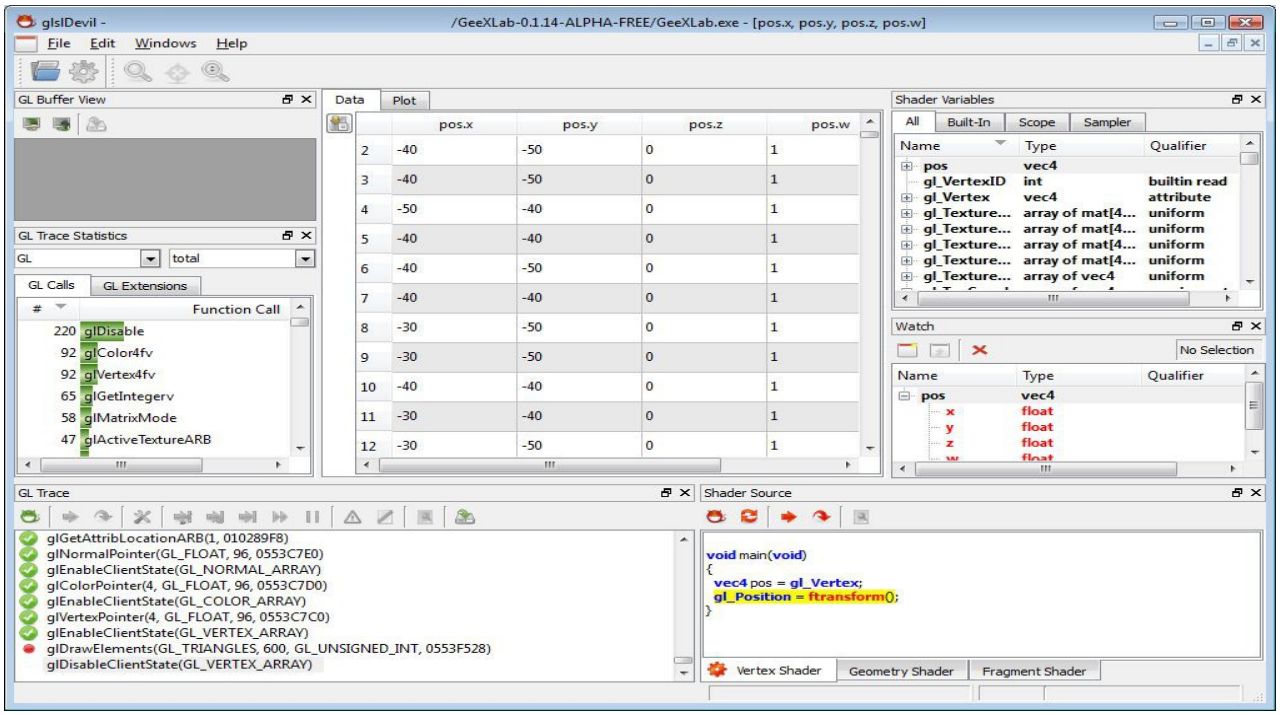
Слика 5а – Изглед алата *BuGLe* приликом посматрања вредности атрибута коначног аутомата библиотеке OpenGL



Слика 5б – Изглед алата *BuGLe* приликом посматрања атрибута једне текстуре

3.3. Алат *glslDevil*

Софтверски алат *glslDevil* [5], чији је изглед приказан на слици 6, примарно се употребљава за анализу и проналажење грешака програма за сенчење, писаних у језику GLSL (OpenGL Shading Language). Међутим, алат може да се користи и за детаљнију анализу оног дела програма који користи библиотеку *OpenGL*. Изабран програм је могуће извршити до неке (означене) наредбе за цртање или до тренутка учитавања програма за сенчење. Тада је могуће дохватити вредност координата свих темена, укључујући и садржај бафера слике. Пресретнутим позивима наредби је могуће изменити вредности стварних параметара, пре њиховог извршавања. Међутим, не постоји могућност интерактивног састављања *OpenGL* програма. Такође, овај алат нема могућност дохватања вредности атрибута и садржаја стекова атрибута коначног аутомата.



Слика 6 - Изглед алата *glslDevil* где се може уочити листа позива *OpenGL* наредби (доле лево) и списак променљивих програма за сенчење (горе десно)

У табели 1, су упоређени алати: *gDEBugger*, *BuGLe*, *glslDevil* и предметни алат *SeeGL*. У табели се може уочити да су особине визуелизације тока обраде података и састављања програма подржане само од стране алата *SeeGL*. Ове две особине значајно доприносе могућностима експериментисања и информисању корисника приликом проучавања рада библиотеке.

Табела 1 – Поређење могућности три издвојена постојећа решења, која се баве анализом рада *OpenGL* библиотеке, са *SeeGL* алатом

	Визуелизација тока обраде података	Састављање *OpenGL* програма	Дохватање вредности атрибута	Анализа садржаја графичке меморије	Мерење перфоманси	Извршавање до задате наредбе
gDEBugger	Не	Пресретање позива	Да	Да	Да	Да
BuGLe	Не	Пресретање позива	Да	Да	Не	Да
glslDevil	Не	Пресретање позива	Да	Да	Не	Да
SeeGL	Да	Интерактивно	Да	Не	Не	Да

3.4. Актуелна истраживања

У овом одељку дат је кратак преглед претходних решења из области унапређивања начина изучавања рачунарске графике, што је примарна намена алата *SeeGL*.

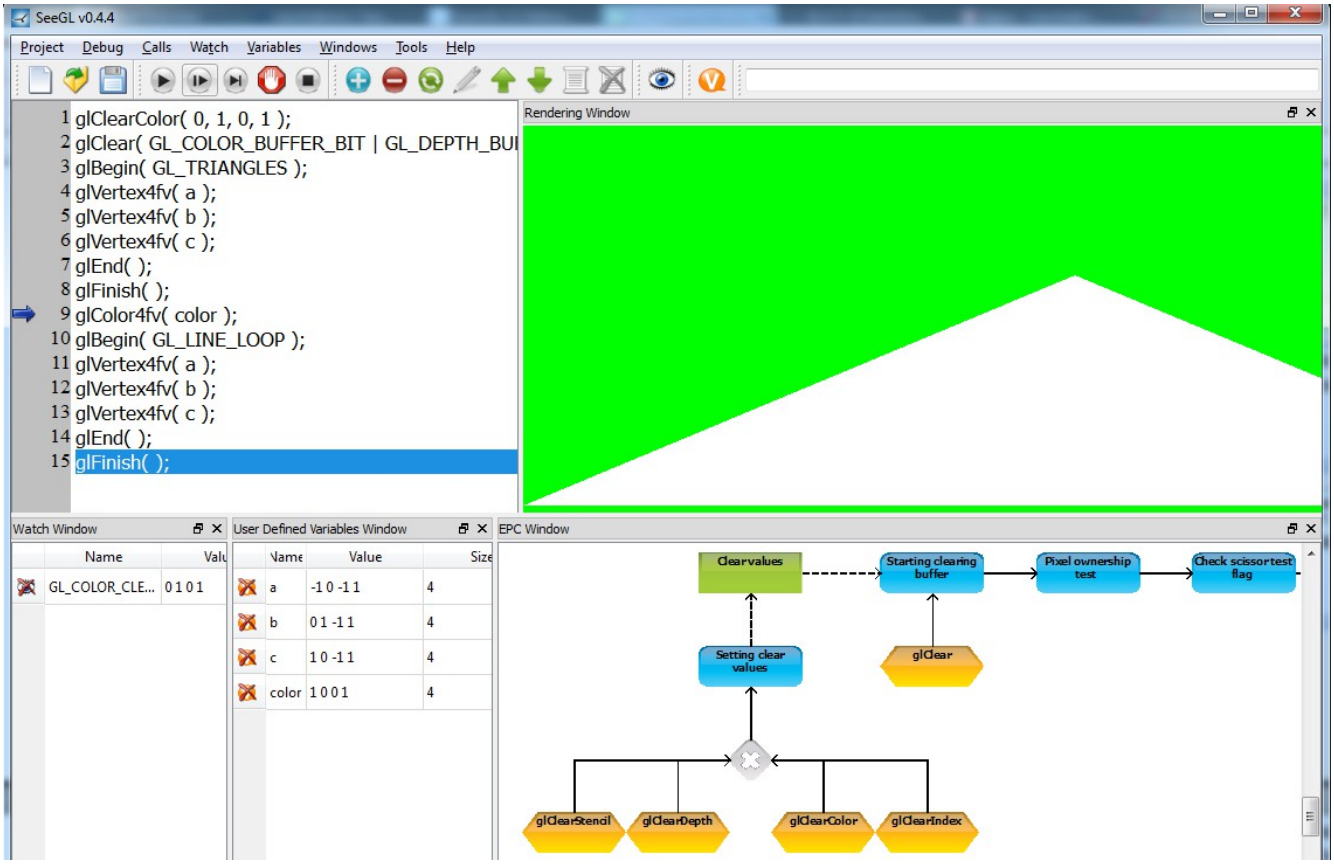
У [7], аутори уочавају проблем напорнијег учења програмирања у области Рачунарске графике, који из године у годину постаје сложенији, јер студенти и поред почетног ентузијазма и интересовања, најчешће одустају од праћења курса због обимног градива, које неретко није директно применљиво у пракси. У раду је упоређена једна уобичајена лекција на предавањима из Рачунарске графике - лекција растеризације криве, са једним примером практичног пројекта – цртање једноставнијег портрета неког лица на основу расположиве слике лица. Истакнуто је како је практични пројекат задржао интересовање студената и да су студенти кроз саму примену, успешно савладали потребно градиво из Рачунарске графике.

У [8] је решаван конкретан проблем разумевања афиних трансформација њиховом визуелизацијом. Систем који је развијен у оквиру [8] приказује резултат цртања у окружењу проширене стварности (енг. *augmented reality*) где су *OpenGL* наредбе за афине трансформације, кодиране у *QR* кодове [9]. Њихов систем се може проширити да подржава и остале *OpenGL* наредбе. Овај систем представља једно атрактивно решење за експериментисање са *OpenGL* наредбама и омогућава кориснику да интуитивно научи и види утицај наредби на крајњи резултат цртања.

Аутор рада [10] препознаје један од најзначајнијих проблема приликом учења састављања програма из области Рачунарске графике. Савремен начин програмирања графички оријентисаног хардвера подразумева рад са проточном обрадом података чије је неке делове могуће репрограмирати програмима за сенчење, за разлику од ранијег решења, где је проточна обрада била непроменљива, због ограничења тада доступног хардвера. Аутор [10] примећује да просечан студент нема знање о традиционалном приступу обради 3D геометрије, који је заступљен у непроменљивој проточној обради. Стога, студентима представља велики напор да проучавање библиотеке *OpenGL* почну од модерног приступа, код којег употреба шејдер програма није само опција, већ неопходност. У [10] се предлаже решење у виду библиотеке *OpenGL Training Wheels* (GLTW), писане на језику C++. Намена библиотеке GLTW јесте да кориснику обезбеди оне функционалности које су теже за разумевање и програмирање, а неопходне су код модерне верзије библиотеке *OpenGL*. Како студент напредује, постепено престаје да користи помоћ библиотеке GLTW.

4. Функционална спецификација

У овом поглављу су приказани најинтересантнији делови функционалне спецификације. Комплетна функционална спецификација софтверског алата *SeeGL* се налази у прилогу А. На слици 7 је приказан уобичајен изглед графичког интерфејса алата *SeeGL*, где је извршен једноставан *OpenGL* програм који боји позадину у зелену боју, а затим црта троугао чију унутрашњост боји белом бојом. Главни прозор, односно прозор са наредбама *OpenGL* програма налази се на горњем левом делу слике. У горњем десном делу слике је приказан прозор са резултатом цртања сцене (Rendering Window). Цртање сцене је завршено *OpenGL* наредбом `glFinish`, чији се позив може приметити на главном прозору. У доњем левом делу интерфејса приказана су два прозора: прозор са атрибутима коначног аутомата и њиховим вредностима (Watch Window), као и прозор за дефинисаним променљивама од стране корисника (User Defined Variables). У прозору са атрибутима коначног аутомата може се приметити резултат наредбе `glClearColor(0,1,0,1)` на атрибут `GL_COLOR_CLEAR_VALUE`. Променљиве приказане у прозору за променљиве могу бити адресиране у наредбама *OpenGL* програма, као што је овде случај са променљивама `a`, `b` и `c`. У доњем десном делу интерфејса (EPC Window) приказан је део шеме *OpenGL* тока обраде где се јасно види да је пре позивања наредбе `glClear` неопходно прво поставити боју којом ће се бојити позадина, ако тренутно постављена боја није задовољавајућа. У наставку објашњене су најбитније функционалности за сваки од прозора.



Слика 7 – Уобичајен изглед графичког интерфејса алата *SeeGL*

4.1. Састављање и извршавање *OpenGL* програма

Као што је већ споменуто, главни прозор садржи секвенцу *OpenGL* наредби која представља тренутно актуелан *OpenGL* програм (учитан из датотеке, или интерактивно састављен). На слици 7 у овом прозору је могуће приметити плаву стрелицу која означава где је програм заустављен, односно указује на наредну наредбу која ће се извршити ако се извршавање програма настави (у даљем тексту – показивач наредне наредбе). На истој слици се може приметити да је плавом бојом обележена наредба `glFinish()`. У даљем тексту плавом бојом обележена наредба је одабрана наредба. Одабрана наредба представља референтну позицију за функционалности едитовања секвенце наредби. Могуће је одабрати и више наредби у секвенци. У наставку ће бити објашњене и могућности састављања и извршавања програма.

4.1.1. Састављање програма

На слици 8 је приказана палета алата за покретање следећих операција: додавање нове наредбе после одабране наредбе, избацивање одабране или одабраних наредби, едитовање одабране наредбе преко графичког дијалога, едитовање одабране наредбе писањем изворног кôда наредбе, промена позиције одабране наредбе са наредбом која претходи (стрелица горе) или која следи (стрелица доле) одабраној наредби, стварање блока од секвенце наредби и растављање блока у секвенцу наредби, респективно гледано са лева на десно.

Слика 8 – Палета алата са функционалностима едитовања програма

Додавањем нове наредбе, корисник има могућност да зада вредности њених параметара. У случају нумеричке вредности параметра, алат проверава грешке прекорачења (енг. *overflow*) и подбацивања (енг. *underflow*). У случају вектора, алат проверава да ли је величина вектора очекивана за тај параметар. У случају симболичких константи, алат нуди само исправне вредности симболичких константи за тај параметар. У случају променљиве, корисник има могућност да адресира постојећу променљиву. Постоји и могућност додавања нове променљиве, која се потом може адресирати и биће приказана у прозору за кориснички дефинисане променљиве (видети одељак 4.3). Нова наредба се додаје после одабране наредбе.

Могуће је едитовати постојећу наредбу тако што јој се само промене параметри или тако што се уместо ње изабере друга наредба (са адекватним параметрима).

Корисник може да едитује наредбу и (ручним) писањем кôда наредбе. Алат проверава исправност написаног изворног кôда наредбе и не дозвољава да се наредба дода у листу позива уколико је неисправна.

Подржана је могућност да се од секвенце наредби формира именовани блок чији се садржај може добити на захтев и који се може расформирати у почетну секвенцу. На сликама 7 и 8 формирање блока наредби је онемогућено, јер је у тренутку настанка слике била одабрана само једна наредба. Такође, могуће је угнездити блокове. Покретањем операције растављања блока у секвенцу наредби, одабрани блок открије њему додељене наредбе и евентуално друге блокове.

Састављен програм је могуће сачувати у датотеци и учитати га из ње. Поред програма у датотеци ће се сачувати све променљиве и њихове вредности.

4.1.2. Извршавање програма

На слици 9 је приказана палета алата којима одговарају операције: извршавање наредби до позиције заустављања (енг. *Breakpoint*), извршавања текуће наредбе, извршавање наредби до прве наредбе за пражњење бафера цртања (наредбе `glFlush` и `glFinish`) не укључујући њу, постављање позиције заустављања и прекид рада коначног аутомата, респективно гледано с лева на десно.

Слика 9 – Палета алата са функционалностима за извршавање наредби, постављање позиције заустављања и ресетовања коначног аутомата

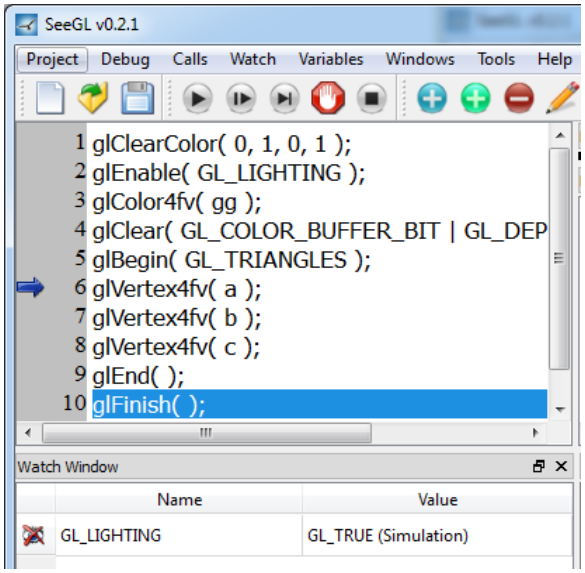
Покретањем било које операције извршавања започиње извршавање од наредбе на коју је претходно указивао показивач наредне наредбе, односно од прве наредбе програма уколико је коначни аутомат претходно био неактиван.

Покретање операције постављања позиције заустављања обележава одабрану наредбу позицијом заустављања уколико наредба није била претходно обележена или поништава обележје одабране наредбе уколико је наредба била претходно обележена.

Покретањем операције заустављања и прекида рада коначног аутомата прво се поставља питање кориснику да ли је сигуран да ли жели да заустави коначни аутомат. Уколико корисник потврди, коначни аутомат ће се зауставити.

4.2. Дохватање вредности атрибута коначног аутомата

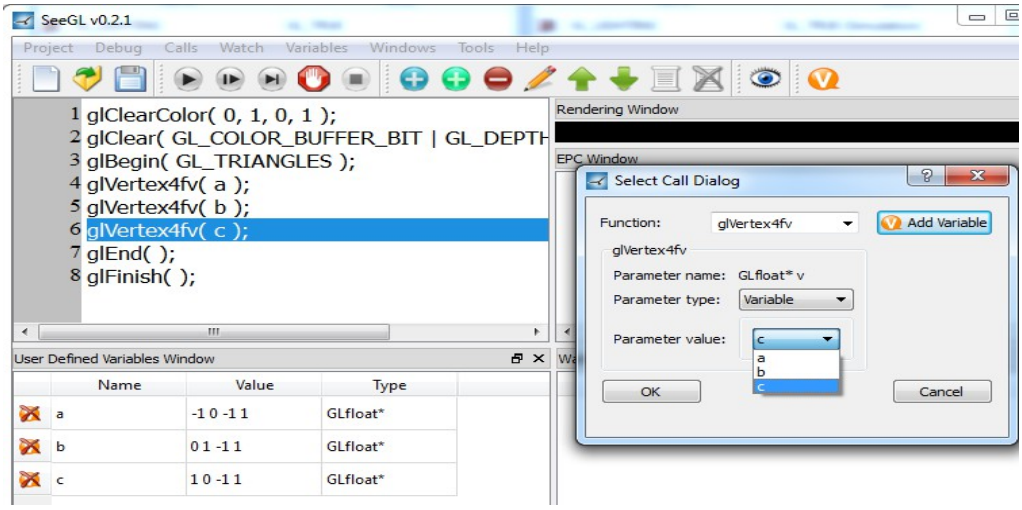
На слици 7 у доњем левом делу главног прозора је приказан прозор за атрибуте коначног аутомата (Watch window). Кад корисник одабере атрибут који жели да посматра, изабран атрибут и његова вредност биће приказани у овом прозору. Уколико је коначни аутомат заустављен између наредби `glBegin` и `glEnd`, он не дозољава да се дохвати стварна вредност било ког атрибута. У том случају, овај алат приказује симулиране вредности атрибута које је корисник одабрао да посматра, са тим да је кориснику назначено да је вредност симулирана (видети слику 10). Пре уласка аутомата у стање у којем није дозвољено дохватање атрибута, ископира се последња вредност атрибута. Ископирана вредност се ажурира после извршавања наредби која има утицаја на тај атрибут, јер вредност атрибута може да се мења и док не може да се чита.



Слика 10 - Приказ симулиране вредности атрибута `GL_LIGHTING` када није могуће дохватити његову вредност из коначног аутомата

4.3. Променљиве које је увео корисник

Прозор за кориснички дефинисане променљиве (User Defined Variables Window) се налази у доњем левом делу слике 11. На истој слици се види отворен дијалог за едитовање наредбе `glVertex4fv`, где корисник бира коју ће променљиву да адресира као параметар наредбе. Кориснику ће бити понуђене само оне променљиве које одговарају типу и величини параметра одговарајуће наредбе.



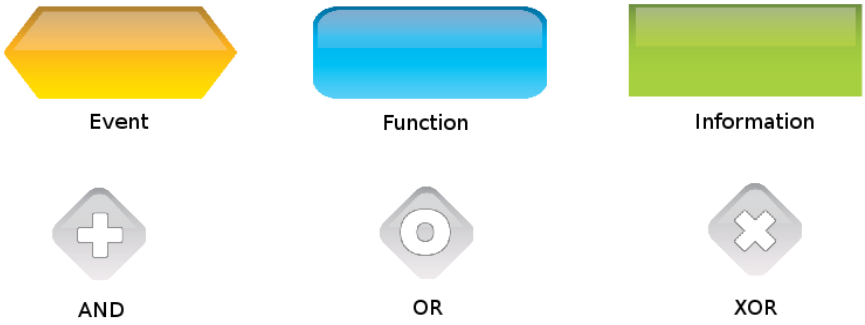
Слика 11 – Прозор за кориснички дефинисане променљиве и дијалог за едитовање наредбе, где је кориснику понуђена листа свих атрибута које може да референцира.

4.4. Шема *OpenGL* тока обраде података

Шематски изглед *OpenGL* тока обраде података је реализован кроз модел ланчања процеса управљаног догађајима (енг. *event-driven process chaining*) [11], у даљем тексту EPC модел. Овај модел се често користи за моделирање пословних процеса (енг. *business processing model*). Модели пословних процеса представљају графичке дијаграме који описују ток или токове неког пословног процеса. EPC модел је изабран због своје једноставности и интуитивне нотације.

4.4.1. Елементи EPC шеме

У овом пододељку су дефинисани елементи EPC модела (видети слику 12) и како ће се користити у овом раду.



Слика 12 – Изглед елемената EPC модела који се користе у шематском приказу *OpenGL* коначног аутомата којег нуди алат *SeeGL*

Елемент “догађај” (енг. *event*) представљен је жутим шестоуглом и дефинише или почетни догађај дијаграма тока или крајњи догађај (исход).

Елемент “функција” (енг. *function*) представљен је плавим правоугаоником заобљених ивица и означава неку активност. Намена активности јесте да за улазне податке генерише излазне податке.

Елемент “информациони објекат” (енг. *information object*) представљен је зеленим правоугаоником и означава скуп неких ентитета који обично представљају улазне или излазне параметре за неку функцију. Све везе с информационим објектима представљају ток података.

Постоје три врсте елемената за контролу тока: "И", "ИЛИ" и "Ексклузивно ИЛИ", у наставку "AND", "OR" и "XOR", респективно. Сваки од ових елемената може бити долазни или одлазни. Долазни елемент има више улаза и један излаз. Одлазни елемент има један улаз и више излаза.

Активан контролни ток на улазу одлазног елемента AND активира све контролне токове на излазу. Долазни елемент AND синхронизује све активне контролне токове на улазима. Тек када су активни сви контролни токови на улазима, активира се контролни ток на излазу.

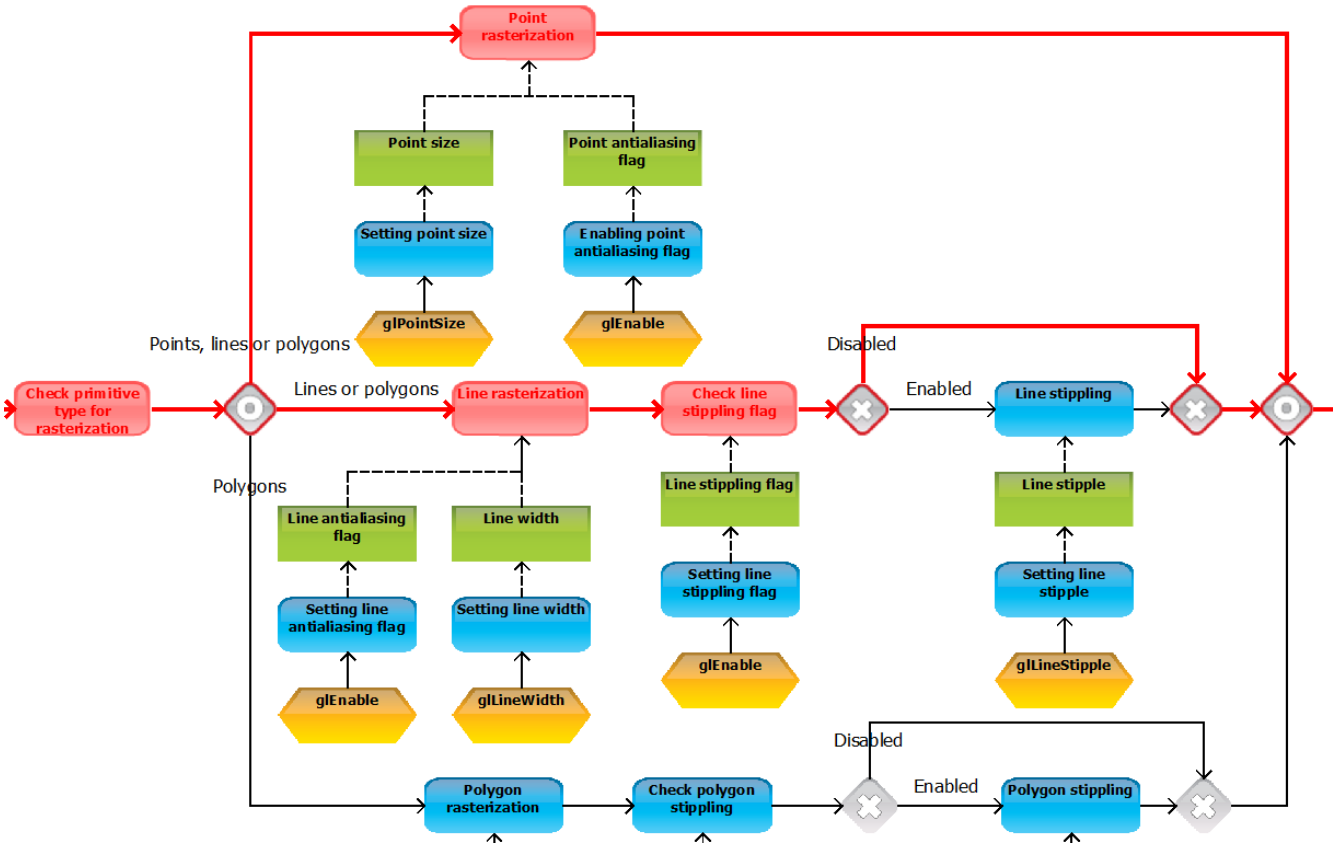
У овом раду, одлазни и долазни елементи OR су увек упарени, тако да је број улаза долазног елемента OR једнак броју излаза упареног одлазног елемента OR. Сваки долазни елемент OR има информацију о томе који су контролни токови активирани од стране упареног одлазног елемента. Активан контролни ток на улазу одлазног елемента OR активира један или више контролних токова на његовим излазима, на основу неког дефинисаног правила. Долазни

OR елемент синхронизује само оне активне контролне токове на улазима који су активирани од стране одлазног елемента OR са којим је упарен. Тек када су на улазима активни сви контролни токови активирани од стране њему упареног одлазног елемента OR, активира се контролни ток на излазу. Скреће се пажња да долазни елемент OR, у ствари, обавља логичку операцију "и" над активираним токовима од стране упареног одлазног елемента OR, вршећи њихову синхронизацију у тачки спајања. Са друге стране, довољно је да буде активан барем један улазни ток долазног елемента, ако је само он активиран од стране упареног одлазног елемента, те у том смислу назив елемента OR одговара.

На слици 13 је приказан део EPC шеме који приказује употребу долазног и одлазног елемента OR, где је потребно одредити које је токове операција потребно активирати на основу типа примитиве која се тренутно обрађује. На слици се може приметити:

- да се приликом обраде тачака активира само ток операција над тачкама (горњи ток),
- да се приликом обраде линија активира ток операција над тачкама (горњи ток) и ток операција над линијама (средњи ток)
- да се приликом обраде полигона активирају сва три тока.

У овом случају се обрађивала примитива типа линије, па је одлазни елемент OR активирао горњи и средњи ток, а долазни елемент OR је синхронизовао та два тока, односно није активирао излазни ток све док оба тока нису завршила све операције.

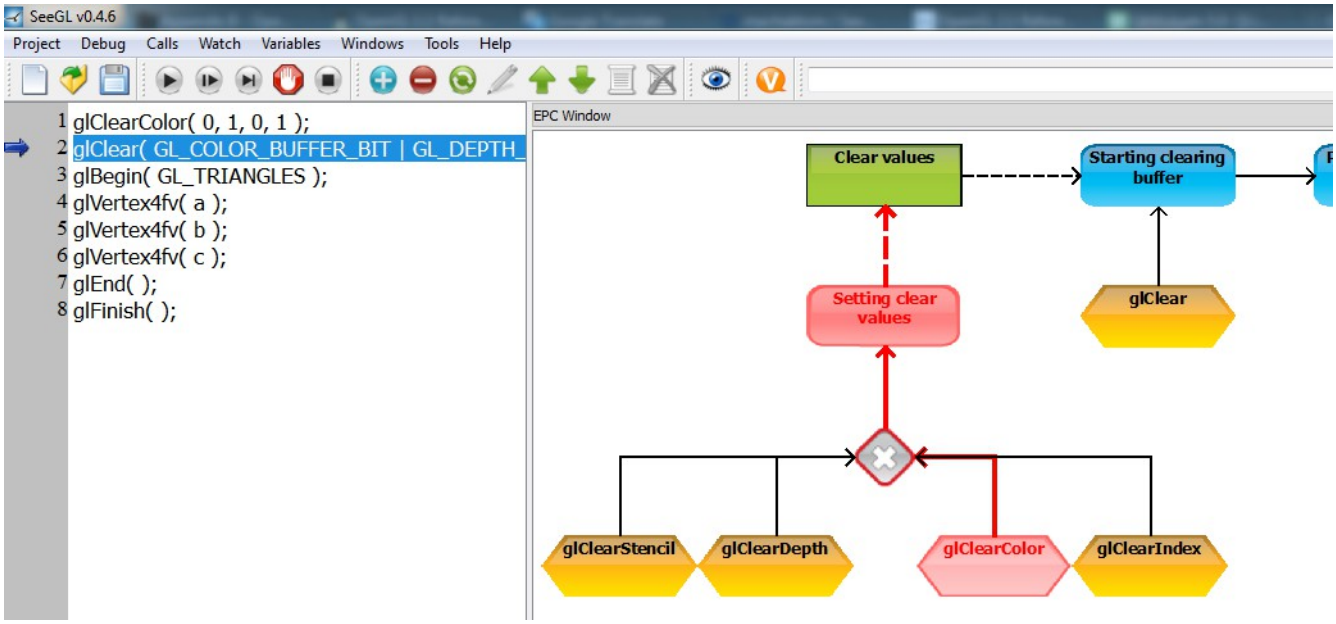


Слика 13 – Део EPC шеме где одлазни елемент OR (лево) активира ток операција над тачкама и ток операција над линијама, јер се у том тренутку обрађује примитива типа линије која захтева операције над тачкама и линијама. Долазни OR, који синхронизује ова два тока, налази се сасвим десно.

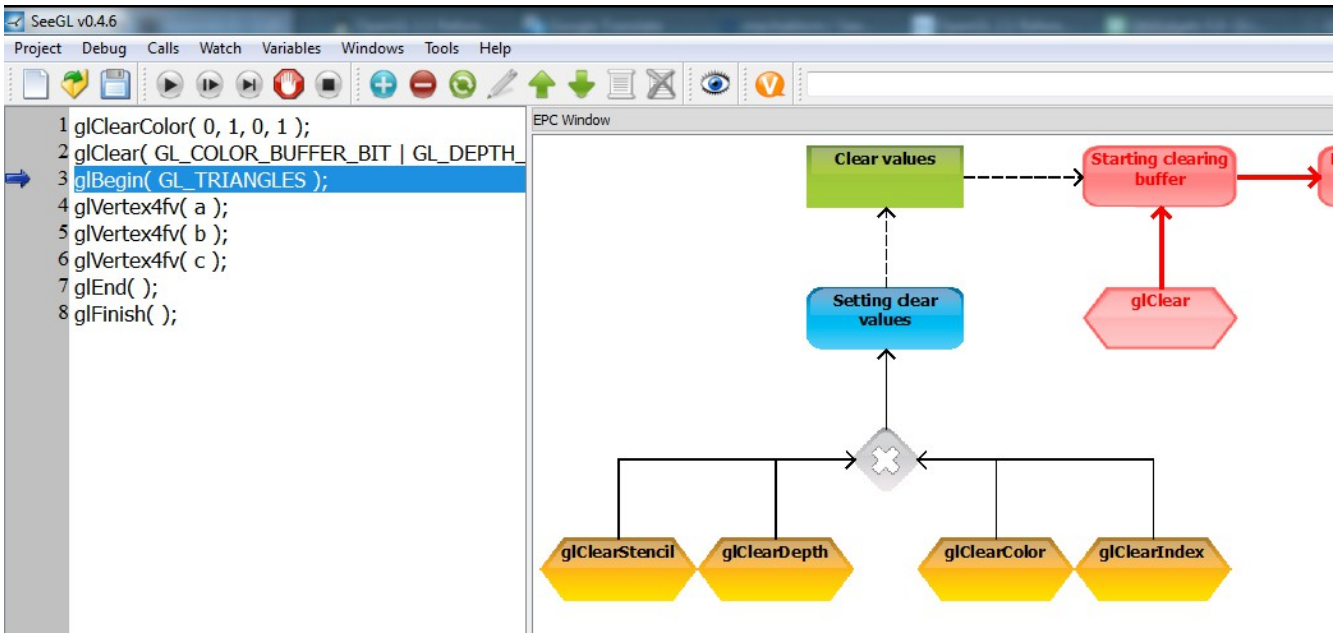
Активан контролни ток на улазу одлазног елемента XOR активира контролни ток на искључиво једном од његових излаза, на основу неког дефинисаног правила. Долазни XOR елемент приликом активирања контролног тока на само једном од улаза, активира контролни ток на излазу.

4.4.2. Визуелизација тока обраде на шематском приказу

Када се коначни аутомат заустави, на шематском приказу ће бити обојен контролни ток који одговара наредби која је последња извршена. На сликама 14 и 15 су приказане две визуелизације на EPC шеми приликом покретања наредби `glClearColor` и `glClear`, респективно. Пуне линије представљају контролне токове. Испрекидане линије представљају токове података. Контролни токови описују редослед операција, док токови података описују кретање информација, које могу бити улазни или излазни параметри неког функционалног елемента. Алат има могућност да се после сваке извршене наредбе, након заустављања аутомата, фокус шематског приказа аутоматски позиционира на елемент шеме који представља дату наредбу у шематском приказу.



Слика 14 – Изглед прозора са шематским приказом након извршене наредбе `glClearColor`.

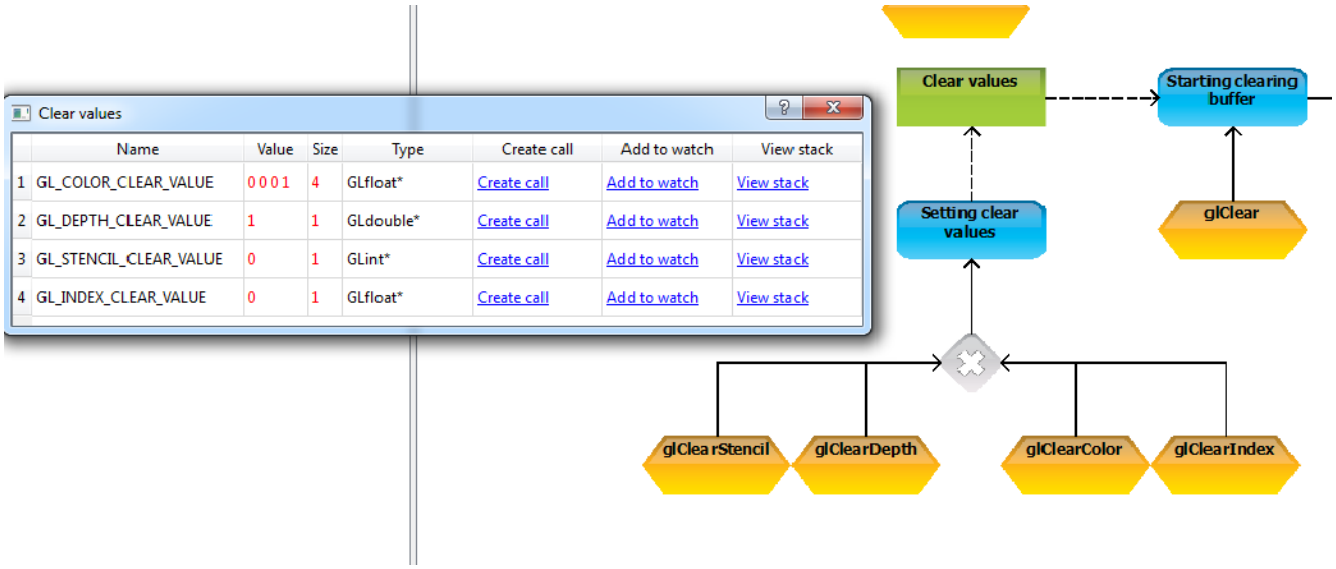


Слика 15 – Изглед прозора са шематским приказом након извршене наредбе `glClear`

4.4.3. Интеракција

Корисник на захтев елементу информационог објекта дохвата скуп атрибута коначног аутомата који одговарају том елементу. На слици 16 је приказан дијалог са скупом атрибута коначног аутомата под називом “вредности за чишћење бафера” (енг. *clear values*), добијен кликом на информациони објекат "ClearValues". За сваки атрибут је могуће прегледати вредност, величину , и тип. Као што је приказано на слици 16, за сваки атрибут су понуђене следеће опције:

- додавање наредбе која има могућност да измени вредност тог атрибута
- додавање тог атрибута у прозор за посматрање атрибута коначних аутомата
- дохватање стека вредности за тај атрибут



Слика 16 – Дијалог скупа атрибута за информациони објекат *ClearValues*

Корисник на захтев елементу почетног догађаја отвара дијалог наредбе која одговара овом догађају. Дијалог је идентичан дијалогу који се отвара приликом додавања нове наредбе у програм (видети пододељак 6.1.3). Додатно, корисник на захтев елементу почетног догађаја, може да отвори мени где свака ставка менија одговара једном, унапред састављеном, позиву наредбе. На избор ставке, одговарајући позив се додаје у програм, након текуће наредбе. Све *OpenGL* наредбе, које као параметре примају симболичке константе, имају унапред састављене позиве наредбе (на пример, `glEnable(GL_NORMALIZE)`).

5. Пројекат и имплементација

У овом поглављу биће дат осврт на технологије коришћене за развој *SeeGL* алата, пројектовање архитектуре софтвера са применама пројектних узорака, решења специфичних проблема при пројектовању софтвера и технички детаљи.

5.1. Технологије

За развој алата SeeGL користио се језик C++ и радни оквир (енг. *Framework*) Qt[12]. Овај радни оквир не зависи од платформе и првенствено је намењен за развој апликација са графичким интерфејсом. Од генералних предности оквира, издваја се механизам за деалоцирање непотребног меморијског садржаја, чиме је програмер, у већини случајева, ослобођен проблема цурења меморије (енг. *memory leak*). За комуникацију између објеката овај оквир нуди механизам сигнала и прикључница (енг. *signal-slot*). За развој *SeeGL* алата посебно је корисна особина овог оквира да подржава све верзије графичке библиотеке *OpenGL*. Треба похвалити и доступну детаљну документацију.

Развојно окружење *QtCreator* представља основно окружење за развој апликација које користе радни оквир Qt. Ово окружење се може користити и за развој C++ апликација које не користе овај оквир. У наставку су наведене неке могућности овог развојног окружења, корисне Qt програмерима:

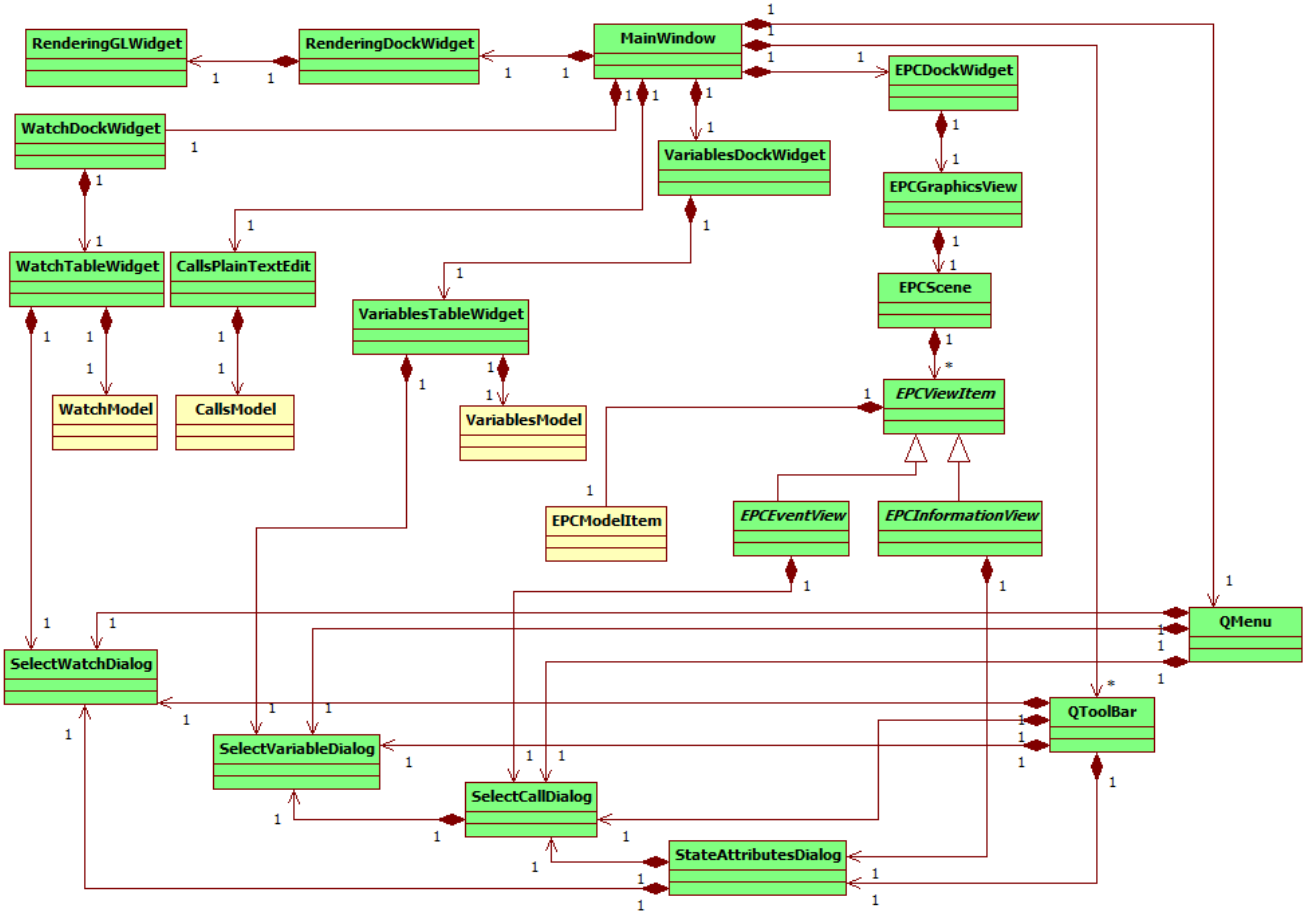
- аутоматско допуњавање и исправљање грешака за време куцања (енг. *auto-complete*) текста изворног кода
- рефакторисање изворног кода
- могућност конфигурисања више преводилаца (*GCC* [13], *MinGW* [14] *Visual C++*[15])
- детаљна документација радног оквира Qt која не захтева везу према Интернету
- подршка за комбиновано визуелно и конвенционално (текстуално) програмирање графичких интерфејса.

5.2. Пројектовање архитектуре софтвера

У овом одељку биће дат осврт на архитектуру софтвера и примере употребе пројектних узорака. Прво ће бити објашњена архитектура дела софтвера који представља интерактивни графички интерфејс, а потом ће бити објашњена архитектура дела софтвера који представља адаптер *OpenGL* библиотеке. На крају одељка су дати примери употребе пројектних узорака.

5.2.1. Пројектовање архитектуре софтвера интерактивног графичког интерфејса

На слици 17 је дат упрошћен UML [16] класни дијаграм софтвера који представља интерактивни графички интерфејс. Изостављене компоненте се могу прегледати у прилогу Б, где су дати обухватнији UML дијаграми класа. Зеленом бојом су обојене класе радног оквира Qt или класе које их наслеђују. Ове класе су задужене за приказ и интеракцију са корисником, док жуто обојене класе представљају логику (модел). Жуто обојене класе не зависе од радног оквира Qt. У наставку текста и на свим сликама, класе чија имена почињу са великим словом "Q" потичу из радног оквира Qt. Остале класе су настале током израде алата *SeeGL*.



Слика 17 – Упрошћен UML класни дијаграм архитектуре софтвера интерактивног графичког интерфејса. Жутом бојом су означене класе језгра апликације (не зависе од радног оквира Qt), а зеленом класе радног оквира Qt, као и оне изведене из њих.

Класа `MainWindow` представља централну компоненту интерактивног графичког интерфејса. Могуће је креирати само један објекат ове класе. Ова класа имплементира главни прозор алата и може да представља посредника у комуникацији између осталих интерактивних графичких компоненти. Ова класа проширује класу `QMainWindow`, која представља главни прозор сваког Qt графичког интерфејса. Класа `RenderingGLWidget` проширује класу `QGLWidget` која омогућава цртање *OpenGL* сцене. Класа `WatchTableWidget` представља интерактивни прозор са табелом атрибута коначног аутомата, које је корисник одабрао да посматра. Ова класа проширује класу `QTableWidget` која омогућава функционалности

приказивања и интеракције са табелом. Класа `CallsPlainTextEdit` представља прозор који приказује *OpenGL* наредбе које чине актуелни програм. Ова класа проширује класу `QPlainTextEdit` и омогућава функционалности приказивања и интеракције са изворним кôдом *OpenGL* програма. Класа `VariablesTableWidget` проширује класу `QTableWidget` и представља интерактивни прозор са табелом променљивих, које је корисник креирао. Класа `EPCGraphicsView` представља интерактивни прозор у коме је нацртана шема тока обраде података. Ова класа проширује класу `QGraphicsView` која представља базну компоненту графичког интерфејса (енг. *widget*) где је могуће сместити неку сцену дводимензионалне графике. Дводимензионална сцена се имплементира користећи класу `QGraphicsScene`. За имплементацију менија и панела са алаткама (енг. *toolbar*) користе се класе `QMenu` и `QToolbar`, респективно.

Може се приметити да су класе `RenderingGLWidget`, `EPCGraphicsView`, `WatchTableWidget` и `VariablesTableWidget` повезане са класом `MainWindow` преко класа `RenderingDockWidget`, `EPCDockWidget`, `WatchDockWidget` и `VariablesDockWidget`, респективно. Објекти класе `QDockWidget` имају могућност да буду независни прозори или уклопљени у главни прозор, усидредни (енг. *docked*) уз његову ивицу (горе, лево, десно или доле). Објекти класа које представљају дијалоге у интерфејсу, наслеђују класу `QDialog`. Дијалози на дијаграму на слици 17 су: `SelectCallDialog`, `SelectVariableDialog`, `SelectWatchDialog` и `StateAttributesDialog`.

Код свих компоненти интерактивног графичког интерфејса, презентациони слој је одвојен од логичког. Презентациони слој искључиво ради са текстуалним подацима (енг. *Strings*), који су добијени из логичког слоја. Ниједна класа логичког слоја не зависи од класа презентационог слоја.

5.2.2. Пројектовање архитектуре софтвера адаптера *OpenGL* библиотеке

На слици 18 је дат упрошћен UML класни дијаграм софтвера адаптера *OpenGL* библиотеке. Изостављене компоненте се могу прегледати у прилогу Б, где је дат комплетан UML класни дијаграм.

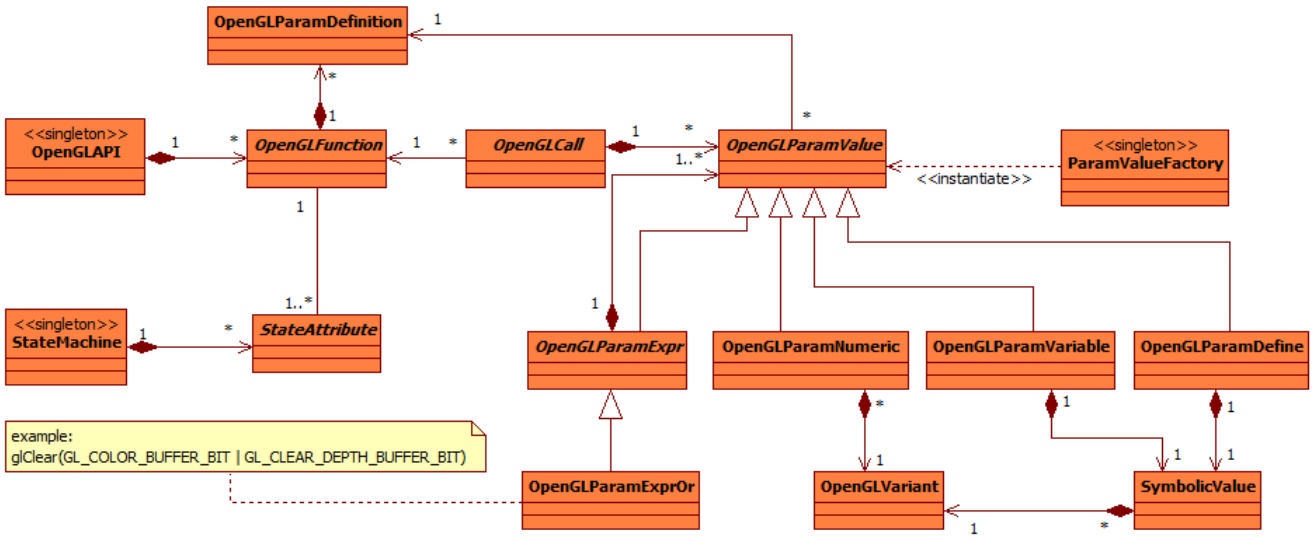
Класе које имплементирају апстрактну класу `OpenGLFunction` пружају програмеру све информације везане за саму функцију (наредбу) библиотеке *OpenGL*. Под таквим информацијама мисли се на: број параметара, дефиниције параметара представљених кроз објекте класа `OpenGLParamDefinition`, назив наредбе, назив наредбе без префикса уколико префикс постоји (нпр. `glVertex` уместо `glVertex4fv`), верзију *OpenGL* библиотеке од које је наредба саставни део библиотеке и тип улазних параметара за наредбе које подржавају више од једног типа улазних параметара (нпр. `glVertex4f`, `glVertex4i`,...). Уникатан објекат класе `OpenGLAPI` представља контејнер где су смештене наредбе односно објекти конкретних класа које имплементирају апстрактну класу `OpenGLFunction`. За сваку наредбу је могуће креирати позив наредбе, представљен апстрактном класом `OpenGLCall`. Претходно је потребно да се одреде вредности параметара, које су представљене апстрактном класом `OpenGLParamValue`. Сама вредност параметара може бити израз. То је решено композитним узорком где је основни елемент класа `OpenGLParamValue`, а композиција апстрактна класа `OpenGLParamExpr`. Као пример, на слици 18 је приказан "или" израз преко конкретне класе `OpenGLParamExprOr`, чија је употреба врло честа у параметрима наредби као што је на пример `glClear(GL_COLOR_BUFFER_BIT | GL_CLEAR_DEPTH_BUFFER_BIT)`.

Класе `OpenGLParamNumeric`, `OpenGLParamVariable` и `OpenGLParamDefine` представљају вредност параметра наредбе, односно нумеричку вредност, променљиву и константу, респективно. Класа `SymbolicValue` представља симболичку вредност која се састоји од имена и вредности. Све вредности типова из *OpenGL* библиотеке (нпр. `GLenum`, `GLint`, `...`) су представљене класом `OpenGLVariant`. При креирању, објекту ове класе се додељује *OpenGL* тип, који се може накнадно изменити.

Класа `OpenGLParamDefinition` омогућава проверу типа, проверу и дохватање расположивих опција уколико се ради симболичкој константи или дозвољених вредности уколико се ради о нумеричкој вредности, проверу величине вектора уколико се ради о векторском или матричном типу.

Уникатни објекти који представљају атрибуте коначног аутомата су класног типа који имплементира апстрактну класу `StateAttribute` и њих је могуће дохватити преко контејнер класе `StateMachine`. Класе које представљају атрибуте коначног аутомата омогућавају дохватање следећих информација: назив атрибута, верзију *OpenGL* библиотеке од када је уведен атрибут, тип атрибута и величину атрибута, додељени елемент EPC модела `EPCModelItem` (видети слику 17) и наредбу (класа `OpenGLFunction`) чији позив подешава вредност овог атрибута.

Из претходног се може закључити да свака наредба (класа `OpenGLFunction`) има информацију о томе које атрибуте (класа `StateAttribute`) коначног аутомата може да модификује, и сваки атрибут има информацију о томе која *OpenGL* наредба може да га модификује. Треба напоменути да библиотека *OpenGL* наизглед има више наредби које модификују неки атрибут (на пример `glColor3f` и `glColor4fv`). Међутим, у питању су сродне наредбе, исте намене, које се разликују једино по типу и броју параметара, па се заправо могу третирати као различити облици једне наредбе.



Слика 18 – Упрошћен UML класни дијаграм архитектуре софтвера адаптера *OpenGL* библиотеке

5.2.3. Примери употребе пројектних узорака

При пројектовању архитектуре софтвера коришћени су следећи пројектни узорци: композиција, класни адаптер, уникат, декоратер и фабрички метод. На слици 19 су приказане употребе ових узорака.

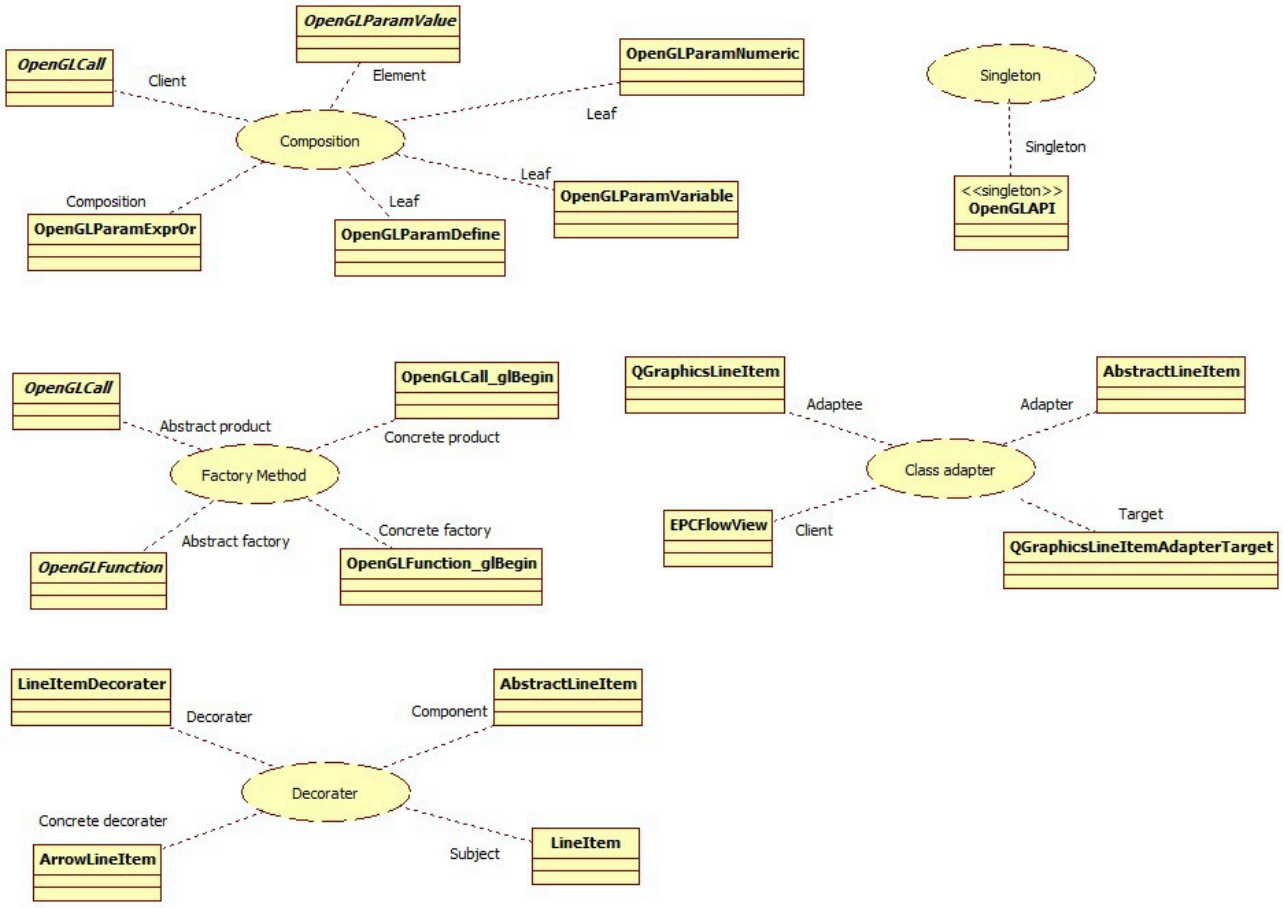
Пројектни узорак композиција (енг. *composition*) употребљен је како би клијент (класа `OpenGLCall`) третирао идентично све објекте у хијерархији односно изразе (класа `OpenGLParamExpr`) као чворове стабла у хијерархији и конкретне вредности (класе `OpenGLParamNumeric`, `OpenGLParamVariable` и `OpenGLParamConstant`) као листове стабла у хијерархији.

Примери узорка униката (енг. *singleton*) могу се видети код класа `OpenGLAPI`, `StateMachine` и `ParamValueFactory`. Могуће је направити само по један објекат сваке од ових класа.

Свака наредба креира одговарајући позив *OpenGL* наредбе. Овде је употребљен узорак фабричке методе (енг. *factory method*) и на слици 19 је приказан пример да конкретна фабрика наредбе `glBegin` (класа `OpenGLFunction_glBegin`) креира позив *OpenGL* наредбе `glBegin` (класа `OpenGLCall_glBegin`, која представља конкретан производ у узорку).

Класни адаптер (енг. *class adapter*) је употребљен због неадекватног интерфејса класе `QGraphicsLineItem`. Конкретно, методе ове класе, које подешавају разне параметре линије нису биле виртуелне, а полиморфизам тих метода је био потребан због имплементације разних допуна при цртању линије. Дакле, креиран је класни адаптер `AbstractLineItem` који је наследио по једну апстрактну методу из класе `QGraphicsLineItemTargetAdapter` за сваку од оних метода за које је био потребан полиморфизам. .

Пројектни узорак декоратер (енг. *decorater*) је употребљен како би се обезбедило цртање линије са разним допунама (конкретно, додавање стрелице на крај линије). Тиме је обезбеђен идентичан интерфејс за цртање обичних линија и линија са допунама.



Слика 19 – Примери употребе пројектних узорака у архитектури софтвера алата *SeeGL*

5.3. Решење специфичних проблема

У овом одељку ће бити објашњено на који начин су решени специфични проблеми, наведени у одељку 2.3, који захтевају посебну анализу.

Основни проблем визуелизације дијаграма тока је одређивање опсега информација које су значајне кориснику, а које алат може да му пружи. Приликом анализе посебно је обраћена пажња на информације које су значајаније почетнику него искусном кориснику. Закључено је да би следеће информације биле врло значајне за корисника:

- шема тока обраде, где је визуелно јасно показано на које операције утичу атрибути коначног аутомата, како би корисник брзо добио представу у којој фази обраде је који атрибут значајан;
- атрибути коначног аутомата треба да буду организовани у смислене целине уколико се тако и користе (нпр. атрибути за подешавање извора светлости);
- шема тока обраде мора јасно да покаже која наредба утиче на које организоване целине атрибута;
- шема тока обраде мора јасно да укаже на редослед наредби када је он битан.

Као што је споменуто у одељку 2.3, постоји више начина како да се одреди редослед операција који није дефинисан документацијом. Током пројектовања апликације, уочена су следећа три решења: (1) операције се извршавају секвенцијално случајним распоредом, (2) операције се извршавају паралелно и (3) операције су приказане као једна сложена операција. Одлучено је да се користи друго решење, односно уколико распоред операција није дефинисан ни условљен *OpenGL* спецификацијом, операције треба представити као да се извршавају паралелно. Прво решење би захтевало да се кориснику на неки начин објасни да је приказан распоред случајан. Иако је овај приступ генерално најближе осликава хардверску имплементацију, од овог решења се одустало јер постоји могућност да би збунила корисника јер би корисник могао да случајан распоред усвоји као да је у питању распоред дефинисан спецификацијом. Од трећег решења се одустало јер би оно захтевало организовања делова тока обраде у блокове, који би сакрили редослед неких операција, а који свакако постоји. На пример израчунати подаци који одређују параметре материјала и израчунати подаци који одређују трансформисане нормале темена су неопходни за израчунавање утицаја извора светла и њихов међусобни редослед није условљен спецификацијом. Што значи да би израчунавање материјала и вредности нормала стајало у истом блоку. Међутим, онемогућавање израчунавања извора светлости онемогућава израчунавање података који одређују параметре материјала, али не сме да онемогући операцију трансформације нормала темена, јер се трансформисане вредности нормала могу користити при другим операцијама (на пример при аутоматском генерисању координата у простору текстура). У оваквим и сличним ситуацијама тешко је организовати операције у блокове, јер се на тај начин сакривају информације кориснику.

Други проблем визуелизације дијаграма тока се односи за начин приказивања атрибута коначног аутомата који се могу омогућити или онемогућити; да ли их треба приказати елементом гранања и обојити активан излаз или без елемента гранања где се излазни ток боји у једну боју када је атрибут омогућен односно у другу када није омогућен? Проблем је решен тако што се комбинују оба решења. Наиме, за приказ атрибута за којег не постоји више појава, а који се може омогућити односно онемогућити (нпр. атрибути као што су `GL_LIGHTING`, `GL_NORMALIZE`, итд.), користи се елемент гранања. За приказ више атрибута који су сродни (нпр. атрибути као што су: `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2` итд.), а који се могу омогућити односно онемогућити, не користи се елемент гранања, него се излазни ток података од организационе целине где су организоване све појаве тог атрибута, боји у зависности од стања (црвено ако су сви атрибути омогућени, плаво ако су сви атрибути онемогућени, љубичасто ако су неки атрибути омогућени а неки онемогућени). Корисник може да на захтев добије тачну информацију који су атрибути омогућени, а који нису.

Проблем дохватања података из коначног аутомата, када је он у стању дефинисања примитива и не дозвољава дохватање вредности атрибута, је морао бити решен симулацијом вредности тих атрибута у том стању. Непосредно пре него што се изврши *OpenGL* наредба, којом коначни аутомат прелази у стање дефинисања примитива, у меморији се запамте вредности свих атрибута коначног аутомата. У стању дефинисања примитива, вредности атрибута коначног аутомата се не дохватају из *OpenGL* библиотеке већ из меморије. Уколико позив неке наредбе треба да упише вредност атрибута коначног аутомата у *OpenGL* библиотеку, уписаће је и у меморију. Кориснику је наглашено да се ради о симулираним вредностима.

Проблем дохватања стека за атрибуте коначног аутомата је решен симулацијом стека за сваки атрибут приликом извршавања програма. Друго решење је захтевало скидање елемената са стварног стека како би се дохватила њихова вредност и затим рестаурирање стека. То не би било компликовано када би библиотека *OpenGL* имала посебну наредбу за сваки атрибут или групу атрибута. Међутим, библиотека *OpenGL* поседује наредбе којима се на стек стављају вредности

атрибута из неког подскупа скупа атрибута, а подскупови на које утичу различите наредбе нису увек дисјунктни. На пример, наредбе `glPushAttrib(GL_ENABLE_BIT)` и `glPushAttrib(GL_COLOR_BUFFER_BIT)` додају вредност атрибута `GL_ALPHA_TEST` на стек. При том, прва наредба на стек додаје и вредност атрибута `GL_AUTO_NORMAL`, док друга наредба то не ради.

5.4. Технички детаљи

У овом одељку биће наведене техничке карактеристике алата *SeeGL*.

Табела 2 – Техничке карактеристике алата *SeeGL*

Број линија изворног кôда	33127
Број фајлова	272
Број наредби	17865
Број дефиниција класа	442

6. Закључак

У овом раду представљен је софтверски алат *SeeGL*, који предлаже нов начин у изучавању рада библиотеке *OpenGL*. У основи имплементације ове библиотеке налази се коначни аутомат. Упознавање и разумевање принципа његовог функционисања захтева значајно време и предзнање. Намена алата *SeeGL* је да кроз интерактиван рад и визуелизацију шеме тока обраде атрибута коначног аутомата корисницима олакша проучавање библиотеке и поједностави откривање евентуалних грешака у њеној употреби. Корисник може интерактивно да састави програм и посматра ефекте извршења на стање аутомата. Посматрањем тока проточне обраде, корисник сазнаје кроз које обраде одговарајући подаци пролазе и какви су утицаји тих обрада на резултујућу слику.

Софтверски алат *SeeGL* поред тога што нуди кориснику да састави програм, извршава га до обележене позиције и прегледа вредности атрибута аутомата, нуди и приказ шеме тока обраде података *OpenGL*. Кориснику су обезбеђене разне могућности интеракције са приказом шеме од којих се издвајају: аутоматско позиционирање приказа шеме на елемент шеме који представља наредбу последњег позива, састављање програма кликом миша на елемент који представља наредбу чији се позив жели додати и дохватање груписаних атрибута кликом миша на информациони елемент који их представља. Приликом састављања програма, алат готово да не дозвољава састављање неисправног позива и корисника ослобађа одговорности о алокацији и деалокацији меморије.

Будућа проширења и надградње овог софтверског алата би могле да се одвијају у различитим правцима:

- Надградња подршке за анализу програма за сенчење. Ова подршка треба да обезбеди и дохватање улазних и излазних података из програма за сенчење као што је наведено у поглављу 3, код алата *glslDevil*. Употреба програма за сенчење је обавезан део модерног начина програмирања графичких адаптера.

- Додавање подршке за визуелизацију *DirectX* библиотеке, при чему би корисник могао да види сличности и разлике у подешавању ових библиотека.

- Увоз сложених 3D модела.

- Увоз дигиталних слика које би се користиле као текстуре

- Имплементација узорка интерпретер за *OpenGL* наредбе који подржава сложене математичке изразе за вредности параметара наредби. Додатно, употреба узорка мува за објекте наредби би могла да смањи потрошњу меморије при раду са великим *OpenGL* програмима.

7. Литература

1. D. Shreiner, *OpenGL Programming Guide: The official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7th edition, Addison-Wesley Professional, 2009.
2. A. Sherrod, W. Jones**,** *Beginning DirectX 10 Game Programming*, Course Technology PTR, 2011.
3. *gDEBugger,* http://www.gremedy.com/, последњи приступ 07.2013.
4. *BuGLe,* http://www.opengl.org/sdk/tools/BuGLe/, последњи приступ 07.2013.
5. *glslDevil,* http://www.vis.uni-stuttgart.de/glsldevil/, последњи приступ 07.2013.
6. R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, S. Tagawa, *The OpenCL Programming Book*, Fixtars Corporation, 2010.
7. S. Li, D. Li, "A Progressive Method on Studying and Practicing of Computer Graphic," *Services Computing Conference (APSCC)*, 2012 IEEE Asia-Pacific, 6-8.12.2012., pp. 399-402.
8. C.H. Teng, J.Y. Chen, "An Augmented Reality Environment for Learning OpenGL Programming" *Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC)*, 2012 9th International Conference on Autonomic and Trusted Computing, 4-7.9.2012., pp. 996-1001.
9. T.W. Kan, C.H. Teng, and W.S. Chou, "Applying QR code in augmented reality applications," *International Conference on Virtual Reality Continuum and Its Applications in Industry*, Yokohama, Japan, December, 2009, pp. 253-257.
10. D. Wolff, "How do we teach graphics with OpenGL?," *Journal of Computing Sciences in Colleges archive*, Volume 28 Issue 1, October 2012, pp. 185-191.
11. W.M.P van der Aalst, "Formalization and Verification of Event-driven Process Chains," *Information and Software Technology*, Volume 41, Issue 10, July 1999, pp. 639-650.
12. J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4*, 2nd edition, Prentice Hall, 2008.
13. GCC, http://gcc.gnu.org/, последњи приступ 07. 2013.
14. MinGW, http://www.mingw.org/, последњи приступ 07. 2013.
15. Visual C++, http://msdn.microsoft.com/en-us/vstudio/hh386302, последњи приступ 07. 2013.
16. UML, http://www.uml.org/, последњи приступ 07.2013