LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

# OPENGL KODOLA KOLAPSA VIZUALIZĀCIJA

BAKALAURA DARBS

Autors: **Kristaps Veitners**
Studenta apliecības Nr.: kv17044
Vadītājs: Dr.Dat. Jānis Zuters

RĪGA 2021

# ANOTĀCIJA

Bakalaura darbs tiek fokusēts uz zinātnisku simulāciju atveidošanu OpenGL grafiskajā vidē. Darbā tiek apskatītas konstrukcijas, metodes un optimizācijas pasākumi, lai vizualizētu pirmszvaigzņu mākoņa saraušanos. Tiek arī dots ieskats montekarlo metodē, lineārā interpolācijā un kubisku splainu interpolācijā.

Darba gaitā ir izveidotas divas programmas, simulācijas vizualizācija un vērtību ģenerēšana izmantojot interpolācijas un integrācijas bibliotēku. Programmām ir apskatīta arī veiktspēja un izpildes laiks.

Atslēgas vārdi: OpenGL, montekarlo, simulācija, trīsstūru pavairošana, astrofizika.

# ABSTRACT

OPENGL CORE COLLAPSE VISUALIZATION

This bachelor work is focused on scientific simulation visualization in OpenGL graphical environment.

In the work, constructions, methods and optimisation are discussed, in order to create prestellar cloud collapse visualization. Monte-carlo method is also considered, and linear interpolation and cubic spline interpolation are examined.

During work, two programms were developed. Simulation visualization and value generation using interpolation and integration library. Performance and execution time of programs is also inspected.

Keywords: OpenGL, monte-carlo, simulation, triangle generation, astrophysics.

# SATURA RĀDĪTĀJS

# APZĪMĒJUMU SARAKSTS

Ēnotājs – OpenGL programma, kas izmanto pati savu programmēšanas valodu un atbild par virsotņu un fragmentu zīmēšanu uz ekrāna.

Poligons – Mazākais primitīvs, ko var uzzīmēt OpenGL logā. Sastāv no trīs punktiem un veido trīsstūri.

OpenGL – Atvērtā grafiskā bibliotēka, kas dod tiešu pieeju videokartes resursiem.

Attēli sekundē – Programmatūras ekrāna attēlu pārzīmēšanas skaits sekundē.

Renderis – Bildes sintēze, jeb attēla ģenerēšana izmantojot 2D vai 3D modeļus un aprēķinot krāsas vērtības katram redzamam objektam.

Uniforma – Konstrukcija, ar kuras palīdzību no programmas var pārsūtīt papildus informāciju uz ēnotāju programmu.

Glsl – Grafiskās bibliotēkas ēnotāju valoda, jeb programmēšanas valoda, ar kuru tiek programmēti ēnotāji. Tā sintaktiski ir ļoti līdzīga C++ valodai.

Anti-aliasings – Process, kurā līniju asās malas tiek izsmērētas, lai līnija izskatītos gludāka un ar mazāk robiem.

Alglib – C++ bibliotēka, kuras sastāvā ir skaitļu analīzes un datu apstrādes funkcijas.

API – Lietojumprogrammas saskarne, kas ir vairāku konstrukciju kopums, ko var izmantot ārējās programmas.

NASA – Nacionālā aeronautikas un kosmosa aģentūra.

ESA – Eiropas kosmosa aģentūra.

# 1. IEVADS

Bakalaura darba ievadā tiek paskaidrots darba konteksts jeb autora motivācija darba veikšanai, sasaiste ar iepriekšējiem darbiem, kā arī sasniedzamie mērķi.

## 1.2. Konteksts

Praksē, datu vizualizācija ir informācijas pārveidošana vizuālā kontekstā, kā kartē, grafā, 3D attēlā vai animācijā. Tas tiek darīts, lai veicinātu datu izpratni un atvieglotu secinājumu veikšanu. Galvenais datu vizualizācijas uzdevums ir iespējot, vai padarīt vieglāku, sakarību, tendenču vai ekstrēmu gadījumu ievērošanu lielās, sarežģītās datu kopās. Datu vizualizāciju ikdienā pielieto visdažādākie profesiju pārstāvji, tajā skaitā arī pasniedzēji, lai labāk komunicētu informāciju studentiem. [1]

Datu vizualizācijas priekšrocības ir

- Spēja ātrāk absorbēt informāciju;
- Uzlabot izpratni;
- Uzlabot spēju noturēt publikas uzmanību;
- Padarīt datus par vieglāk pieejamiem un vieglāk saprotamiem;
- Uzlabot spēju reagēt uz atradumiem ātri, tādā veidā sasniedzot panākumus ātrāk un ar mazāk kļūdām.

Lai vizualizētu dabas notikumus vai fenomenus ar sarežģītām, daudzpakāpju cēloņsakarībām, dati parasti tiek iegūti ar simulāciju palīdzību. Simulācijas izmanto, lai atveidotu reālās pasaules apstākļus, un balstoties uz kādiem noteikumiem radītu datus, kuros varētu novērot kādas iezīmes, īpašības vai uzvedību, ko nevar pa tiešo izmērīt. Šajos datos pēc tam tiek meklēta korelācija, vai sakarības, kas paskaidrotu datu uzvedību, izmantojot datu vizualizācijas metodes.

Tomēr, tiek novērots informācijas vakuums starp simulāciju, vizualizēšanu un secinājumu veikšanu. Respektīvi, starp simulāciju un vizualizēšanu tiek palaista garām informācija, kas varētu

dot papildus ieskatu notikuma vai fenomena cēloņos un uzvedībā. Kā arī starp vizualizāciju un secinājumu veikšanu, vai informācijas komunikāciju publikai, kas rodas dēļ dažādām priekšzināšanām, vai tīri no subjektīvas interpretācijas.

Risinājums ir jau šobrīd novērots trends – datu vizualizācijas attīstīšana, izmantojot jaunus risinājumus. Viens šāds risinājums ir interaktīvas vizualizācijas, kas atļauj lietotājam tieši manipulēt ar informācijas grafisko reprezentāciju. Otrs risinājums ir pašas simulācijas zinātniski korekta un akurāta vizualizācija. Mūsdienu datu vizualizācijā abi risinājumi ir relatīvi neizpētīta joma, un populārzinātniskas simulācijas nereti ir mākslinieciski, nevis zinātniski projekti. [2][3]

Turpmākajās nodaļās tiek apskatītas divas simulācijas OpenGL grafiskajā vidē, kas līdzinās reāla, fiziska modeļa simulācijas vizualizēšanai. Pamatideja ir, ka datoriem, īpaši video-kartēm, kļūstot spējīgākām, ir iespējams izveidot ambiciozas vizualizācijas, izmantojot grafiskās bibliotēkas un konstrukcijas, kas oriģināli ir paredzētas apjomīgu un vizuāli iespaidīgu video spēļu realizēšanai.

### 1.2. Sasaiste ar citiem darbiem

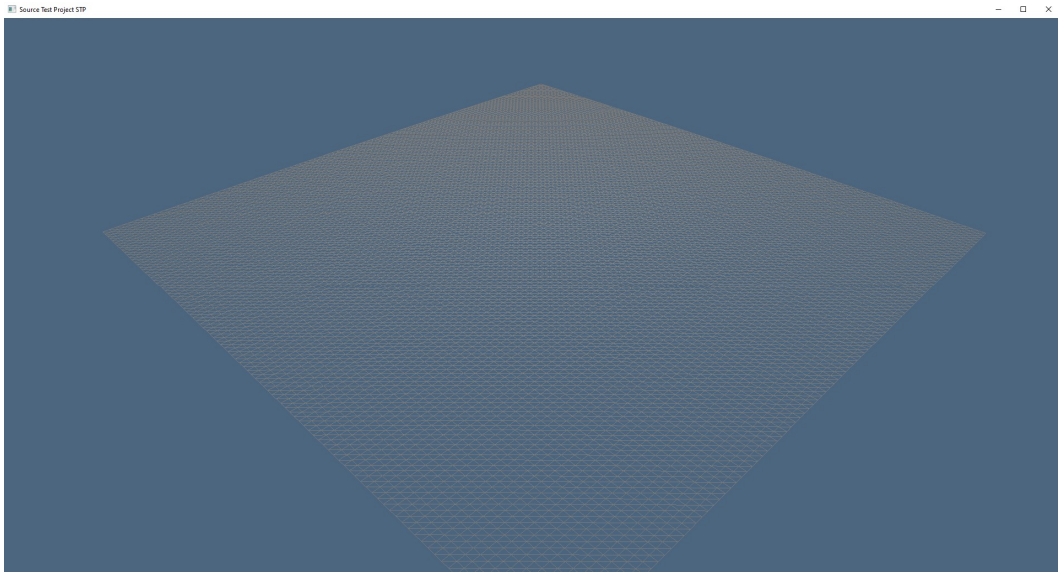
Bakalaura darbam par pamatu tiek ņemti divi iepriekš izveidoti darbi – “OpenGL datu vizualizācijas sistēma” (kvalifikācijas darbs) un “Kodola kolapsa vizualizācija” (kursa darbs).

No kvalifikācijas darba tiek izmantots izveidotais OpenGL grafiskais dzinis, kurā ir realizēta 2D datu vizualizēšana 3D vidē, izmantojot laukuma ģenerēšanas, distances no centra un lineārās interpolācijas algoritmus. Dzinis ietver:

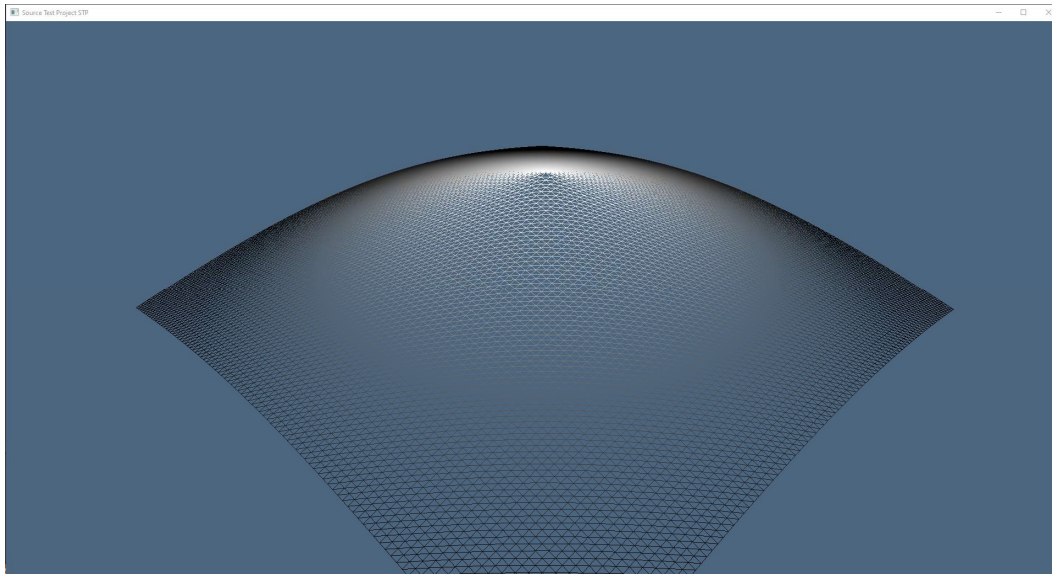
- Grafiskā loga izveidošanu;
- Dinamisku grafiskā loga izmēru mainīšanu;
- Funkcijas ārēju datu ielasīšanai programmā;
- Virsotņu un elementu bufera objektu izveidošanu;
- Ēnotāju programmas izveidošanu un kompilēšanu;
- Renderēšanas ciklu;
- Lietotāja klaviatūras un peles ievada apstrādāšanu;
- Modeļa, skata un projekcijas matricas;

- Datu plūsmas sasaisti starp datiem, renderēšanas ciklu un ēnotāju programmām.

Dzinī ir arī realizēta brīva kameras kustība, kas atļauj apskatīt modeli no visiem leņķiem. Kameras kustība simulācijas kontekstā kalpo par interaktīvās vizualizācijas rīku. Dzinis sastāv no divām klasēm – shader(ēnotāju) klases un kameras klases, kuras nav nepieciešams modificēt šim projektam. 1.2.1. Attēlā un 1.2.2. attēlā ir redzams nedaudz atjaunota kvalifikācijas darba programmatūras darbība, kurā ir novērojams kustīgs, 3D funkcijas renderis.




1.2.1 attēls – Ģenerēts funkcijas laukums kustības sākumā.




1.2.2 attēls – Ģenerēts funkcijas laukums kustības beigās.

Kursa darbā tika izpētīta simulācijas izveidošana OpenGL grafiskajā dzinī, kas atspoguļotu fizisku modeli. Simulācijas princips ir balstīts uz starpzvaigžņu mākoņa saraušanos gravitācijas spēka ietekmē, kā rezultātā veidojas zvaigzne.

Kursa darba ietvaros tika apskatīti veidi, kā izpildīt:

- Sfēras renderēšana izveidotā OpenGL grafiskajā sistēmā;
- Sfēras izmēru mainīšana izmantojot aprēķinātus datus un animācijas;
- Sfēras fragmentu krāsas un caurspīdīguma mainīšana, atkarībā no tekošās rādiusa vērtības;
- Vairāku, savstarpēji ietvertu sfēru instanču renderēšana;
- Vērtību pārnešana starp sfērām, jeb sava veida sfēru sadursmes;
- Sfēru rotācija un rotācijas pārnese sarusmēs;
- Kā arī kopējas sistēmas sfēru atkarība, jeb kā tiktu realizēta gravitācija.

Pirmajā simulācijā tiek realizēts un testēts aprakstītais sfēras šūnu simulācijas modelis OpenGL grafiskajā vidē, ar brīvu kameras kustību un lietotāja ievadu.

### 1.3. Sasniedzamie mērķi

Šī darba ietvaros autors ir paredzējis izstrādāt un aprakstīt izmantotās konstrukcijas "Kodola kolapsa vizualizācijai", kā arī veikt izveidotās simulācijas testēšanu un datu kalibrēšanu. Simulācija ir paredzēta kā demonstrācija tam, kā varētu izveidot izveidot uz sakarībām un likumiem, nevis tikai datiem balstītu simulācijas vizualizāciju.

# 2. KODOLA KOLAPSA VIZUALIZĀCIJA

Šajā nodaļā tiek virspusēji aprakstīta OpenGL grafiskā vide, pirmās vizualizācijas konstrukciju realizēšana, kalibrēšana un testēšana.

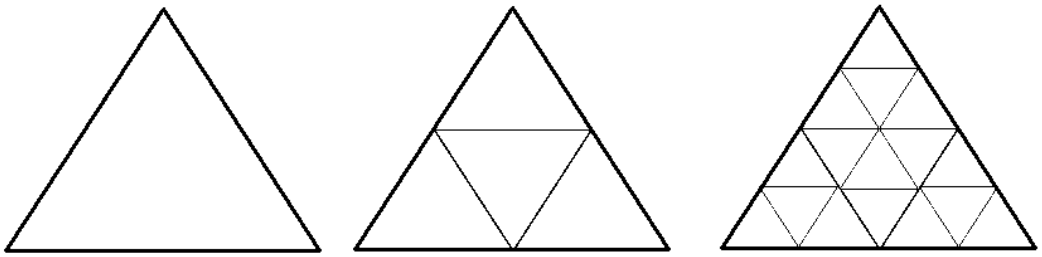
## 2.1 OpenGL grafiskā vide.

OpenGL (Atvērtā grafiskā bibliotēka) ir vairākvalodu, vairākplatformu lietojumprogrammas saskarne. To izmanto, lai efektīvi piekļūtu un izmantotu videokartes resursus un sasniegtu paātrinātu 2D un 3D vektoru grafikas renderēšanu. OpenGL ir grafikas API, kam vajag programmēšanas valodu, lai ar to strādātu. Tāpēc programmēšanas priekšzināšanas ir nepieciešamas. Vektoru grafikas renderēšana izmanto daudz lineāro algebru, ģeometriju un trigonometriju, tāpēc zināšanas šajās jomās arī ir nepieciešamas. OpenGL specifikācija precīzi paskaidro katru funkciju un tās rezultātus, bet implementācija ir atstāta izstrādātāju ziņā. Sākot ar 3.2. OpenGL versiju tiek pielietota kodola-profila metode, kas ietver sevī modernu praksi un efektīvu viektspēju.

OpenGL grafiku zīmēšanas cikls sākas ar koordinātēm – vektora punktiem. Tie tiek strukturēti ielasīti video kartes atmiņā izmantojot virsotņu un elementu buferu objektus. Buferu objekti tālāk tiek nodoti virsotņu ēnotājam un fragmentu ēnotājam. Pēc fragmentu ēnotāja notiek rasterizācija un zīmēšana uz ekrāna. Ēnotāji ir atsevišķas programmas, kuras var programmēt izmantojot glsl – grafiskās bibliotēkas ēnotāju valodu.

## 2.2. Trīsstūru pavairošanas algoritms.

Kursa darbā tika apskatīts rekursīvs trīsstūru pavairošanas algoritms. 2.2.1. Attēlā ir redzama pavairošanas algoritma iterāciju vēlamais darbības rezultāts 0, 1 un 2 iterācijām.



2.2.1 attēls – Trīsstūru pavairošana.

Algortima realizācija izskatītos šādi:

1. Sāk ar virsotnēm a, b, c.
2. Iegūst visu trīs šķautņu viduspunktus – v1, v2, v3 un saskaita četru trīsstūru virsotnes (a, v1, v2), (b, v1, v3), (c, v2, v3), (v1, v2, v3).
3. Rekursīvi katram trīsstūrim atrod nākamos 4 trīsstūrus.[5]

2.2.2. Attēlā var redzēt, kā tas izskatītos kodā. Katrs trīsstūris tiek rekursīvi sadalīts 4 trīsstūros, un funkcija atkārtoti tiek piesaukta katram trīsstūrim ar palielinātu dziļuma skaitu, līdz vēlamais dziļums ir sasniegts.

```
1  atrastTrissturus(float[] trissturis, int dzilums, int currentdepth)
2  {
3      //Dzilums ir sasniegts
4      if(currentDepth == dzilums)
5      {
6          saglabat(trissturis);
7          return;
8      }
9
10     //sadala punktus
11     float a = {trissturis[0],trissturis[1],trissturis[2]};
12     float b = {trissturis[3],trissturis[4],trissturis[5]};
13     float c = {trissturis[6],trissturis[7],trissturis[8]};
14
15     //Izrēķina viduspunktus
16     float v1 = middle(a,b);
17     float v2 = middle(b,c);
18     float v3 = middle(c,a);
19
20     //Izveido jaunos trīsstūrus
21     float[] pirmais = concatenate(a, v1, v2);
22     float[] otrais = concatenate(b, v1, v3);
23     float[] tresais = concatenate(c, v2, v3);
24     float[] ceturtais = concatenate(v1, v2, v3);
25
26     //Rekursīvi piesauc funkciju.
27     atrastTrissturus(pirmais, dzilums, currentdepth+1);
28     atrastTrissturus(otrais, dzilums, currentdepth+1);
29     atrastTrissturus(tresais, dzilums, currentdepth+1);
30     atrastTrissturus(ceturtais, dzilums, currentdepth+1);
31 }
```

2.2.2 attēls – Programmas kods rekursīvai trīsstūra sadalīšanai.

Tomēr, rekursīvajai pieejai ir daudz mīnusi. Iterācijām augot, rekursīvo piesaukumu daudzums aug exponenciāli, kas nav tas labākais risinājums priekš dinamiskas grafiskās sistēmas. Pavērojot trīsstūru iterācijas var ieraudzīt, ka punktu skaits uz trīsstūra šķautnes aug ar formulu, $2 + (2^n) - 1$, (n = iterācijas), un trīsstūru līmeņu skaits, ar katru nākamo rindu samazinās konstanti ar -1. To var izteikt ar for-ciklu, kas ietaupītu datora resursus, un padarītu arī vienkāršāku indeksu ģenerēšanu priekš elementu bufera.

For ciklā ir jāiekodē speciālie gadījumi, un jāizmanto iepriekš definētas trīsstūra koordinātas. Speciālie gadījumi sastāv no visiem malējiem punktiem, kas nozīmē – punkti uz apakšējās līnijas, augšējais punkts, punkti katrai līnijai abos sānos. Vidējiem punktiem – katrai līnijai var izmantot ciklu. Pilns punktu ģenerācijas kods ir 1. pielikumā. Šis punktu ģenerācijas algoritms ar tik daudz īpašajiem gadījumiem varētu šķist neintuitīvs, bet tas ievērojami ietaupa datora resursus salīdzinājumā ar rekursīvo punktu algoritmu, kas nozīmē vairāk iespējamās iterācijas un detalizētāku simulācijas pakāpi.

2.2. Tabulā var redzēt punktu skaitu attiecīgajai pieejai. Rekursīvā algoritma punktu aprēķināšanas formula – $4^x * 3$ . Ģenerācijas algoritma punktu aprēķināšanas formula – $\frac{(2+(2^x)-1)(2+2^x-1+1)}{2}$ . x = iterāciju skaits.

2.2. tabula –Iterāciju skaits pret punktu skaitu katram algoritmam.

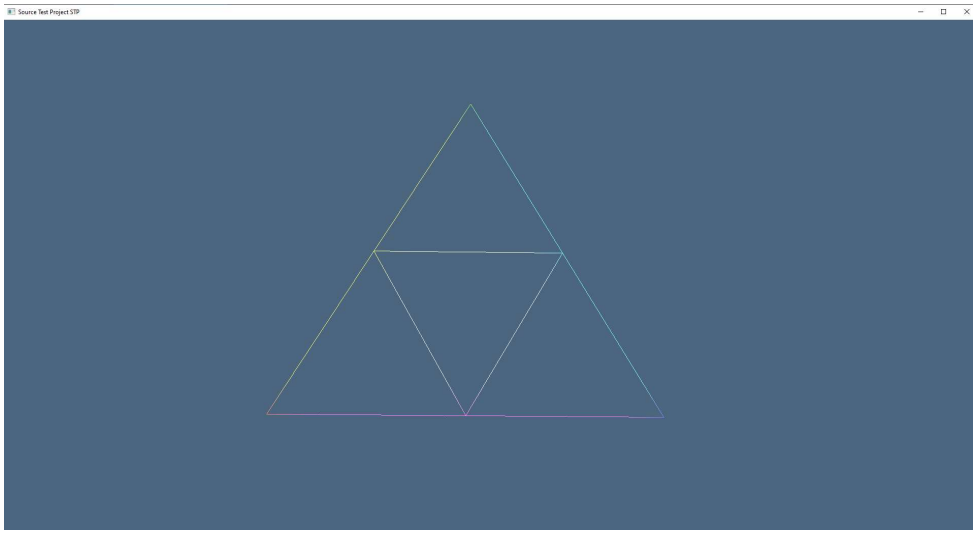
| Iterācijas | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Rekursīvais algoritms (punktu skaits) | 12 | 48 | 192 | 768 | 3072 |
| Punktu ģenerācija (punktu skaits) | 6 | 15 | 45 | 153 | 561 |

2.2.3. Attēlā var redzēt izveidotā trīsstūra stāvokli pēc vienas iterācijas. 2.2.4. Attēlā var redzēt izveidotā trīsstūra stāvokli pēc divām iterācijām. 2.2.5 Attēlā var redzēt izveidotā trīsstūra stāvokli jau pēc 10 iterācijām, kur monitora rezolūcijas dēļ punkti sāk saplūst, tāpēc 2.2.6. attēlā ir pietuvināts kreisais stūris. 10 rekursīvām iterācijām būtu nepieciešams glabāt 3,145,728 virsotņu

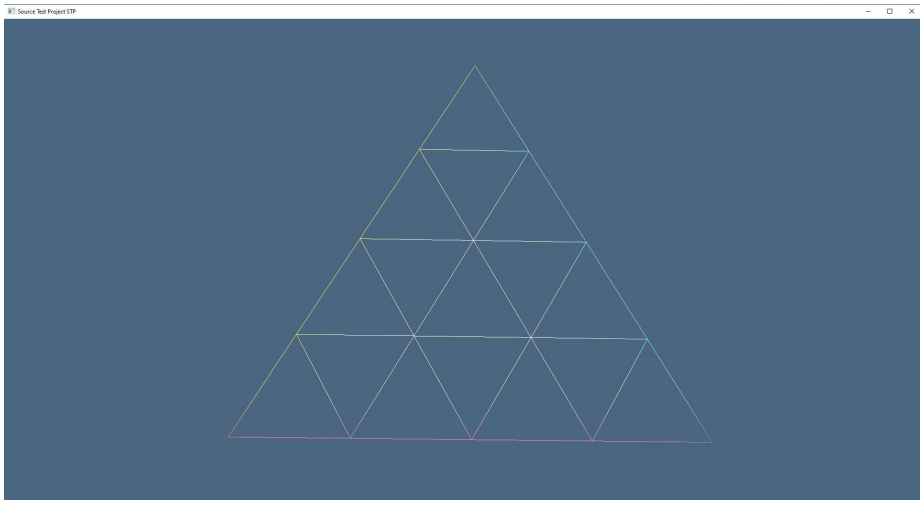
vērtības, ko vairs nebūtu iespējams glabāt vienā float punktu masīvā un vajadzētu izmantot papildus dinamiskās atmiņas konstrukcijas.

Ir vērts pieminēt, ka simulācijas vizualizācijas gaitā, katru sekundi tiek pārzīmēta visa scēna, kas nozīmē ievērojamus veiktspējas samazinājumus, ja tiek izmantoti vairāk punkti.

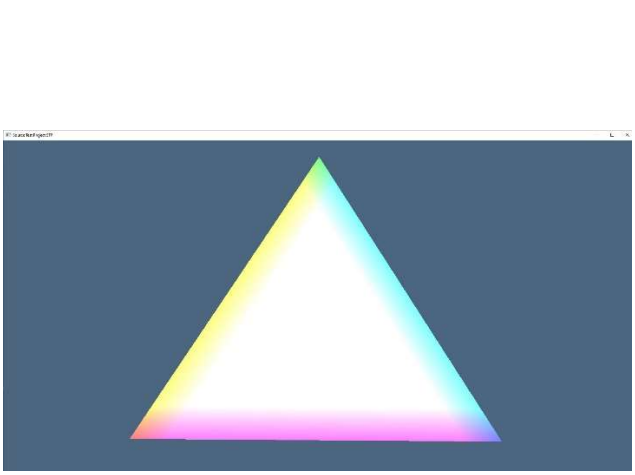
Darba autors testēja ģenerācijas algoritmu ar dažādām iterācijām, un tikai ap 12. Iterāciju tika novēroti mazāki attēli sekundē (144 vietā 50±2). Testēšana notika uz RTX 2060 sērijas video kartes.

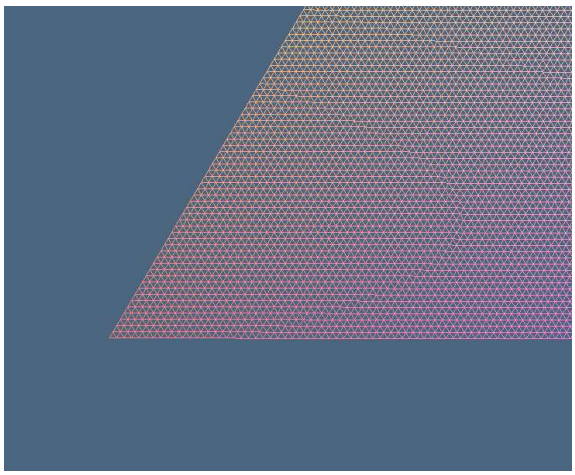


2.2.3 attēls – Trīsstūra ģenerācijas algoritma rezultāts OpenGL vidē, 1 iterācija.



2.2.4 attēls – Trīsstūra ģenerācijas algoritma rezultāts OpenGL vidē, 2 iterācijas.

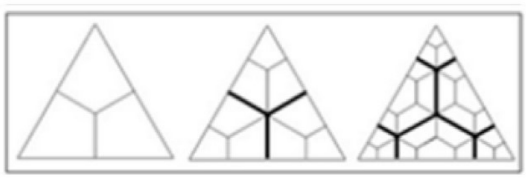




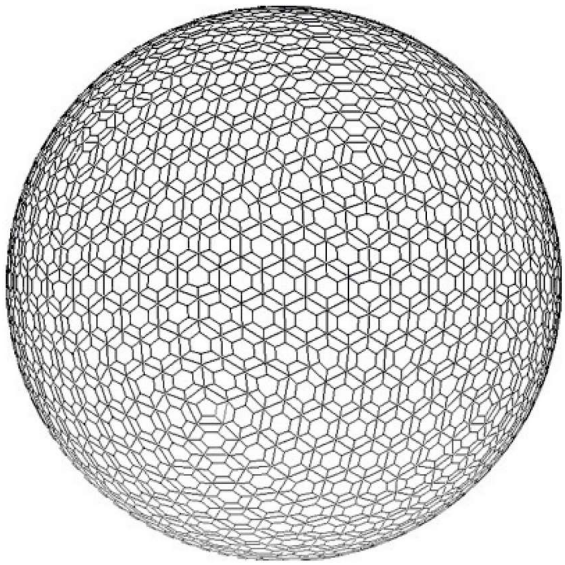
2.2.5 attēls – Trīsstūra ģenerācijas algoritma rezultāts OpenGL vidē, 10 iterācijas.

2.2.6 attēls – Trīsstūra ģenerācijas algoritma rezultāts OpenGL vidē, 10 iterācijas, pietuvināts kreisais stūris.

Trīsstūru sadalīšana nav vienīgais veids kā sadalīt laukumu mazākos segmentos. To var izdarīt izmantojot sešstūru sadalījumu. Balstoties uz pētījumu, sešstūru sadalījumam ir ģeometriskas priekšrocības modelējot nevienmērīgo zemes virsmu. Sešstūru sadalījumam kopā ar ikosaedra sfēras modelēšanu būtu ģeometriski mazāk nevienmērību. 2.2.6. Attēlā ir redzams trīsstūra rekursīvs sadalījums ar sešstūriem. 2.2.7. Attēlā ir redzama sfēra, kas veidota no sešstūru sadalījumiem. [4][6]



2.2.3 attēls – Trīsstūra rekursīvs sadalījums ar sešstūru metodi



2.2.3 attēls – Sfēra, kas veidota no ikosaedra, kurš sastāv no trīsstūriem ar sešstūra sadalījumu.

Saistībā ar grafiskajām sistēmām, sešstūru sadalījums nav izdevīgs, jo uz ekrāna vienmēr tiek renderēti poligoni – trīsstūri. Tas nozīmē papildus četrstūru sadalīšanu vismaz divos trīsstūros, un sešstūru sadalīšanu vismaz četros trīsstūros. Tomēr, nav nepieciešami papildus punkti, kas atļauj lietot elementu bufferi un ietaupīt resursus.

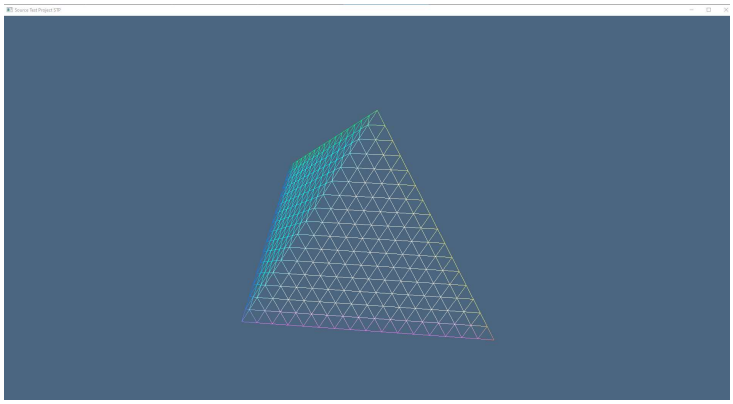
# 3. OKTAHEDRONA IZVEIDOŠANA UN RĀDIUSA ALGORITMS

## 3.1. Oktahedrona izveidošana

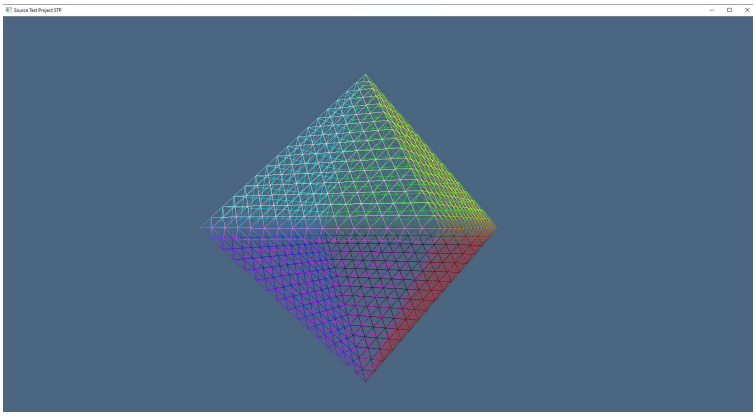
Lai izveidotu oktahedronu, trīsstūra pavairošanas algoritmu abstrahē funkcijā vai klasē, un piesauc ar atšķirīgām trīsstūra malām. Ja oktahedrona centrs ir (0,0,0), tad astoņu trīsstūru koordinātes būs visas iespējamās koodinātes ar 0, 1 un -1.

Elementu bufera masīvu nav jāpavairo atsevišķi, jo tas sastāv tikai no punktu indeksiem, kas jau ir pareizā secībā. Var izveidot for-ciklu, kurš iterē cauri esošajiem elementiem tos kopējot, un pieskaitot klāt iepriekšējā trīsstūra punktu skaitu.

3.1.1. Attēlā var redzēt kā izskatās divas oktahedrona puses pavairotas šādā veidā. 3.1.2. Attēlā ir gatavs oktahedrons. 2. Pielikumā ir pavairošanas funkcijas piesaukumu un elementu bufera cikla pirmkods.



3.1.1. Attēls -Divas oktaheodona puses.

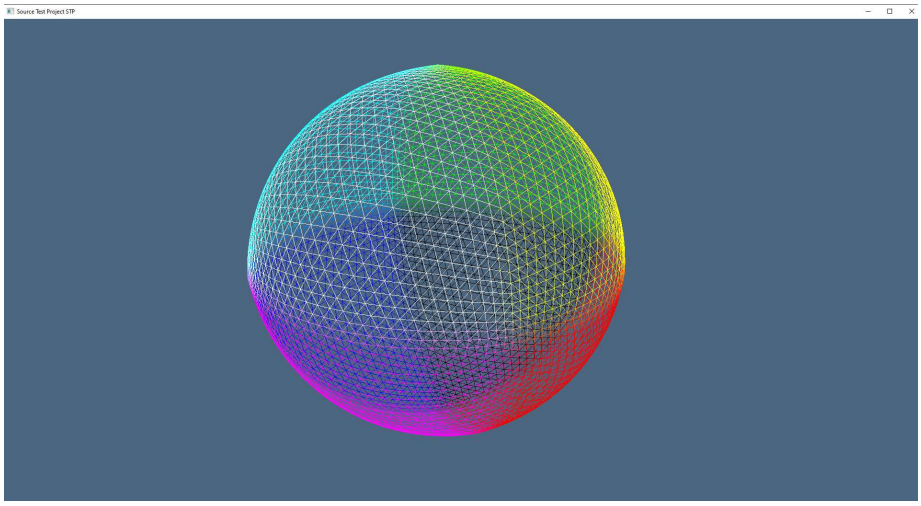


3.1.2. Attēls -Gatavs oktahedrons, izveidots pavairojot trīsstūra oktahedrona puses.

Lai testētu veiktspēju, autors izmainīja oktahedrona pušu trīsstūru pavairošanas iterācijas. Sākot ar 10 iterācijām 144 attēlu sekundē vietā bija 99±100. Ar 11 iterācijām attēli sekundē nokritās jau uz 25±10 attēliem sekundē. Testēšana tika veikta uz RTX 2060 sērijas videokartes. 3. pielikumā ir attēli no oktahedrona trīsstūru iterāciju testēšanas.

## 3.2. Rādiusa algoritms

Lai liktu oktahedronam izplesties un ieņemt sfēras formu, ir jāpielieto normalizācija katrai koordinātei attiecībā pret oktahedrona centra punktu (0,0,0), un izvēlēto rādiusa vērtību. Šī normalizācija sākas ar trīs esošo koordinašu distanču vērtību izrēķināšanu, no kurām šajā gadījumā nekas nav jāatņem, jo centra punkti visi ir nulles. Izrēķina distanci, kas ir pitagora teorēma 3D punktiem. Vēlamo rādiusu izdala ar izrēķināto distanci, tādā veidā iegūstot starpvērtību, kas norāda attiecību starp vēlamo rādiusu un esošo rādiusu. Ar starpvērtību reizina katras koordinātes distanci, un pieskaita esošajai koordinātei, tādā veidā iegūstot jaunas koordinātes ar esošo rādiusa lielumu. 3.2.1. Attēlā var redzēt oktahedronu, kurš ir ieņēmis sfēras formu. 4. Pielikumā ir normalizācijas pirmkods.



3.2.1. Attēls -Divas oktaheodona puses.

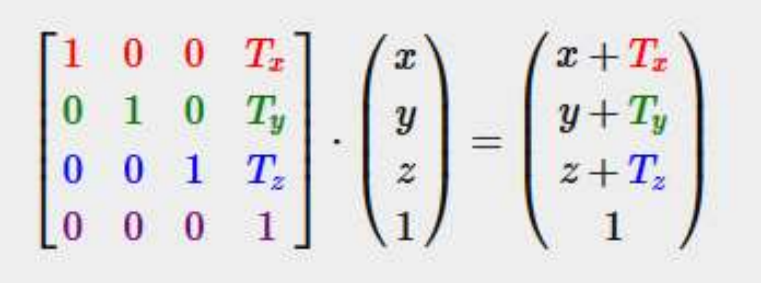
Oktahedrona punktu rādiusa palielināšana no centra neietekmē programmatūras veiktspēju, tāpēc to nav nepieciešams testēt.

# 4. SFĒRAS KUSTĪBAS ANIMĀCIJA UN KRĀSU IZMAIŅAS

## 4.1. Transformācijas

Objekta kustības animāciju var radīt divos veidos. Viens veids ir pa taisno virsotņu ēnotājam sūtīt kaut kādu vērtību, kas katrā ekrāna zīmēšanas ciklā tiek parrēķināta un izmantota objekta virsotņu koordinātu vērtībām. Šajā gadījumā notiek objekta koordināšu manipulācija virsotņu ēnotājā, kas vienādi iedarbojās uz visām objekta koordinātēm. Šis pielietojums ir noderīgs, ja ir nepieciešamība mainīt visas scēnas izmērus, vai iekodēt kādu īpašu uzvedību objektiem, izveidojot un pielietojot tiem atsevišķu ēnotāju.

Tomēr, katram objektam veidot atsevišķu ēnotāju nav labākais risinājums, ja ir nepieciešams dažādus objektus dažādi manipulēt. Virsotņu ēnotājā, virsotņu zīmēšanas brīdī, objektu virsotņu pozīcija attiecībā pret ekrānu tiek aprēķināta izmantojot modeļa, skata un projekcijas matricas. Šīs matricas tiek definētas un pārrēķinātas zīmēšanas cikla laikā, un viņas var izmantot, lai dinamiski mainītu skata leņķi, apgrieztu fragmentus, vai mainītu modeļa pozīcijas. Modeļa pozīcijas maiņu sauc par transformāciju, kas vienmērīgi izpildīta kļūst par animāciju, kuru var pielietot neatkarīgi katram modelim atsevišķi. Šīs transformācijas tiek izpildītas virsotņu ēnotājā, bet tiek piekārtotas katram modelim atsevišķi. Atmiņā, OpenGL matricas ir 16-vērtību masīvi ar bāzes vektoriem, kas novietoti blakus viens otram atmiņā. Transformācijas komponentes aizņem 13., 14., 15., un 16. elementus matricā. Tātad, transformācijas var ģenerēt ar vec3 konstrukciju, kas ir vektors ar 3 vērtībām, normalizētām starp 0 un 1. Pielietojot vektoru modeļa identitātes matricai, iegūst modeļa transformācijas matricu ar jaunajām koordinātēm. Šo transformēšanas matricu pēc tam pārsūta izmantojot uniformas virsotņu ēnotājam, kurš tālāk piekārto viņu objektam balstoties uz tekošo tā vērtību objekta zīmēšanas brīdī. 4.1.1. Attēlā var redzēt transformēšanas matricas un vektora reizinājumu.[7]



4.1.1 Attēls – Transformācijas matrica.

Pielietojot transformācijas nav manuāli jāpārrēķina visi objekta punkti, jo OpenGL tas notiek virsotņu ēnotājā.

Primitīva transformācija, kas liktu sfērai samazinātos, sastāv no sfēras rādiusa vērtības un soļa, kas norāda sfēras samazināšanās ātrumu. Katrā zīmēšanas ciklā tiek piesaukta funkcija, kas samazina šo vērtību par soli. Pēc tam, tajā pašā zīmēšanas ciklā atjaunotā vērtība tiek piekārtota transformāciju matricai un tiek reizināta ar modeļa koordinātēm, kas pēc tam tiek pārsūtītas uz virsotņu ēnotāju. Rezultātā ir izveidota konstanti lineāra sfēras transformācija, kas ekrānā izskatās kā sfēra, kas samazina savu lielumu.

## 4.2. Krāsu izmaiņas

Objekta krāsu izmainīšana notiek fragmentu ēnotājā. Fragmentu ēnotājam, tāpat kā virsotņu ēnotājam, var pārsūtīt vērtības izmantojot uniformas. Ja tiktu lietotas tekstūras fragmentu krāsu piešķiršanai, krāsu koordinātes tiktu piešķirtas izmantojot iebūvēto glsl funkciju *texture*, kas ņem divus parametrus. Pirmais parametrs ir 2d krāsu koordinātes sampleris jeb paraugs, kas norāda uz atmiņā ielasītu attēla pikseļu koordinātēm. Otrais parametrs ir tekstūras koordinātes, kas norāda uz 2d krāsu koordinātes uzklājumu. Otrais parametrs parasti tiek ģenerēts kopā ar objekta koordinatēm un iesūtīts ēnotājam izmantojot buffera objektu, kopā ar objekta virsotņu koordinātēm Ja netiek pielietotas tekstūras, objekta krāsu var kontrolēt ar četrām fragmentu krāsu vērtībām, kas atbilst sarkanam, zaļam, zilam, un caurspīdīgumu. Fragmentu krāsas vērtības fragmentu ēnotājā ir normalizētas stap 1.0f un 0.0f., tāpat kā virsotņu vērtības. Tāpēc virsotņu vērtības var izmantot arī fragmentu ēnotājā, lai iegūtu krāsu pārejas balstoties uz objekta koordinātēm. Šīs krāsas vērtības var arī reizināt ar intensitātes vērtību, kas tiek mainīta katrā zīmēšanas ciklā, tādā veidā iegūstot krāsu pāreju, kas izpildās kā animācija. Fragmentu krāsas ēnotājs interpretē kā float krāsu vērtību

standartu, tāpēc var izmantot internetā atrodamus krāsu atlasītājus, lai iegūtu vēlamās krāsas. Piemēram, sarkanai krāsai OpenGL vidē būs tādas pašas RGB vērtības, kā web izstrādes vidē, tikai normalizētas starp 1.0f un 0.0f. (255.0.0) un (1.0f, 0.0f 0.0f).[8][9]

Causpīdīguma vērtība ir ceturtā fragmentu krāsas vērtība, kas pēc noklusējuma tiek ignorēta OpenGL. Lai izmantotu caurspīdīguma alfa-vērtības, ir jāieslēdz GL_BLEND un jākonfigurē Blend funkcija.[10] 4.2.1. Attēlā var redzēt koda fragmentu.

```
// Enable blending
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

4.2.1 Attēls – OpenGL blend funkcijas iestatīšana.

glEnable() un glDisable() ir funkcijas, kuras ieslēdz vai izslēdz vairākas OpenGL iespējas.[11]

glGet() var izmantot, lai iegūtu šī brīža OpenGL izstādījuma statusu.

glBlendFunc() uzstāda pikseļu aritmētiku. Var izmantot glBlendFunci(), kas ņem papildus vienu parametru – buffera objektu, tādā veidā atļaujot uzstādīt individuālu caurspīdīguma režīmu katram objektam.

glBlendFunc() ņem divus parametrus – sfactor, kas ir OpenGL tipa specifikācija un norāda uz to, kā rgb un alfa sākuma vērtību sajaukšanas faktori tiek aprēķināti. Dfactor, kas ir OpenGL tipa specifikācija un norāda uz to, ka rgb un alfa gala sajaukšanas faktori tiek aprēķināti.

GL_SRC_ALPHA norāda uz sākuma vērtību sajaukšanas formulu, kas atbilst formulai 4.2.2. attēlā.

A vērtība atbilst (RGBA) alfa vērtībai.

K vērtība atbilst formulai 4.2.3. attēlā.

M vērtības norāda uz (mR, mG, mB, mA), kas atbilst rgb un alfa bitu plaknēm.

$$\left( \frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A}, \frac{A_{s0}}{k_A} \right)$$

4.2.2 Attēls – GL_SRC_ALPHA formula.

$$k_c = 2^{m_c} - 1$$

4.2.3 Attēls – k vērtības formula.

GL_ONE_MINUS_SRC_ALPHA norāda beigu vērtību sajaukšanas formulu, kas atbilst formulai 4.2.4. attēlā.

$$(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$$

4.2.4 Attēls – GL_ONE_MINUS_SRC_ALPHA formula.

Šīs ir tikai divas no 19 definētajām OpenGL konfigurācijām priekš caurspīdīguma sajaukšanas. Sarežģītākas konfigurācijas ir nepieciešamas, kad vizualizācijā tiek pielietots anti-aliasings vai tiek haotiskā secībā renderētas līnijas. Haotisku līniju secības gadījumā katrā ekrāna zīmēšanas brīdī ir jāpārrēķina visu uz ekrāna redzamu trīsstūru pozīcijas attiecībā pret dziļuma plakni un tekošajām pikseļu vērtībām. Tas ir skaitliski dārgs risinājums, tāpēc praksē tiek reti pielietots.

Vēlviens veids, kā iegūt caurspīdīguma vērtības neizmantojot funkcijas un ietaupot skaitļošanas resursus, ir izmantojot tekstūras ar caurspīdīgiem pikseļiem. Fragmentu ēnotājā, tāpat kā tiek iegūtas rgb vērtības, var iegūt arī alfa-caurspīdīguma vērtību un to uzklāt objektam. Fragmentu ēnotājā ir iespējams katram objektam uzlikt divas vai vairāk tekstūras, kur viena tekstūra atbilst fragmentu krāsai, bet cita fragmentu intensitātei, krāsu intensitātei, vai caurspīdīgumam. Šo pieeju sauc par spekulāru karšu, vai atspulgu karšu izmantošanu. Tāpat kā virsotņu ēnotājā tiek reizinātas modeļa, skata un projekcijas matricas, lai iegūtu pikseļu koordinātas atbilstoši ekrāna skatapunktam, tāpat arī tiek rēķinātas fragmentu krāsas. Sareizina tekstūras, apgaismojuma, intensitātes un caurspīdīguma vērtības, lai iegūtu gala fragmentu vērtības. Ar šādu pieeju var uzzīmēt relatīvi vienkāršus objektus ar ļoti augstu detalizācijas pakāpi. 4.2.5. Attēlā var redzēt trīs pakāpju tekstūru izmantošanu fragmentu vērtību iegūšanai.[12]

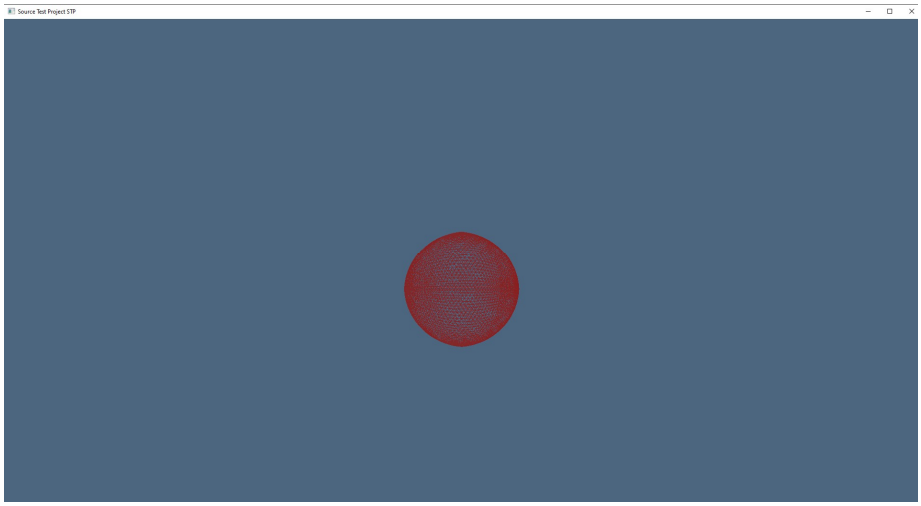
```
vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

4.2.5 Attēls – Tekstūru pikseļu vērtību reizināšana fragmentu vērtībām.

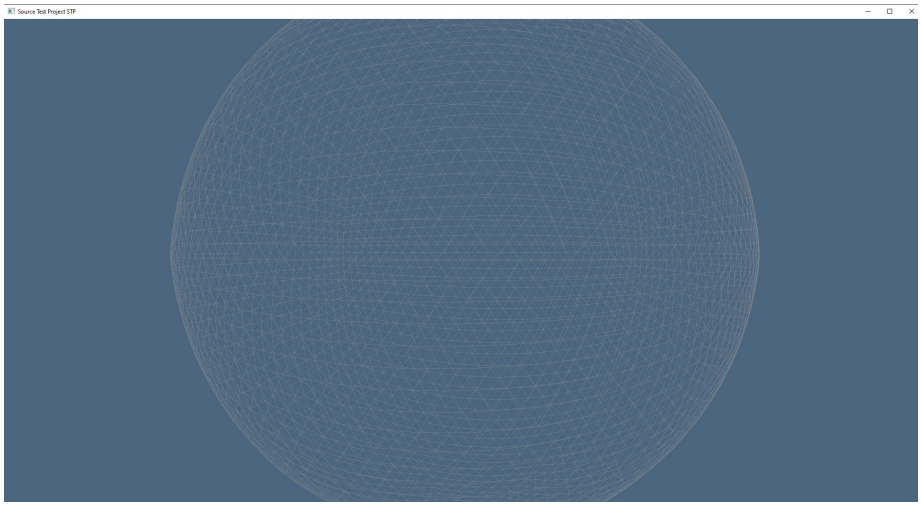
4.2.6. Attēlā un 4.2.7. attēlā ir redzams rezultāts pēc transformāciju funkcijas izveidošanas un piesaukšanas caur zīmēšanas ciklu, kā arī krāsas un caurspīdīguma nomaiņu balstoties uz

transformācijas funkcijas vērtībām. Attēlos ir redzams vienslāņa sfēras modelis, kas jau aptuveni šajā brīdi varētu simbolizēt pirmszvaigžņu mākoņa apvalku, kuram saraujoties temperatūra palielinās, līdz ar to sarkanā krāsa kļūst intensīvāka.

Kad pietiekami daudz materiāls ir sakritis centrā, izveidojas protozvaigzne. Dzīves gaitā, protozvaigzne turpina savākt materiālu no apkārtējā mākoņa, līdz tā kļūst par galvenās secības zvaigzni. Galvenās secības zvaigznei ir atmosfēra, no kuras tiek aizpūsts prom materiāls – protoni, elektroni un smagāku vielu atomi. Šis aizpūstais materiāls veido zvaigžņu vēju, kurš uzreiz, zvaigznei rodoties, aizpūš prom tās molekulāro mākoni, kurā tā atradās. Sfēras izplešanās simbolizē šo aizpūsto mākoņa apvalku, kurš paliekot lielāks maina krāsu uz oranžu – pelēku – baltu, līdz pavisam izgaist.



4.2.6 Attēls – Sarāvusies sfēra ar intesīvu, sarkanu krāsu.



4.2.7 Attēls – Izpletusies sfēra, ar baltu krāsu un caurspīdīgumu.

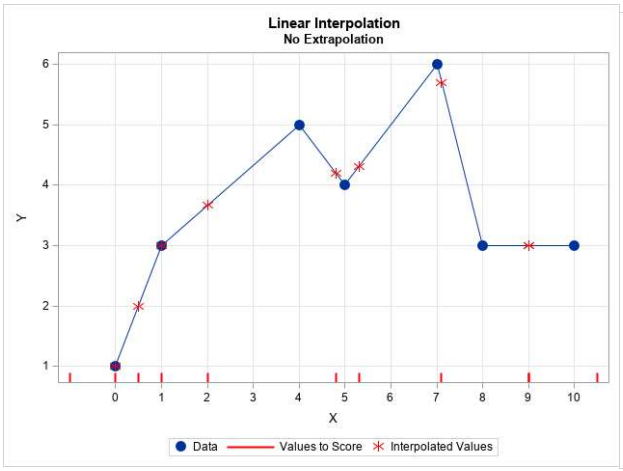
# 5. SFĒRAS INSTANCES UN SIMULĀCIJA

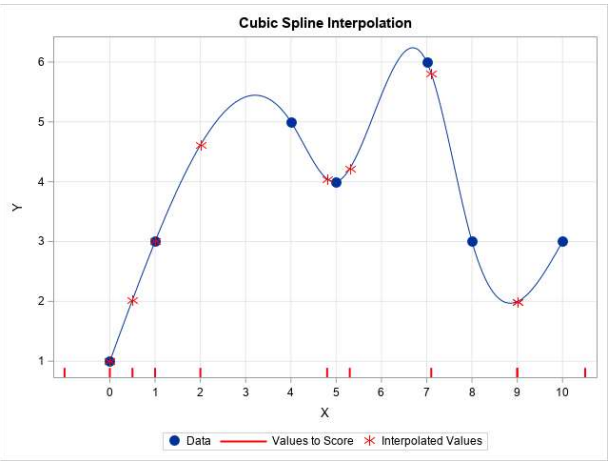
## 5.1. Montekarlo metode, interpolācija un integrēšana

Montekarlo metode ir skaitļošanas tehnika, kas meklē atrisinājumu (parasti – aptuvenu) matemātiskām problēmām, kuras pamatprincips ir nejauši izvēlētas vērtības. Montekarlo metode parasti tiek pielietota divām problēmām – integrēšanai un optimizācijai. Tā tiek arī plaši pielietota fiziskas, ķīmijas un bioloģijas simulācijās. Daudziem modeļiem ir neprātīgi sarežģītas struktūras, kuras nav iespējams atrisināt izmantojot tradicionālas metodes. Principā, montekarlo metodes var pielietot, lai atrisinātu jebkuru problēmu, kurai ir varbūtiska interpretācija. No statistikas teorijas, balstoties uz lielā skaitļa likumu, integrāļi, kas aprakstīti ar to sagaidāmajām vērtībām un kādu nejaušu vērtību, var tikt tuvināti izmantojot empīrisko, jeb vienkāršo vidējo no neatkarīgiem vērtību rezultātiem. Kad varbūtiskā vērtības izkliede ir noteikta, tai var izmantot markova ķēžu montekarlo vērtību iegūšanai.[13]

Lineārā interpolācija ir metode, ar kuras palīdzību var pietuvināt skaitļus nezināmām vērtībām, kuras atrodas starp diviem citiem skaitļiem. Metode strādā izveidojot no dotajiem skaitļiem attiecības vērtību, un balstoties uz to piešķirot vērtību nezināmajam punktam. Tas ir vienkāršs, bet neprecīzs veids, kā noteikt nezināmos punktus starp x un y vērtībām, saglabājot kaut kādu x un y vērtību atkarību. 5.1.1. Attēlā var redzēt lineārās interpolācijas tredus.[14]

Cubic spline, jeb kubiskā splainu interpolācija ir augstākas pakāpes interpolācija, kur interpolants ir polinomiālas daļas, kuras sauc par splainiem. Splainu interpolācija pielīdzina zemas pakāpes polinomiālus nelielām vērtību apakškopām. Splainu interpolācija ir ar daudz augstāku precizitātes pakāpi nekā lineārā vai polinomiālā interpolācija, jo interpolācijas kļūda var tikt samazināta pat ar zemas pakāpes polinomiāļiem. Salīdzinājumā ar lineāro interpolāciju, kas ir nevienmērīga, kubiskā splainu interpolācija ir gluda, saglabā datu trendus un var izveidot jaunus ekstrēmos punktus. 5.1.2. Attēlā var redzēt cubic spline trendus, līnijas ir daudz gludākas.[15]




5.1.1 attēls – Lineārā interpolācija. 5.1.2 attēls – Kubiskā splainu interpolācija.

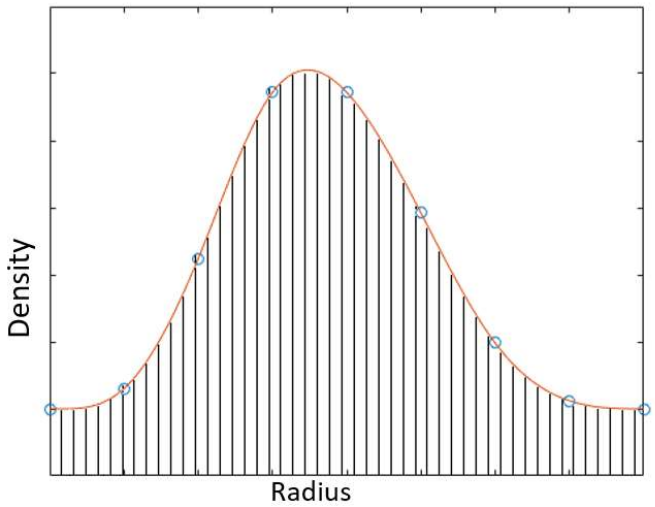
Modelējot dabas procesus, vērtīgi ir izmantot tuvinājumus ar vismazākajām kļūdām, tāpēc kubiskā splainu interpolācija ir labākais variants pirmszvaigžņu blīvuma profila modelēšanai. Pirmszvaigžņu mākoņa raksturojošus parametrus mēra ar ļoti liela, un ļoti maza apjoma vērtībām. Mākonis atrodas aukstā vidē, kur rodas ūdeņraža molekulas. Ūdeņraža atoma svars ir 1.0079, kas gramos izsakās 1,6735575E-24. Molekulā ir divi atomi, kas nozīmē, ka svaru var tuvināti reizināt ar divi. Pirmszvaigžņu mākonim raksturīga centra masa ir 100 000 – 400 000 atomi, un 20 000 atomi mākoņa malās, kas izsakās ar lielumiem 3.347115e-19 un 3.34738E-0020 gramos. Mākoņa rādiuss tiek izteikts centimetros, kas nozīmē milzīgus lielumus kosmiskos mērogos. No zinātniskās literatūras, pirmszvaigžņu masu centrā, ar mūsdienu instrumentiem nav iespējams noteikt, jo mākonim paliekot blīvākam tas kļūst necaurredzams.[16] Tomēr, ar dažādu noteicēju savienojumu palīdzību var aptuveni noteikt koncentrācijas dažādos mākoņa rādiusos. Šīs rādiusa vērtības tiek izteiktas astronomiskajās vienībās, kuras ir jākonvertē uz centimetriem. 507.648 AU = 7.59432E+0015 cm, 20882.650 = 3.12400E+0017 ir tipiskas rādiusu vērtības pirmszvaigžņu mākonim.

Alglib ir C++ bibliotēka, kas satur ātras un optimizētas funkcijas viendimensionālai kubiskā splaina interpolācijai un splaina integrācijai. Alglib Free Edition sastāvā ir funkcijas, kas nav optimizētas vairāku procesora pavedienu izmantošanai, bet tās apmerina prasības lielu datu montekarlo integrācijas vērtību meklēšanai.[17]

Lai ģenerētu datu failu, kura sastāvā ir matemātiski vienotas, hidrostatiska mākoņa sfēru šūnu masas un rādiusa vērtības, ir jāsāk ar punktu vērtību ielasīšanu, kas atbilst rādiusa(centimetros) un blīvuma masas(gramos) koordinātēm. Koordinātes ielasa Alglib real_1d_array struktūrā. Datu ielasīšana notiek definējot real_1d_array objektus AX, AY, un

izmantojot funkciju AX.setcontent, kas ņem divus parametrus – masīva lielumu un masīvu. Ar izveidotām vērtībām var nodefinēt spline1dinterpolant struktūru, un izveidot kubisko splainu ar spline1dbuildcubic funkciju, kas ņem real_1d_array vērtības un spline struktūru. Spline integrācija notiek izmantojot spline1dintegrate funkciju, kuras parametri ir spline struktūra un x ass punkts, līdz kuram notiks integrēšana.

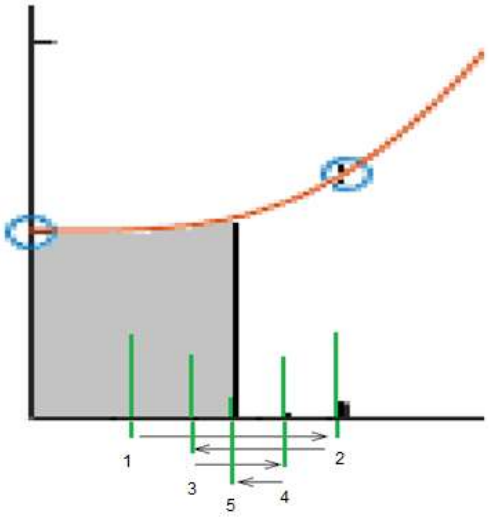
Lai atrastu mākoņa kopējo masu, tiek integrēts blīvums visa rādiusa garumā. Integrētais blīvums tiek izdalīts ar kopējo rādiusu, lai iegūtu vidējo blīvumu, kurš pēc tam tiek reizināts ar riņķa tilpuma formulu, lai iegūtu kopējo mākoņa masu. Izdalot mākoņa masu ar šūnu skaitu, iegūst vienas šūnas masas vērtību. 5.1.3. Attēlā ir redzama aptuvenā blīvuma funkcijas sadalīšana. Realitātē masas funkcija sastāv no kubiskas blīvuma rādiusa atkarības, kas nozīmē, ka šūnas būtu platākas pie x=0 un progresīvi kļūtu šaurākas x tiecoties uz gala vērtību.



5.1.3 attēls – blīvuma funkcijas sadalīšana šūnās.

Montekarlo minēšana notiek izvēloties sākuma punktu. Lai izvairītos no liekiem soļiem, to var pieņemt par rādiusu, izdalītu uz šūnu skaitu. Lai atrastu katru rādiusa vērtību, kas atbilst kopējai masai attiecīgajā šūnā, tiek veidots cikls katrai šūnai. Katrai šūnai cikls izmanto savādākas salīdzināšanas un sākuma rādiusa vērtības, kas ir šūnas masa, reizināta ar šūnas kārtas skaitli. Un rādiuss, izdalīts uz šūnu skaitu un reizināts ar šūnas kārtas skaitli. Rādiusu izmanto, lai katrā iterācija integrētu blīvumu un pārveidotu masas punktā. Masa tiek salīdzināta ar vēlamo šūnu masu punktā, un tiek izrēķināta starpība. Balstoties uz šo starpību rādiusa vērtība tiek apstiprināta, ja kļūda ir pietiekami maza. Ja kļūda ir lielāka, priekš nākošās iterācijas rādiusu pakustina par soli.

Soļa vērtība sākumā ir rādiuss, izdalīts uz šūnu skaitu. Ja kļūda, iterāciju gaitā, ir mainījusi zīmi, tas nozīmē, ka ir pārkāpts pāri vēlamajam punktam, un soļa vērtība tiek dalīta uz 2, un tai tiek mainīta zīme. Rādiusa vērtības minēšana izmantojot soļa vērtības, ļauj veikt minējumus 1 - $10^{20}$ diapazonā tikai 64 iterācijās, kas ļoti ātri atrod vērtības. Šī iemesla dēļ minēšanas metode strādā saprātīgā laikā arī tad, ja vajag atrast ļoti daudz (100000) punktus. Ir vērts pieminēt, ka iesaistīto, lielo skaitļu dabas dēļ, pieļaujamās kļūdas pieņemtajai vērtībai nedrīkst būt mazākas par 10+E15 no pašas vērtības. Savādāk programmatūras un aparatūras limitāciju dēļ no rādiusa vairs netiek atņemtas soļa vērtības, jo solis ir pārāk mazs, kas liek programmai iestrēgt ciklā un nekad neatrast meklēto vērtību. 5.1.4. Attēlā var redzēt aptuvenu vēlamās rādiusa vērtības meklēšanas algoritma darbību.




5.1.4 attēls – Montekarlo minējumu algoritms. X ass atbilst rādiusam un y ass atbilst blīvumam. 1. Solī vērtība neatbilst, tāpēc rādius tiek pakustināts pa soļa vērtību. 2. Solī vērtība arī neatbilst, bet kļūda ir ar negatīvo zīmi, kas nozīmē, ka ir pāriets pāri vēlamajai vērtībai, tāpēc soļa vērtība tiek samazināta un zīme tiek mainīta. 3. Solī notiek tas pats, kas pirmajā. 4. Solī tas pats, kas 2. Solī. 5. Solī vērtība ir ar pietiekami mazu kļūdu, un to var apstiprināt.

## 5.2. Simulācija

Izmantojot atrastās montekarlo pirmszvaigžņu mākoņa šūnu vērtības un modeļa skalāro reizinājumu, var uzģenerēt matemātiski un fiziski akurātas, un saistītas šūnas.

Lai veiktu simulāciju, izveido uz masas balstītu gravitācijas funkciju, kas katrā zīmēšanas ciklā, izmantojot gravitācijas formulu, samazina šūnu rādiusu. Tā kā visu šūnu masas lielums ir vienāds, gravitācijas spēku ir viegli aprēķināt. Ja kāda šūna pārsniedz kritisko bonor-ebert masu,

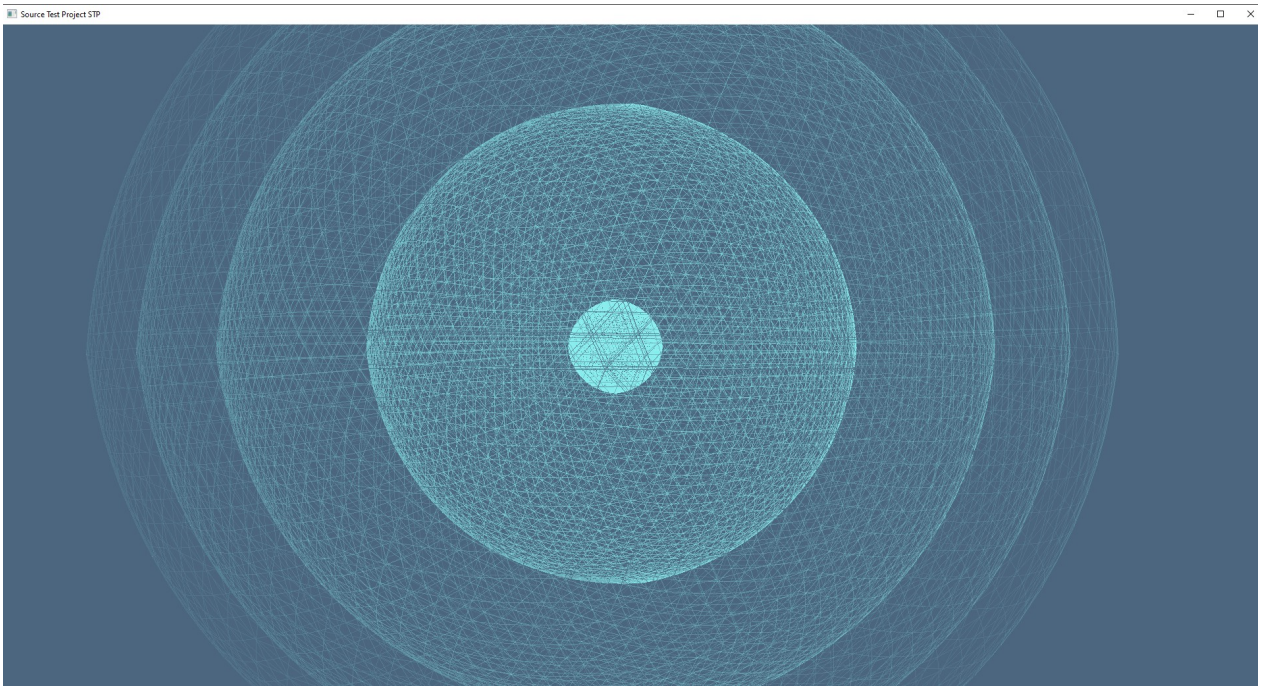
var uzskatīt, ka zvaigzne ir izveidojusies un apstādināt gravitāciju centra šūnām. Visām pārējām šūnām sāk simulēt zvaigznes vēju, kas pūš ārējās šūnas prom un to rādiuss palielinās. 5.2.1. Attēlā var redzēt gravitācijas formulu, pēc kuras var aprēķināt šūnu rādiusa samazinājumu. [18]

$$F = G\frac{Mm}{r^2}$$

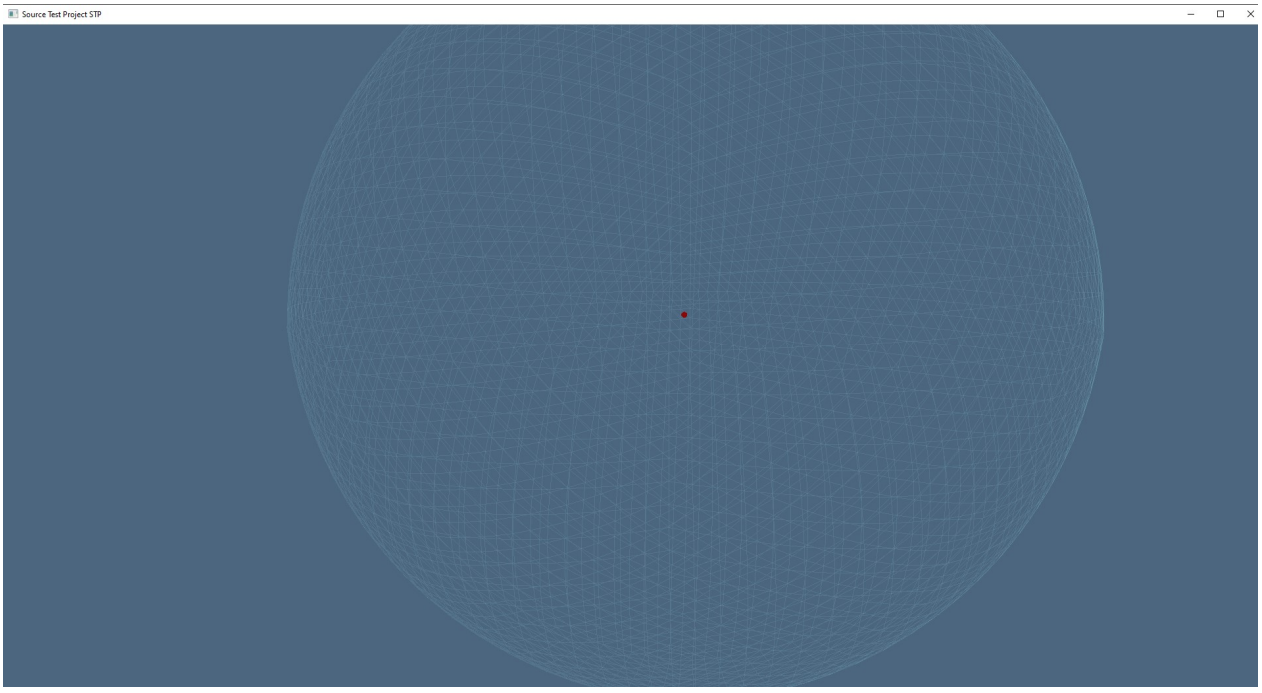
5.2.1 attēls - Gravitācijas formula.

Attēlos 5.2.2. un 5.2.3. var redzēt simulācijas vizualizācijas sākuma un beigu stāvokli. Sākuma stāvoklī šūnas lēnām krīt uz centru. Krišanas ātrums lēnām palielinās. Šūnām esot tuvāk centram tās maina krāsu uz gaiši zilu, tad uz oranžu, tad uz koši sarkanu. Brīdī, kad rodas zvaigzne centra šūnas apstājas un to krāsa paliek koši sarkana, bet ārējais apvalks tiek pūsts prom un attālinās no centra šūnām. Ārējais apvalks turpina attālināties, un maina krāsu no koši sarkanas uz oranžu, zilu, baltu, līdz sāk kļūt caurspīdīgs, un beigās pilnībā izdziest. Pašā beigu stāvoklī ir palicis viens sarkans objekts – jauna galvenās secības zvaigzne. [19]

Simulācijā bonor-ebert masa ir pārvērtēta, jo realitātē zvaigzne būtu tik maza, ka to nebūtu iespējams uzrenderēt kopā ar ārējiem apvalkiem uz esošās rezolūcijas. Zvaigznes vējš arī ir vienkāršots, jo realitātē tas nepūš vienmērīgi uz visām pusēm. Akurāta vēja reprezentācija prasītu īpaša gadījuma iekodēšanu, kas vairs nebūtu sfēriska arhitektūra.




5.2.2 attēls – Simulācijas vizualizācijas sākums.




5.2.3 attēls – Simulācijas vizualizācijas beigas.

# 6. REZULTĀTI UN DISKUSIJA

Darba gaitā tika izveidotas divas programmas. Galvenā programma ir OpenGL kodola kolapsa simulācijas vizualizācija, kas reallaikā simulē kodola sabrukšanu. Vienīgie ārējie dati, kas simulācijā tiek ielasīti, ir sākuma stāvoklis. Simulācijas gaitā, stāvoklis nomainās tikai balstoties uz iekodētiem aprēķiniem un sakarībām, bez ārējiem ievaddatiem.

Šī brīža simulācijas vizualizācijai ir arī augsta optimizācijas pakāpe, jo to ir iespējams divu trīsstūru iterācijas robežās palaist uz daudz vajākiem datoriem. Simulācijas vizualizācija tika testēta uz diviem datoriem ar dažādām trīsstūru iterācijām. Testēšanas rezultātus var redzēt 6.1 Tabulā. Attēli sekundē intensīvākajā brīdī norāda uz brīdi tieši pirms galvenās secības zvaigznes izveidošanās.

1. Datora parametri:

- Ryzen 5 2400g, 3.6GHz, 4 kodolu, 8 pavedienu procesors;
- RTX 2060, 1680 MHz, 6GB GDDR6 video karte;
- 16GB DDR4, 3000MHz RAM.

2. Datora specifikācijas:

- I5-7200U, 2.5GHz, 2 kodolu, 4 pavedienu procesors;
- HD Graphics 620, 1000MHz, līdz 4GB DDR4 videokarte;
- 8GB DDR4 2400MHz RAM.

3.1. tabula

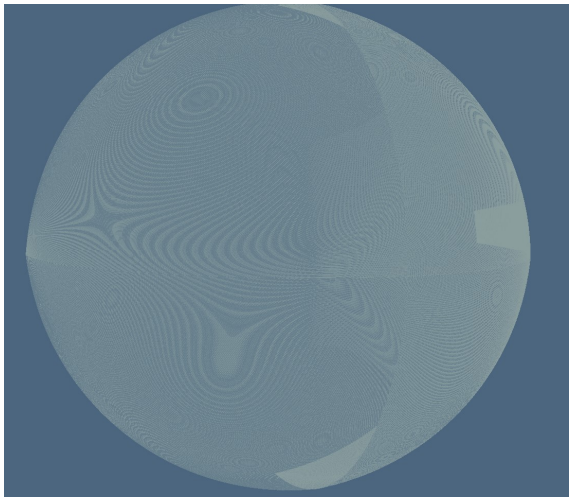
| Dators | Iterācijas | Punktu skaits | Attēli sekundē | Attēli sekundē intensīvākajā brīdī |
|---|---|---|---|---|
| 1. | 7 | 8 385 | 144 | 144 |
| 1. | 8 | 33 153 | 144 | 144 |
| 1. | 9 | 131 841 | 144 | 76 |
| 1. | 10 | 525 825 | 54 | 19 |
| | | | | |
| 2. | 7 | 8 385 | 60 | 60 |
| 2. | 8 | 33 153 | 43 | 27 |
| 2. | 9 | 131 841 | 16 | 5 |
| 2. | 10 | 525 825 | 5 | 1 |

Ir sasniegts darba galvenais mērķis, kas ir prezentēt vizuālu simulāciju un tās izstrādes procesu. Simulācija netika izstrādāta pārāk ilgā laikā, un ar maz trīsstūru iterācijām to ir iespējams palaist pat uz vājākajiem datoriem. Simulācija vizuāli atbilst vēlamajam rezultātam, un pie tās piestrādājot varētu to padarīt par pilnīgi reprezentatīvu reālam pirmszvaigžņu mākoņa dzīves ciklam. Simulācijā pielietotās konstrukcijas nav pārāk sarežģītas, un tās izstrādi ir iespējams atkārtot ar minimālām programmēšanas zināšanām. Neskatoties uz to, lielākajai daļai zinātnieku un pasniedzēju, kuriem šādas simulācijas varētu būt noderīgas atklājumu prezentēšanai vai informācijas nodošanai, varētu būt grūtības ar programmēšanu dinamiskajā OpenGL grafiskajā vidē. Problēmas galvenokārt rastos no vājās dokumentācijas un lietotāju bāzes, kas nozīmē daudz eksperimentēšanu, lai nonāktu pie vēlamā rezultāta.

Ir pamats domāt, ka šādas simulāciju vizualizācijas būs pieprasītas nākotnē, jo NASA un ESA aizvien biežāk publicē sociālajos tīklos simulāciju vizualizācijas, kas ir veidotas šādā stilā. Starp citu valstu bakalaura, maģistra un doktorantūras studentiem arī ir novēroti diplomdarbi, kas saistās ar šādu simulācijas vizualizācijas izveidošanu. Ir arī gadījumi, kad individuāli zinātnieki pievieno šādi izveidotu simulāciju rakstam, lai labāk vizualizētu rakstā aprakstītos fizikas procesus.[20]

Otrā programma ir datu ģenerēšana ar montekarlo metodi. Programma strādā pietiekami ātri, lai to varētu ievietot vizualizācijās un iegūt papildus matemātisko atkarību. To arī ir iespējams pielāgot citiem, ar matemātiku un fiziku saistītiem uzdevumiem. 10 000 vērtību ģenerēšana programmai aizņēma 5.9 sekundes. Vidējais vienas vērtības atrašana prasīja 50 iterācijas.

Nākotnē ir plānots apskatīt jauna modeļa izveidošanu, kurš saistās ar punktu mākoņa konstrukcijām. Pašreizējam modelim ir vairāki trūkumi, piemēram, tajā nav iespējams iekodēt protozvaigznes disku. Tas nozīmē, ka tiek palaists garām vesels posms zvaigžņu veidošanās ciklā. Punktu mākonī papildus protozvaigznes diskam varētu iekodēt arī turbulenci un akurātu zvaigžņu vēja reprezentāciju. 6.1 attēlā arī var novērot sfēras artifaktus, kas rodas no pārāk blīva trīsstūru pārklājuma. Punktu mākoņa modelim nebūtu šādas problēmas. Šobrīd, programmatūras limitējošais faktors nav trīsstūru punktu daudzums, bet indeksu daudzums. Punktu mākoņa modelī renderēšana notiktu tikai ar punktiem.



6.1 attēls, sfēras artifakti.

# 7. SECINĀJUMI

No izstrādātās simulācijas vizualizācijas var secināt, ka simulāciju vizualizēšana izmantojot OpenGL vidi ir labs, vienkāršs, un efektīvs datu vizualizācijas risinājums.

No izstrādātās datu ģenerācijas programmas var secināt, ka pareizi implementēta monte karlo metode spēj ar relatīvi maz minējumiem (40-60) atrast integrācijas punktus.

Darba sākumā apskatītā trīsstūra pavairošana ar trīsstūriem un sešstūriem liecina par to, ka laukumu ģenerēšanas algoritmi nav pienācīgi izpētīti. Sešstūru pavairošanas algoritmam vispār nebija atrodami piemēri internetā, un trīsstūru algoritmam bija pieejami tikai neefektīvi, rekursīvi varianti.

Viena no šī darba īpatnībām, bija projekta realizācija OpenGL dzinī, kurā jau bija implementēta brīva 3D kameras kustība. Darba gaitā autoram bija aizraujoši lidināties apkārt, un apskatīt izveidotās konstrukcijas novisām pusēm. Brīva 3D kameras kustība arī atļāva pieredzēt simulācijas vizualizācju no dažādiem skatpunktiem, kā no centra, kas deva pavisam savādāku notikumu priekšstatu, kā vērojot darbību no ārpuses. Turpmākajos plānos radās arī ideja darbu realizēt virtuālās realitātes vidē, lai vēl vairāk iegremdētos vizualizācijas pieredzē.

Darba izstrādes gaitā tika arī nejaušu uzietas dažādi vizualizācijas rakstu fenomeni, kas radās no kļūdaini uzrakstīta koda.

# IZMANTOTĀ LITERATŪRA UN AVOTI


[1] Kate Bush, Ed Burns - data visualization – february 2020. [tiešsaiste]. Pieejams internetā: https://searchbusinessanalytics.techtarget.com/definition/data-visualization

[2] Matthew Steward – The Power of Visualization in Data Science – may 2019. [tiešsaiste] Pieejams internetā: https://towardsdatascience.com/the-power-of-visualization-in-data-science-1995d56e4208

[3] Visual components – Simulation vs. Visualization – what's the difference? – may 2017. [tiešsaiste] Pieejams internetā: https://www.visualcomponents.com/resources/blog/simulation-vs-visualization-difference/

[4] Denis White, A.John Kimerling, Kevins Sahr, Lian Song – Comparing area and shape distortion on polyhedral-based recursive partitions of the sphere – february 1998. [tiešsaiste] Pieejams internetā: https://www.researchgate.net/publication/220649496_Comparing_Area_and_Shape_Distortion_on_Polyhedral-Based_Recursive_Partitions_of_the_Sphere

[5] Java Recursion Triangle with Deviation, komentāra autors ir lietotājs ar segvārdu "Jeewantha" – november 2012 [tiešsaiste] Pieejams internetā: https://stackoverflow.com/questions/13242050/java-recursion-triangle-with-deviation

[6] Li Meng, Xiaochong Tong, Shuaibo Fan, Chengqi Cheng, Bo Chen, Weiming Yang, Kaihua Hou – A Universal Generating Algorithm for the Polyhedral Discrete Grid Based on Unit Duplication – march 2019 [tiešsaiste] Pieejams internetā: https://www.mdpi.com/2220-9964/8/3/146/htm

[7] Joey de Vries – LearnOpenGL – june 2014 [tiešsaiste] Pieejams internetā: https://learnopengl.com/Getting-started/Transformations

[8] Webarhīvs – color chart – march 2018 [tiešsaiste] Pieejams internetā: https://web.archive.org/web/20180301041827/https://prideout.net/archive/colors.php

[9] OpenGL Super Bible [tiešsaiste] Pieejams internetā: http://opengl.czweb.org/ch03/040-043.html

[10] opengl-tutorial [tiešsaiste] Pieejams internetā: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-10-transparency/

[11] Khronos OpenGL dokumentācija [tiešsaiste] Pieejams internetā: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glEnable.xml

[12] Joey de Vries – LearnOpenGL – june 2014 [tiešsaiste] Pieejams internetā: https://learnopengl.com/Lighting/Lighting-maps

[13] A.M. Johansen - Monte Carlo Methods – 2010 [tiešsaiste] Pieejams internetā: https://www.sciencedirect.com/topics/medicine-and-dentistry/monte-carlo-method

[14] Rick Wicklin – Linear interpolation in SAS – may 2020 [tiešsaiste] Pieejams internetā: https://blogs.sas.com/content/iml/2020/05/04/linear-interpolation-sas.html

[15] Rick Wicklin – Cubic spline interpolation in SAS – may 2020 [tiešsaiste] Pieejams internetā: https://blogs.sas.com/content/iml/2020/05/11/cubic-interpolation-sas.html

[16] Melissa L. Enoch, Neal J. Evans II, Anneila I. Sargent, Jason Glenn, Erik Rosolowsky, Philip Myers – The Mass Distribution and Lifetime of Prestellar Cores in Perseus, Serpens, and Ophiuchus – 2008 – [tiešsaiste] Pieejams internetā: https://iopscience.iop.org/article/10.1086/589963

[17] S. Bochkanov, V. Bystritsky – Alglib net – [tiešsaiste] Pieejams internetā: https://www.alglib.net

[18] Lumenlearning – Newton's Law of Universal Gravitation – [tiešsaiste] Pieejams internetā: https://courses.lumenlearning.com/boundless-physics/chapter/newtons-law-of-universal-gravitation/

[19] Modeling Molecular Cores as Bonnor-Ebert Spheres and Isothermal Spheres [tiešsaiste] Pieejams internetā: http://astro1.physics.utoledo.edu/~megeath/ph6820/lecture6_ph6820.pdf

[20] Video klips – Three-Dimensional Core-Collapse Supernova – october 2012 [tiešsaiste]
Pieejams internetā: https://www.youtube.com/watch?v=oxGajNoPz8c

# PIELIKUMI

## 1. Pielikums. Punktu ģenerācijas kods.

```
//Iteraciju skaits
int triangleIterations;
triangleIterations = 10;

//Trissturu skaits
int triangles;
triangles = pow(4, triangleIterations);

//Punktu skaits uz linijas
int pointLineCount = (2 + pow(2, triangleIterations) - 1); //Oktahedrona vienas puses sakaribas formula 2+(2^x)-1 punktu skaitam.

//Kopejais punktu skaits
int totalPointCount = pointLineCount * (pointLineCount + 1) / 2;

//Punktu masivs
float *triangleCoordinates;
triangleCoordinates = new float[totalPointCount * 3];

float top[3] = {0.0f, 10.0f, 0.0f};
float left[3] = {10.0f ,0.0f, 0.0f};
float front[3] = {0.0f ,0.0f, 10.0f};

float t = 1.0 / (pointLineCount - 1);

int coordcount = 0;
```

```
for (int x = pointLineCount; x > 0; x--)
{
	//Tris gadijumi = linija ir bottom, linija ir top = nav jaizmanto t. linija ir middle - izmanto t.

	//Linija ir bottom
	if (x == pointLineCount)
	{
		for (int y = 0; y < pointLineCount; y++)
		{
			//first point
			if (y == 0)
			{
				triangleCoordinates[0] = left[0];
				triangleCoordinates[1] = left[1];
				triangleCoordinates[2] = left[2];
				coordcount += 3;
				//std::cout << "Test1" << std::endl;
			}

			//last point
			if (y == pointLineCount - 1)
			{
				triangleCoordinates[coordcount] = front[0];
				triangleCoordinates[coordcount+1] = front[1];
				triangleCoordinates[coordcount+2] = front[2];
				coordcount += 3;
				//std::cout << "Test2" << std::endl;
			}

			//middle point
```

```
                        if (y != 0 && y != pointLineCount - 1)
                        {
                                float cx = left[0] * (1 - t * y) + front[0] * t * y;
                                float cy = left[1] * (1 - t * y) + front[1] * t * y;
                                float cz = left[2] * (1 - t * y) + front[2] * t * y;
                                triangleCoordinates[coordcount] = cx;
                                triangleCoordinates[coordcount + 1] = cy;
                                triangleCoordinates[coordcount + 2] = cz;
                                coordcount += 3;
                                //std::cout << "Test3 " << std::endl;
                                //std::cout << "Test3 " << left[0] * (1 - t * y) << " " << y << " " <<
(1 - t * y) << " " << t << std::endl;
                                //std::cout << pointLineCount << " " << pointLineCount - 1 << " "
<< 1.0 / (pointLineCount - 1) << std::endl;
                        }
                }
        }

        //Linija ir top
        if (x == 1)
        {
                triangleCoordinates[coordcount] = top[0];
                triangleCoordinates[coordcount + 1] = top[1];
                triangleCoordinates[coordcount + 2] = top[2];
                coordcount += 3;
                //std::cout << "Test4" << std::endl;
        }

        //Linija ir middle
        if (x != 1 && x != pointLineCount)
        {
```

```
float lx = top[0] * (1 - t * (x - 1)) + left[0] * t * (x - 1);
float ly = top[1] * (1 - t * (x - 1)) + left[1] * t * (x - 1);
float lz = top[2] * (1 - t * (x - 1)) + left[2] * t * (x - 1);

//std::cout << top[0] * (1 - t * x) << " " << left[0] * t * (x - 1) << std::endl;
//std::cout << top[1] * (1 - t * (x-1)) << " " << left[1] * t* x << std::endl;

float fx = top[0] * (1 - t * (x - 1)) + front[0] * t * (x - 1);
float fy = top[1] * (1 - t * (x - 1)) + front[1] * t * (x - 1);
float fz = top[2] * (1 - t * (x - 1)) + front[2] * t * (x - 1);

for (int y = 0; y < x; y++)
{
	//first point
	if (y == 0)
	{
		triangleCoordinates[coordcount] = lx;
		triangleCoordinates[coordcount+1] = ly;
		triangleCoordinates[coordcount+2] = lz;
		coordcount += 3;
		//std::cout << "Test5" << std::endl;
	}

	//last point
	if (y == x - 1)
	{
		triangleCoordinates[coordcount] = fx;
		triangleCoordinates[coordcount + 1] = fy;
		triangleCoordinates[coordcount + 2] = fz;
		coordcount += 3;
```

```
                    //std::cout << "Test6" << std::endl;
                }

                //std::cout << "The Y value for newt " << y << " or x " << x << std::endl;
                float nt = 1.0 / (x - 1);

                //middle point
                if (y != 0 && y != x - 1)
                {
                    float cx = lx * (1 - nt * y) + fx * nt * y;
                    float cy = ly * (1 - nt * y) + fy * nt * y;
                    float cz = lz * (1 - nt * y) + fz * nt * y;
                    triangleCoordinates[coordcount] = cx;
                    triangleCoordinates[coordcount + 1] = cy;
                    triangleCoordinates[coordcount + 2] = cz;
                    coordcount += 3;
                    //std::cout << "Test7" << std::endl;
                }
            }
        }

}


int *triangleIndices;
triangleIndices = new int[triangles * 3];


int indexoffset = pointLineCount;
```

```
int indeks = 0;
int topline = pointLineCount; //start from 0
int bottomline = 0; //start from 0
//Indeksi.
for (int x = pointLineCount-1; x > 0; x--)
{
	triangleIndices[indeks] = bottomline;
	triangleIndices[indeks + 1] = bottomline+1;
	triangleIndices[indeks + 2] = topline;
	indeks += 3;

	bottomline += 1;
	//std::cout << "Test1 " << x << std::endl;

	if (x > 1)
	{
		for (int y = 0; y < x-1; y++)
		{
			triangleIndices[indeks] = bottomline;
			triangleIndices[indeks + 1] = topline;
			triangleIndices[indeks + 2] = topline +1;

			triangleIndices[indeks + 3] = bottomline;
			triangleIndices[indeks + 4] = topline + 1;
			triangleIndices[indeks + 5] = bottomline + 1;
			indeks += 6;

			bottomline += 1;
			topline += 1;
			//std::cout << "Test2 " << y << std::endl;
```

```
                }
                bottomline += 1;
                topline += 1;
            }
        }
```

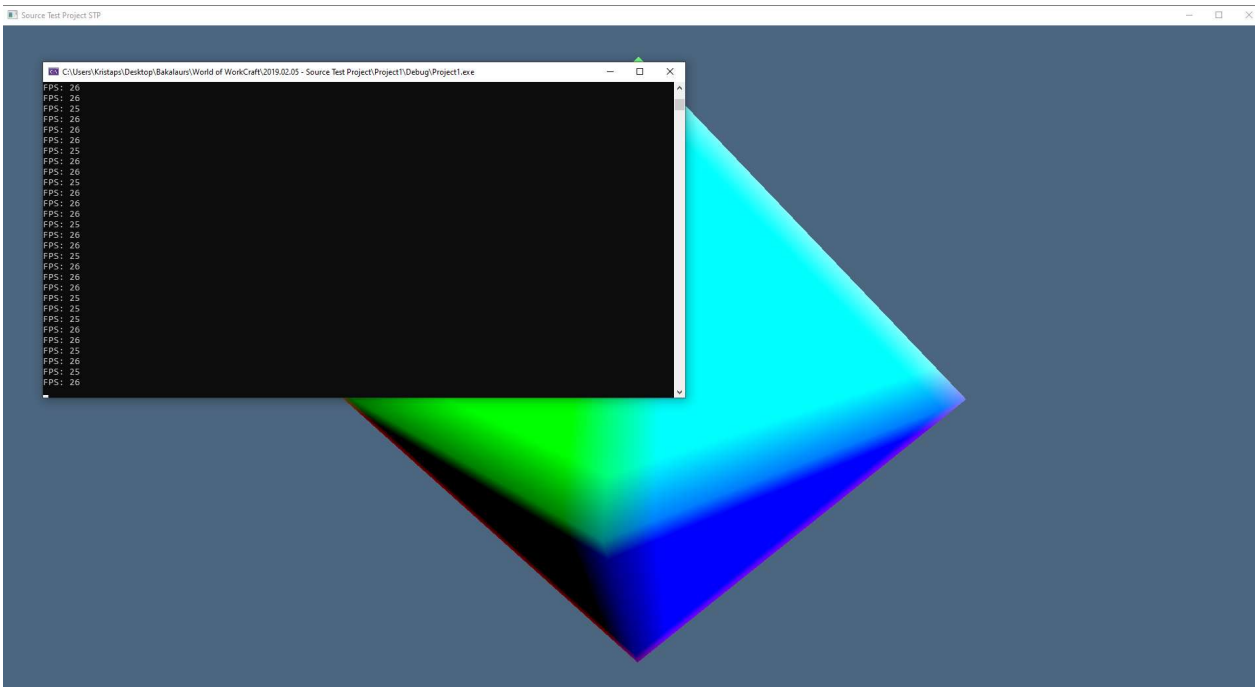
## 2. Pielikums ģenerācijas funkcijas piesaukumi un buferu objektu pavairošana

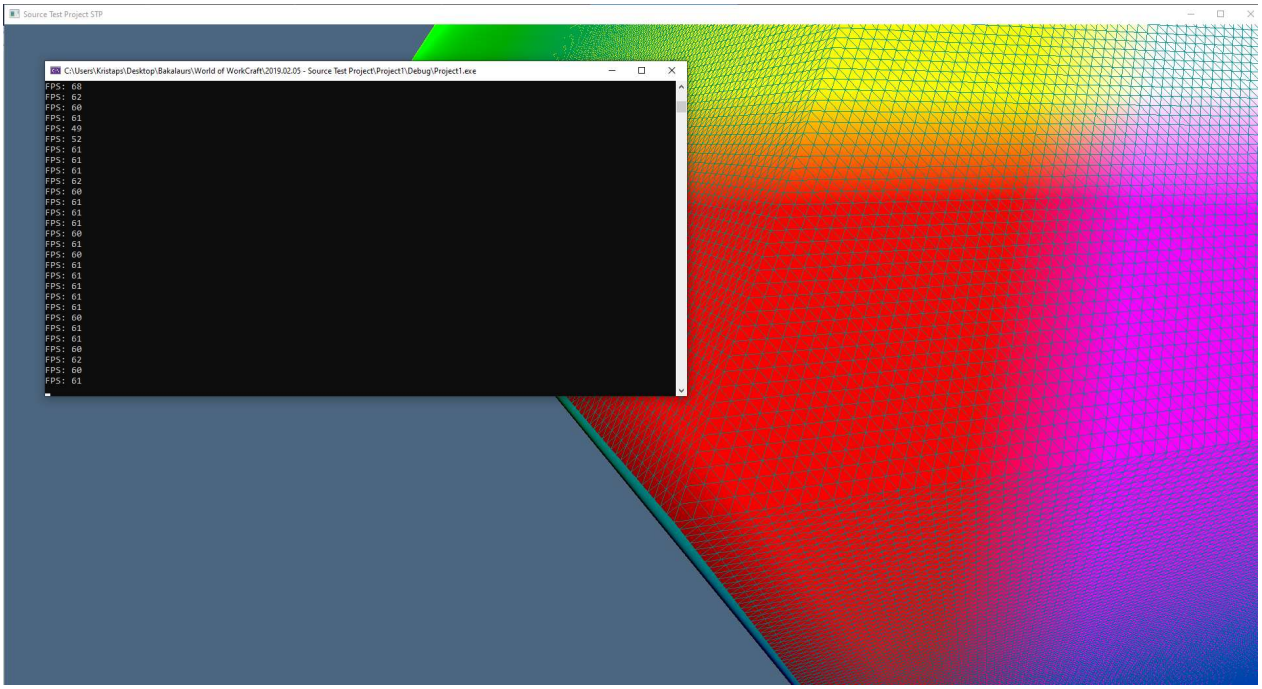
```
float top[3] = {0.0f, raddist, 0.0f};
float left[3] = { raddist ,0.0f, 0.0f};
float front[3] = {0.0f ,0.0f, raddist };

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = 0.0f; left[1] = 0.0f; left[2] = raddist;
front[0] = -raddist; front[1] = 0.0f; front[2] = 0.0f;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = -raddist; left[1] = 0.0f; left[2] = 0.0f;
front[0] = 0.0f; front[1] = 0.0f; front[2] = -raddist;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = 0.0f; left[1] = 0.0f; left[2] = -raddist;
front[0] = raddist; front[1] = 0.0f; front[2] = 0.0f;
```

```
triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);



//bottom 4.
coordcount += totalPointCount * 3;
top[0] = 0.0f; top[1] = -raddist; top[2] = 0.0f;
left[0] = raddist; left[1] = 0.0f; left[2] = 0.0f;
front[0] = 0.0f; front[1] = 0.0f; front[2] = raddist;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = 0.0f; left[1] = 0.0f; left[2] = raddist;
front[0] = -raddist; front[1] = 0.0f; front[2] = 0.0f;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = -raddist; left[1] = 0.0f; left[2] = 0.0f;
front[0] = 0.0f; front[1] = 0.0f; front[2] = -raddist;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);

coordcount += totalPointCount * 3;
left[0] = 0.0f; left[1] = 0.0f; left[2] = -raddist;
front[0] = raddist; front[1] = 0.0f; front[2] = 0.0f;

triangleCoords(triangleIterations, top, left, front, triangleCoordinates, coordcount);
coordcount += totalPointCount * 3;
```

```
//Just a casual duplicate
for (int x = 0; x < triangles * 3; x++)
{
	for (int y = 0; y < 7; y++)
	{
		triangleIndices[triangles * 3 * (y+1) + x] = triangleIndices[x] + totalPointCount *
(y+1);
		//triangleIndices[triangles * 3 * 2 + x] = triangleIndices[x] + totalPointCount * 2;
		//triangleIndices[triangles * 3 * 3 + x] = triangleIndices[x] + totalPointCount * 3;
	}
}
```

## 3. Pielikums oktahedrona trīsstūru iterāciju testēšana






## 4. Pielikums normalizācijas funkcija

```
void exradius(float radius, float* triangleCoordinates, int coordcount)
{
	float center = 0.0f;
	for (int x = 0; x < coordcount; x += 3)
	{
		float dx = triangleCoordinates[x];
		float dy = triangleCoordinates[x + 1];
		float dz = triangleCoordinates[x + 2];

		float d = sqrt(pow(dx, 2) + pow(dy, 2) + pow(dz, 2));
```

```
                dx = dx * radius / d;

                dy = dy * radius / d;

                dz = dz * radius / d;


                triangleCoordinates[x] = triangleCoordinates[x] + dx;

                triangleCoordinates[x+1] = triangleCoordinates[x+1] + dy;

                triangleCoordinates[x + 2] = triangleCoordinates[x + 2] + dz;

        }

}
```

Bakalaura darbs „OPENGL KODOLA KOLAPSA VIZUALIZĀCIJA” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti.

Autors: Kristaps Veitners 31.05.2021.

Rekomendēju darbu aizstāvēšanai

Vadītājs: profesors Dr. dat. Jānis Zuters 31.05.2021.

Recenzents: profesors Dr. dat. Leo Seļāvo

Darbs iesniegts Datorikas fakultātē 31.05.2021.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

__.06.2021. prot. Nr. ___.

Komisijas sekretārs: lektors Ivo Odītis