Универзитет у Београду
Електротехнички факултет

Паралелизација и визуелизација ForceAtlas2 алгоритма на графичком процесору

Мастер рад

Ментор:
доц. др Марко Мишић

Кандидат:
Теодора Алексић 2016/3144

Београд, септембар 2018.

САДРЖАЈ

1. Увод

Мреже се сусрећу у великом броју области и њима се могу описати разни типови проблема. Могу се наћи у физици, хемији, биологији, па чак и лингвистици и друштвеним наукама. Користе се да представе комплексне релације и процесе. Постоји велик број алгоритама чија је сврха откривање корисних информација које се налазе у мрежама. Међутим, некад је најбољи начин за њихову анализу не компјутерска обрада, већ ручна анализа спроведена од стране човека.

Да би овакве анализе биле могуће, потребно је приказати мрежу на одговарајући начин. Технике које се овим баве се зову технике визуелизације мреже. Визуелизација мреже представља концептуално захтеван проблем. Лоша визуелизација тешко може бити од користи, али зато добра може много тога да открије о мрежи: обрасце, комуне, граничне случајеве, итд. Самим тим постоји велик број алгоритама и апликација које се баве овим проблемом.

Визуелизација мреже је такође и рачунски захтеван проблеме. Већина решења која се баве овим проблемом користе централне процесоре и не разматрају могуће алтернативе. Графички процесори су годинама уназад непревазиђени у погледу компјутерске графике. Такође, њихов потенцијал за захтевна израчунавања је генерално слабо искоришћен. Поред коришћења централних процесора, већина постојећих решења ограничава визуелизацију на 2D простор и занемарује информације које можемо добити из 3D репрезентације.

Циљ овог рада је имплементација апликације за визуелизацију мрежа у 3D простору на графичком процесору. За визуелизацију мреже ће се користи паралелна имплементација *ForceAtlas2* алгоритма који је део *Gephi* софтвера. *Gephi* је алат за визуелизацију и анализу мрежа. *ForceAtlas2* је главни алгоритам који користи за визуелизацију мреже. Заснован је на алгоритмима који користе усмерене силе за израчунавање интеракција између тела. Предвиђен је да пружи већу флексибилност при визуелизацији мрежа.

За израчунавање ће се користити OpenCL технологија, а за визуелизацију мреже у 3D простору OpenGL технологија. OpenCL и OpenGL су технологије које одржава *Khronos Group* конзорцијум. OpenCL је стандард за писање програма на хетерогеним уређајима и платформама. OpenGL је стандард за компјутерску графику на хетерогеним платформама. Ове технологије пружају велик степен интероперабилности и дељења ресурса на графичким процесорима.

У првом поглављу ће се изложити *ForceAtlas2* алгоритам. Изложиће се постојеће методе визуелизације мреже и оне на којима је овај алгоритам базиран. Објасниће се његови параметри и начини на који они утичу на резултујућу визуелизацију мреже. У другом поглављу ће се изложити коришћене технологије. Објасниће се основе OpenCL технологије и начин на који она може да се користи за захтевна израчунавања. Потом ће се објаснити OpenGL технологија и начин на који се она користи за прављење компјутерске графике. На крају ће се представити интероперабилност између ове две технологије.

У трећем поглављу ће се изложити детаљи имплементације. Објасниће се општа структура апликације, као и начин функционисања њених делова. У четвртом поглављу ће се изложити корисничка апликација и њен начин коришћења. Објасниће се параметри са којима се апликација покреће и начин коришћења графичког окружења. У петом поглављу ће бити приказани резултати тестирања перформанси апликације.

На крају рада ће бити изложене предности и мане коришћења представљених технологија. Биће понуђено алтернативно решење за визуелизацију мрежа у односу на већ постојеће. Биће изложене препреке за имплементирање наведених алгоритама на графичком процесору. Циљ рада је да буде почетна тачка за размишљање о проблему визуелизације мрежа и даље коришћење представљених технологија.

2. FORCEATLAS2 АЛГОРИТАМ

Мреже су заступљене у великом броју области. Користе се у хемији и биологији да представе комплексна једињења. Од великог су значаја у изучавању друштвених мрежа. У рачунарској техници се сусрећу при илустровању разних процеса и рачунарских мрежа. Самим тим, идеална визуелизација мреже увелико зависи од њене природе и информација које желимо да добијемо из ње.

Постоји велик број алгоритама који покушавају да пруже универзално решење за овај проблем. Међутим, једно решење често не може да покрије све случајеве, те се модерни алати не фокусирају на нуђење једног решења, већ више њих од којих корисник може сам да изабере најбоље. Неки алата који се овим баве су *Tulip* [1], *Graphviz* [2], *Pajek* [3] и *Cytoscape* [4], али ће се у овом раду фокусирати на *Gephi* и његове алгоритме.

Gephi [5] је *open-source* софтвер за визуелизацију мрежа. Пружа брз и лак начин за манипулисање подацима. Приказује мреже на начин који је уједно естетски леп и погодан за анализу. Служи за анализу мрежа разних типова, за откривање образаца и аномалија у мрежама и разумевање релација између њихових елемената. *Gephi* пружа велик број алгоритама које корисници апликације могу користити за визуелизацију мрежа. Његова главна одлика је могућност праћења извршавања алгоритама уживо и подешавања њихових параметара зарад најбољег резултата.

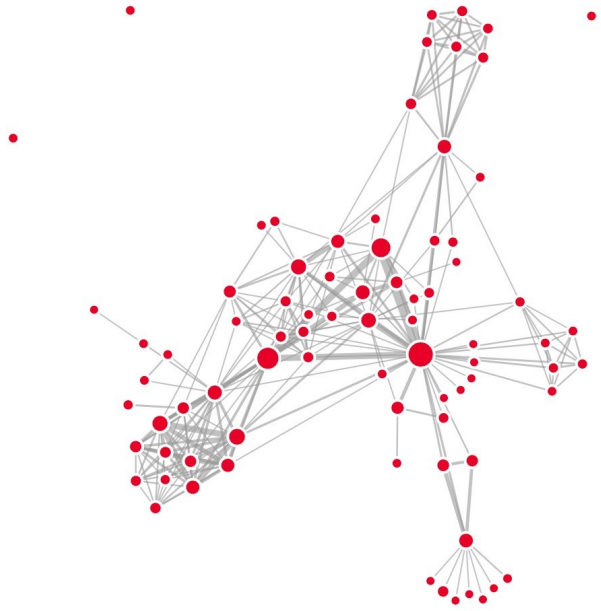
На том принципу је заснован и *ForceAtlas2* алгоритам [6] - алгоритам који су творци *Gephi* апликације дизајнирали. *ForceAtlas2* је заснован на алгоритмима који користе усмерене силе за израчунавање интеракција тела у простору. Није замишљен да буде усавршена верзија тих алгоритама, већ специјализован алгоритам који омогућава флексибилну и ефикасну визуелизацију мрежа.

У даљем тексту ће укратко бити представљене постојеће методе за визуелизацију мреже и алгоритми који користе усмерене силе. Потом ће бити објашњен сам *ForceAtlas2* алгоритам. На крају ће бити објашњене технике које *ForceAtlas2* алгоритам користи за контролисање брзине и прецизности симулације.

2.1. Алгоритми за визуелизацију мрежа

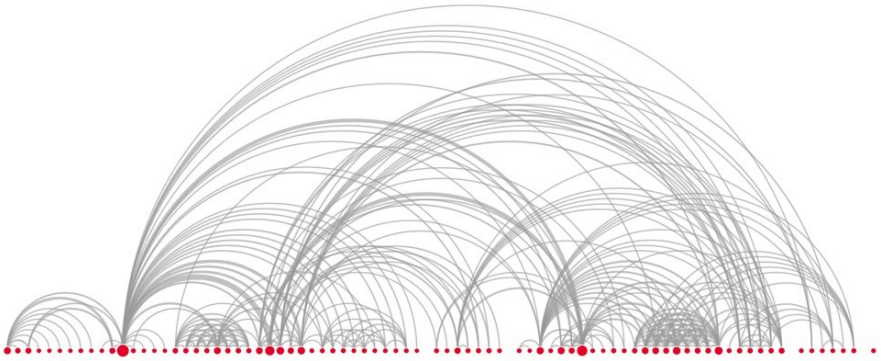
Свака мрежа се састоји из две компоненте: из скупа чворова и скупа грана које повезују те чворове. Традиционално се чворови представљају као кругови, а гране као линије које их повезују. Међутим, овакав приказ не мора увек бити најбољи. Како се мрежа и њене компоненте представе може знатно да утиче на количину и тип информација који можемо да сазнамо из ње. Самим тим постоји и велик број различитих метода које се баве визуелизацијом мрежа [7].

Најчешће методе за визуелизацију мрежа су оне које користе усмерене силе. Ове методе посматрају процес визуелизације мреже као физичку симулацију. Раде по правилу да се чворови графа одбијају, а гране привлаче. Најбоље су за приказ топологије целе мреже. Ова група метода је најзаступљенија и сам *ForceAtlas2* алгоритам спада у њу. На слици 2.1. се може видети пример мреже визуелизоване методом која користи усмерене силе.



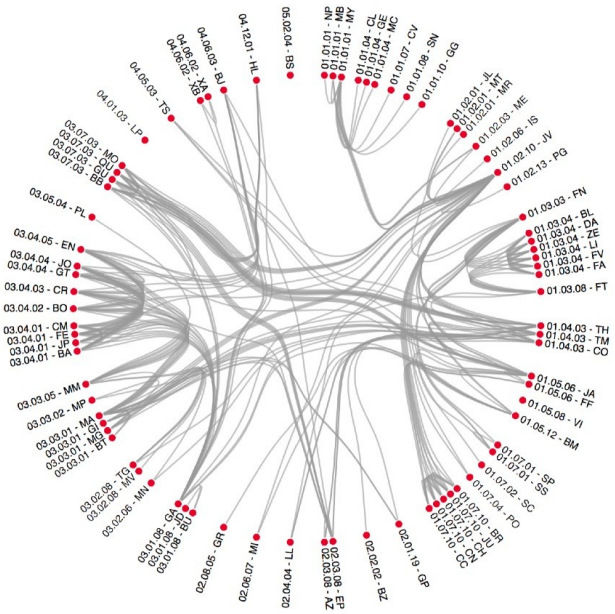
Слика 2.1. Мрежа визуелизована методом која користи усмерене силе [8].

Некада је потребно линеаризовати чворове мреже да би се најбоље уочиле релације између њих. Метода која ово ради је лучна метода репрезентације мреже. Лучна метода распоређује чворове у линију према произвољном редоследу, а гране представља као лукове који их спајају. Распоред чворова може бити азбучни, према степену чвора, или некој другој особини, и знатно утиче на изглед мреже. На слици 2.2. се може видети пример мреже визуелизоване лучном методом.



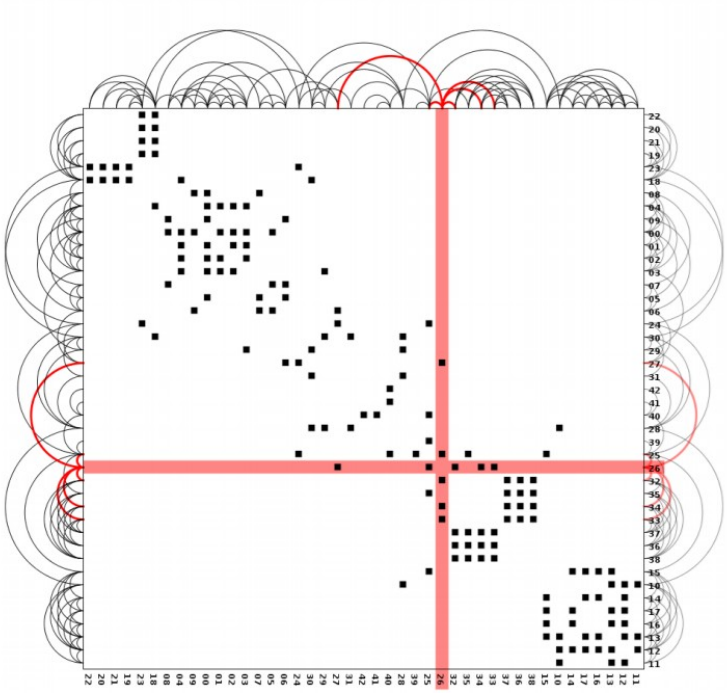
Слика 2.2. Мрежа визуелизована лучном методом [8].

Следећа значајна метода је кружна метода. Она такође распоређује чворове према неком редоследу, али овог пута у круг, а не у линију. Од користи је да прикаже групе чворова и кружне релације између њих. Као и лучна метода, знатно зависи од начина распоређивања чворова. На слици 2.3. се може видети пример мреже визуелизоване кружном методом.



Слика 2.3. Мрежа визуелизована кружном методом [8].

Претходно представљене методе све имају истуману, а то је да могу постати непрегледне у случају да дође до преклапања великог броја грана. Матрична метода решава овај проблем. Она представља мрежу помоћу квадратне матрице где су редови и колоне листе свих чворова мреже. Пресек неког реда и неке колоне представља грану између два чвора који одговарају том реду и тој колони. Ако је пресек попуњен, значи да грана између та два чвора постоји. На слици 2.4. се може видети пример мреже визуелизоване матричном методом.



Слика 2.4. Мрежа визуелизована матричном методом [8].

Свака од наведених метода има своје предности и мане и у складу са њима их треба користити. Такође, ни једна од њих не решава проблем визуелизације великих мрежа и може постати нечитљива за дату огромну количину података. За то, а за и визуелизације мрежа које се фокусирају на њихове друге особине, постоји велик број других метода.

2.2. Алгоритми који користе усмерене силе

Као што је речено, велик број алгоритама који врше визуелизацију мрежа су базирани на алгоритмима који користе усмерене силе. Ови алгоритми су првобитно намењени за физичке симулације и користе се за израчунавање кретања тела у простору под утицајем неке усмерене силе. Ове силе представљају силе које постоје у природи (гравитациона, електромагнетна, итд.), али се могу кориговати за визуелизацију мрежа користећи одлике елемената мреже.

Сам *ForceAtlas2* алгоритам представља имплементацију *Barnes-Hut* алгоритма, а *Barnes-Hut* алгоритам представља оптимизацију директне симулације проблема гравитације *n* тела [9]. Проблем гравитације *n* тела се тиче одређивања трајекторија *n* тела кроз време, у систему у коме делује гравитациона сила. Најједноставнија симулација овог проблема представља директна симулација проблема гравитације *n* тела.

Симулација се одвија из два корака. У првом се израчунавају гравитационе силе којом свако тело делује на свако друго у систему. У другом се израчунавају нове позиције тела у систему на основу добијених сила. За израчунавање гравитационе силе се користи Њутнова једначина силе гравитације:

$$Fij = \frac{G\, mi\, mj}{rij^3}\, rij \qquad (2.1.)$$

где i и j представљају индексе тела у систему, F_{ij} гравитациону силу између њих, G гравитациону константу, m_i и m_j њихове масе, а r_{ij} растојање између њих.

С обзиром да рачуна деловање сваког тела на свако друго, ова симулација је уједно и најзахтевнија симулација овог проблема и има сложеност од $O(n^2)$. Пошто није погодна за примењивање на већим мрежама, постоји велик број њених оптимизација. Најпознатија од тих оптимизација је *Barnes-Hut* алгоритам [10]. Он смањује сложеност симулације вршећи апроксимације помоћу структуре стабла.

Пре почетка симулације, *Barnes-Hut* алгоритам дели простор у коме се симулација извршава у квадранте. Наставља да дели квадранте на подквадранте све док у њима има више од једног тела. При израчунавању гравитационих сила, *Barnes-Hut* алгоритам не пореди свако тело са сваким другим телом, него свако тело са сваким квадратном. Уколико је центар масе квадранта предалеко од тела, израчунава се гравитациона сила којом тело и квадрант делују једно на друго. Уколико није предалеко, тело по истом принципу наставља да се пореди са подквадрантима датог квадранта. Ово смањује сложеност симулације са $O(n^2)$ на $O(nlog(n))$. Прецизност која се губи коришћењем *Barnes-Hut* алгоритма у односу на директну симулацију је занемарљива у односу на побољшање у перформансама. Због тога је овај алгоритам и коришћен за имплементацију *ForceAtlas2* алгоритма.

Међутим, *ForceAtlas2* алгоритам је у *Gephi* алату имплементиран за рад на централним процесорима, док ће се овај рад бавити његовом имплементацијом на графичким процесорима. Алгоритми који конструишу и користе структуре стабла нису погодни за извршавање на графичким процесорима због своје тенденције да дивергирају у току контроле између различитих нити [11]. Због тога, а и због одлуке да се израчунавање и визуелизација извршавају на графичком процесору без међукорака на централном процесору, за имплементацију *ForceAtlas2* алгоритма у овом раду ће бити коришћена директна симулација проблема гравитације n тела.

2.3. Параметри ForceAtlas2 алгоритма

ForceAtlas2 алгоритам се заснива на алгоритмима који користе усмерене силе. Међутим, у мрежама не постоје природне силе које се користе у поменутим алгоритмима за израчунавање кретање тела. Такође, оно што би се сматрало тачном расподелом тела у простору према деловању силе не би нужно чинило и визуелно најбољи приказ тог система. Због тога *ForceAtlas2* алгоритам модификује формуле за рачунање силе које се користе у симулацијама проблема гравитације *n* тела. Води се по једноставном принципу да се чворови графа одбијају, а гране привлаче.

Формула коју *ForceAtlas2* алгоритам користи за израчунавање усмерених сила се састоји из две компоненте: силе одбијања и силе привлачења [6]. Као што назив каже, сила одбијања (означена са F_r) је сила која утиче на одбијање чворова, а сила привлачења (означена са F_a) сила супротног смера која утиче на привлачење чворова. Ове силе зависе од удаљености чворова, тј. ближа тела се више одбијају, а мање привлаче и обрнуто за даља тела.

Основна формула силе привлачења F_a за два чвора n_1 и n_2 је једнака:

$$Fa(n1,n2)=d(n1,n2) \tag{2.2.}$$

где d представља удаљеност између чворова n_1 и n_2. Основна формула силе одбијања F_r за два чвора n_1 и n_2 је једнака:

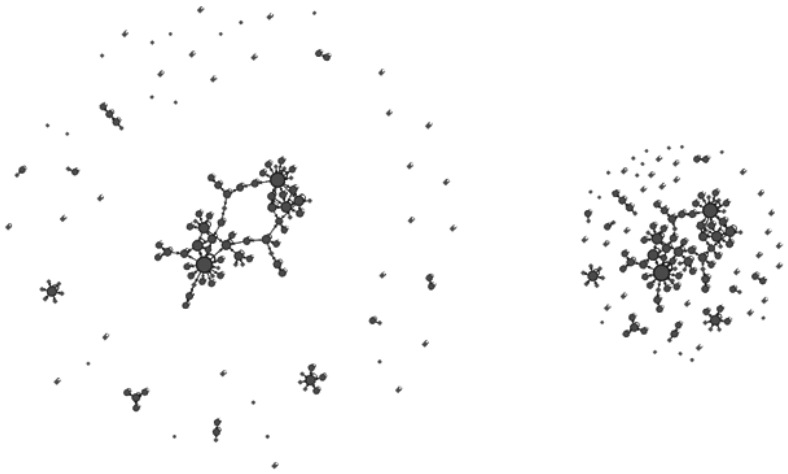
$$Fr(n1,n2)=kr\frac{(deg(n1)+1)(deg(n2)+1)}{d(n1,n2)} \tag{2.3.}$$

где d представља удаљеност између чворова n_1 и n_2, а deg степен чвора. Параметар k_r задаје корисник и утиче на јачину силе одбијања. Што је параметар већи, то је сила одбијања већа. Може се приметити да се степен чвора сабира са један. Ово је урађено да би чворови који немају гране и даље имали силу одбијања. Коначна сила између два чвора се рачуна као збир ове две силе.

ForceAtlas2 алгоритам је дизајниран са великим бројем параметара које корисник може променити и тиме утицати на визуелизацију мреже. Најутицајнији од тих параметара је је додавање треће силе: силе гравитације *Fg*. Сила гравитације спречава мање повезане чворове да се превише раздвоје од остатка графа и чворове да се скупљају око више повезаних чворова. Формула силе гравитације је:

$$Fg(n) = kg(deg(n) + 1) \tag{2.4.}$$

где *deg* представља степен чвора. Параметар k_g задаје корисник и утиче на јачину силе гравитације. Што је параметар већи, то је сила гравитације већа. Могуће је проширити формулу силе гравитације множењем десне стране једначине са вредношћу *d(n)*. Ова вредност представља удаљеност чвора од центра графа и чини силу гравитације јачом, тј. граф визуелно компактнијим. На слици 2.5. се може видети утицај силе гравитације на визуелизацију мреже. На слици лево се може видети утицај силе гравитације где је вредност параметра k_g 2, а на слици десно утицај силе гравитације где је вредност параметра k_g 5.



Слика 2.5. Утицај силе гравитације на визуелизацију мреже [6].

Да би се постигао распоред чворова графа у коме су чворови са већим степеном ближи центру графа, а они са мањим ближи рубовима графа, могуће је додатно модификовати формулу силе привлачења. Модификована формула силе привлачења је:

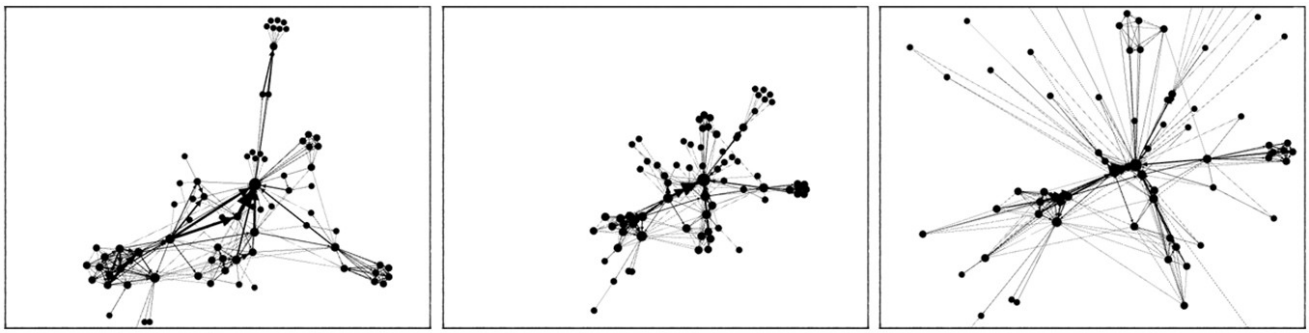
$$Fa(n1, n2) = \frac{d(n1, n2)}{deg(n1) + 1} \tag{2.5.}$$

где *d* представља удаљеност између чворова n_1 и n_2, а *deg* степен чвора. Ова модификација је битна за анализу мрежа јер су често више повезани чворови они који су најзначајнији у мрежи.

У случају да гране графа имају тежину, оне могу утицати на силу привлачења. Што је тежина гране између два чвора већа, то је сила привлачења између та два чвора већа. Формула силе привлачења која узима у обзир тежину грана је:

$$Fa(n1, n2) = w(e)^{\delta} d(n1, n2) \tag{2.6.}$$

где d представља удаљеност између чворова n_1 и n_2, а w тежину гране e. Параметар δ има вредност 1 када алгоритам допушта утицај тежине грана графа на силу привлачења, а вредност 0 када не допушта тај утицај. У случају да гране графа немају тежину, подразумева се да је њихова тежина 1. На слици 2.6. се може видети утицај тежине грана графа на визуелизацију мреже. Са лева на десно, приказан је изглед графа где је вредност параметра δ 0, 1 и 2.



Слика 2.6. Утицај грана графа на визуелизацију мреже [6].

При визуелизацији мрежа потребно је обратити пажњу и на преклапање чворова. Да би се оно спречило, уместо раздаљине између њихових центара, потребно је израчунати раздаљину између њихових ивица. Формула за раздаљину између ивица два чвора је:

$$d'(n1,n2)=d(n1,n2)-s(n1)-s(n2) \tag{2.7.}$$

где d' представља удаљеност између ивица чворова n_1 и n_2, d удаљеност између центара чворова n_1 и n_2, а s функцију полупречника чвора.

Користећи d' могуће је модификовати формуле силе одбијања и привлачења у зависности од три случаја у коме се два чвора могу наћи. У случају да се чворови не преклапају, фомруле су:

$$Fa(n1,n2)=d'(n1,n2) \tag{2.8.}$$

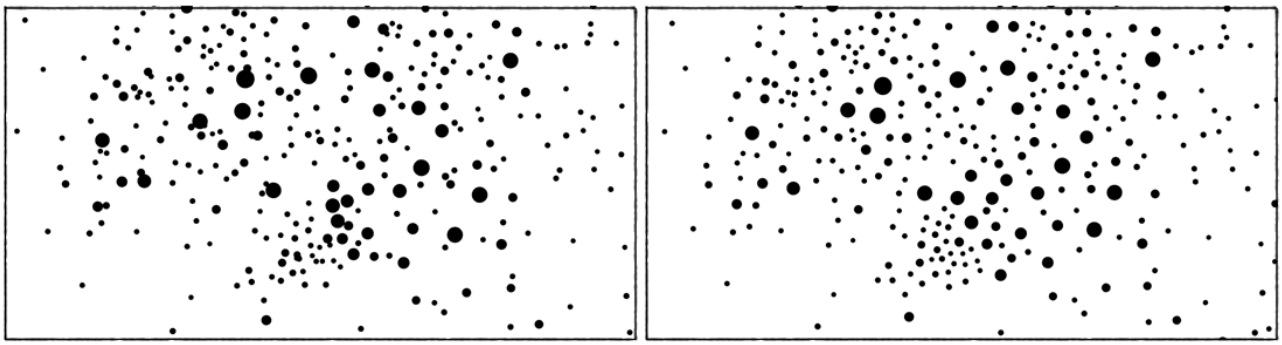
$$Fr(n1,n2)=kr\frac{(deg(n1)+1)(deg(n2)+1)}{d'(n1,n2)} \tag{2.9.}$$

где d' представља удаљеност између ивица чворова n_1 и n_2, а deg степен чвора. Ове формуле представљају стандардне формуле силе одбијања и привлачења. У случају да се чворови преклапају, формуле су:

$$Fa(n1,n2)=0 \tag{2.10.}$$

$$Fr(n1,n2)=kr'(deg(n1)+1)(deg(n2)+1) \tag{2.11.}$$

где *deg* представља степен чвора. Параметар k_r' у *Gephi* алату има вредност 100, али га може задати и корисник. Циљ ових формула је да чворове што пре удаљи један од другог. У случају да се ивице чворова додирују, силе одбијања и привлачења имају вредност 0. На слици 2.7. се са леве стране може видети мрежа при чијој се визуелизацији није водило рачуна о преклапању чворова, а са десне мрежа при чијој се визуелизацији водило рачуна.

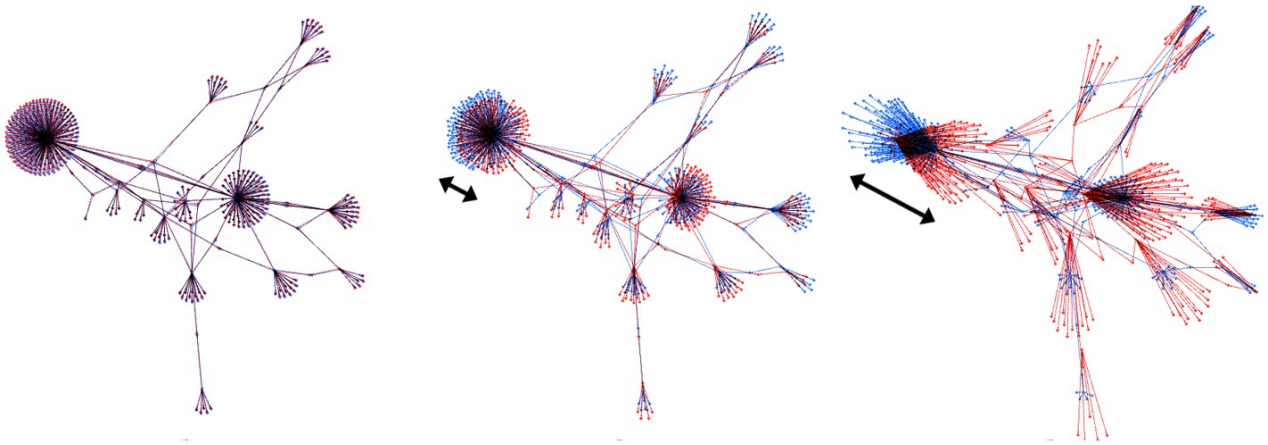


Слика 2.7. Утицај преклапања чворова на визуелизацију мреже [2].

2.4. Брзина и прецизност ForceAtlas2 алгоритма

Брзина и прецизност су два појма која се у оквиру *ForceAtlas2* терминологије користе да опишу кретања чворова графа у симулацији [6]. Што већу раздаљину чворови пређу у току једног корака симулације, то је брзина симулације већа, а прецизност мања. Такође, што мању раздаљину чворови пређу, то је брзина симулације мања, а прецизност већа. Ова два параметра немају идеалне вредности и могу се разликовати од мреже од мреже. Циљ *ForceAtlas2* алгоритма је да подеси ове вредности тако да учине искуство корисника током коришћења апликације што бољим.

Брзина и прецизност долазе до значаја када посматрамо скупове чворова у којима је температура висока. Под тим се сматра већи број збијених чворова који делују великом сумом сила једни на друге. Брзина ових чворова је велика, а прецизност мала. На слици 2.8. се може видети утицај брзине, тј. прецизности на визуелизацију мреже. Са лева на десно приказан је изглед мреже визуелизоване помоћу *Fruchterman-Rheingold* алгоритма са задатим брзинама 100, 500 и 2500. Плавом и црвеном бојом су означени чворови у два сукцесивна корака симулације.



Слика 2.8. Утицај брзине и прецизности на визуелизацију мреже [6].

Да би повећао прецизност симулације и изједначио брзину кретања чворова у целом графу, *ForceAtlas2* алгоритам у току симулације користи технике за кориговање брзине мреже. Ове технике могу бити локалне за дати чвор или глобалне за целу мрежу.

Што се тиче локалне брзине, *ForceAtlas2* дефинише променљиву која одређује осциловање чвора (енгл. *swinging*) поредећи два сукцесивна корака. Формула за осциловање чвора *n* је:

$$swg\,t(n)=|F\,t(n)-F\,t-1(n)| \tag{2.12.}$$

где swg_t представља осциловање чвора у кораку t, F_t силу којом се делује на чвор у кораку t, а F_{t-1} силу којом се делује на чвор у кораку $t-1$. Чвор чије је осциловање близу нуле се скоро уопште неће померити током тренутног корака симулације, док чвор чије је осциловање велико треба кориговати.

Померај чвора *n* се дефинише као:

$$D(n)=s(n)F(n) \tag{2.13.}$$

где D представља померај чвора, s његову брзину, а F примењену силу. Даље се брзина чвора $s(n)$ израчунава формулом:

$$s(n)=\frac{ks\,s(G)}{1+s(G)\sqrt{swg(n)}} \tag{2.14.}$$

где је k_s параметар који у *Gephi* имплементацији има вредност 0.1, *swg* осциловање чвора, а $s(G)$ глобална брзина графа о којој ће се причати у наставку. Као додатна предострожност, брзина се још ограничава формулом:

$$s(n)=\frac{ksmax}{|F(n)|} \tag{2.15.}$$

где је k_{smax} параметар који у *Gephi* имплементацији има вредност 10.

Што се тиче глобалне брзине, у сваком кораку се израчунавају две променљиве: глобална осцилација графа и глобална вучна снага графа (енлг. *effective traction*). Глобална осцилација графа се рачуна као:

$$swg(G) = \sum_1^n (deg(n) + 1)\, swg(n) \qquad (2.16.)$$

где *swg(G)* представља глобалну осцилацију графа, *deg* степен чвора *n*, а *swg* осциловање чвора *n*. Глобална вучна снага се рачуна као:

$$tra(G) = \sum_1^n (deg(n) + 1) \frac{|Ft(n) - Ft-1(n)|}{2} \qquad (2.17.)$$

где *tra(G)* представља глобалну вучну снагу, *deg* степен чвора *n*, F_t силу којом се делује на чвор у кораку *t*, а F_{t-1} силу којом се делује на чвор у кораку *t-1*. Ако чвор настави да се креће истом брзином, његова вучна снага ће бити једнака у сваком кораку. Ако је вучна снага чвора нула, чвор се неће померити. Користећи ове две променљиве, глобална брзина графа се рачуна као:

$$s(G) = \tau \frac{tra(G)}{swg(G)} \qquad (2.18.)$$

где *s(G)* представља глобалну брзину графа, *tra(G)* глобалну вучну снагу, а *swg(G)* глобалну осцилацију. Параметар τ представља толеранцију осцилације и задаје га корисник.

Коришћењем глобалне и локалне брзине могуће је наћи компромис између жељене брзине и прецизности мреже. Примећени су најбољи резултати где осцилација између сила у два сукцесивна корака не прелази 50%, али ова вредност се може разликовати од мреже до мреже. Као што је случај за све параметре *ForceAtlas2* алгоритма, не постоји ни једна идеална вредност која је примењива за сваку мрежу, већ се до жељене визуелизације долази испробавањем и комбиновањем свих ових вредности.

3. Коришћене технологије

Иако су централни процесори и даље задужени за вршење већине израчунавања, графички процесори су све заступљенији и све се чешће користе. Иако не флексибилни као централни процесори, графички процесори су прилагођени за извршавање истих инструкција над велим скуповима података - према такозваном SIMD (*Single Instruction Multiple Data*) моделу. То их чини моћним алаткама и због тога су доминанти на пољу компјутерске графике, а и све чешће за сврхе општег израчунавања. Увелико се користе за обраду видеа и слика, за графичке симулације и компјутерске игре.

Постоји велик број стандарда и технологија које омогућавају писање програма за извршавање на графичким процесорима. Они се разликују почев од саме сврхе, до уређаја и окружења на којима могу да се извршавају. Због тога их је тешко комбиновати и писати програме који се у исто време баве општим израчунавањима и компјутерском графиком. Често је потребно подржати више стандарда да би се обезбедило извршавање неке апликације на уређајима различитих произвођача и различитих оперативним системима.

Делимично решење овом проблему пружају технологије *Khronos Group* конзорцијума. *Khronos Group* [12] је конзорцијум задужен за стварање и одржавање великог броја стандарда за извршавање на хетерогеним платформама. Међу њима спадају стандарди за компјутерску графику, виртуелну стварност, паралелну обраду, неуралне мреже, итд. Неки од најпознатијих, као и они коришћени у овом раду, су OpenCL (*Open Computing Language*) и OpenGL (*Open Graphics Library*).

OpenCL [13] је стандард за писање програма за паралелну обраду за извршавање на хетерогеним платформама и хетерогеним уређајима. Подржавају га платформе великих произвођача као што су: *Nvidia*, AMD (*Advanced Micro Devices*), *Intel*, итд. Такође, може се извршавати на различитим уређајима као што су: централни процесори, графички процесори, акцелератори, процесори за обраду дигиталних сигнала, итд.

OpenGL [14] је тренутно водећи стандард за писање програма за компјутерску графику. Омогућава извршавање програма на графичким процесорима великог броја произвођача: *Nvidia*, AMD, *Intel*, итд. Има широку примену: од научних симулација, до виртуелне стварности и компјутерских игара.

Највеће уско грло у програмима који део свог израчунавања врше на графичким процесорима је копирање података са централног процесора на графички и назад. У случају OpenCL технологије, то представља копирање података на графички процесор ради паралелне обраде и копирање назад на централни процесор ради интерпретације. У случају OpenGL технологије, то представља вршење израчунавања на централном процесору и копирање података на графички процесор ради визуелизације.

Оно што је специфично за ове две технологије, јесте да пружају опције за интероперабилност. Она се огледа у могућности OpenCL и OpenGL технологије да деле исте ресурсе на графичком процесору. На овај начин је потребно копирати податке са централног на графички процесор само на почетку програма, а после вршити израчунавање и визуелизацију на графичком процесору.

Поред интероперабилности, највећа предност ових технологија је њихова портабилност. Имплементације OpenCL и OpenGL стандарда зависе од произвођача уређаја на којима се програми извршавају. Због тога је могуће написати програм који се може извршити на свим уређајима свих произвођача који подржавају специфицирану верзију стандарда. Највећа мана ових технологија уједно проистиче из њихове портабилности. Квалитет њихових имплементација увелико зависи од произвођача уређаја. То може довести до великих разлика у перформансама између различитих уређаја.

Овај рад се неће бавити предностима и манама портабилности ове две технологије. Уместо тога ће се фокусирати на њихову интероперабилност. Прво ће се објаснити основни појмови OpenCL и OpenGL технологија и дати примери програма писаних помоћу њих. Потом ће бити објашњен начин на који ове две технологије могу да се користе заједно и дати примери програма.

3.1. OpenCL

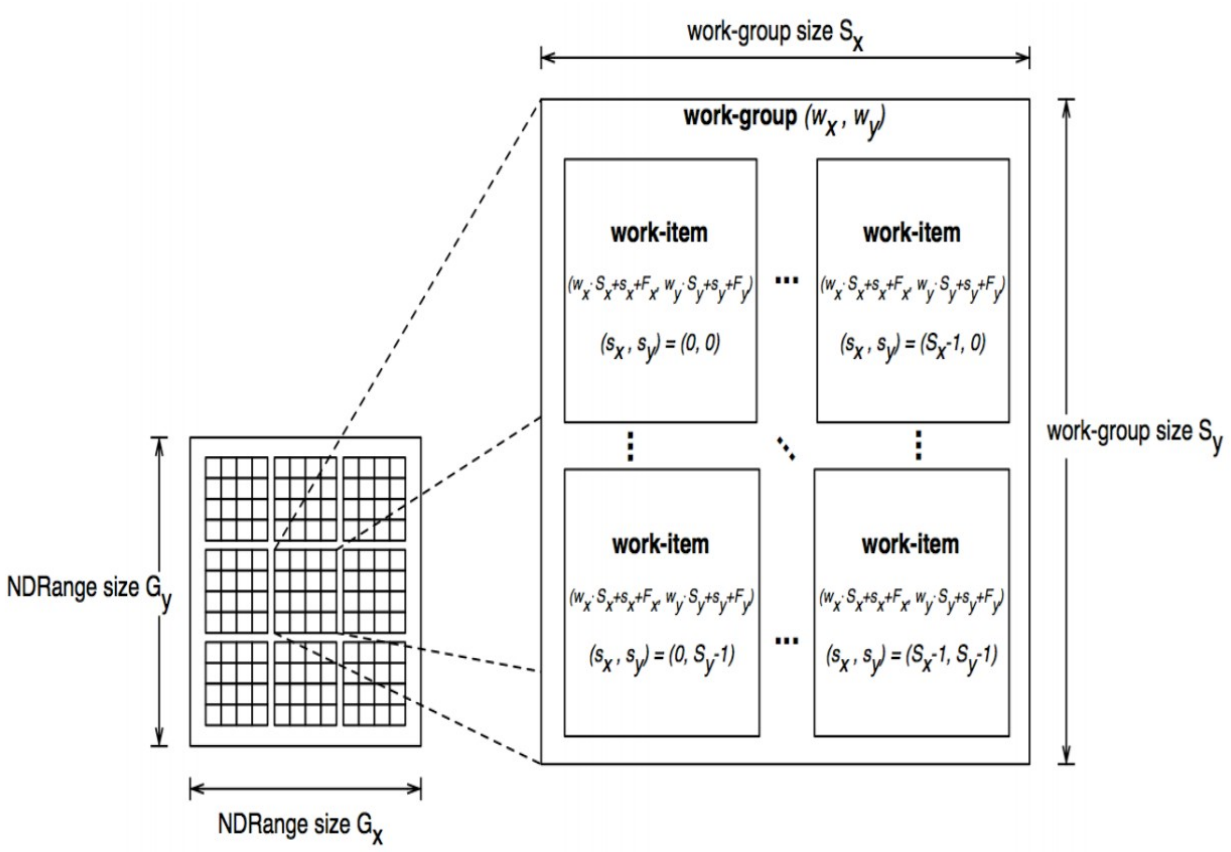
Иако у порасту, коришћење графичких процесора за опште израчунавање и даље није достигло исти ниво као и њихово коришћење за компјутерску графику. Разлог за то може бити што је компјутерска графика проблем за који су графички процесори првобитно дизајнирани, док њихова ефикасност за опште израчунавање зависи искључиво од датог проблема. Због тога на овом пољу не постоји велик број доступних технологија. Тренутно је доминанта CUDA [15] - NVIDIA технологија за опште израчунавање на графичким процесорима, а скорије и *Vulkan* [16] - стандард за компјутерску графику и опште израчунавање на графичким процесорима.

Поред њих, заступљен је и OpenCL - стандард који је прво развио *Apple* и после га предао *Khronos Group* конзорцијуму 2008. године. OpenCL је замишљен да буде стандард за писање програма на хетерогеним платформама и уређајима, али је поготово од значаја његова могућност за извршавање на графичким процесорима великог броја произвођача. Ово му даје знатну флексибилност и предност над својим конкурентима.

Програми који врше паралелну обраду користећи OpenCL технологију се састоје из три дела: копирања података на графички процесор, вршења паралелне обраде на графичком процесору и копирања података назад са графичког процесора [13]. *Kernel* је основна јединица извршавања и представља функцију у којој се врши паралелна обрада. Једна инстанца *kernel* функције која се извршава на датом уређају се зове *work-item* (у даљем тексту: *нит*). Вишедимензионални низ података на коме се врши паралелна обрада у *kernel* функцији се зове *NDRange*. Скуп свих нити на уређају се мапира на елементе *NDRange* низа.

Постоје два начина на који се нити могу груписати на графичком процесору: хардверски и софтверски. Хардверска група чини фиксан број нити на који програмер не може да утиче. Овај број варира од произвођача до произвођача. На NVIDIA графичким процесорима се ова група зове *warp* и величине је 32 нити, док се на AMD графичким процесорима зове *wavefront* и величине је 64 нити. Све нити које упадају у једну хардверску групу извршавају се у истом тренутку на једној извршној јединици уређаја. Ова расподела служи да замаскира кашњење при приступу меморији. Такође, хардверска група представља најнижу јединицу поделе нити на коју утиче ток контроле. Ако бар једна нит унутар ње извршава различит ток контроле од осталих, онда све нити морају да изврше оба тока.

У OpenCL терминологији, софтверска група нити се зове *work-group* (у даљем тексту: група). Број група и број нити у њој су параметри које може да специфицира програмер. Препоручено је да број нити у групи буде дељеник броја нити које садржи једна хардверска група. Група се додељује једној извршној јединици уређаја и нити у оквиру те групе се извршавају само на њој. Служи као виши степен грануларности при расподели посла и синхронизацији нити.

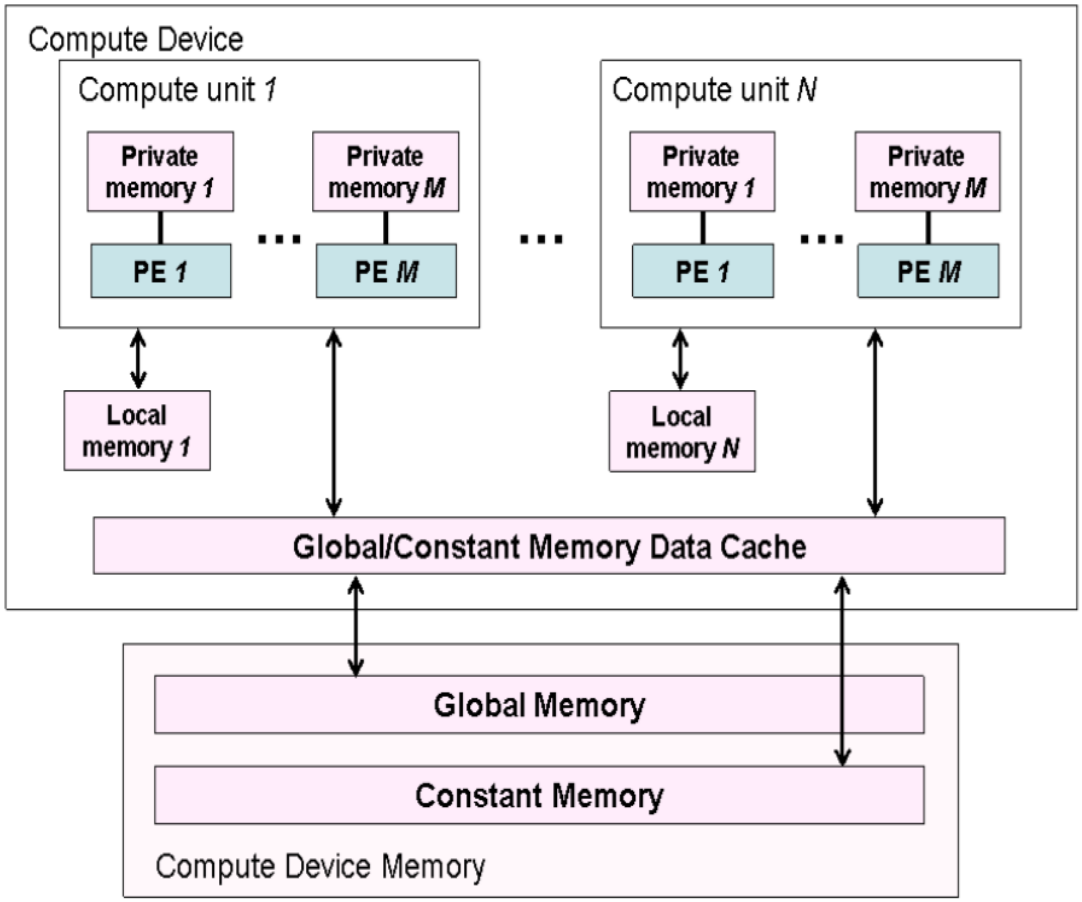


Слика 3.1. Однос *NDRange* низа, нити и групе [13].

3.1.1. Хијерархија меморије

OpenCL дефинише четири типа меморије. Имплементације ових типова меморије зависе од произвођача и типа уређаја. Ови типови меморије су:

- приватна меморија (енгл. *private*),
- локална меморија (енгл. *local*),
- глобална меморија (енгл. *global*),
- меморија константи (енгл. *constant*).



Слика 3.2. OpenCL хијерархија меморије [13].

Приватна меморија је меморија специфична за сваку нит. Представља најбржи тип меморије. На графичким процесорима је најчешће имплементирана кроз регистре опште намене који се налазе на извршним јединицама. Служи за смештање података које користи само једна нит при извршавању. Ако подаци који треба да се сместе у приватну меморију превазилазе њен капацитет, вишак се смешта у посебан део глобалне меморије, што драстично утиче на перформансе.

Локална меморија је меморија специфичну за сваку групу. Представља следећи најбржи тип меморије. Она може бити имплементарана као посебна меморија на извршним јединицама или посебан део глобалне меморије. Служи за смештање података које користе све нити које припадају једној групи.

На модерним графичким процесорима, локална меморија је подељена у меморијске банке. Број ових банки и њихова величина зависе од произвођача уређаја. У једном циклусу се може извршити *load*/*store* операција над свим меморијским банкама. Ако две нити у истом тренутку затраже приступ истој банки, долази до конфликта приликом приступа меморијским банкама. Он се разрешава хардверски тако што се приступи банки серијализују. У случају да нема конфликата, локална меморија је читав ред величине бржа од глобалне.

Глобална меморија је јединствена за цео уређај и могу јој приступи нити које се извршавају на њему као и хост уређај. Дозвољено је читање и писање у произвољне локације глобалне меморије. Нити јој углавном приступају путем алоцираних меморијских објеката.

Меморија константи је такође јединствена за цео уређај. Она је искључиво *read-only* и посао је хост уређај да на њој алоцира и иницијализује простор за променљиве. Њихове вредности се не могу мењати у току извршавања *kernel* функције.

3.1.2. OpenCL програми

OpenCL дефинише C API (*Application Programming Interface*) за писање дела програма који се извршава на хост уређају [13]. Дефинише и C++ API који представља омотач за постојећи C API. Омотач не уводи никакав додатни *overhead* и олакшава алоцирање и ослобађање меморије. Због тога ће у даљим примерима бити коришћен C++ API.

OpenCL програми се извршавају следећим редоследом [13]:

1) Дохватају се све OpenCL платформе на систему;
2) Дохватају се сви OpenCL уређаји на изабраној платформи;
3) Над изабраним уређајима се ствара контекст;
4) Над изабраним уређајем у контексту се прави командни ред (енгл. *command queue*);
5) На основу *kernel* функције се прави се и преводи програм;
6) На уређају се алоцира и иницијализује меморија за аргументе *kernel* функције;
7) На нивоу командог реда се покреће и извршава *kernel* функција;
8) Резултати *kernel* функције се копирају назад на хост уређај.

На слици 3.3. се може видети пример једноставног OpenCL програма.

```
1. // 1) Dohvatanje svih platformi na sistemu
2. std::vector<cl::Platform> platforms;
3. cl::Platform::get(&platforms);
4. // 2) Dohvatanje svih GPU uređaja na prvoj platformi
5. std::vector<cl::Device> devices;
6. platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);
7. // 3) Pravljenje konteksta nad prvim uređajem
8. cl::Context context = cl::Context(devices[0]);
9. // 4) Pravljenje komandnog reda
10. cl::CommandQueue queue = cl::CommandQueue(context, devices[0]);
11. // 5) Prevođenje kernel funkcije
12. cl::Program::Sources sources(
13.     1, std::make_pair(func.c_str(), func.length()));
14. cl::Program program = cl::Program(context, sources);
15. program.build({ devices[0] });
16. // Pravljenje kernel objekta
17. cl::Kernel kernel = cl::Kernel(program, "kernelFunction");
18. // 6) Alociranje memorije na uređaju za argumente funkcije
19. cl::Buffer out = cl::Buffer(context, CL_MEM_WRITE_ONLY,
20.     sizeof(cl_float) * ndRange, NULL);
21. // Postavljanje argumenata kernel funkcije
22. kernel.setArg(0, out);
23. // 7) Pokretanje kernel funkcije
24. queue.enqueueNDRangeKernel(kernel, cl::NDRange(),
25.     cl::NDRange(ndRange), cl::NDRange(workgroup));
26. // 8) Kopiranje rezultata sa uređaja nazad na host
27. queue.enqueueReadBuffer(
28.     out, CL_TRUE, 0, sizeof(cl_float) * ndRange, buffer);
```

Слика 3.3. Пример једноставног OpenCL програма [13].

У оквиру једног система може постојати већи број OpenCL платформи, а у оквиру једне платформе већи број OpenCL уређаја. Могуће је дохватити и приказати особине сваке платформе или уређаја. Контекст се може направити над произвољним бројем уређаја са једне платформе. У оквиру једног контекста, OpenCL може да синхронизује уређаје и контролише ток података између њих. Све обраде које се врше на уређају се раде преко интерфејса командног реда.

OpenCL користи OpenCL C језик за писање *kernel* функција. OpenCL C је базиран на језику C и поред његових основних функционалности поседује додатне вишедимензионалне типове података, уграђене функције, квалификаторе типова, итд. *Kernel* функције се могу писати у одвојеним фајловима или као вредност програмских промењиви, а преводе се у току извршавања коришћењем компајлера специфичног произвођача. На слици 3.4. је приказан пример једне једноставне *kernel* функције.

```
1. // Kernel funkcija koja prima tri niza
2. // smeštena u globalnu memoriju
3. __kernel void kernelFunction(
4.     __global float *a, __global float *b, __global float *c)
5. {
6.     // Dohvatanje globalnog identifikatora niti
7.     int id = get_global_id(0);
8.     // Izračunavanje odgovarajućeg elementa niza
9.     c[id] = a[id] + b[id];
10. }
```

Слика 3.4. Пример тела *kernel* функције [13].

Испред имена функције се ставља кључна реч: *__kernel*. Типови OpenCL меморије се специфицирају стављањем једне од кључних речи испред промењивих: *__private*, *__local*, *__global* или *__constant*. Ако се тип меморије не специфицира, подразумева се да промењива припада приватној меморији. У датом примеру, *kernel* функција прима три низа смештена у глобалну меморију. Свака нит дохвата свој идентификатор путем уграђене функције *get_global_id*. Потом свака нит врши израчунавање над одговарајућим елементима низова.

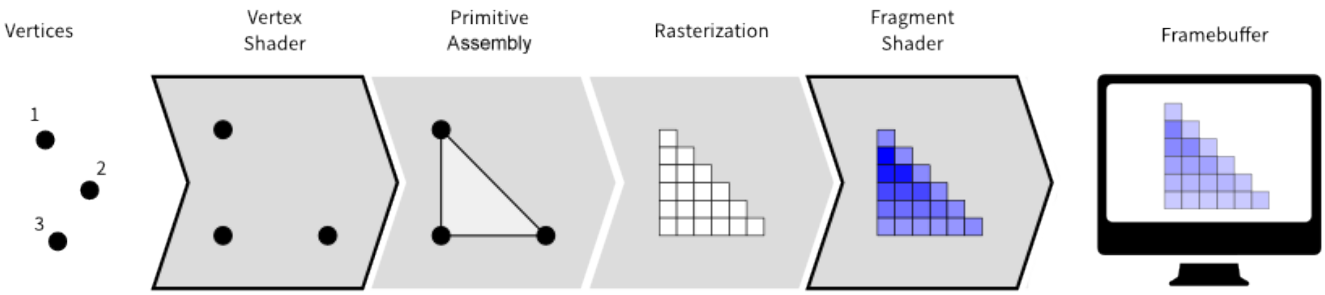
3.2. OpenGL

OpenGL је настао од IRIS GL (*Integrated Raster Imaging System Graphical Library*) стандарда за компјутерску графику [17]. Њега је развила фирма *Silicon Graphics* током раних деведесетих. Касније је стандард променио име и 2006. године био предат *Khronos Group* конзорцијуму за развој и одржавање.

OpenGL је током свог постојања прошао кроз доста фаза. Пре верзије 3.0, свака OpenGL верзија је подржавала све OpenGL функционалности. Од верзије 3.0 па надаље уведен је концепт укидања застарелих функционалности. Са тим концептом уведене су и две врсте OpenGL профила: *core* и *compatibility*. Од верзије 3.0, заједно са OpenGL верзијом потребно је специфицирати један од ових профила. *Core* профил дозвољава само оне функционалности које нису означене као застареле, док *compatibility* омогућава све. У овом раду се користи OpenGL верзија *core* 3.3.

Модерни OpenGL је тренутно један од водећих стандарда за компјутерску графику. Непревазиђен је за писање програма за извршавање на уређајима различитих произвођача и различитим оперативним системима. Поготово је заступљен у индустрији компјутерских игара. Од већих конкурената има *Direct3D* [18] - *Microsoft* API за компјутерску графику направљен за *Windows* оперативне системе и *Mantle* [19] - AMD API за компјутерску графику направљен за AMD графичке процесоре. Такође, од скора постоји и *Vulkan* [16] - још један стандард *Khronos Group* конзорцијума који се бави компјутерском графиком и општим израчунавањем на графичким процесорима различитих произвођача.

OpenGL обрађује све објекте у 3D простору [14]. Прозор апликације се третира као 2D прозор којим се гледа у 3D свет. Сваким позивом неке функције за цртање, OpenGL израчунава позиције објеката у 3D простору, пресликава их на 2D прозор апликације и одређује боју сваког пиксела на екрану. Цео овај процес се извршава на графичком процесору и зове се графички пајплајн (енгл. *graphics pipeline*). Графички пајплајн се састоји из низа функција које се увек извршавају истим редоследом. Излазни параметри једне функције представљају улазне параметре следеће, итд. У OpenGL терминологији, једна функција која се извршава на графичком процесору се зове *shader*. Поједностављен приказ графичког пајплајна се може видети на слици испод:



Слика 3.5. Поједностављен приказ графичког најплајна [20].

Сваки објекат у 3D простору је дефинисан низом темена које га чине. Објекти се конструишу користећи по три темена за склапање једног троугла. Улаз графичког најплајна представља низ темена једног објекта, а излаз матрицу која представља боју сваког пиксела на екрану.

Први корак графичког најплајна представља *vertex shader*. *Vertex shader* као улаз прима по једно теме (енгл. *vertex*) полигона који се црта. Врши трансформације над тим теменом и таквог га прослеђује на свој излаз. Свако теме се налази негде у 3D простору, међутим је само део тог 3D простора видљив кроз прозор апликације. Зато се у кораку између првог и другог врши одбацивање темена која нису видљива кроз прозор апликације. Други корак графичког најплајна представља *primitive assembly*. *Primitive assembly* узима по три темена и од њих склапа једну примитиву: троугао.

Трећи корак графичког најплајна представља *rasterization*. *Rasterization* корак над сваким пикселом на екрану врши проверу да ли се он налази преко неке примитиве у 3D простору. Пиксели који пролазе ову проверу одлазе у слећи корак. Четврти корак графичког најплајна представља *fragment shader*. *Fragment shader* одређује финалну боју пиксела на екрану. Вредност пиксела се на крају обраде уписује у *framebuffer* - бафер чијим се вредностима исцртава нова слика на екрану.

Дужност је програмера да имплементира *vertex* и *fragment shader*. Поред њих, могуће је, али не и обавезно, имплементирати *tessellation* и *geomtry shader* функције, које представљају кораке графичког најплајна између *vertex shader* и *primitive assembly* корака. Остале кораке графичког најплајна није могуће мењати.

3.2.1. GLSL

OpenGL *shader* функције се пишу у језику GLSL (*OpenGL Shading Language*) [21]. GLSL језик је базиран на језику C. Поред основних C функционалности, GLSL поседује додатне векторске и матричне типове података, уграђене функције, квалификаторе типова, итд. Као што је и случај са OpenCL *kernel* функцијама, OpenGL *shader* функције се преводе у време извршавања на циљаном уређају.

Тело *shader* функције се пише у оквиру функције *main*. Изнад ње се специфицира верзија GLSL језика која се користи, као и улазни и излазни параметри *shader* функције. Улазни параметри се означавају кључном речју *in*, а излазни кључном речју *out*. Могуће је специфицирати и параметре *shader* функције који су константе. Ти параметри се означавају кључном речју *uniform*. У односу на OpenCL, OpenGL нема опције за специфицирање типа меморије у којој се налази нека променљива.

GLSL има посебне векторске и матричне типове података који се користе да поједноставе математичке операције. Векторски типови могу имати од две до четири компоненте: *vec2*, *vec3* и *vec4*, а матрични од два пута два, до четири пута четири елемента. Поред ових типова, GLSL има и посебне уграђене функције које врше израчунавања са векторским, тј. матричним типовима података.

Једноставан пример *vertex shader* функције се може видети на слици испод:

```
1. #version 330 core
2. layout(location = 0) in vec3 pos;
3. void main()
4. {
5.     gl_Position = vec4(pos, 1);
6. }
```

Слика 3.6. Једноставан пример *vertex shader* функције [21].

Пошто је *vertex shader* функција први корак графичког најплајна, њени улазни параметри се задају у корисничкој апликацији. Због тога у односу на остале *shader* функције, улазни параметри *vertex shader* функције имају задате индексе који одговарају онима који су им задати у корисничкој апликацији. Ови индекси се означавају кључним речима *layout* и *location*.

Vertex shader као излазни параметар мора да зада вредност уграђеној променљиви *gl_Position* типа *vec4*. Ова вредност представља финалну позицију темена кога *vertex shader* обрађује. Без задавања ове вредности, графички пајплајн се неће извршити. Поред овог обавезног излазног параметра, *vertex shader* може дефинисати додатне излазне параметре који могу бити улазни параметри наредних корака у графичком пајплајну.

Поред *vertex shader* функције, од програмера се тражи да имплементира и *fragment shader* функцију. У односу на *vertex shader*, *fragment shader* може, а и не мора да има улазне параметре. Међутим, као и *vertex shader*, *fragment shader* мора да специфицира излазни параметар типа *vec4*, који представља финалну боју пиксела. Једноставан пример *fragment shader* функције се може видети на слици испод:

```
1. #version 330 core
2. out vec4 outColor;
3. void main()
4. {
5.     outColor = vec4(1.0, 0.0, 1.0, 1.0);
6. }
```

Слика 3.7. Једноставан пример *fragment shader* функције [21].

3.2.2. OpenGL стања и бафери

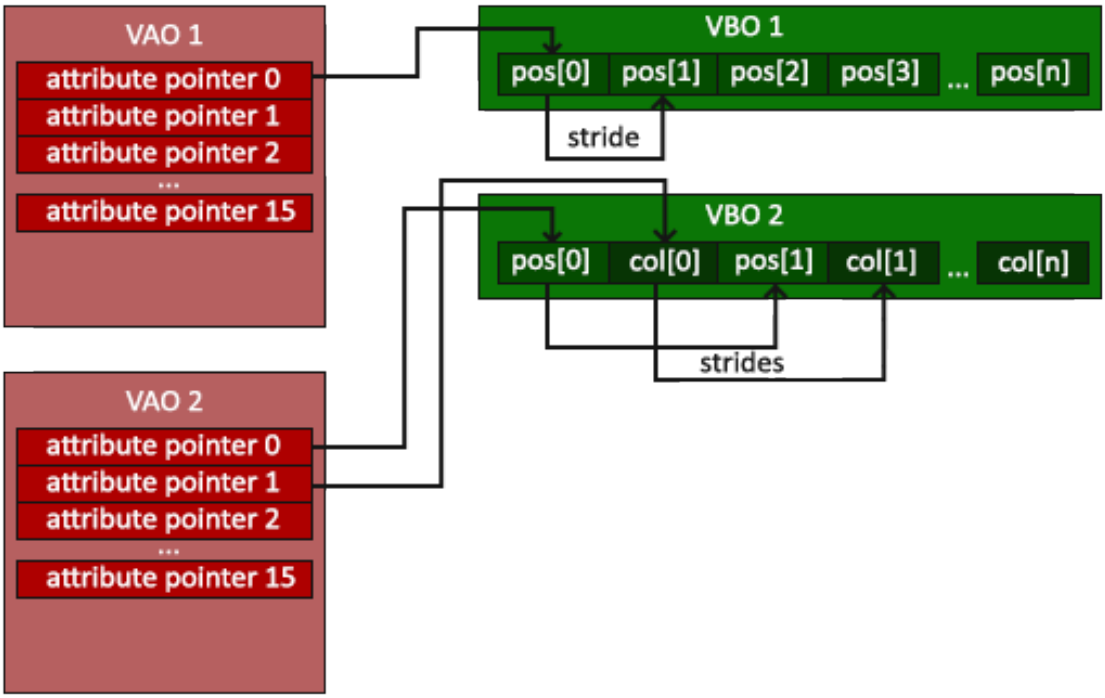
При свакој промени неког од објеката на 3D сцени, OpenGL мора поново да исцрта целу сцену. OpenGL кориснички програми се зато често извршавају у оквиру бесконачне петље која се зове *rendering loop*. У оквиру ње се врши обрада корисничких захтева, израчунавање нових вредности објеката на сцени и позиви OpenGL функција које су задужене за цртање примитива.

Због тога OpenGL ради по принципу коначног аутомата. При сваком извршавању графичког пајплајна, користе се параметри који су последњи специфицирани у корисничкој апликацији. У те параметре спадају *shader* функције, улазни параметри *vertex shader* функције, *uniform* параметри, специјална подешавања, итд.

Улазни параметри се *vertex shader* функцији прослеђују путем VBO (*Vertex Buffer Objects*) бафера [14]. Помоћу VBO бафера се на графичком процесору алоцира меморија и у њу копирају подаци са хост уређаја. Пошто OpenGL ради по принципу коначног аутомата и

пошто се за сваку функцију цртања могу користити различите *shader* функције, самим тим се и тренутно активни VBO бафери морају мењати пре сваког цртања.

Да би се овај процес олакшао, могуће је везати већи број VBO бафера за VAO (*Vertex Array Object*) бафер. Један VAO у себи памти све улазне параметре једне *vertex shader* функције. На тај начин се пре позива неке функције за цртање, уместо везивања сваког VBO бафера понаособ, везује само одговарајући VAO бафер. На слици 3.8 се може видети једноставан приказ VAO и везаних VBO бафера.



Слика 3.8. Једноставан приказ VAO и VBO бафера [22].

3.3. OpenCL и OpenGL интероперабилност

Да би се OpenCL и OpenGL користили заједно, потребно је иницијализовати њихове контексте на истом графичком процесору [23]. У односу на OpenCL, OpenGL API не пружа опције за излиставање и бирање уређаја на ком ће се програм извршавати, већ је оперативни систем задужен за додељивање уређаја OpenGL контексту. Због тога се при успостављању OpenCL и OpenGL интероперабилности прво креира OpenGL контекст. Потом се користећи OpenCL API проналази уређај додељен OpenGL контексту и са њим прави OpenCL контекст. На слици 3.11 се може видети пример иницијализовања OpenCL контекста.

```
1. // Dohvatanje ručke na OpenGL kontekst i dodeljen uređaj
2. cl_context_properties properties[] = {
3.     CL_GL_CONTEXT_KHR,
4.     (cl_context_properties)wglGetCurrentContext(),
5.     CL_WGL_HDC_KHR,
6.     (cl_context_properties)wglGetCurrentDC(),
7.     CL_CONTEXT_PLATFORM,
8.     (cl_context_properties)platform,
9.     0};
10. // Dohvatanje pokazivača na funckiju clGetGLContextInfoKHR
11. clGetGLContextInfoKHR_fn clGetGLContextInfoKHR =
12.     (clGetGLContextInfoKHR_fn)
13.     clGetExtensionFunctionAddressForPlatform
14.     (*platformId, "clGetGLContextInfoKHR");
15. // Dohvatanje uređaja dodeljenog OpenGL kontekstu
16. cl_device_id device;
17. cl_int errorCode = clGetGLContextInfoKHR(
18.     properties, CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR,
19.     sizeof(cl_device_id), &device, nullptr);
```

Слика 3.9. Иницијализовање OpenCL контекста коришћењем OpenGL контекста [23].

За дохватање уређаја додељеног OpenGL контексту користи се фунцкија *clGetGLContextInfoKHR*. На њу је потребно дохватити показивач коришћењем функције *clGetExtensionFunctionAddressForPlatform* у току извршавања. Показивач на функцију се мора дохватати за сваку пронађену OpenCL платформу понаособ.

Разлог за ово представља OpenCL екстензија *cl_khr_gl_sharing*. Присуство ове екстензије на OpenCL уређају, тј. платформи, показује могућност тог уређаја да извршава OpenCL и OpenGL интероперабилне програме. Ако ова екстензија не постоји ни на једном

уређају неке OpenCL платформе, показивач функције ће имати вредност *NULL*. Иначе ће показивати на одговарајућу имплементацију функције.

Позивом функције *clGetGLContextInfoKHR* се могу добити сви уређаји на тренутној OpenCL платформи који имају екстензију *cl_khr_gl_sharing* или само уређај који је додељен OpenGL контексту, ако он припада датој OpenCL платформи. Додатни аргументи функције потребни за проналажење уређаја су ручка на OpenGL контекст, ручка на уређај додељен OpenGL контексту и тренутна OpenCL платформа. Функције за дохватање ручки на OpenGL контекст и њему додељен уређај су зависне од оперативног система. У примеру на претходној слици су ради једноставности коришћене функције *Windows* оперативног система.

Када се успоставе OpenCL и OpenGL контексти, следећи корак представља дељење ресурса између њих. Ово се ради тако што се прво алоцира OpenGL бафер, а потом његов идентификатор користи при прављењу OpenCL бафера. Тако направљен OpenCL бафер представља само омотач око постојећег OpenGL бафера и нема никакву контролу над алоцираном меморијом.

Да би OpenCL и OpenGL програми били коректни, потребно је обезбедити да OpenCL контекст не приступа дељеним баферима док то ради OpenGL контекст и обрнуто. Ово се ради позивима одговарајућих функција које чекају на завршетак OpenCL, тј. OpenGL операција. У случају OpenCL операција, ово чекање се врши на нивоу командног реда. У случају OpenGL операција ово чекање се врши на глобалном нивоу свих покренутих OpenGL операција. На слици 3.12 се може видети пример коректног програма.

```
1. queue.clEnqueueAcquireGLObjects();
2. // OpenCL izračunavanje ...
3. queue.clEnqueueReleaseGLObjects();
4. queue.clFinish();
5. // OpenGL crtanje ...
6. glFinish();
```

Слика 3.10. Пример коректног OpenCL и OpenGL програма [23].

Пре и после OpenCL израчунавања потребно је позвати функције OpenCL командног реда које добијају права, тј. одричу се права над дељеним баферима: *clEnqueueAcquireGLObjects* и *clEnqueueReleaseGLObjects*. После одрицања права над дељеним баферима, потребно је позвати и функцију *clFinish* OpenCL командног реда која чека док се све операције покренуте над тим командним редом не заврше. После тога је могуће вршити OpenGL цртање. После OpenGL цртања потребно је позвати *glFinish* функцију која чека док се све OpenGL операције не приведу крају. Све док су OpenCL и OpenGL операције инкапсулиране функцијама за чекање, овај процес се може понављати произвољним редоследом.

4. Опис имплементације

Програм је развијен у језику C++. Коришћени су OpenCL C++ API и OpenGL C API. Развијан је и тестиран на *Windows* 10 оперативном систему, у алату *Visual Studio* 2017. Тестиран је на NVIDIA и AMD графичким процесорима. Целокупан код, заједно са инструкцијама за покретање и инсталацију и листом коришћених библиотека се може наћи на *GitHub* репозиторијуму [24]. У даљем тексту је описана сама имплементација.

4.1. Главни програм

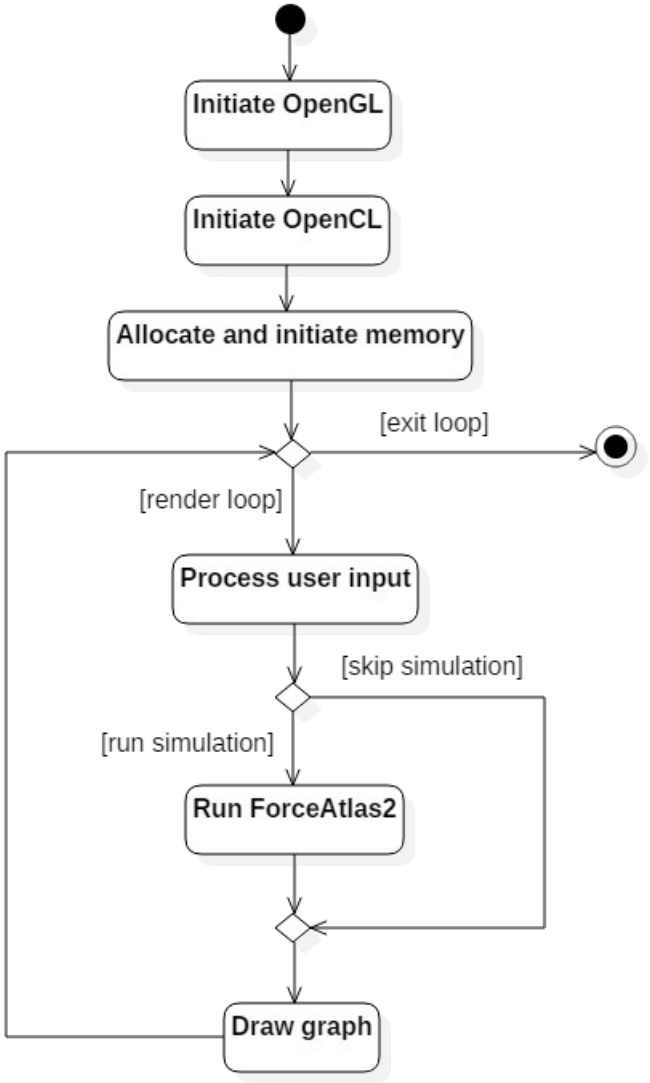
Главни програм се одвија у оквиру петље која је описана у претходном поглављу и зове се *rendering loop*. У њој се извршава *ForceAtlas2* алгоритам и црта граф. Из петље се излази када корисник затражи затварање прозора апликације.

На самом почетку програма се парсирају аргументи команде линије, па потом парсира улазни граф фајл. Подржани формати граф фајла су GEXF (*Graph Exchange XML Format*) [25] и GML (*Graph Modeling Language*) [26]. Више детаља о овим граф форматима се може наћи у прилогу А.

После тога се редом иницијализују OpenGL и OpenCL контексти. У случају да не може да иницира ова два контекста, апликација ће се завршити пре уласка у петљу. Потом се на графичком процесору алоцира меморија и са хост уређаја у њу копирају подаци. Потом се улази у петљу.

На почетку тела петље се очитавају команде корисника. Подржане команде су кретање кроз 3D простор, селектовање чвора графа и покретање *ForceAtlas2* алгоритма. Детаљнија упутства о коришћењу графичке апликације се могу наћи у следећем поглављу. Команда за покретање *ForceAtlas2* алгоритма ће извршити један корак алгоритма. Потом ће се, небитно од претходне акције, исцртати граф. Пошто се сви ресурси налазе на графичком процесору пре уласка у петљу, и пошто се и израчунавање и исцртавање врши на графичком процесору, између ова два корака не постоји никакве додатне синхронизације или потребе за чекањем.

На слици испод се може видети поједностављен приказ главног програма:

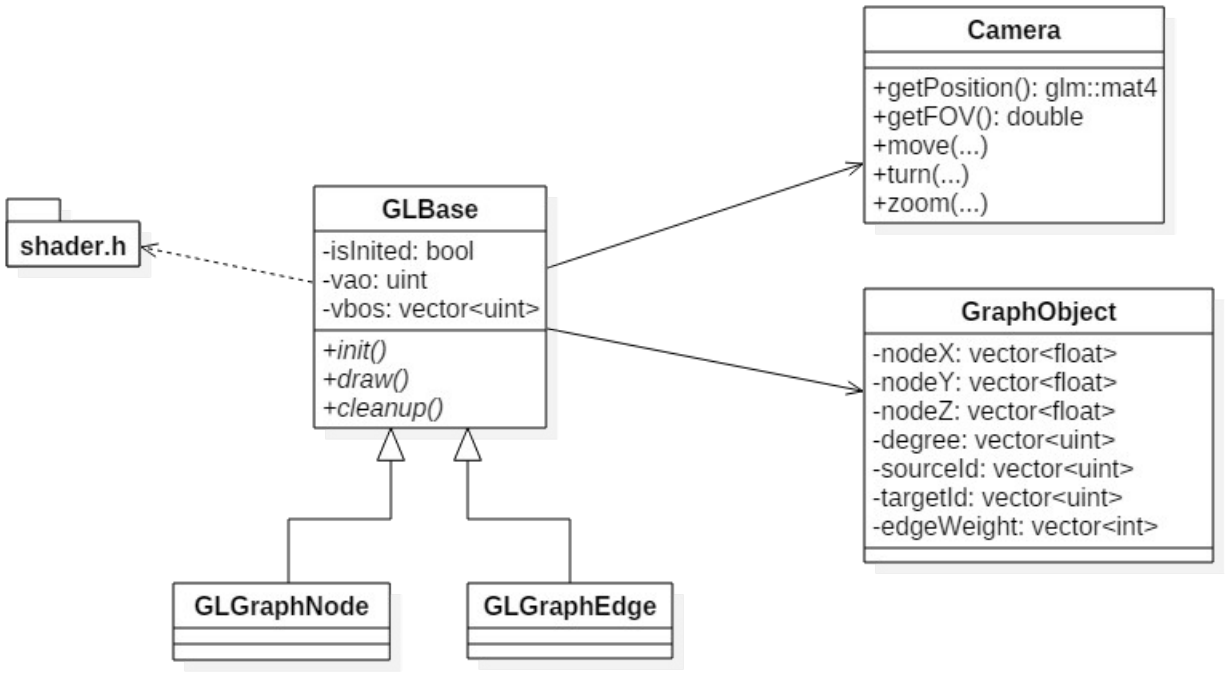


Слика 4.1. Дијаграм главног програма.

4.2. OpenGL класе

Све OpenGL класе које врше исцртавање се изводе из основне $GLBase$ класе. $GLBase$ класа декларише три апстрактне методе које имплементирају њене изведене класе: $init$, $draw$ и $cleanup$. У методи $init$ се иницијализују све променљиве потребне за исцртавање. Метода $draw$ врши исцртавање, а метода $cleanup$ се зове у оквиру деструктора класе и задужена је за ослобађање ресурса.

Поред тога, класа $GLBase$ садржи идентификатор VAO и идентификаторе свих VBO објеката који се користе као аргументи *shader* функција. Сама тела *shader* функција се налазе у оквиру заглавља $shader.h$. Једноставан дијаграм ове класе се може видети на слици испод:



Слика 4.2. Дијаграм OpenGL класа.

Две класе се изводе из класе *GLBase*: *GLGraphNode* и *GLGraphEdge*. Класа *GLGraphNode* је задужена за цртање чворова графа, а класа *GLGraphEdge* за цртање грана графа. Заједно, ове две класе исцртавају целу сцену.

Поред тога, класа *GLBase* садржи референце на објекте друге две класе: *Camera* и *GraphObject*. Класа *Camera* служи за позиционирање корисничког прозору у оквиру 3D простора апликације. Класа *GraphObject* садржи све податке о графу прочитаног из улазног граф фајла.

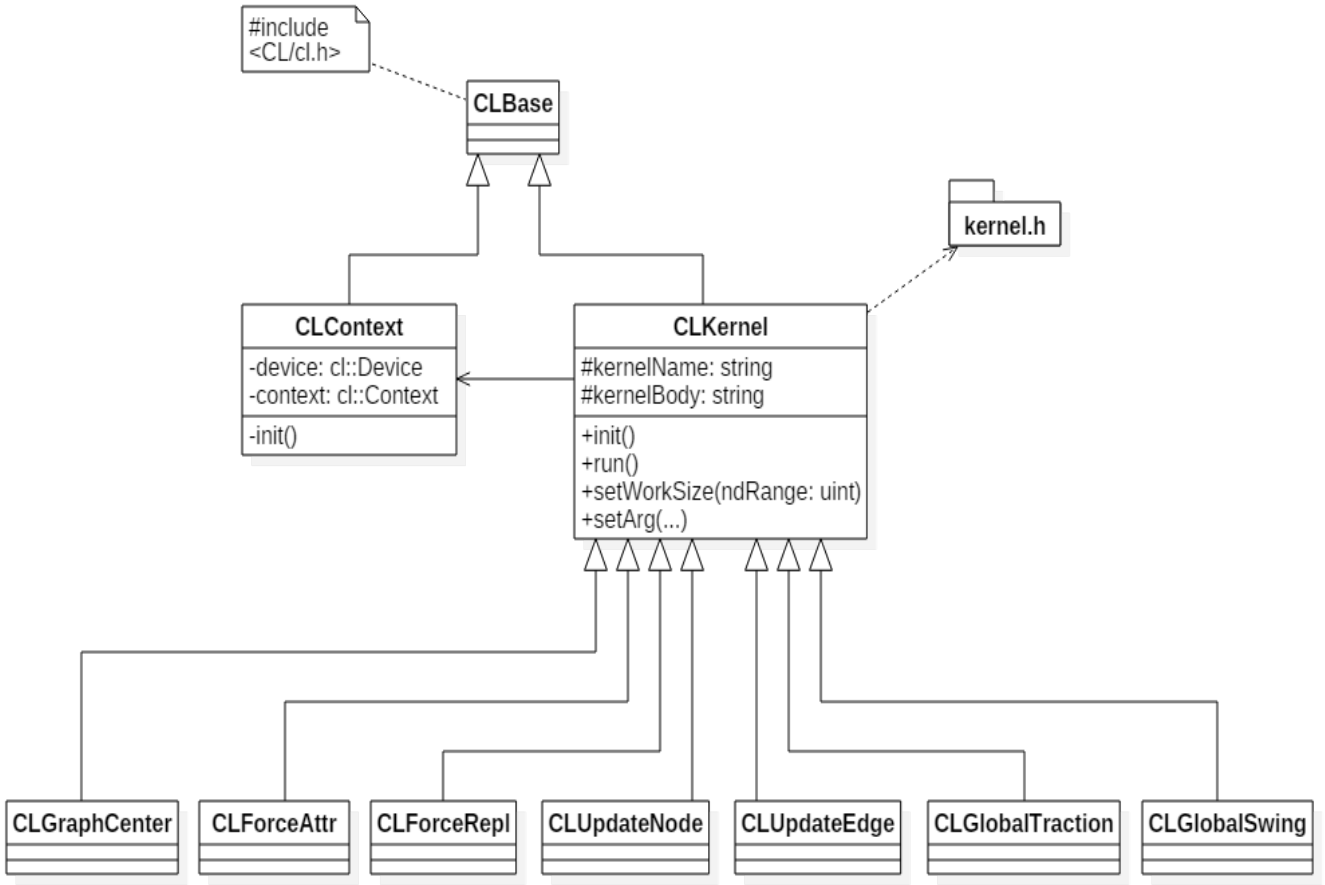
4.3. OpenCL класе

Све OpenCL класе се изводе из основне *CLBase* класе. Ова класа увози OpenCL заглавља и садржи глобалне OpenCL методе и подешавања. Из ње се изводе две класе: *CLContext* и *CLKernel*.

Класа *CLContext* је задужена за иницијализацију OpenCL контекста користећи уређај додељен OpenGL контексту. Садржи иницијализован OpenCL контекст и објекат изабраног уређаја. Користе је све остале OpenCL класе.

CLKernel је основна класа за све OpenCL класе које врше израчунавања. Једна изведена класа ове класе одговара једној *kernel* функцији. Класа *CLKernel* дефинише две

методе: *init* и *run*. У методи *init* се компајлира *kernel* функција и иницијализују промењиве потребне за израчунавање. Сама тела *kernel* функција се налазе у оквиру заглавља *kernel.h*. Метода *run* извршава један корак симулације. Поред тога, класа *CLKernel* садржи методе за постављање аргумената *kernel* функција. Једноставан дијаграм ових класа се може видети на слици испод:



Слика 4.3. Дијаграм OpenCL класа.

Изведене класе из класе *CLKernel* су: *CLGraphCenter*, *CLForceAttr*, *CLForceRepl*, *CLUpdateNode*, *CLUpdateEdge*, *CLGlobalSwing* и *CLGlobalTraction*. Класа *CLGraphCenter* израчунава центар масе графа. Класе *CLForceAttr* и *CLForceRepl* израчунавају силу привлачења, тј. силу одбијања. Класе *CLUpdateNode* и *CLUpdateEdge* ажурирају чворове, тј. гране графа после извршавања алгоритма. Класе *CLGlobalSwing* и *CLGlobalTraction* израчунавају глобалну осцилацију графа, тј. глобалну вучну снагу.

4.4. *Kernel* и *shader* функције

Свака класа изведена из класе $CLKernel$ има себи придружену једну *kernel* функцију, а свака класа изведена из класе $GLBase$ има себи придружену једну *vertex shader* и једну *fragment shader* функцију. Бафери које OpenCL и OpenGL класе деле су бафери за позиције чворова графа, степен чвора графа и позиције грана графа. Поред тога, OpenCL класе поседују бафере за резултујуће силе, бафере за глобалне променљиве и помоћне бафере за израчунавање. OpenGL класе садрже бафере са координатама за исцртавање чворова, тј. грана графа.

Kernel функције се извршавају следећим редоследом: израчунавање силе гравитације, израчунавање силе привлачења, израчунавање силе одбијања, израчунавање глобалне осцилације графа, израчунавање глобалне вучне снаге графа, ажурирање чворова графа и ажурирање грана графа. *Kernel* функције за израчунавање силе гравитације, глобалне осцилације графа и глобалне вучне снаге графа су упарене са посебном *kernel* функцијом за израчунавање суме низа. Свака нит ове три *kernel* функције се извршава над једним чвором графа и свој резултат уписује у помоћни низ величине пропорционалне броју чворова графа. На крају сваке се зове *kernel* функција која израчунава суму свих елемената у помоћном низу и резултат смешта у посебан низ који садржи глобалне променљиве у оквиру алгоритма.

Свака нит *kernel* функција за израчунавање силе привлачења и силе одбијања се такође извршава над једним чвором графа. Свака нит узима себи додељен чвор графа и рачуна силу привлачења, тј. одбијања којом тај чвор делује на сваки други чвор у графу. Резултујућа сила се додаје на укупну силу која делује на тај чвор.

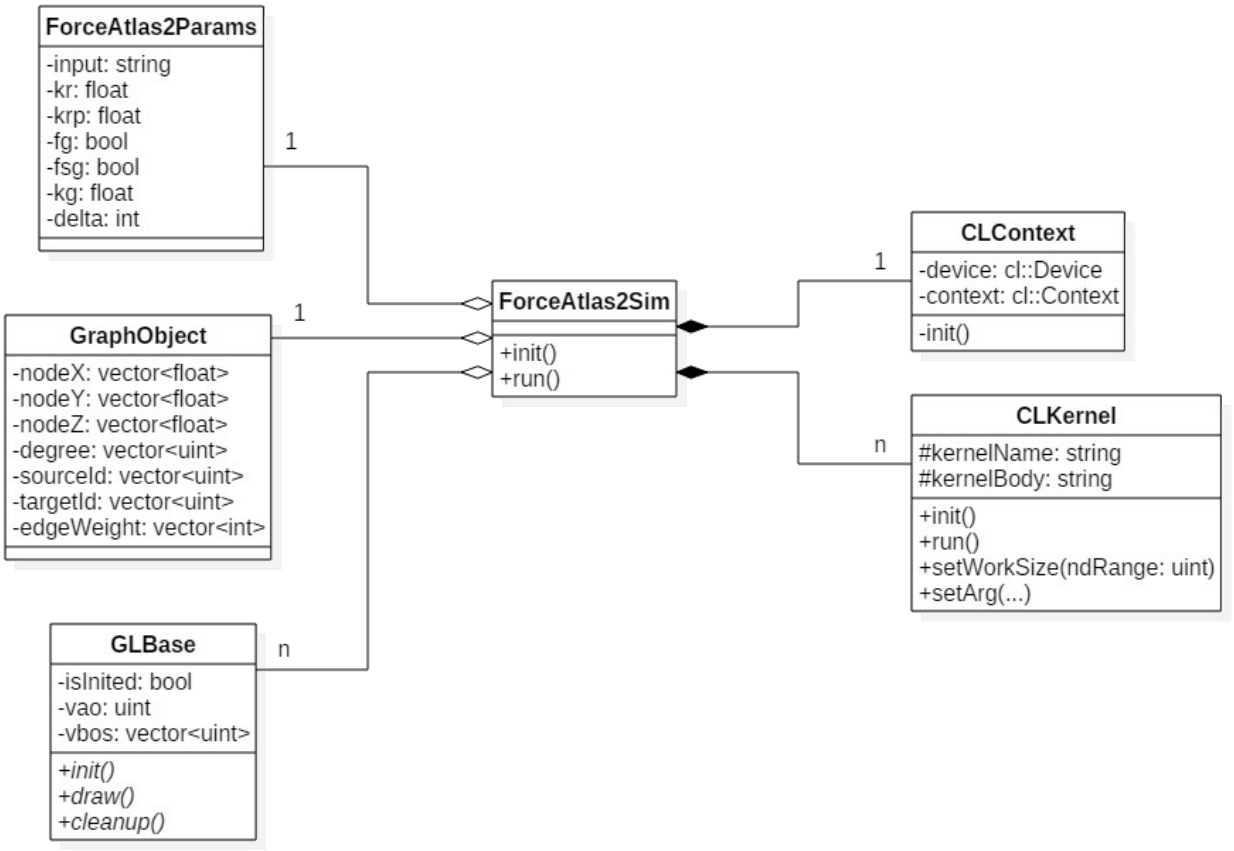
Kernel функције за ажурирање чворова графа и грана графа ажурирају позиције чворова и грана графа услед резултујућих сила. Свака нит *kernel* функције за ажурирање чворова графа се извршава над једним чвором графа, а свака нит *kernel* функције за ажурирање грана графа се извршава над једном граном графа. На крају ове две *kernel* функције се релевантне вредности ресетују.

Свака *vertex shader* функција се извршава над једном координатом 3D објекта. Укупан број извршавања *vertex shader* функције је једнак укупном броју координата чвора, тј. гране графа помножен са укупним бројем чворова, тј. грана графа. Свака *fragment shader* функција се извршава над једним пикселом екрана на који се пресликава 3D објекат и њен укупан број извршавања зависи од тренутне позиције објеката у 3D простору.

4.5. *ForceAtlas2* класе

Класа која је задужена за покретање целокупне симулације је *ForceAtlas2Sim*. Ова класа у себи садржи и иницијализује све OpenCL класе. Има методу *run* у оквиру које позива све *run* методе свих *CLKernel* објеката у зависности од аргумената апликације.

Садржи референце на класе: *ForceAtlas2Params*, *GraphObject* и *GLBase*. Класа *ForceAtlas2Params* садржи све аргументе апликације који су прослеђени кроз командну линију. Класа *GraphObject* и изведене класе основне класе *GLBase* се користе ради прослеђивања података *CLKernel* класама. Једноставан дијаграм ових класа се може видети на слици испод:



Слика 4.4. Дијаграм *ForceAtlas2* класа.

5. Коришћење алата

Апликација је развијана за оперативни систем *Windows* 10. За покретање је потребно имати графички процесор који подржава OpenCL и OpenGL интероперабилност. Захтевана OpenCL верзија је 1.2, а OpenGL верзија 3.3. Апликација се покреће из командне линије, а потом користи кроз графичко окружење.

5.1. Командна линија

Покретање и постављање аргумената апликације се врши кроз команду линију. Упутство за употребу се може приказати покретањем апликације без аргумената или задавањем аргумента *-h* иза имена апликације. Испис команде се може видети на слици 5.1.

Први наведен аргумент апликације је *-i*. Иза овог аргумента се у командној линији наводи путања до улазног GEXF или GML граф фајла. Аргумент *-i* је обавезан и без њега апликација не може да ради.

Следећа два аргумента апликације су *-kr* и *-krp*. Аргументи *-kr* и *-krp* представљају константе *kr* и *kr'* које се користе за израчунавање силе одбијања. Константа *kr* се користи за израчунавање силе одбијања када се чворови не преклапају, а константа *kr'* када се поклапају. Уколико се не задају као аргументи програма, њихове вредности су 0.01 и 100.

Следећа три аргумента се тичу силе гравитације. Задавање аргумента *-fg* доводи до коришћења гравитационе силе при израчунавању сила које делују на чвор графа. Задавање аргумента *-fsg* доводи до коришћења јаке гравитационе силе. Ова два параметра не могу да се задају у исто време. Ако се ни један не зада, у програму се неће користити сила гравитације. Аргумент *-kg* представља параметар k_g који утиче на јачину силе гравитације. Уколико се не зада као аргумент програма, његова вредности је 1.0.

```
Usage: forceatlas2sim.exe (-i INPUT) [-kr FLOAT] [-krp FLOAT] [-fg]
                          [-fsg] [-kg FLOAT] [-tau FLOAT] [-ks FLOAT]
                          [-ksmax FLOAT] [-delta INT] [-h]
GPU-based parallelization and graphical simulation of the
ForceAtlas2 algorithm.
Options:
-i INPUT Input GEFX or GML graph file. Required.
-kr FLOAT Repulsion force coefficient. Default: 0.01.
-krp FLOAT Repulsion force overlap coefficient. Default: 100.
-fg Uses gravitational force to attract nodes to graph center.
-fsg Uses strong gravitational force to attract nodes to graph center.
-kg FLOAT Gravitational force coefficient. Default: 1.0.
-tau FLOAT Tolerance to swinging coefficient. Default: 0.5.
-ks FLOAT Global speed coefficient. Default: 0.1.
-ksmax FLOAT Max global speed coefficient. Default: 10.
-delta INT Edge weight influence coefficient. Default: 1.
-h Prints usage.
```

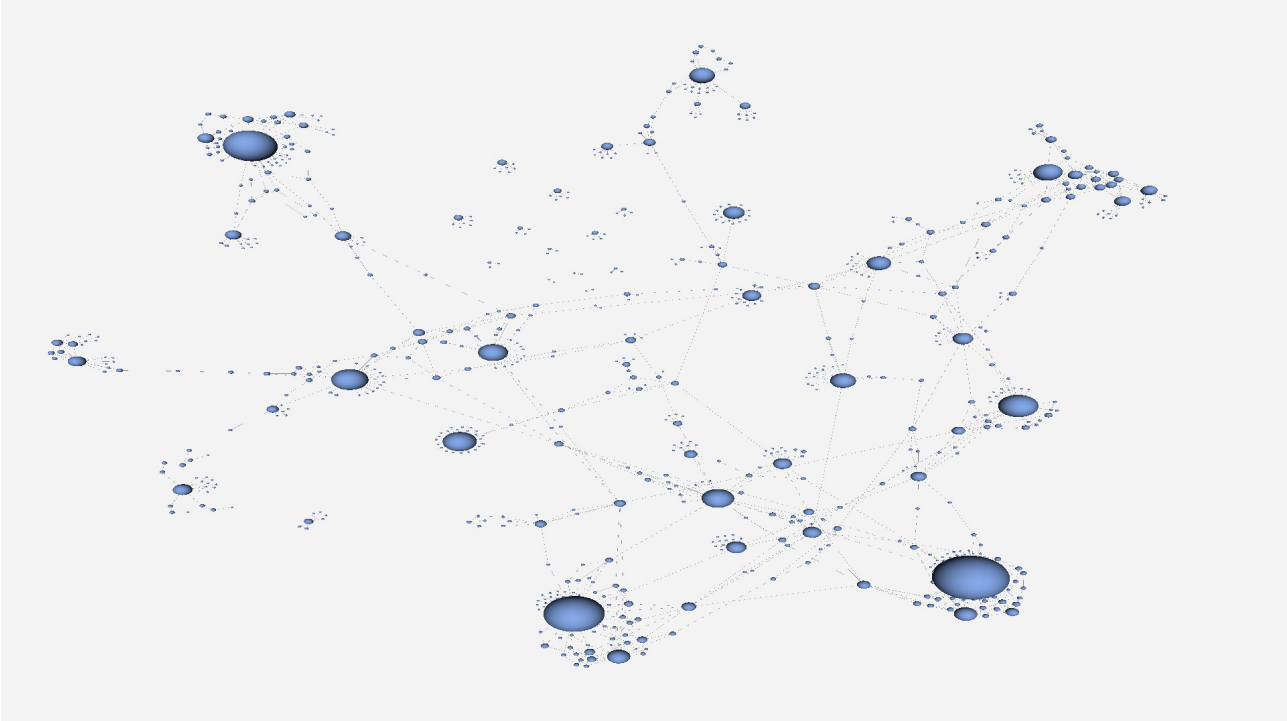
Слика 5.1. Упутство за употребу апликације.

Следећа три аргумента се тичу брзине и прецизности *ForceAtlas2* алгоритма. Аргумент *-tau* представља параметар τ који утиче на толеранцију глобалне осцилације графа. Уколико се не зада као аргумент програма, његова вредности је 0.5. Аргументи *-ks* и *-ksmax* представљају параметре k_s и k_{smax} који утичу на глобалну брзину графа. Уколико се не задају као аргументи програма, њихове вредности су 0.1 и 10.

Последњи аргумент програма је *-delta*. Он представља параметар δ који контролише утицај тежине грана графа на силу привлачења. Уколико се не зада као аргумент програма, његова вредности је 1.

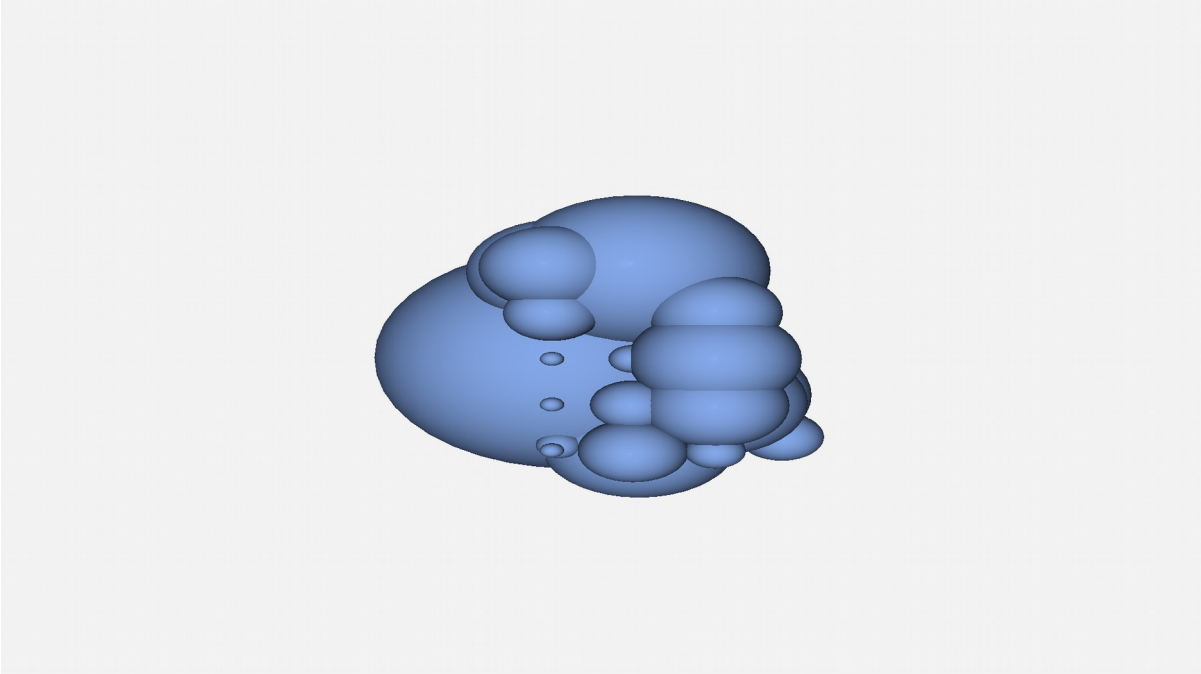
5.2. Графичка апликација

После задавања жељених аргумената у командној линији, покреће се графичка апликација. У оквиру терминала ће бити исписане информације о улазном граф фајлу и информације о изабраном графичком процесору. У случају да су у улазном граф фајлу специфициране координате чворова графа, те координате ће се користи за почетне позиције чворова графа при покретању апликације. На слици 5.2. се може видети приказ графичке апликације покренуте са *codeminer.gexf* улазним граф фајлом, у ком су специфициране координате чворова графа.



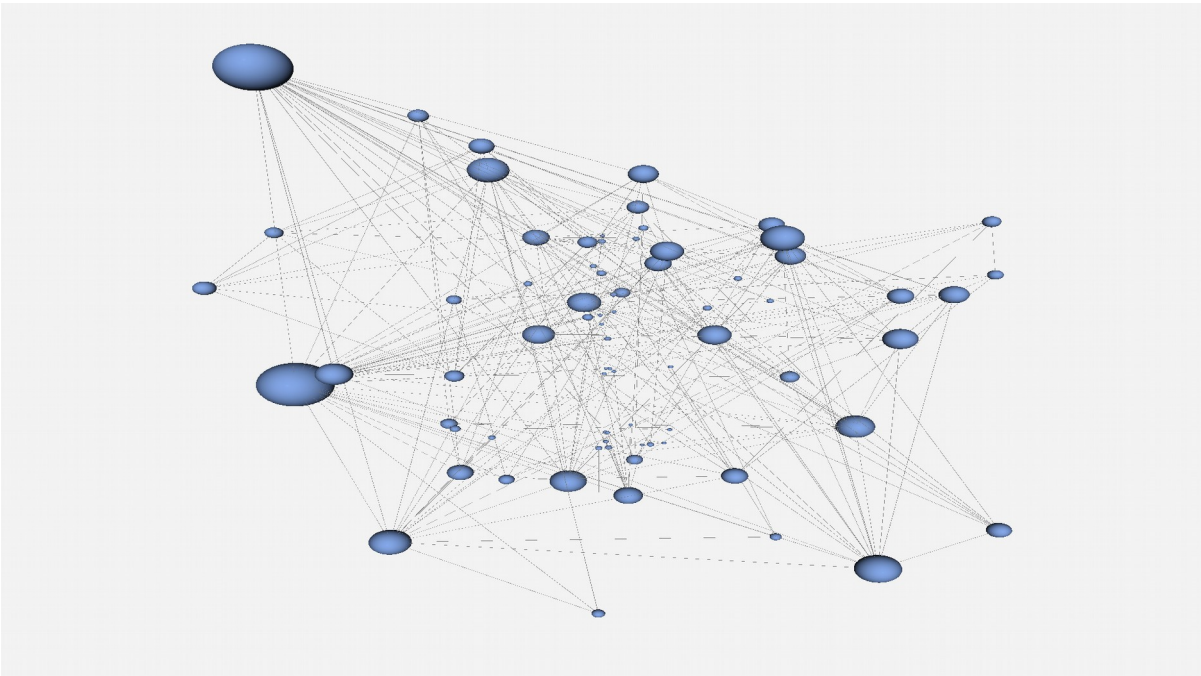
Слика 5.2. Изглед графичке апликације покренуте са *codeminer.gexf* улазним граф фајлом.

У случају да у улазном граф фајлу нису специфициране координате чворова графа, апликације ће поређати чворове у структуру коцке, редоследом којим су чворови читани из улазног граф фајла. Почетна позиција корисника у оквиру апликације ће бити таква да гледа на поређане чворове графа. На слици 5.3. се може видети приказ графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом, у ком нису специфициране координате чворова графа.



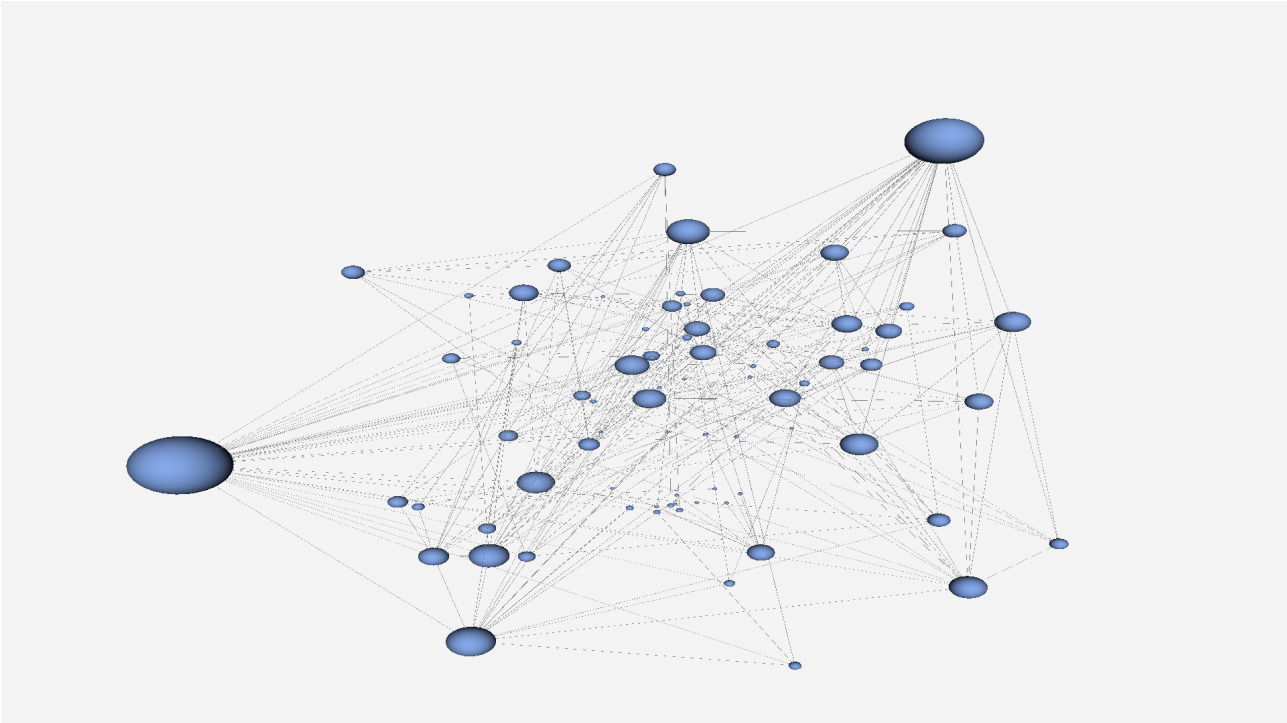
Слика 5.3. Изглед графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом.

ForceAtlas2 алгоритам се покреће притиском тастера *P* на тастатури. Алгоритам ће се извршавати уживо све док се не заустави поновним притиском тастера *P* на тастатури. Алгоритам се може покретати и заустављати произвољан број пута. На слици 5.4. се може видети приказ графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом после пуштања *ForceAtlas2* алгоритма.



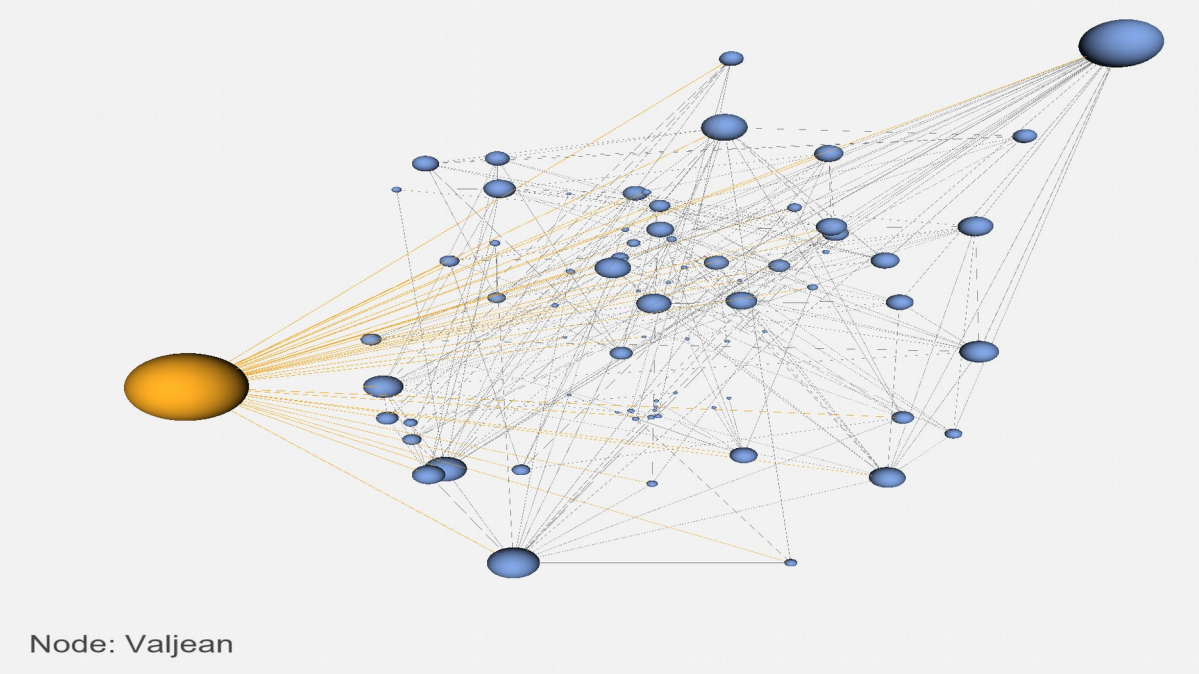
Слика 5.4. Изглед графичке апликације после пуштања *ForceAtlas2* алгоритма.

Могуће је кретати се кроз 3D простор апликације. Померање напред, лево, назад и десно је могуће притиском тастера *W*, *A*, *S* и *D* на тастатури, респективно. Ротирање камере је могуће урадити стиском левог тастера миша и повлачењем миша у жељеном смеру. На слици 5.5. се може видети приказ графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом у истом тренутку као и на слици 5.4. само из другачије позиције у 3D простору.



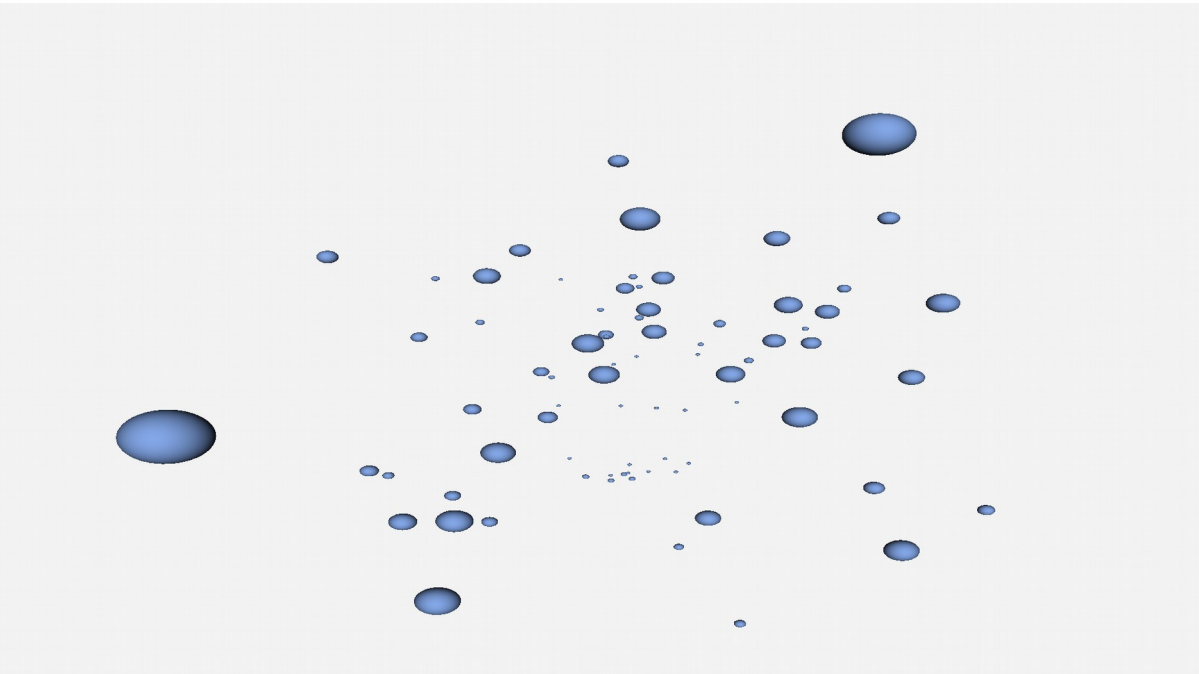
Слика 5.5. Изглед графичке апликације из другачије позиције у 3D простору.

Могуће је селектовати чвор графа. То се ради стиском левог *Ctrl* тастера на тастатури, померањем курсора преко жељеног чвора графа и стиском левог тастера миша. Селектовање чвора графа ће обојити дати чвор и његове гране другом бојом. Поред тога ће исписати лабелу додељену чвору у доњем левом углу апликације. Могуће је селектовати само један чвор графа у једном тренутку. Стискањем левог *Ctrl* тастера на тастатури и левог тастера миша док курсор није преко неког чвора ће обрисати селекцију. На слици 5.6. се може видети приказ графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом када је селектован чвор графа.



Слика 5.6. Изглед графичке апликације када је селектован чвор графа.

Могуће је искључити гране графа притиском тастера *E* на тастатури. Све акције могуће у апликацији са укљученим гранама су могуће и са искљученим гранама. Гране је могуће укључити поновним притиском тастера *E* на тастатури. На слици 5.7. се може видети приказ графичке апликације покренуте са *lesmiserables.gml* улазним граф фајлом када су гране графа искључене.



Слика 5.7. Изглед графичке апликације када су гране графа искључене.

6. Перформансе

Најчешћа мерна јединица за перформансе графичке апликације је FPS (*Frames Per Second*). Представља број пута који се исцрта графичка сцена у року од једне секунде, тј. број различитих слика приказаних у року од једне секунде. Вредности које се најчешће срећу у графичким апликацијама или видеима су 24, 30 и 60 FPS.

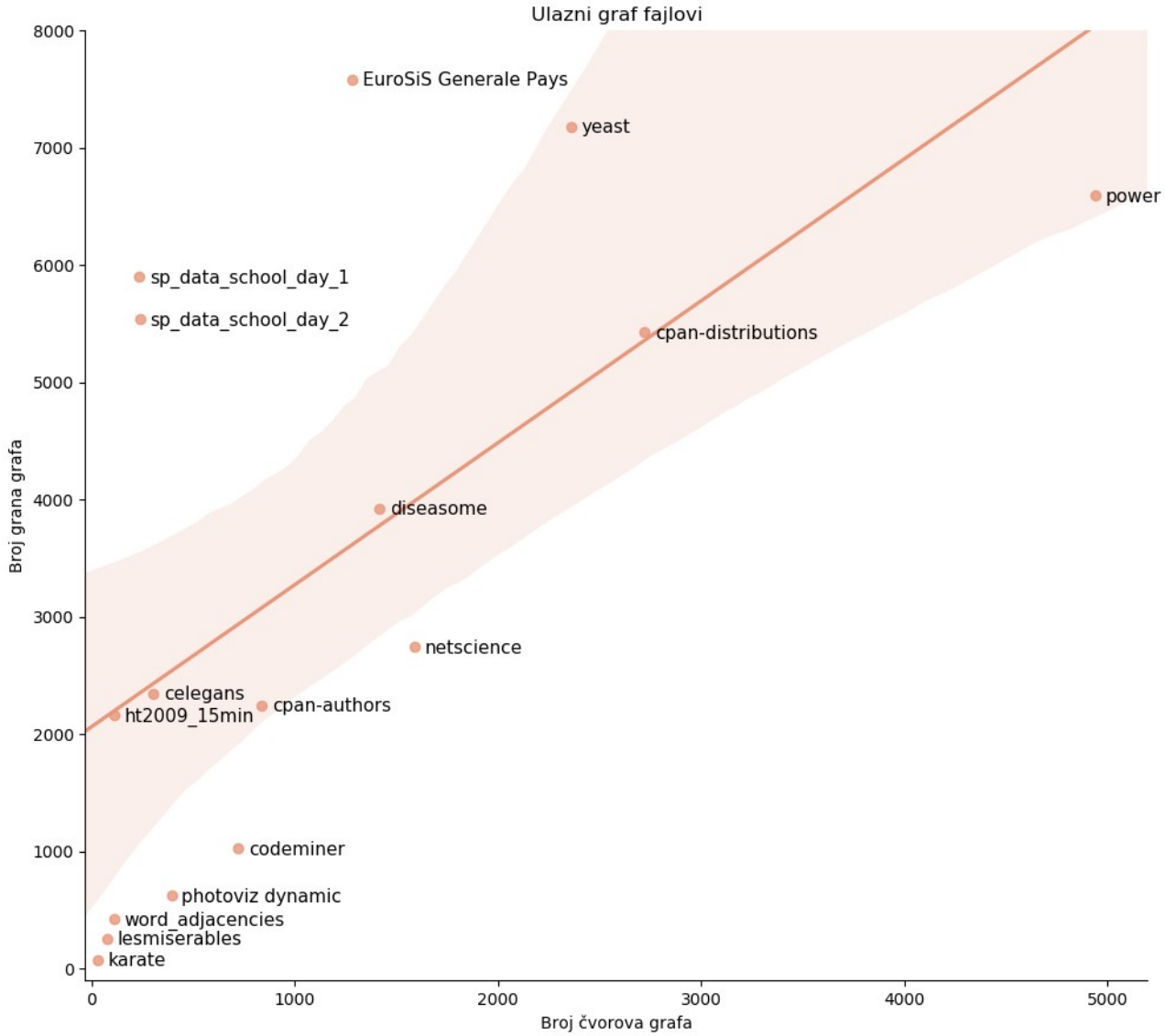
Међутим, пошто представља инверзну функцију времена, FPS није идеална мерна јединица за израчунавање перформанси [27]. Нпр. у случају да имамо графичку апликацију која се пре додавања неког позива за цртање извршавала брзином од 180 FPS, а после брзином од 160 FPS, разлика у времену извршавања једног исцртавања графичке сцене ће бити: 1/180 - 1/160 = 0.69 ms. С друге стране, ако имамо графичку апликацију која се пре додавања неког позива за цртање извршавала брзином од 60 FPS, а после брзином од 40 FPS, разлика у времену извршавања једног исцртавања графичке сцене ће бити: 1/60 - 1/40 = 8.33 ms. Као што се може видети, разлика у брзини од 20 FPS не доводи до исте разлике у времену извршавања.

Због тога ће се уместо користећи FPS, перформансе ове апликације мерити као број милисекунди потребан за исцртавање графичке сцене, тј. у нашем случају, број милисекунди потребан за извршавање једне итерације главне петље програма. Као референтне вредности су коришћене 16.67 ms и 33.33 ms које одговарају 60 FPS и 30 FPS, респективно. Парсирање улазног граф фајла и копирање података са централног на графички процесор није мерено.

Перформансе су мерене на резолуцији екрана 1920 x 1080, са искљученом опцијом Vsync (*Vertical Sync*). Vsync опција омогућава синхронизацију брзине освежавања монитора са брзином исцртавања графичке сцене изражене са FPS. Нпр. монитор са брзином освежавања од 60 Hz има могућност да прикаже максимално 60 различитих слика у секунди. У случају да је брзина исцртавања графичке сцене већа од 60 FPS, може доћи до такозване *screen tearing* појаве, где једну слику на екрану могу чинити резултати два или више исцртавања графичке сцене. Међутим, за сврхе мерења перформанси, тј. да би избегли ограничавање перформанси на 60 FPS, Vsync опција ће бити искључена.

За тестирање перформанси је узето у обзир седамнаест улазних граф фајлова. На слици 6.1. се може видети однос броја грана и броја чворова улазних граф фајлова. На хоризонталној оси се налази број грана, а на вертикалној оси број чворова графа. Граф фајлови су означени тачком и својим именом.

Да би резултати тестирања перформанси били конзистентни, од приказаних граф фајлова су за тестирање изабрани они чији однос броја чворова и броја грана упада у одређен опсег. Тај опсег је израчунат регресионом анализом, а на слици 6.1. је означен обојеним делом графика. На тај начин се од седамнаест граф фајлова изабрало седам који упадају у обојену област, и као изузетак осми фајл *netscience*, услед своје близине обојеном региону.



Слика 6.1. Однос броја грана и броја чворова улазних граф фајлова.

Даље, мерење перформанси је обављено на два графичка процесора: NVIDIA GeForce GTX 1050 Mobile и AMD RX VEGA 56. Пошто су ова два графичка процесора избачена у различитом периоду и не упадају у исте категорије перформанси, добијене резултате не треба посматрати као поређење између њих. Уместо тога, из добијених резултата треба да се уочи исти тренд понашања апликације са скалирањем улазних података, као и да буде демонстрација функционисања апликације на две различите платформе. У табели 6.1. се могу видети карактеристике ова два уређаја:

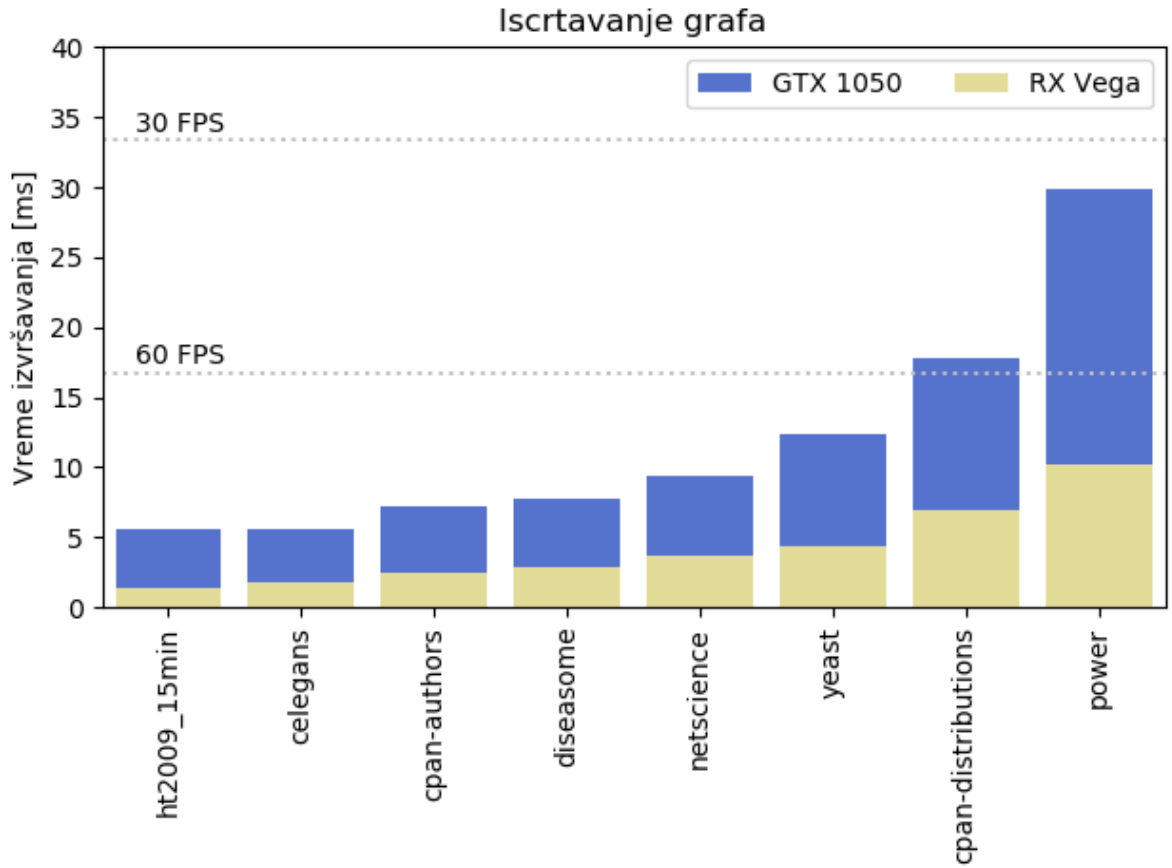
Карактеристике	GTX 1050 Mobile	RX VEGA 56
Радни такт	1.354 MHz	1.156 MHz
Радни такт меморије	1,752 MHz	800 MHz
Меморија	4.096 MB	8.192 MB
Меморијска магистрала	128 bit	2.048 bit
Проток меморије	112.1 GB/s	409.6 GB/s
Број језгара	640	3.584

Табела 6.1. Карактеристике уређаја коришћених за мерење перформанси.

Мерена су два аспекта апликације: брзина извршавања једне итерације главне петље програма само са исцртавањем графичке сцене, и са исцртавањем графичке сцене у комбинацији са извршавањем *ForceAtlas2* алгоритма. Пошто брзина исцртавања графичке сцене зависи од количине објеката који се налазе на екрану, резултати могу незнатно варирати од теста до теста. При мерењу резултата приказаних у наставку, потенцирало се на приказивању већине графа на екрану, колико год је то било у могућности. Сва мерења су рађења са укљученом *-fg* опцијом. Сви остали параметри имају подразумеване вредности, јер не утичу на брзину извршавања алгоритма.

На слици 6.2. се могу видети резултати мерења брзине извршавања једне итерације главне петље програма само са исцртавањем графичке сцене. На хоризонталној оси се налазе улазни граф фајлови сортирани према броју чворова графа. На вертикалној оси се налази време извршавања изражено у милисекундама. Зеленом бојом је приказано време извршавања на NVIDIA GeForce GTX 1050 графичком процесору, а црвеном време извршавања на AMD RX VEGA графичком процесору. Две хоризонталне линије означавају време извршавања које одговара лимиту од 60 FPS и 30 FPS.

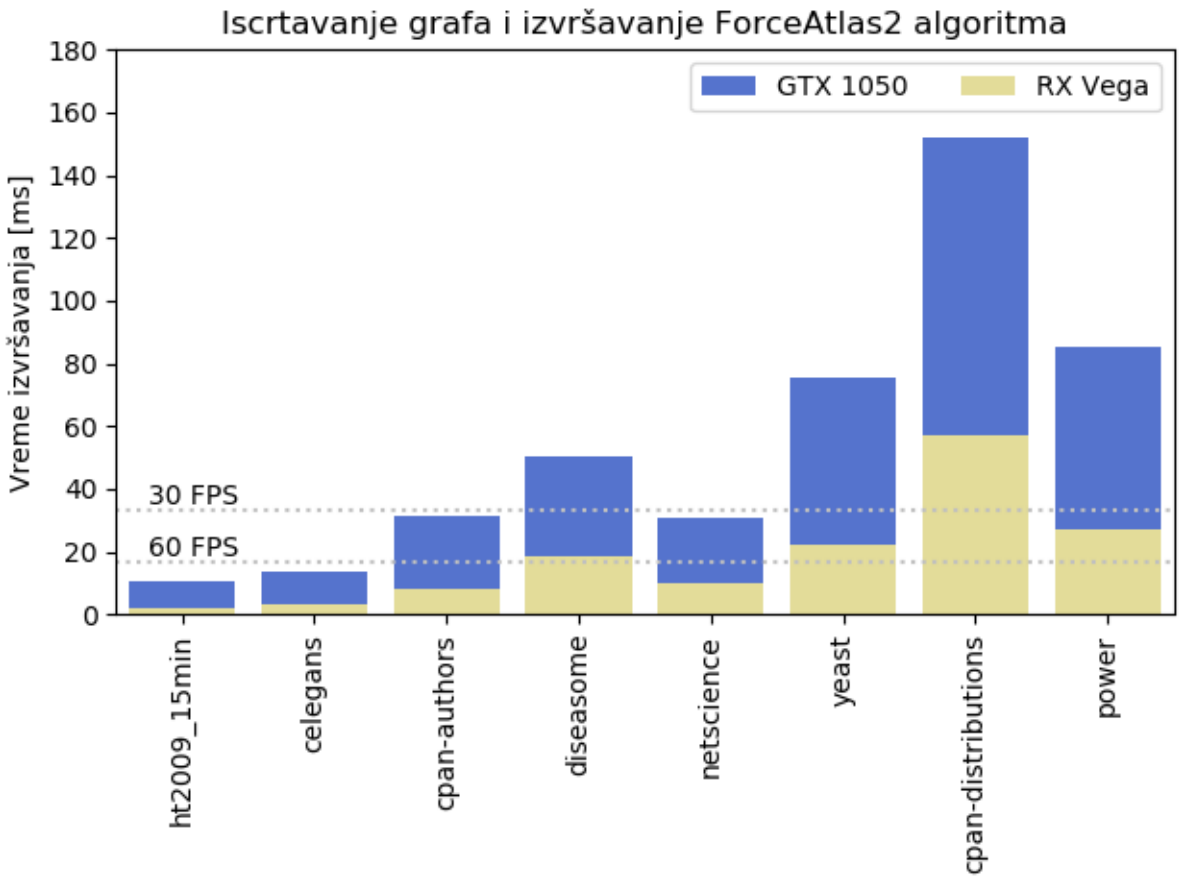
Као и очекивано, на слици 6.2. се може видети повећање времена извршавања заједно са величином графа. Време извршавања се на NVIDIA GeForce GTX 1050 графичком процесору креће испод лимита од 60 FPS за првих шест графова, а између лимита од 60 FPS и 30 FPS за последња два графа. Што се тиче AMD RX VEGA графичког процесора, време извршавања је увек испод лимита од 60 FPS. Ове перформансе омогућавају глатко коришћење графичке апликације за анализу графа.



Слика 6.2. Брзина извршавања исцртавања графичке сцене.

На слици 6.3. се могу видети резултати мерења брзине извршавања једне итерације главне петље програма са исцртавањем графичке сцене у комбинацији са извршавањем *ForceAtlas2* алгоритма. Ознаке и редослед графова су исти као на слици 6.2. У случају NVIDIA GeForce GTX 1050 графичког процесора, времена извршавања су испод лимита од 30 FPS за четири од осам графова, а преко те линије за остатак. Посебно су приметна изузетно велика времена извршавања за последња три графа. У случају AMD RX VEGA графичког процесора, времена извршавања су испод лимита од 30 FPS за све графове осим претпоследњег, где време извршавања знатно прелази лимит.

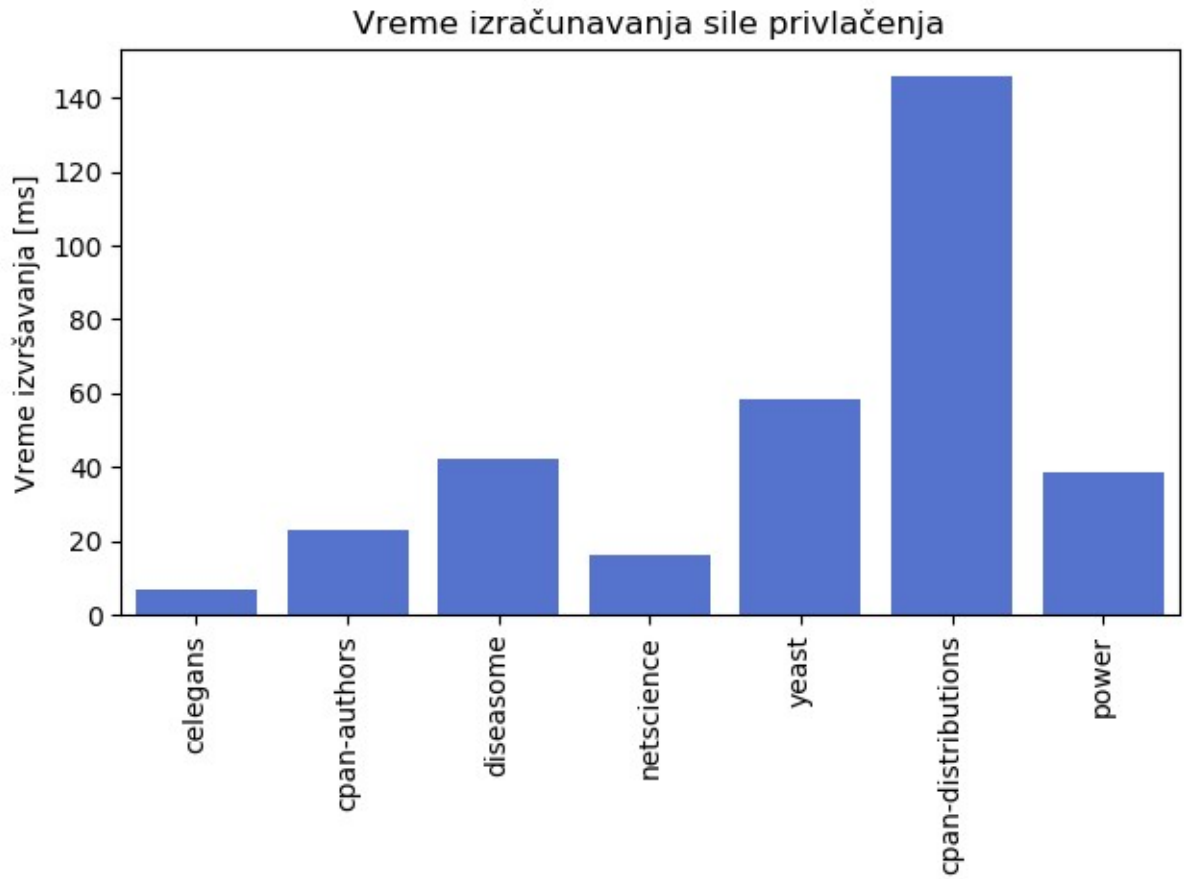
Оно што се прво примећује јесте да брзина извршавања више не важи од величине графа. Прво овакво одступање се може видети у времену извршавања три графа: *diseasome*, *netscience* и *yeast*. Времена извршавања између графова *diseasome* и *yeast* се воде трендом линеарног раста, док граф *netscience* одступа од тога. Објашњење за ово се може наћи на слици 6.1. Сва три графа су у сличном опсегу броја чворова, међутим граф *netscience* има мањи број грана.



Слика 6.3. Брзина извршавања исцртавања графичке сцене и извршавања *ForceAtlas2* алгоритма.

Друго одступање се може видети у времену извршавања *cpan-distributions* графа, које је знатно веће од осталих. На први поглед, граф *cpan-distributions* се налази тачно на линији регресионе анализе приказане на слици 6.1 и не би требало да одступа од осталих графова. Анализа времена извршавања појединих *kernel* функција указује на функцију која врши израчунавање силе привлачења. Резултат ове анализе на NVIDIA GeForce GTX 1050 графичком процесору се може видети на слици 6.4.

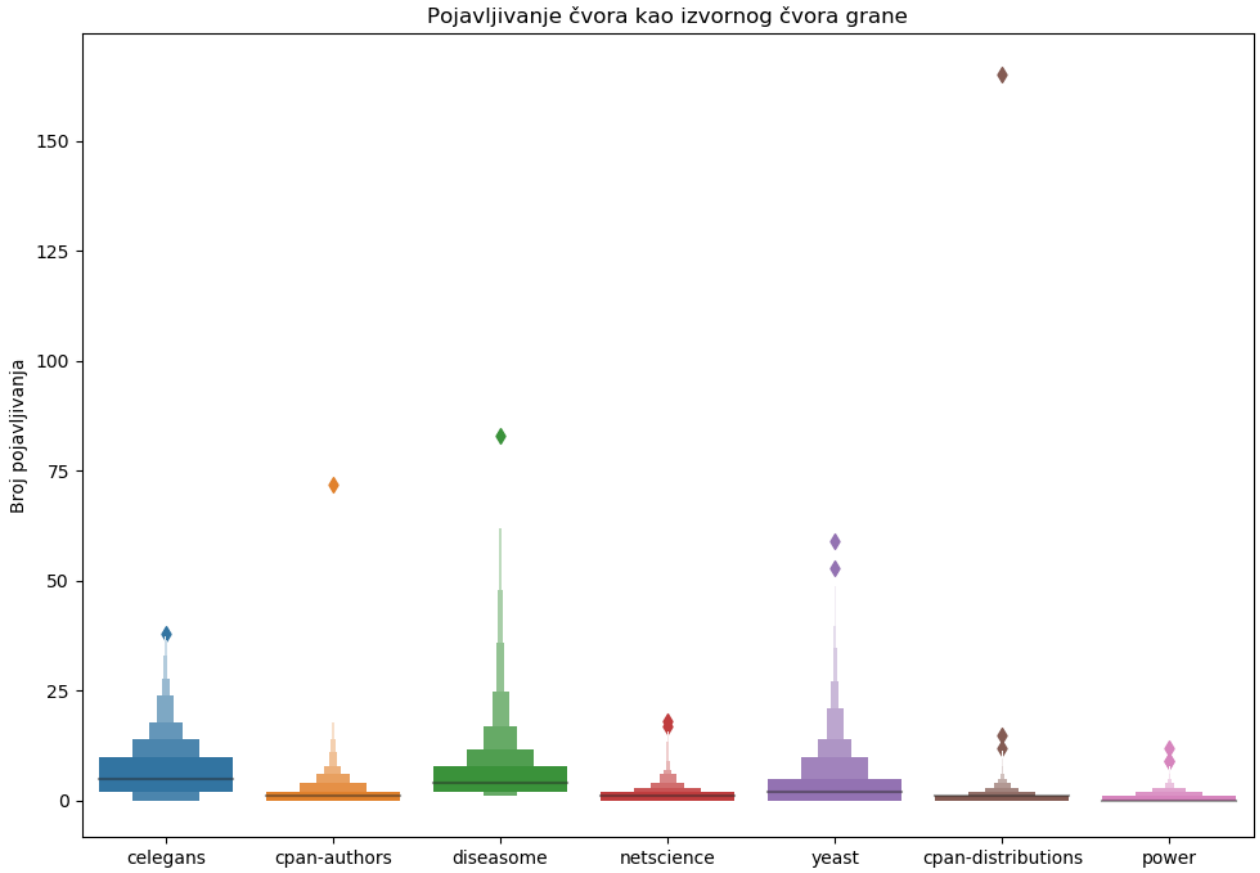
Главно уско грло *ForceAtlas2* алгоритма на графичком процесору, представља претрага постојања гране између два чвора графа током израчунавања силе привлачења. Овај корак је неопходан, али доводи до дивергирања у току контроле између различитих нити на графичком процесору. Због тога, промена броја грана знатно више утиче на време извршавања *ForceAtlas2* алгоритма него промена броја чворова графа.



Слика 6.4. Анализа времена израчунавања силе привлачења.

Са слике се може видети да времена извршавања *kernel* функције која врши израчунавање силе привлачења прате исти тренд као укупна времена извршавања апликације на слици 6.3. При израчунавању силе привлачења између два чвора потребно је пронаћи ако постоји грана између њих. Ова претрага уједно представља једини део извршавања *kernel* функције у ком нити дивергирају у току контроле. Што се неки чвор чешће појављује као изворни чвор грана, то је претрага за одредишним чвором те исте гране дужа.

Анализом улазних граф фајлова можемо видети број појављивања чворова као изворног чвора неке гране графа. Резултати те анализе се могу видети на слици 6.5. Граф *cpan-distributions* има највећу екстремну вредност, док граф *power* има вредности у јаком малом опсегу. Ово објашњава зашто је време извршавања првог веће од времена извршавања другог, иако граф *power* има више чворова и грана. Граф *yeast* такође има пар екстремних вредности, што објашњава зашто је његово време извршавања толико близу времена извршавања *power* графа.



Слика 6.5. Анализа појављивања чворова као изворног чвора гране.

Из приказаних резултата се може видети да је време исцртавања графа конзистентно, тј. да расте са повећањем улазног граф фајла, тј. опада са повећањем ресурса. Што се тиче времена извршавања *ForceAtlas2* алгоритма, оно највише зависи од самог граф фајла. Непредвидиво је и није га лако предвидети ослањајући се само на број чворова и грана улазног граф фајла. Израчунавање силе привлачења представља највеће уско грло апликације, тј. део чије перформансе треба највише усавршити.

7. Закључак

Као што је изложено на почетку рада, визуелизација мреже представља изазован проблем. Један од главних разлога за то се тиче самог приступа визуелизацији мреже. Постоји велик број различитих метода за визуелизацију, свака са својим предностима и манама. Одабир методе диктира тип информација које можемо сазнати из мреже. У комбинацији, ове методе могу покрити различите аспекте мреже, али се све и даље боре са проблемом визуелизације великих мрежа.

Читљивост података је једна од највећих препрека при визуелизацији великих мрежа. Са скалирањем података, читљивост драстично опада. То смањује корисност и лакоћу интерпретације визуелизоване мреже. Што се тиче алгоритама који користе усмерене силе, овај проблем се делимично решава груписањем сродних чворова једних поред других. Ово међутим не решава тешкоћу приказа целе топологије мреже или непрегледност коју може изазвати велик број грана.

Други велики изазов при визуелизаији мреже се тиче перформанси. Визуелизаија мреже је рачунски захтеван проблем. Лоше перформансе могу подједнако да утичу на квалитет визуелизације мреже као и непрегледност. Видели смо да се у алгоритмима који се извршавају на централним процесорима ове лимитације превазилазе вршењем разних апроксимација при израчунавању.

У овом раду се проблему визуелизације мреже приступило на начин који се не среће често у постојећим решењима. Мрежа је визуелизована у 3D простору, а израчунавање обављено на графичком процесору. Циљ рада је био да се додавањем нове димензије допринесе информацијама које можемо сазнати из мреже, а да се коришћењем графичког процесора понуди алтернатива за захтевна израчунавања.

Видели смо да се на графичким процесорима могу постићи перформансе конкурентне онима на централном процесору. Такође, графички процесори омогућавају визуелизовање мреже без вршења апроксимација, што повећава прецизност крајњег резултата. Међутим, графички процесори нису флексибилни као и централни. Алгоритам који је погодан за

извршавање на једном, често није погодан за извршавање на другом. У овом раду је то примећено у делу алгоритма који се бави претрагом грана. Побољшана имплементација би користила и централни и графички процесор за расподелу рада, тј. балансирала предности и мане оба уређаја.

Још једна мана имплементације која се тиче перформанси, јесте захтевност израчунавања у 3D простору. Увођење још једне осе чини и израчунавање и исцртавање графичке сцене знатно захтевнијим. Што се тиче исцртавања, перформансе би се могле побољшати када би чворови графа били приказани на неки други начин, нпр. помоћу коцки које се у односу на лопте састоје од знатно мањег броја темена за цртање.

Мана од које ова имплементација такође пати се тиче прегледности визуелизације. Као што је проблем и са визуелизацијом у 2D простору, приказ већих мрежа брзо постаје непрегледан. Алтернативна презентација грана графа би могла да помогне са решавањем овог проблема. Такође би од значаја био и већи број опција за кориговање у оквиру самог графичког окружења, које би корисницима дале већу контролу над визуелизацијом мреже.

Поред наведених, још један начин за побољшање би био истраживање алтернативних технологија. Од посебног интереса је *Vulkan*, стандард *Khronos Group* конзорцијума за компјутерску графику и општа израчунавања на графичким процесорима. Овај стандард обећава знатно већу контролу над подацима и током извршавања функција на графичким процесорима. Поред тога, с обзиром да се бави и компјутерском графиком и општим израчунавањем, нема толико грубу потребу за синхронизацијом између операција као што је случај са OpenCL и OpenGL технологијама.

Упркос свим препрекама, изучавање визуелизације мреже у 3D простору представља одличан увод за даље размишљање о овом проблему. Додавање нове димензије такође пружа додатни увид при анализи мрежа. Поред тога, OpenCL и OpenGL технологије представљају одличан почетак за разумевање и коришћење графичких процесора. Помоћу њих се могу сазнати могућности и лимитације хардвера, као и тренутно стање програмирања на графичким процесорима у индустрији.

Литература

[1] *Tulip* [Online]. Available: http://tulip.labri.fr/TulipDrupal/ (17.9.2018.)

[2] *Graphviz* [Online]. Available: https://www.graphviz.org/ (17.9.2018.)

[3] *Pajek* [Online]. Available: http://mrvar.fdv.uni-lj.si/pajek/ (17.9.2018.)

[4] *Cytoscape* [Online]. Available: http://www.cytoscape.org/ (17.9.2018.)

[5] *Gephi* [Online]. Available: https://gephi.org/about/ (17.9.2018.)

[6] M. Jacomy, T. Venturini, S. Heymann, M. Bastian, "*ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software",* PLoS ONE 9(6): e98679, јун 2014. (https://doi.org/10.1371/journal.pone.0098679, 17.9.2018.)

[7] M. J. McGuffin. "*Simple Algorithms for Network Visualization: A Tutorial*", Tsinghua Sciene and Technology 17(4), август 2012. (https://pdfs.semanticscholar.org/9f0f/5a1507b83f96bcedbf2b8971fde21948b086.pdf, 17.9.2018.)

[8] *6 Ways to Visualize Graphs* [Online]. Available: https://www.twosixlabs.com/6-ways-visualize-graphs/ (17.9.2018.)

[9] *N-body simulations (gravitational)* [Online]. Available: http://www.scholarpedia.org/article/N-body_simulations_(gravitational) (17.9.2018 .)

[10] T. Ventimiglia, K. Wayne, *The Barnes-Hut Algorithm* [Online]. Available: http://arborjs.org/docs/barnes-hut (17.9.2018 .)

[11] Т. Алексић, "*Паралелизација проблема гравитације n тела на графичком процесору*", септембар 2016.

[12] *About The Khronos Group* [Online]. Available: https://www.khronos.org/about/ (17.9.2018.)

[13] *The OpenCL Specification* [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf (17.9.2018.)

[14] *The OpenGL Graphics System: A Specification* [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf (17.9.2018.)

[15] *CUDA* [Online]. Available: https://developer.nvidia.com/cuda-zone (17.9.2018.)

[16] *Vulkan* [Online]. Available: https://www.khronos.org/vulkan/ (17.9.2018.)

[17] *What is OpenGL?* [Online]. Available: http://openglbook.com/chapter-0-preface-what-is-opengl.html (17.9.2018.)

[18] *Direct3D* [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/direct3d (17.9.2018.)

[19] *Mantle* [Online]. Available: https://www.amd.com/Documents/Mantle_White_Paper.pdf (17.9.2018.)

[20] *Modern OpenGL* [Online]. Available: https://glumpy.github.io/modern-gl.html (17.9.2018.)

[21] *The OpenGL Shading Language* [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf (17.9.2018.)

[22] *Hello Triangle* [Online]. Available: https://learnopengl.com/Getting-started/Hello-Triangle (17.9.2018.)

[23] *The OpenCL Extension Specification* [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2-extensions.pdf (17.9.2018.)

[24] *forceatlas2sim* [Online]. Available: https://github.com/TeodoraAleksic/forceatlas2sim (17.9.2018.)

[25] *GEXF File Format* [Online]. Available: https://gephi.org/gexf/format/ (17.9.2018.)

[26] *GML Format* [Online]. Available: https://gephi.org/users/supported-graph-formats/gml-format/ (17.9.2018.)

[27] *Performance* Available: https://www.khronos.org/opengl/wiki/Performance (17.9.2018.)

Списак скраћеница

SIMD - *Single Instruction Multiple Data*

OpenCL - *Open Computing Language*

OpenGL - *Open Graphics Library*

AMD - *Advanced Micro Devices*

API - *Application Programming Interface*

IRIS GL - *Integrated Raster Imaging System Graphical Library*

GLSL - *OpenGL Shading Language*

GEXF - *Graph Exchange XML Format*

GML - *Graph Modeling Language*

XML - *Extensible Markup Language*

FPS - *Frames Per Second*

Vsync - *Vertical Sync*

Списак слика

Списак табела

А. Граф фајл формати

Постоји велик број формата за граф фајлове. Ради једноставности, апликација подржава два најчешћа: GEXF и GML. У наставку су ови формати описани.

А.1. GEXF

GEXF [25] је фајл формат направљен за *Gephi* софтвер. Базиран на је XML (*Extensible Markup Language*) фајл формату. Има велик број опција за дефинисање параметара графа. Поред чворова и грана, има могућност за дефинисање хијерархија чворова. Пружа доста опција за уношење метаподатака, попут лабела и графичких карактеристика графа. Једноставан пример GEXF фајл формата се може видети на слици испод:

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
  <graph mode="static" defaultedgetype="directed">
    <nodes>
      <node id="0" label="Hello" />
      <node id="1" label="Word" />
    </nodes>
    <edges>
      <edge id="0" source="0" target="1" />
    </edges>
  </graph>
</gexf>
```

Слика А.1. Једноставан пример GEXF фајл формата [25].

A.2. GML

GML [26] је једноставан текстуални фајл формат. Поред дефинисања чворова и грана графа, допушта уношење метаподатака. Мање је флексибилан у односу на GEXF, али зато једноставнији и читљивији. Једноставан пример GML фајл формата се може видети на слици испод:

```
graph
[
  node
  [
   id A
   label "Node A"
  ]
  node
  [
   id B
   label "Node B"
  ]
   edge
  [
   source B
   target A
   label "Edge B to A"
  ]
]
```

Слика А.2. Једноставан пример GML фајл формата [26].