Sami Moisio

2D-grafiikkamoottori

Opinnäytetyö

Tietotekniikka / Peliohjelmointi

Toukokuu 2016

KYAMK
University of Applied Sciences

KYAMK
University of Applied Sciences

Tekijä/Tekijät	Tutkinto	Aika
Sami Moisio	Tietotekniikan insinööri	Toukokuu 2016
Opinnäytetyön nimi		31 sivua
2D-grafiikkamoottori		
Toimeksiantaja		
Kyamk		
Ohjaaja		
Yliopettaja Paula Posio		

Tiivistelmä

Tässä opinnäytetyössä käsitellään kaksiulotteisen grafiikkamoottorin rakenteiden ja toiminnan suunnittelua ja toteutusta. Työssä selvennetään syitä oman grafiikkamoottorin tekoon ja esitellään moottorien yleisiä ominaisuuksia. Toteutuksen osalta työssä keskitytään vahvasti grafiikkamoottorin eri ominaisuuksien toimintaan ja suorituskykyä parantaviin tekniikoihin OpenGL-ympäristössä. Lopuksi esitellään demonstraatio-ohjelman toimintaa ja analysoidaan sillä mitattuja suorituskykyyn liittyviä tuloksia.

Grafiikkamoottori kirjoitettiin C-kielellä ja sen päälle kasattu demonstraatio-ohjelma kirjoitettiin C++-kielellä. Moottorin alustariippumattomuus saavutettiin käyttämällä ainoastaan alustariippumattomia rajapintoja ja kirjastoja. Moottorilla pystytään tehokkaasti esittämään kaksiulotteista grafiikkaa ohjelmoijan haluamalla tavalla. Moottorin toiminta on suunniteltu siten, että vältyttäisiin mahdollisimman monelta suorituskykyongelmalta monimutkaistamatta moottorin käyttöä.

Demonstraatio-ohjelman suorituskykymittauksista selviää, että käytetyt tekniikat parantavat moottorin suorituskykyä selvästi. Moottoriin saatiin toteutettua suunnitellut ominaisuudet. Moottorin tehokkuus ja selkeys vastasivat myös vaatimuksia.

Asiasanat

2D, alustariippumattomuus, C, C++, grafiikkamoottori, OpenGL

KYAMK
University of Applied Sciences

Author (authors)	Degree	Time
Sami Moisio	Bachelor of Information Technology	May 2016

Thesis Title	
2D Rendering Engine	31 pages

Commissioned by

Kyuas

Supervisor

Paula Posio, Principal Lecturer

Abstract

The objective of this thesis was to design and implement a 2D graphics engine that is capable of efficient graphics rendering. The thesis covers reasons for making a graphics engine and introduces basic graphics engine features. The implementation section of the thesis has a strong focus on explaining the OpenGL-related features and performance optimizations that have been implemented into the graphics engine. Towards the end, the focus of the thesis moves into the operation of demonstration program and analyzing benchmark results measured with the program.

Graphics engine was written in C language and the demonstration program was built on top the engine with C++ language. Cross-platform capabilities of the engine were achieved by using only cross-platform libraries and APIs. The main objective of the engine is to present 2D graphics efficiently and in the way that the programmer desires. The engines operation is designed to avoid common performance reducing pitfalls without compromising the ease of using the engine.

Demonstration program benchmarks show that the performance improving techniques succeeded in speeding up the engine. All the goals set for the engine were met with ease.

Keywords

2D, C, C++, Cross-platform, OpenGL, Rendering Engine,

SISÄLLYS

1 JOHDANTO 6
2 REAALIAIKAISET GRAFIIKKAMOOTTORIT 6
2.1 Pelimoottorit 6
2.2 Kaksiulotteisuus 7
3 ALUSTARIIPPUMATTOMAT GRAFIIKKARAJAPINNAT 7
3.1 OpenGL 7
3.1.1 Toiminta 8
3.1.2 Vahvuudet 9
3.1.3 Heikkoudet 10
3.1.4 OpenGL ES 11
3.1.5 GLSL 11
3.2 Vulkan 11
4 TOTEUTUS 11
4.1 Tavoitteet 11
4.2 Suunnittelu 12
4.3 Tekstuurit 13
4.4 Varjostinohjelmat 14
4.5 Kamerat 16
4.6 Spriten piirto 17
4.7 Staattinen ja dynaaminen data 20
4.8 Matematiikka 20
4.8.1 Vektorit 20
4.8.2 Matriisit 21
4.9 Suorituskykyä parantavat tekniikat 21
4.9.1 Datan monipuskurointi 22
4.9.2 SIMD-käskykantalaajennokset 23
4.9.3 Data-orientoitunut suunnittelu 26
4.10 Demonstraatio-ohjelma 26

4.10.1 Esittely 26
4.10.2 Suorituskykymittaukset 27
5 PÄÄTELMÄT JA JATKOKEHITYS 29
LÄHTEET 31

1 JOHDANTO

Tässä opinnäytetyössä esitellään kaksiulotteisen grafiikkamoottorin toteutusta. Keskeisimmät moottorissa käytetyt teknologiat ovat OpenGL-rajapinta (Open Graphics Library) ja SDL-kirjasto (Simple DirectMedia Layer). Työssä perehdytään OpenGL-rajapinnan toimintaan ja sen käyttöön grafiikkamoottorissa. Työssä analysoidaan moottorin suorituskykyä ja käydään läpi tekniikoita, jotka vaikuttavat suorituskykyyn.

Motiiveita grafiikkamoottorin tekoon olivat osaamisen vahvistaminen ja grafiikkamoottorin hyödyntäminen tulevaisuuden projekteissa. Oman grafiikkamoottorin voi räätälöidä oman graafisen tyylin tehokkaaseen esittämiseen, kun taas valmiit grafiikkamoottorit pyrkivät olemaan mahdollisimman yleiskäyttöisiä ja usein keskittyvät realistisen kolmiulotteisen grafiikan esittämiseen. Työssä käsitelty moottori on suunniteltu esittämään suuria määriä kaksiulotteisia grafiikkaobjekteja.

2 REAALIAIKAISET GRAFIIKKAMOOTTORIT

Grafiikkamoottorilla tarkoitetaan korkean tason koodin abstraktiotasoa, joka mahdollistaa grafiikan esittämisen. Reaaliaikaiset grafiikkamoottorit usein kätkevät alleen yhden tai useamman grafiikkarajapinnan, jonka käyttö sellaisenaan olisi hyvin työlästä tai haastavaa. Tarkoitus on siis helpottaa loppukehittäjää grafiikan esittämiseen liittyvissä asioissa. Moottorin yleisiä ominaisuuksia ovat muistinhallinta, tekstuurien ja materiaalien hallinta, matemaattiset funktiot, grafiikkadatan generointi ja piirto.

2.1 Pelimoottorit

Pelimoottorit sisältävät lähes poikkeuksetta jonkin tasoisen grafiikkamoottorin, jolla pelimaailmaa piirretään. Yleisimmin käytetyt pelimoottorit kuten Unity, Unreal Engine ja CryENGINE ovat vahvasti orientoituneita kolmiulotteisen grafiikan esittämiseen. Kyseiset moottorit ovat peliteollisuudessa hyvin suosittuja, mutta eivät välttämättä tarjoa kovin kattavia työkaluja kaksiulotteisen pelimaailman luontiin (Game engines used by the video game developers in the United Kingdom 2014). Jos valmiiden grafiikkamoottorien ominaisuudet tai suorituskyky eivät vastaa asetettuja vaatimuksia, voidaan harkita oman grafiikkamoottorin kirjoittamista.

2.2 Kaksiulotteisuus

Peleissä käytetyissä grafiikkamoottoreissa kaksiulotteinen grafiikka on usein esillä pelin käyttöliittymän elementeissä tai itse pelimaailmassa. Kolmiulotteisen pelimaailman esittäminen voi laskennallisesti olla raskaampaa kuin kaksiulotteisen maailman. Kolmiulotteiselle grafiikalle ominaiset asiat, kuten realistiset varjostimet ja yksityiskohtaiset 3D-mallit, voivat helposti osoittautua liian raskaiksi esimerkiksi mobiililaitteille ja vanhemmille tietokoneille. Kaksiulotteinen grafiikka ei yleensä vaadi yhtä paljon laskentatehoa ja muistia. Myös grafiikan tuotantoprosessista jää 3D-mallinnus ja -animaatio pois. Pienine vaatimuksineen 2D-pelit soveltuvat hyvin nettisivuille ja mobiililaitteille.

3 ALUSTARIIPPUMATTOMAT GRAFIIKKARAJAPINNAT

Ohjelmien ja pelien graafiset vaatimukset ovat vuosien saatossa kasvaneet siihen pisteeseen, että on kehitetty erillisiä grafiikkapiirejä, joiden ainoa tehtävä on avustaa grafiikan piirtämistä. Moderneissa tietokoneissa grafiikkapiiri löytyy näytönohjaimelta, joka on taas yksinkertaistetusti vastuussa piirtämisen orkestroinnista ja kuvan siirtämisestä näytölle. Näytönohjaimien ajureissa on ohjelmointirajapinnat, mutta ne ovat liian laiteorientoituneita siten, että ohjelmoijan olisi järkevää käyttää suoraan ajurien ohjelmointirajapintoja. Niinpä lopputuotteen ja ajureiden välille on kehitetty niin sanottu grafiikkarajapinta. Grafiikkarajapinnan tarkoitus on tarjota ohjelmoijalle abstrakteja rakenteita ja funktioita, joiden avulla näytönohjaimen resurssit voidaan valjastaa grafiikan piirtoon. (Sellers, Richard & Haemel 2014, 4.)

3.1 OpenGL

OpenGL on Khronos Group konsortion ylläpitämä ja kehittämä grafiikkarajapinta. Konsortiossa on yli 100 jäsentä, joista monet ovat hyvin tunnettuja ja isoja yhtiötä kuten Adobe, Microsoft, Nintendo ja Pixar. (Khronos Contributing Members 2016.)

OpenGL-rajapinnan abstraktiotaso on pyritty luomaan niin, että se ei olisi liian matalalla tasolla, jolloin ohjelmoija saattaisi joutua perehtymään laitekohtaisiin eroihin. Liian korkeaa abstraktiota on myös vältelty, koska se saattaisi rajoittaa

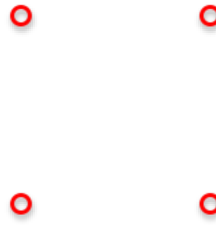
ohjelmoijaa edistyneempien ominaisuuksien ja tekniikoiden käytössä. (Sellers, Richard & Haemel 2014, 4.)

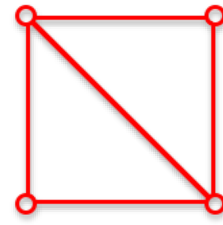
3.1.1 Toiminta

OpenGL-rajapinta koostuu monista abstrakteista operaatioista ja rakenteista, joita ohjelmoija pystyy tapauskohtaisesti manipuloimaan. Abstraktiotaso on sidottu vahvasti näytönohjaimen piirtoputken toimintaan. Piirtoputki koostuu monista grafiikkadatan käsittelyvaiheista, joista osa on ohjelmoitavissa ja osa taas suljettuja. Renderöintiin eli kuvan muodostamiseen on useita tekniikoita, mutta reaaliaikaiseen renderöintiin suosituin vaihtoehto on rasterointi, johon OpenGL-rajanpinnan renderöintikin perustuu. (Sellers ym. 2014, 10.) Rasterointi on tekniikkana huomattavasti nopeampi kuin esimerkiksi säteiden jäljittämiseen perustuvat tekniikat, joilla pyritään fotorealistisempaan lopputulokseen (Sherrod 2008, 9).

Kuvassa 1 on esitetty piirtoputken toimintaa yksinkertaistetusti. Ensimmäisessä vaiheessa noudetaan piirrettäväksi yksi tai useampi verteksi eli tietorakenne, joka sisältää pisteen informaatiot, kuten sijainnin, värin ja uv-koordinaatin. Toisessa vaiheessa ohjelmoijan luoma verteksivarjostin (engl. vertex shader) laskee pisteiden lopulliset sijainnit. Kolmannessa vaiheessa pisteet tulkitaan muodoksi ohjelmoijan määrittämällä tavalla. Neljännessä vaiheessa muoto rasteroidaan eli kartoitetaan muodon peittämä alue. Viidennessä vaiheessa kartoitettu alue väritetään ohjelmoijan luomalla pikselivarjostimella (engl. fragment shader). Viimeisessä vaiheessa testataan pikselikohtaisesti, onko pikseliä tarve värittää ruutupuskuriin. Ensimmäisessä testissä selvitetään, onko pikseli ohjelmoijan määrittämän sapluunan sisällä, jos sellainen on määritelty. Seuraavaksi pikselin syvyyttä testataan vertaamalla sitä syvyyspuskuriin, koska jokin muu muoto saattaa peittää pikselin. Lopuksi testatut pikselit väritetään ruutupuskuriin tai sekoitetaan ruutupuskurin pikselien värien kanssa. (Sellers ym. 2014, 28–46.)

```
Vertex verts[4] =
{
  {0.0f, 0.0f, 0.0f, 0.0f},
  {1.0f, 0.0f, 1.0f, 0.0f},
  {1.0f, 1.0f, 1.0f, 1.0f},
  {0.0f, 1.0f, 0.0f, 1.0f}
};
```

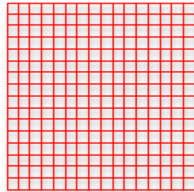


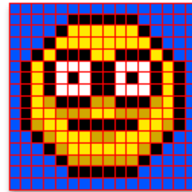


1. Verteksien nouto

2. Verteksivarjostin

3. Muodon kasaus





4. Rasterointi

5. Pikselivarjostin

6. Testit ja sekoitus

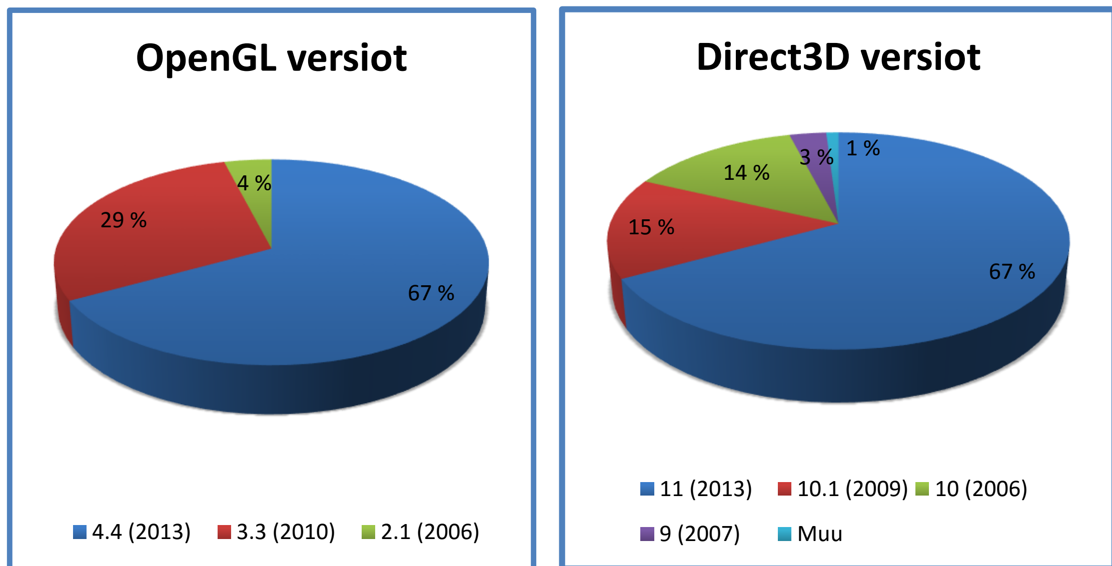
Kuva 1. Yksinkertaistettu piirtoputki

Moderneissa OpenGL-versioissa on myös valinnaisia vaiheita, kuten tesselaatio- ja geometriavarjostimet, joilla voidaan lisätä ja manipuloida verteksidataa ennen muodon kasausta (Sellers ym. 2014, 32–36). Osa OpenGL-rajapinnan ominaisuuksista on suunniteltu 3D-ympäristöä silmällä pitäen. Aikaisemmin mainituista valinnaisista varjostimista ja syvyystestauksesta on vaikea löytää hyötyä 2D-ympäristössä.

3.1.2 Vahvuudet

OpenGL-rajapinta on alustariippumaton grafiikkarajapinta. Alustariippumattomuus mahdollistaa saman lähdekoodin kääntämisen usealle eri käyttöjärjestelmälle (Shreiner ym. 2013, 835.) Alustariippumattomuuden ansiosta ohjelmoijan ei siis tarvitse kirjoittaa ollenkaan alustakohtaista koodia.

OpenGL-tuki löytyy lähes kaikilta moderneilta käyttöjärjestelmiltä. Kuten kuvasta 2 nähdään, Microsoftin kehittämän Direct3D-rajapinnan versioiden tuki vastaa OpenGL-rajapinnan versioiden tukea.



Kuva 2. Rajapintojen versio-osuudet (Steam Hardware Survey 2013)

Hyvä esimerkki rajapintatuesta on Kiina, jossa kuluttajilla on ollut uudet tietokoneet, mutta niihin asennettuina iäkäs Windows XP-käyttöjärjestelmä. Windows XP:n DirectX-rajapinnan päivitykset loppuivat vuonna 2007 versioon 9.0c (Mitchell ym. 2014, 10–12). Koska OpenGL on vapaa standardi niin laitevalmistajat voivat halutessaan luoda vanhoillekin käyttöjärjestelmille ajurit ja paremman OpenGL-tuen. Eli OpenGL-rajapinnalla voidaan Kiinan tapauksessa käyttää laitteiden uusimpia ominaisuuksia, samaa ei voida sanoa DirectX-rajapinnan osalta.

3.1.3 Heikkoudet

Laitevalmistajat saattavat implementoida OpenGL:n toimintoja eri tavoin aiheuttaen suorituskykyeroja. Yleensä suurimmat erot ilmenevät varjostinohjelmien kääntämisessä, koska jokaisen eri laitteen ajuri sisältää oman kääntäjän. Sama varjostinohjelma ei siis välttämättä toimi kaikilla laitteilla. Varjostinohjelmien kanssa tulee siis olla tarkkana ja tarkistaa varjostinohjelmat tarkistustyökalulla kuten glslang. (Mitchell ym. 2014, 30–35.)

OpenGL-rajapinnassa vianetsintä voi välillä osoittautua hyvinkin hankalaksi, koska rajapinta antaa hyvin niukasti informaatiota tapahtuneista virheistä ja väärinkäytöksistä.

3.1.4 OpenGL ES

OpenGL Embedded Systems on sulautetuille järjestelmille räätälöity versio OpenGL-rajapinnasta. Mobiilikäyttöjärjestelmät kuten iOS ja Android tukeutuvat vahvasti OpenGL ES:n käyttöön. (Sellers ym. 2014, 705.)

3.1.5 GLSL

OpenGL Shading Language on OpenGL-rajapinnan käyttämä varjostinkieli, joka muistuttaa syntaksiltaan C-kieltä. Kieli sisältää vakiona matemaattiset perustyypit kuten matriisit ja vektorit ja niiden eri operaatiot. (Sellers ym. 2014, 17.)

3.2 Vulkan

Vulkan-rajapinta on uunituore Khronos Group konsortion kehittämä grafiikka-rajapinta, jonka olisi tarkoitus tarjota OpenGL-rajapintaan verrattuna pääsy näytönohjaimen huomattavasti matalamman tason toimintoihin. Vulkanissa ei ole OpenGL-tyylistä kontekstia, johon objektit olisivat sidottuina. Kontekstivapaa ja kevyt rakenne mahdollistavat tehokkaan rinnakkaislaskennan ja komentojen syötön. Vulkanissa ohjelmoija on vastuussa myös muistinhallinnasta, tekstuurien käsittelystä ja koko piirtoputken alustamisesta. (Olson ym. 2016, 13–14.)

Vulkan on tarkoitettu projekteihin, joissa on aikaa ja resursseja kehittää kunnollista grafiikkamoottoria jykevälle pohjalle. Vulkan-rajapintaa on mahdollista käyttää näytönohjainpainotteiseen peruslaskentaankin, jos sovelluksella ei ole graafisia tarpeita.

4 TOTEUTUS

4.1 Tavoitteet

Grafiikkamoottorin toteutusta pitäisi tehdä yksinkertaisuutta silmällä pitäen. Käytön ja toiminnan yksinkertaisuus on kuitenkin haaste, kun alla oleva rajapinta ei ole yksinkertainen. Rajapintaakaan ei pitäisi piilottaa siltä varalta, että ohjelmoija osaa sitä käyttää. OpenGL-rajapinnan alustariippumattomuuskin olisi hyvä säilyttää välttämällä alustakohtaisten kirjastojen ja funktioiden käyttöä.

Suorituskyky vaatimuksena moottorin pitäisi kyetä tehokkaasti piirtämään satojatuhansia kaksiulotteisia bittikarttoja (sprite). Aikaisempien ominaisuuksien lisäksi moottorissa olisi hyvä olla seuraavat ominaisuudet:

- sisäänrakennetut yleiskäyttöiset varjostinohjelmat
- pieni matematiikkakirjasto
- kameroiden hallinta
- tekstuurien lataus
- varjostinohjelmien lataus
- alustariippumattomuus

4.2 Suunnittelu

Ohjelmointiprojekti rakennettiin Visual Studio ohjelmankehitysympäristössä, koska se on äärimmäisen suosittu ja hyväksi todettu ohjelma. Ohjelmointikielenä käytettiin moottorin osalta C-kieltä ja demo-ohjelmassa C++-kieltä. C-kieli mahdollistaa yksinkertaisuudellaan helpomman linkityksen muihin kieliin, jos sellaista joskus moottorilta tarvittaisiin. C++-kieli sen sijaan tarjoaa todella kattavat toiminnot ja tuen C-kielelle. Grafiikkamoottorille annettiin nimeksi Stercus Graphics Library eli lyhennettynä SGL.

Moottorissa päätettiin käyttää SDL-kirjastoa ja sen lisäosia alustariippumattomaan ikkunointiin, kuvatiedostojen lataamiseen ja syöttölaitteiden lukemiseen. OpenGL-laajennosten hallintaan käytettiin GLEW-kirjastoa. Demossa käyttöliittymän esittämiseen käytettiin AntTweakBar-kirjastoa, jonka avulla voidaan luoda yksinkertainen ikkuna muokattavine muuttujineen.

Versiohallintaan käytettiin Git-versionhallintaohjelmistoa ja projektia ylläpidetään GitHub verkkosivuston ilmaisilla palvelimilla. Git-versiohallintaa käytettiin projektissa, koska se on hyvin integroitu Visual Studio 2015 ohjelmaan. SGL-kirjaston lähdekoodi löytyy avoimena lähdekoodina GitHub verkkosivustolta. (Stercus engine repository, 2016).

Projektissa käytetty koodaustyyli on yhtenäinen SDL-kirjaston lähdekoodin tyylin kanssa. Eli koodin muuttujien, rakenteiden, luokkien ja funktioiden nimeämissä käytetään variaatioita camelCase-tyylistä, jossa ensimmäistä sanaa lukuun ottamatta jokainen sana kirjoitetaan isolla alkukirjaimella ilman välejä.

4.3 Tekstuurit

Grafiikkamoottoreissa värikkään ja mielenkiintoisen ympäristön esittämiseen tarvitaan tekstuureita eli bittikarttakuvia. Tekstuuria voidaan käyttää esimerkiksi rasteroitujen kuvioiden värittämiseen pikselivarjostimessa.

SDL_image-kirjaston (SDL-kirjaston laajennos) avulla pystytään lataamaan kuvatiedostoja SDL-kirjaston käyttämään SDL_Surface rakenteeseen, joka sisältää kuvan pikselidatan ja siihen liittyvää tietoa. SDL_Surface rakenteesta voidaan poimia OpenGL:lle olennainen data eli kuvan leveys, korkeus, pikseliformaatti ja pikselidata. SGL-kirjastossa tekstuurit ladataan näytönohjaimen muistiin yhdellä funktiolla. (kuva 4)

```
void SGL_CreateTextures(SGL_Tex2D* textures, SGL_TexParams* params, U32 count)
{
U32* texHandles = SDL_malloc(sizeof(U32)*count);
glGenTextures(count, texHandles);
for (U32 i = 0; i < count; i++)
{
   textures[i].handle = texHandles[i];
   glBindTexture(GL_TEXTURE_2D, texHandles[i]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, params[i].wrapModeS);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, params[i].wrapModeT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, params[i].minFilter);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, params[i].magFilter);
   if (params[i].wrapModeS == GL_CLAMP_TO_BORDER ||
   params[i].wrapModeT == GL_CLAMP_TO_BORDER)
   {
      glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
      params[i].borderColor);
   }
   glTexImage2D(GL_TEXTURE_2D, 0, textures[i].format, textures[i].width,
   textures[i].height,0, textures[i].type, GL_UNSIGNED_BYTE, textures[i].data);
   SDL_free(textures[i].data);
   glGenerateMipmap(GL_TEXTURE_2D);
   glBindTexture(GL_TEXTURE_2D, 0);
}
SDL_free(texHandles);
}
```

Kuva 4. Tekstuurien latausfunktio (SGL-kirjaston lähdekoodi 2016)

Funktio SGL_CreateTextures (kuva 4) ottaa parametrina osoittimen tekstuureihin, osoittimen tekstuurikohtaisiin parametreihin ja tekstuurien lukumäärän. Funktio luo annetun määrän OpenGL tekstuureja, asettaa niille ohjelmoijan määrittämät tekstuuriparametrit ja siirtää annettujen tekstuurien pikselidatan näytönohjaimelle. Funktio ei palauta mitään, mutta se tallettaa luotujen tekstuurien tunnisteet parametrina saatuihin tekstuureihin.

4.4 Varjostinohjelmat

SGL-kirjaston sisäänrakennetut varjostinohjelmat koostuvat varjostinohjelman toiminnan kannalta pakollisista verteksi- ja pikselivarjostimista. SGL-kirjasto sisältää 5 sisäänrakennettua varjostinparia, jotka käyttävät GLSL-varjostinkielen versiota 3.3.

Jokainen verteksivarjostin sisältää kameramatriisin, joka on määritetty uniform-puskuriobjektin (engl. uniform buffer object) sisälle. Uniform-tyyppiset muuttujat eroavat muista muuttujista niin, että niiden sisältämä data pysyy samana piirrävän objektin vaihtuessa (Sellers ym. 2014, 103). Normaaliin uniform-muuttujaan verrattuna uniform-puskuriobjektin etuna on se, että sitä ei tarvitse päivittää varjostinkohtaisesti, vaan sen voi jakaa varjostimien kesken. (Sellers ym. 2014, 108). Eli kameramatriisin päivitys vaikuttaa kaikkiin varjostimiin automaattisesti.

Kuvassa 5 on esimerkki yhdestä SGL-kirjaston sisältämästä verteksivarjostimesta. Varjostimessa vakiopuskuriobjektin nimeksi on annettu globalMatrices ja se sisältää 4x4-matriisin, johon on tarkoitus tallettaa kameramatriisi.

```
#version 330 core
layout (std140) uniform globalMatrices
{
   mat4 vPMatrix;
};
layout(location = 0) in vec2 vPosition;
layout(location = 2) in vec2 vTexCoord;
out vec2 texCoord;
void main()
{
   texCoord = vTexCoord;
   gl_Position = vPMatrix * vec4(vPosition, 1.0, 1.0);
}
```

Kuva 5. Verteksivarjostin (SGL-kirjaston lähdekoodi 2016)

Verteksin sisältämät uv-koordinaatit siirretään sellaisenaan pikselivarjostimelle ja kameramatriisi kerrotaan verteksin sisältämällä sijainnilla. Koska sijainti on kaksiulotteinen, niin sijainti muunnetaan neliulotteiseksi vektoriksi, jossa z- ja w-komponentit saavat arvon 1. Vektorin täytyy olla neliulotteinen, jotta sitä voidaan käyttää matriisikertolaskussa.

Kuvassa 6 on esimerkki pikselivarjostimesta, joka on linkitettävissä kuvan 5 esittämään varjostimeen. Pikselivarjostimessa uv-koordinaatit esittävät tiettyä

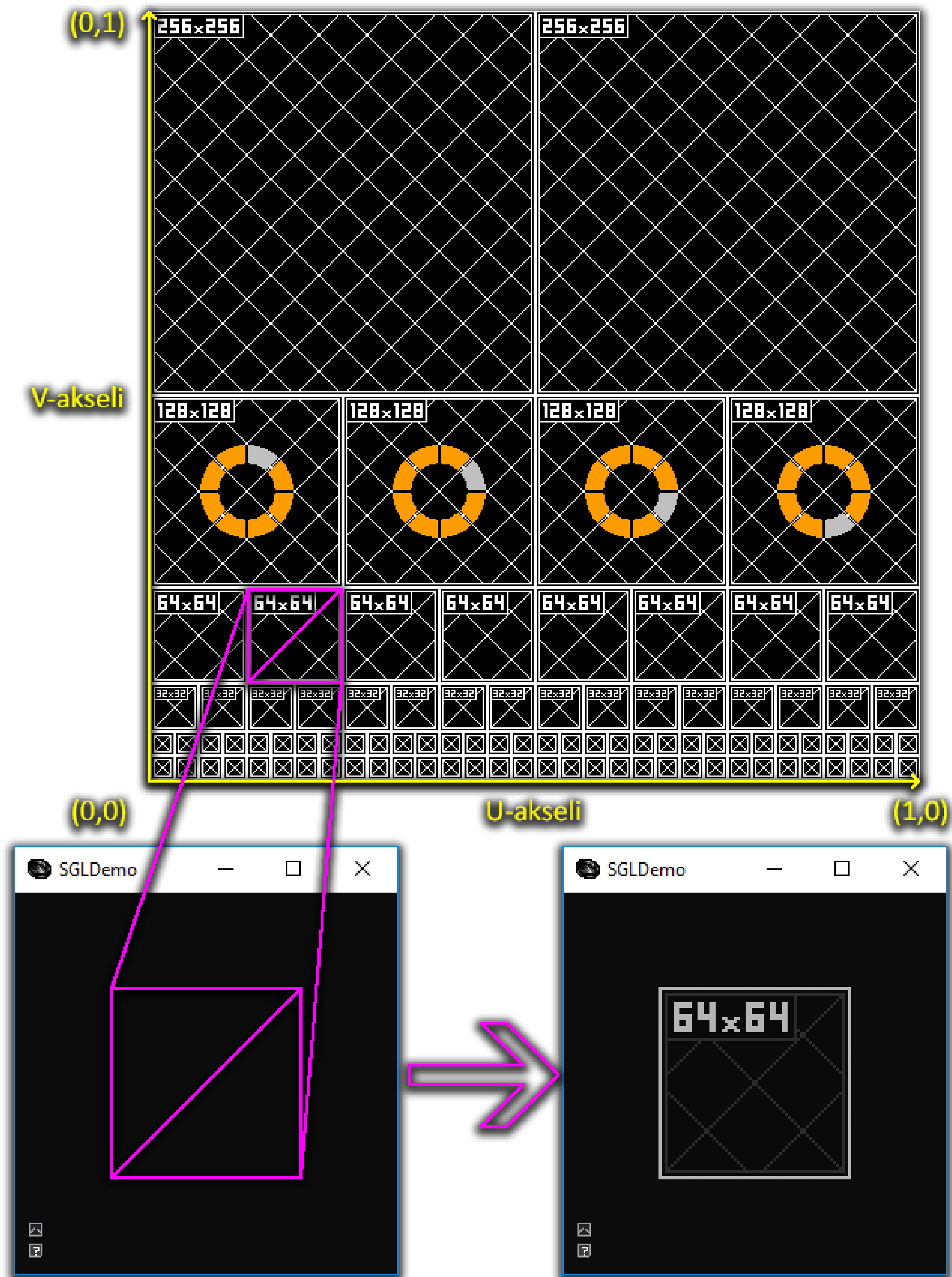
aluetta tekstuurista. Alueelta poimituilla väreillä väritetään rasteroinnissa määritelty alue.

```
#version 330 core
uniform sampler2D tex;
in vec2 texCoord;
out vec4 fColor;
void main()
{
   fColor = texture(tex, texCoord);
}
```

Kuva 6. Pikselivarjostin (SGL-kirjaston lähdekoodi)

Pikselivarjostimessa käytetty ”texture”-funktio ottaa parametrina tekstuurisamplerin ja koordinaattipisteen. Funktio palauttaa tekstuurin värin kyseisen koordinaatin kohdalla.

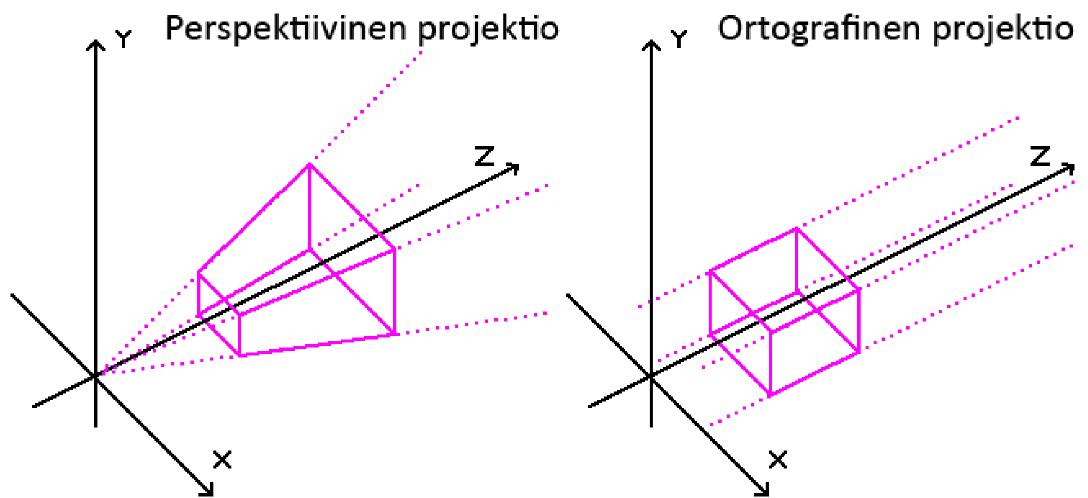
Kuvasta 7 nähdään, miten koordinaatit tulkitaan tekstuurista rasteroidun muodon alueelle. Tekstuuri on aina levitettynä koordinaatiston molempien akselien 0 – 1-alueelle. Kun mennään koordinaatistossa yli tekstuurin peittämän alueen, niin ”texture”-funktio palauttaa värin tekstuuriin määritetyn pakkaustilan mukaisesti.



Kuva 7. Uv-koordinaattien tulkinta pikselivarjostimessa

4.5 Kamerat

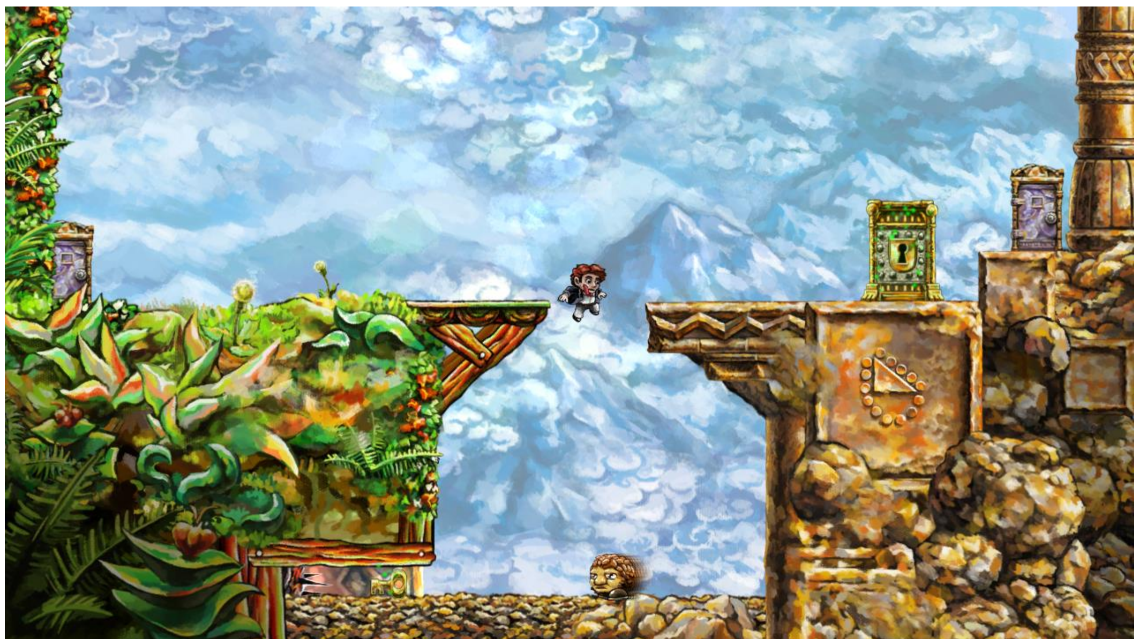
Projektio- ja transformaatiomatriisien ymmärtäminen ja käsittely voi olla haastaa. Grafiikkamoottoreissa kyseisillä matriiseilla voidaan luoda kuvitteellisia kameroita, joiden käyttö on huomattavasti helpompaa kuin matriisien suora muokkaus. SGL-kirjaston sisältämät kamerat kykenevät esittämään ortografisia ja perspektiivisiä projektioita. Kameroita on tarpeen mukaan mahdollista siirtää, kääntää ja skaalata. Kuvassa 8 näkyy, kuinka projektiot eroavat toisistaan.



Kuva 8. Projektiot

4.6 Spriten piirto

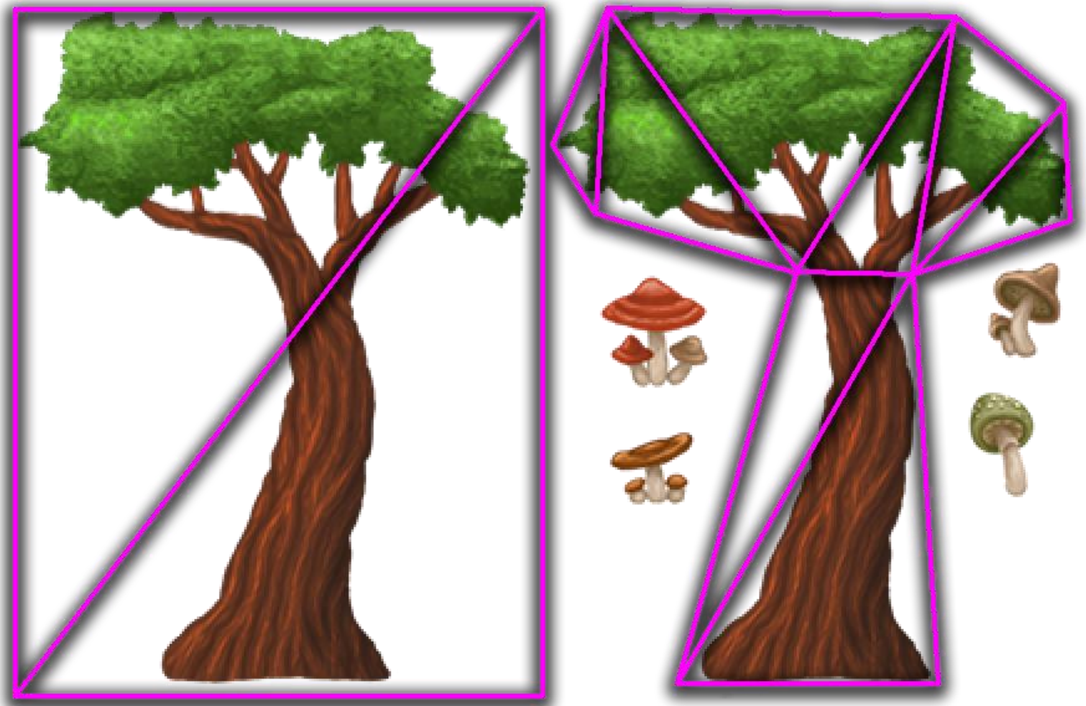
Sprite eli kaksiulotteinen bittikartta on äärimmäisen olennainen osa 2D-grafiikkamoottorin toimintaa. Kuvassa 9 on kuvankaappaus videopelistä Braid. Pelin maailma koostuu useista spriteistä, jotka yhdessä muodostavat hyvinkin näyttävän pelimaailman.



Kuva 9. Kuvankaappaus pelistä Braid

Verteksien sisältämien uv-koordinaattien sijoittelulla voi olla vaikutus suorituskykyyn. Etenkin, jos käytetyssä tekstuurissa on paljon tyhjää tilaa. Kuvasta 10

voi huomata kuinka vasemmanpuoleinen puuta esittävän spriten uv-koordinaatit peittävät ison alueen, josta noin puolet on tyhjää. Oikeanpuoleisessa spritessä on lisätty verteksien määrää ja koordinaateilla on myötäilty objektia. Ratkaisu tuo lisää tilaa muille spriteille ja vähentää rasteroitavan alueen kokoa.



Kuva 10. Spriten uv-koordinaattien sijoittelu

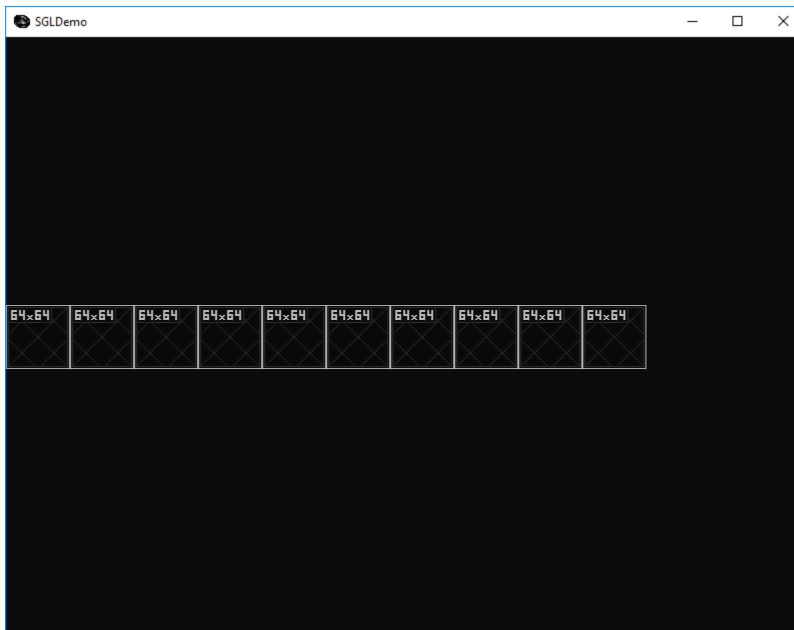
Oikeanpuoleinen ratkaisu kuitenkin tarvitsee enemmän verteksejä ja siten enemmän muistia. Verteksit on myös sijoiteltava käsin, tämä voi lisätä ohjelmoijan tai graafikon työmäärää, jos spritejä on paljon. Näiden kahden vaihtoehdon välillä on tasapainoteltava tapauskohtaisesti.

SGL-kirjaston sprite-grafiikan piirto tapahtuu isoissa erissä, jotta vältytään turhilta OpenGL-komennoilta. Kuvassa 11 on esimerkkiohjelma, joka piirtää 10 spriteä. Puolet koodista on puhtaasti pelkkää asioiden alustusta, kuten ikkunan, kameran, tekstuurin ja renderöijan luonti. Silmukan sisällä tapahtuu itse renderöinti, joka vaatii vain 9 riviä koodia. Silmukassa synkronoidaan renderöijän puskuri, lisätään 10 spriteä renderöijään ja piirretään kyseiset 10 spriteä.

```
SGL_Init();
SGL_Window window = SGL_CreateWindow("SGLDemo", 4, 5, 3, SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
SGL_Camera* camera0 = SGL_CreateCamera(&window.rContext, SGL_CAMERA_TYPE_ORTHO,
SGL_CAMERA_ORTHO, 0.1f, 2.0f, degreesToRadians(45.0f), 1.0f);
SGL_BindCamera(&window.rContext, SGL_CAMERA_ORTHO);
SGL_DataSelect("Demo_data");
SDL_Surface* surf = SGL_DataLoadImage("spriteGridTex");
SGL_Tex2D tex = { surf->pixels, 0, GL_BGRA, GL_RGBA8, surf->w, surf->h };
SGL_CreateTextures(&tex, &SGL_ParamsNearest, 1);
SGL_SimpleSpriteRenderer ssr = SGL_CreateSimpleSpriteRenderer(10, &tex, &win-
dow.rContext);
SGL_TexRegion texReg = { 64.0f,  64.0f,  64.0f,  64.0f };
SGL_Vec2 scale = { 64.0f, 64.0f };
bool quit = false;
while (!quit)
{
   SGL_RendererSync(ssr.syncs[ssr.bufferOffset]);
   for (size_t i = 0; i < 10; i++)
   {
      SGL_Vec2 pos = { i*64.0f-window.rContext.windowHalfSizef.x+32.0f, 0.0f };
      SGL_AddSpritePS(&ssr, pos, scale, texReg);
   }
   SGL_StartRender(&window);
   SGL_SimpleSpriteRendererDraw(&ssr, &window.rContext);
   SGL_EndRender(&window);
}
```

Kuva 11. Esimerkkiohjelma

Kuvassa 12 näkyy esimerkkiohjelman piirtämät 10 spriteä. Grafiikkamoottori pystyy modernilla tietokoneella piirtämään sulavasti satojatuhansia spritejä.



Kuva 12. Kuvankaappaus esimerkki ohjelmasta

4.7 Staattinen ja dynaaminen data

SGL-kirjastossa on erikseen renderöijat dynaamisille ja staattisille spriteille. Dynaamiset renderöijät ylikirjoittavat edellisen ruudunpäivityksen yhteydessä olleen verteksidatan. Spriteihin liittyvä verteksidata luodaan siis jokaisella kuvan ruudunpäivityksellä uudelleen. Eli dynaamisen renderöijän data voi muuttua koko ajan, mutta data pitää lähettää näytönohjaimelle ennen jokaista kuvan piirtoa. Liikkuvien asioiden kuten pelihahmojen on hyvä olla dynaamisia.

Staattiselle renderoijalle sen sijaan luodaan spritet yhden kerran alustuksen yhteydessä, eikä spritejä voida enää muokata jälkeenpäin. Esimerkiksi taivasta tai maastoa esittävät spritet eivät juurikaan muutu, jolloin ne voisivat olla staattisia. Aina kun mahdollista, tulee käyttää staattisia renderöijiä, koska niiden käyttämää dataa ei tarvitse siirrellä jatkuvasti keskusmuistista näytönohjaimen muistiin. Kaikki datan siirtoon käytetty aika viivästyttää muita piirtoputken vaiheita.

4.8 Matematiikka

SGL-kirjasto sisältää pienimuotoisen matematiikkakirjaston, josta löytyy geometriaan, matriisilaskentaan ja yksikkömuunnoksiin soveltuvia operaatioita ja rakenteita. Oman matematiikka kirjaston kirjoittaminen ei ollut mitenkään tarpeellista, koska vapaan lähdekoodin hyviä ja kattavia matematiikkakirjastoja on olemassa. Mutta matematiikan oppimisen kannalta kirjasto oli hyvää harjoitusta ja itse kirjoitettu kirjasto sisältää vain tarvittavat rakenteet ja operaatiot.

Kirjastossa käytetty C-kieli ei sisällä operaattorien kuormitusta kuten C++-kieli, jolloin matemaattiset operaatiot oli pakko kirjoittaa funktioin. Funktioiden helppokäyttöisyys tai siisteys on paikka paikoin hyvin kyseenalaista, koska funktioiden nimet paisuivat suhteellisen pitkiksi.

4.8.1 Vektorit

Matematiikkakirjasto sisältää 2–4 komponenttisia vektoreita. Vektorit sisältävät joko kokonaislukuja tai liukulukuja eli desimaalilukuja. Kuvassa 13 on normalisointifunktion lähdekoodi. Funktiossa muutetaan parametrina saatu kaksiulotteinen vektori yksikkövektoriksi.

```
inline const SGL_Vec2 SM_V2Normalize(const SGL_Vec2 a)
{
   SGL_Vec2 r;
   float m = SDL_sqrtf(a.x * a.x + a.y * a.y);
   r.x = a.x / m;
   r.y = a.y / m;
   return r;
}
```

Kuva 13. Matematiikkakirjaston funktio (SGL-kirjaston lähdekoodi)

4.8.2 Matriisit

Matriisilaskenta on huomattavasti monimutkaisempaa ja raskaampaa vektorilaskentaan verrattuna. Toistuvimmat matriisilaskut ovat verteksivarjostimissa näytönohjaimen taakkana, koska näytönohjaimet osaavat käsitellä laskuja nopeasti. Matriiseja käytetään SGL-kirjastossa kameraprojektioiden esittämiseen ja verteksidatan manipulaatioon.

Kuvan 14 esittämässä funktiossa matriisia siirretään kaksiulotteisen vektorin osoittama määrä. Kyseisen funktion luomalla matriisilla voitaisiin esimerkiksi siirrellä jonkin objektin verteksien sijainteja.

```
inline const SGL_Mat3 SM_M3Translate(const SGL_Mat3* m, const SGL_Vec2 v)
{
   SGL_Mat3 r = *m;
   r.m20 = m->m00 * v.x + m->m10 * v.y + m->m20;
   r.m21 = m->m01 * v.x + m->m11 * v.y + m->m21;
   r.m22 = m->m02 * v.x + m->m12 * v.y + m->m22;
   return r;
};
```

Kuva 14. 3x3-matriisin siirto funktio (SGL-kirjaston lähdekoodi)

4.9 Suorituskykyä parantavat tekniikat

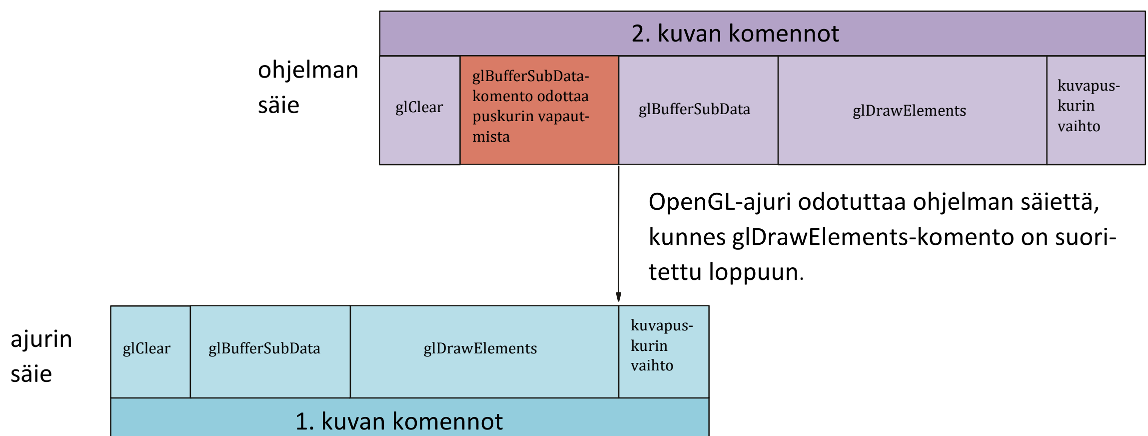
Ohjelmoinnissa termillä optimointi tarkoitetaan ohjelman suorituskyvyn tai resurssien käytön parantamista etsimällä kustannustehokkainta ratkaisua ongelmaan. Optimoinnissa on tärkeää löytää ohjelman pahimmat ongelmakohdat ja keskittyä niihin. Pienten yksityiskohtien syynääminen ja hierominen harvemmin parantavat ohjelman suorituskykyä näkyvästi, koska kääntäjä usein tekee pienet optimoinnit ohjelmoijan puolesta. Algoritmit ja hyvin suunnitellut tietorakenteet usein auttavat hyvän suorituskyvyn saavuttamisessa.

4.9.1 Datan monipuskurointi

Monipuskuroinnissa luodaan useampi puskuri datalle, jotta vältyttäisiin yhdelle puskurille ominaisilta datan synkronointiongelmilta. Kuvapuskuri on yleensä monipuskuroitu, koska tällöin voidaan piirtää uutta kuvaa piilotetulle puskurille ja samalla esittää näytöllä toisen puskurin sisältämää kuvaa. Kun piilotetun puskurin kuva on piirretty loppuun, niin puskureiden paikkoja vaihdetaan eli piilotetun puskurin kuva esitetään näytöllä ja aiemmin näkyvillä ollut puskuri piilotetaan. (Sellers ym. 2014, 634.) SGL-kirjastossa käytetty SDL-kirjasto hoitaa kuvapuskurin kaksoispuskuroinnin automaattisesti.

SGL-kirjastossa on käytetty monipuskurointia myös verteksidatan siirron nopeuttamiseen. Kuvassa 15 näkyy kuinka ilman monipuskurointia verteksidatan muokkaaminen voi aiheuttaa turhaa odottamista ohjelman säikeessä, johon OpenGL-konteksti on sidottu.

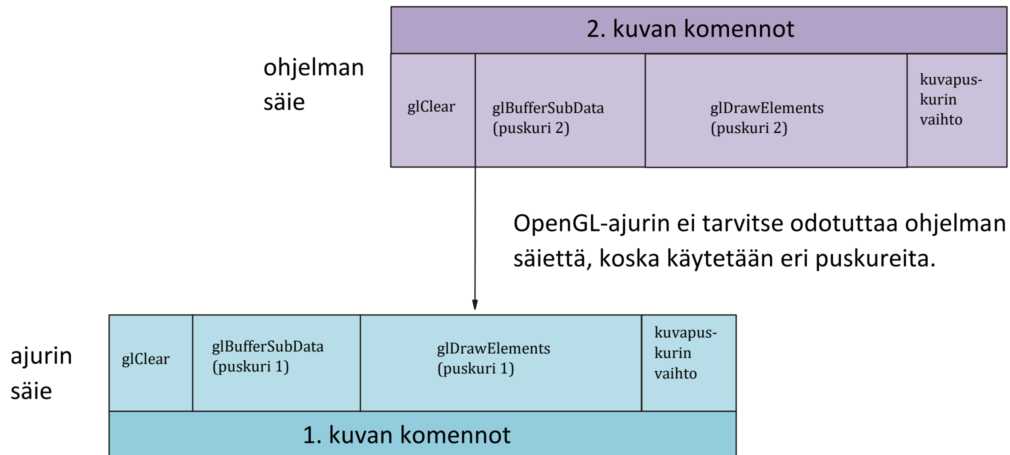
On mahdollista, että OpenGL-ajuri käyttää piirtämiseen puskurin sisältämää verteksidataa samalla kun ohjelmasäie haluaisi päivittää kyseisen puskurin dataa. Ajuri ei anna päivittää puskuria, ennen kuin piirto on suoritettu loppuun asti. (Cozzi & Riccio, 2012, 396–397.)



Kuva 15. Verteksidatan yksöispuskurointi

Kuvasta 16 voidaan huomata, kuinka useamman puskurin käyttö hävittää turhan odottelun ohjelmasäikeestä. Ohjelmasäikeen ei tarvitse odottaa piirron loppumista, koska tällä kertaa puskurista 1 piirretään ja samanaikaisesti puskuriin 2 siirretään dataa. (Cozzi & Riccio, 2012, 398). Useamman puskurin käyttö ei kuitenkaan tule täysin kuluitta, sillä ylimääräiset puskurit tietenkin

moninkertaistavat datan tarvitseman muistin määrän. On siis syytä miettiä, onko monipuskurointiin varaa laitteella, jossa muistista voi olla pulaa.

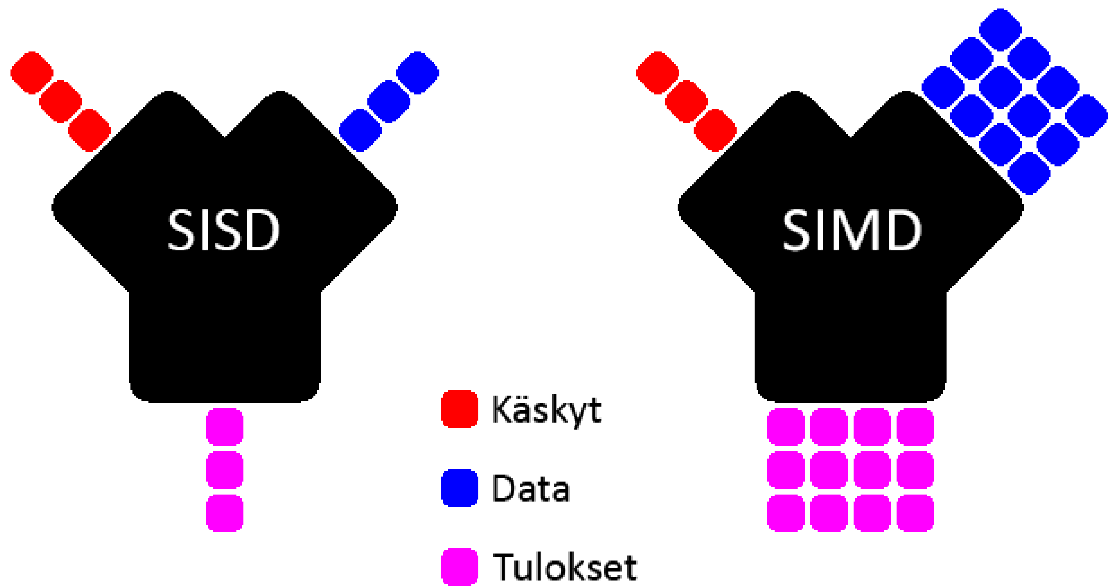


Kuva 16. Verteksidatan kaksoispuskurointi

SGL-kirjastossa monipuskurointia on mahdollista käyttää kaikilla dynaamisissa renderöijillä. Monipuskurointi on toteutettu varaamalla 1–3-kertainen määrä tilaa yhdelle puskurille. Puskurin muokkaus ja piirto tapahtuvat niin, että puskuri on jaettu 1–3 osaan, joiden käyttöä vuorotellaan.

4.9.2 SIMD-käskykantalaajennokset

SIMD-käskykantalaajennokset mahdollistavat rinnakkaislaskennan käytön prosessorilla. Normaalisti prosessori käsittelee yhtä datajoukkoa yhdellä käskyjonolla SISD-mallin mukaisesti. Modernit prosessorit pystyvät käsittelemään rinnakkain useampaa datajonoa, joille suoritetaan yhtä käskyjonoa SIMD-mallin mukaisesti. (Gregory, 2014, 118.) Kuvasta 17 voi havaita kuinka SIMD-mallin käsittelemä datamäärä on huomattavasti suurempi.



Kuva 17. SISD- ja SIMD-malli

SGL-kirjastossa on käytetty neljän liukuluvun käsittelyyn tarkoitettuja SSE-käskykantalaajennoksia. SSE on prosessorivalmistaja Intelin kehittämä laajennos. (Intel Streaming SIMD Extensions Technology Defined, 2016). Monien matriisi- ja vektorilaskujen suoritusaika puolittui laajennosten ansiosta. SGL-kirjastoa käännettäessä ARM-alustalle SSE-käskyjä ei käytetä, koska ARM-prosessorit käyttävät omia NEON-käskyjä SSE-käskyjen sijaan (NEON ARM, 2016).

Kuvassa 18 on SGL_Vec4 rakenteen koodi. Kyseessä on neliulotteinen vektori, joka sisältää komponentit x, y, z ja w 32-bittisinä liukulukuarvoina. Komponenttien kanssa on samassa unionissa 128-bittinen muuttuja v, jota voidaan käyttää SSE-funktioissa.

```
#if !defined(ANDROID)
__declspec(align(16))
#endif
typedef struct _SGL_Vec4
{
union
{
   struct
   {
      F32 x, y, z, w;
   };
#if !defined(ANDROID)
   struct
   {
      __m128 v;
   };
#endif
};
} SGL_Vec4;
```

Kuva 18. SGL_Vec4 rakenne (SGL-kirjaston lähdekoodi)

Käsiteltäessä __m128-tyyppisiä muuttujia on syytä huolehtia muuttujien sijainnista muistissa. Muuttujien tulee olla 16 tavun kohdistuksella muistissa, jotta niitä ei tarvitse kohdistaa enää käsittelyssä. Kyseinen kohdistaminen on hoidettu __declspec-syntaksin avulla. SSE-tukea ei ole ARM-laitteilla, jolloin SSE-datatyyppien käyttö estetään makrojen avulla.

Kuvassa 19 on kahden neliulotteisen vektorin kertolaskufunktion lähdekoodi. Normaalisti molempien vektorien samat komponentit pitäisi kertoa toisillaan aiheuttaen 4 erillistä kertolaskuoperaatiota. SSE-laajennoksiin kuuluvalla _mm_mul_ps-funktiolla voidaan sen sijaan laskea 4 kertolaskua rinnakkain.

```
inline const SGL_Vec4 SM_V4Multiply(const SGL_Vec4* a, const SGL_Vec4* b)
{
   SGL_Vec4 r;
#if defined(ANDROID)
   r.x = a->x - b->x;
   r.y = a->y - b->y;
   r.z = a->z - b->z;
   r.w = a->w - b->w;
#else
   r.v = _mm_mul_ps(a->v, b->v);
#endif
   return r;
}
```

Kuva 19. Vektori kertolasku (SGL-kirjaston lähdekoodi)

4.9.3 Data-orientoitunut suunnittelu

Data-orientoidussa suunnittelussa suositaan tehokasta lineaarista datankäsittelyä ja eristettyjä rakenteita. Data pyritään sekvensoimaan niin, että prosessorin olisi mahdollisimman nopea noutaa ja käsitellä sitä. Tämän tyyppinen suunnittelu usein helpottaa säikeiden ja rinnakkaislaskennan käyttöä. Data-orientoitu suunnittelu ei sulje pois muita ohjelmoinnin malleja. (Fabian, 2013.) On kuitenkin hyvin mahdollista, että esimerkiksi olio-ohjelmointimallin mukaisesti kirjoitettua koodia joutuu hieman muokkaamaan, jotta data-orientoidun mallin kriteerit täyttyisivät.

4.10 Demonstraatio-ohjelma

Demonstraatio-ohjelman eli demon avulla pyritään esittämään jotain teknistä ominaisuutta tai ominaisuuksia. SGLDemo-ohjelmassa näytetään, miten staattiset ja dynaamiset objektit eroavat toisistaan ja miten monipuskurointi vaikuttaa suorituskykyyn. Suorituskykymittauksissa käytettiin Intel Core i5-4670K -prosessoria ja AMD Radeon HD 7870 -näytönohjainta.

4.10.1 Esittely

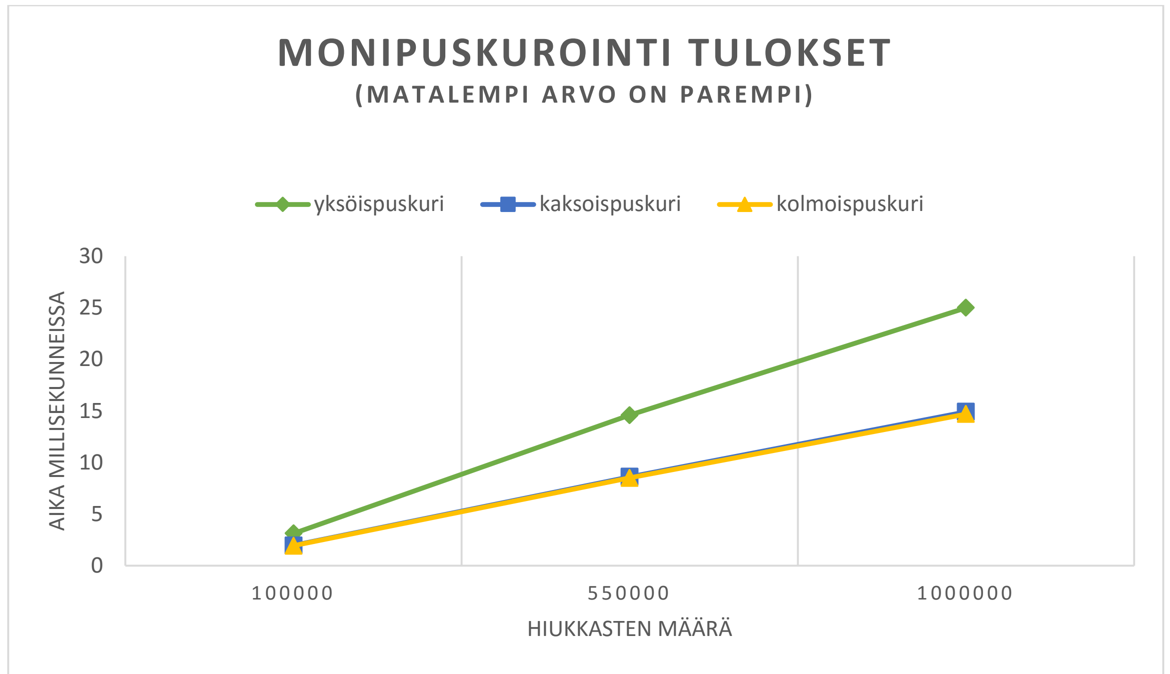
Kuvassa 20 on kuvankaappaus demosta, joka käyttää SGL-kirjastoa renderöintiin. Demossa on taustalla taivas ja vuoria, jotka on piirretty staattisella renderöijällä. Taivas on renderöity käyttäen yksinkertaista varjostinohjelmaa, joka käyttää verteksien värejä piirrossa. Vuoriin käytetty varjostinohjelma poimii värit tekstuurista, joka tässä tapauksessa sisältää vuorien kuvia.

Kuva 20. Kuvankaappaus demonstraatio-ohjelmasta

Kuvassa näkyvät keltaiset hiukkaset ovat simuloitu liikkumaan kohti kursoria lisäämällä hiukkasen nopeusvektoriin hiukkasen ja kursorin välinen vektori. Hiukkasten sijainteja liikutetaan eteenpäin joka ruudunpäivityksen yhteydessä nopeusvektorin verran. Tämän jälkeen hiukkasten uudet sijainnit annetaan renderöijälle. Hiukkaset ovat renderöity dynaamisella datalla ja niissä käytetään samaa varjostinohjelmaa kuin vuorien piirrossa. Kuvassa on 550 000 hiukkasta ja niiden siirtämiseen ja renderöintiin kuluu noin 8 millisekuntia. Eli kuva ehditään renderöidä noin 120 kertaa sekunnissa.

4.10.2 Suorituskykymittaukset

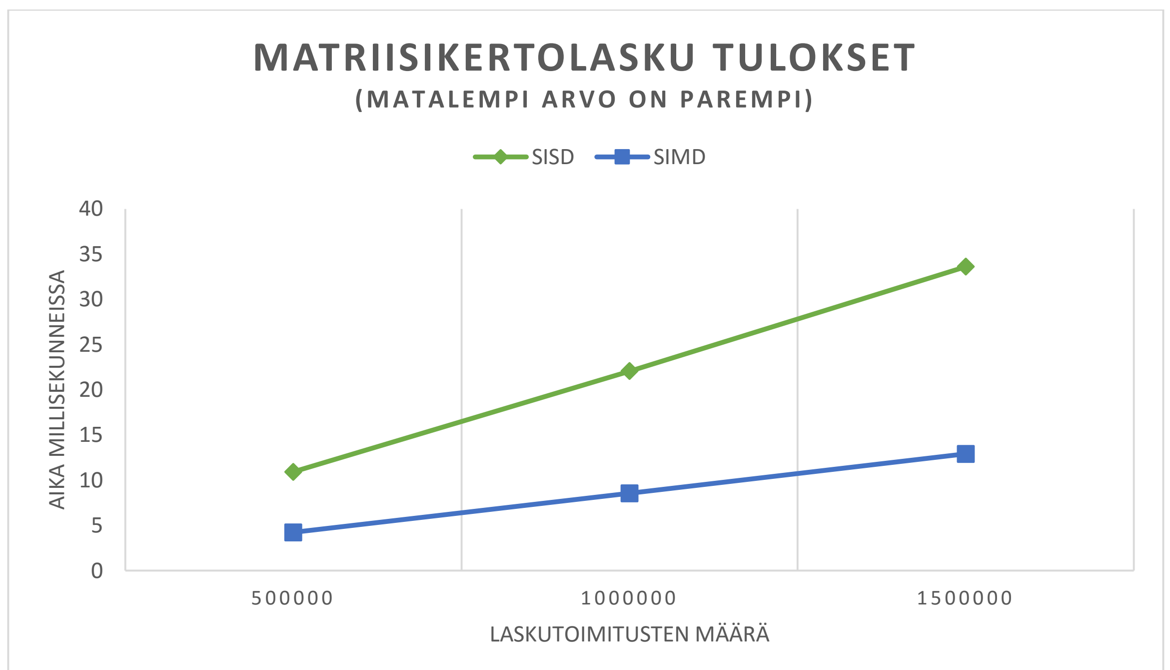
Demossa hiukkasia esittävä verteksidata on monipuskuroitu ja kuvasta 21 voidaan huomata, kuinka puskurien määrät vaikuttavat suorituskykyyn. 1 000 000 hiukkasta piirrettäessä yksöispuskuri joutui odottamaan piirron päättymistä jopa 15 millisekuntia. Kuvaajalla ilmoitetulla ajalla tarkoitetaan koko piirtosilmukan suorittamiseen kulunutta aikaa, eikä pelkästään puskurin datansiirtoon kulunutta aikaa.



Kuva 21. Monipuskurointitulokset

Lisäpuskurit vievät kuitenkin muistia ja kuten kuvaajasta huomaa niin kaksois- ja kolmoispuskuroinnin ero suorituskyvyssä on äärimmäisen pieni. Kolmoispuskuroinnin käyttö vaikuttaakin hyvin pitkälti pelkältä muistin haaskaukselta.

Ennen demon tekoa testattiin pienemmällä sovelluksella SIMD-käskykantalaajennosten tuomaa etua. Ilmeni, että pienemmät funktiot kuten neliulotteisille vektoreille suoritettavat SSE-käskyt, eivät tuoneet järisyttävän suuria suorituskykyetuja. Isommissa funktiossa kuten 4x4-matriisien laskutoimituksissa pystyi jo näkemään huomattavia eroja, kuten kuvasta 22 voidaan huomata.



Kuva 22. 4x4-matriisikertolasku tulokset

SIMD-käskykantalaajennoksien käytön avulla matriisikertolasku nopeutui noin 61 prosentilla. 4x4-matriiseja ei valitettavasti 2D-grafiikkamoottorissa käytetä paljoa, koska kaksiulotteiset transformaatiot voidaan suorittaa 3x3-matriiseilla. SSE-käskyjä olisi mahdollista käyttää myös 3x3-matriisien laskutoimituksissa, edellyttäen, että matriisit varaisivat todellisuudessa muistia 3x4-matriisien tarpeisiin. SSE-käskyissä käytetty __m128-tyyppi voi sisältää 4 kappaletta 32-bittisiä liukulukuarvoja. 3x4-matriisin muuttujia ei siis tarvitsisi siirrellä muistissa käytettäessä SSE-käskyjä, vaan ne olisivat suoraan oikein aseteltuna muistissa.

5 PÄÄTELMÄT JA JATKOKEHITYS

Grafiikkamoottorin toteutukseen käytetty aika yllätti. Työmäärä oli valtava ja isojen rakenteellisten muutosten takia useita tuhansia rivejä koodia piti välillä hävittää. Pelkän Visual Studio -projektin luonti ja tarvittavien kirjastojen lisäämisessä meni viikko. Myös satunnaiset ongelmat koodin kääntämisen kanssa saattoivat hidastaa ohjelmoinnin etenemistä tunneista päiviin.

Helmikuussa 2016 julkaistu Vulkan-rajapinta vaikutti mielenkiintoiselta ja sen parissa tuli taisteltua useampi viikko. Lopulta luovuttiin ideasta, koska rajapinta oli todella työläs käyttää ja siihen liittyvä dokumentaatio oli hyvin niukkaa. Rajapinnan ajuritkin olivat vasta beta-vaiheessa ja niiden toiminta epävarmaa, tämä vaikeutti huomattavasti ongelmien paikantamista.

Koko SGL-kirjasto oli kirjoitettu C-kielellä, jonka käyttö oli usein nopeaa ja helppoa. Tästä huolimatta jotkin kirjaston rakenteet tai osa-alueet olisivat saattaneet hyötyä kattavamman kielen ominaisuuksista. Esimerkiksi matemaattisia funktioita olisi voinut yksinkertaistaa C++-kielen operaattoreiden kuormituksella.

Grafiikkamoottoria voisi jatkokehittää lisäämällä siihen ominaisuuksia, kuten materiaalien käsittely, luupohjainen animaatio ja tekstin renderöinti. Moottorin rakenteitakin voisi vielä siistiä ja dokumentoida paremmin. Demo-ohjelmaa tehdessä huomattiin, että yksinkertainenkin kenttäeditori saattaisi helpottaa pelimaailmojen luomisessa. Spriten lopullisen ulkomuodon hahmottaminen voi paikoittain olla haastavaa ilman visuaalista editoria.

Tämän hetkinen grafiikkamoottori sisältää opinnäytetyössä suunnitellut ominaisuudet ja sitä olisi mahdollista käyttää pohjana jonkin graafisen tyylin esittämisessä. Moottoria tehtäessä oppi ymmärtämään erittäin hyvin minkälaisista asioista grafiikkamoottorit koostuvat. Osa lähdemateriaalista käsitteli grafiikkamoottoreita hyvin kattavasti. Tämä auttoi hahmottamaan myös muiden grafiikkamoottorien toimintaa ja rakenteita. Tulevaisuudessa oman grafiikkamoottorin kirjoittamisen sijaan olisi myös mahdollista yrittää laajentaa olemassa olevia moottoreita.

LÄHTEET

Cozzi, P. & Riccio C. 2012. OpenGL Insights. Bota Raton: CRC Press.

Fabian, R. 2013. Data-Oriented Design. Verkkojulkaisu. Saatavissa: http://www.dataorienteddesign.com/dodmain/dodmain.html [viitattu 10.5.2016].

Game engines used by the video game developers in the United Kingdom. 2014. Verkkojulkaisu. Saatavissa: http://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/ [viitattu 10.5.2016].

Gregory, J. 2014. Game Engine Architecture. 2. painos. Boca Raton: CRC Press.

Intel Streaming SIMD Extensions Technology Defined. 2016. Verkkojulkaisu. Saatavissa: http://www.intel.com/content/www/us/en/support/processors/000005779.html [viitattu 10.5.2016].

Mitchell, J., Ginsburg, D., Geldreich, R. & Lohrmann, P. 2014. Moving to OpenGL. PowerPoint-diaesitys 15.1.2014. Seattle. Saatavissa: http://media.steampowered.com/apps/steamdevdays/slides/movingtoopengl.pdf [viitattu 10.5.2016].

NEON ARM. 2016. Verkkojulkaisu. Saatavisssa: http://www.arm.com/products/processors/technologies/neon.php [viitattu 10.5.2016].

Olson, T., Andersson, J., Griffais, P., Baker, D., Smedberg, N., Pranckevičius, A. & McDonald J. 2016. Vulkan Overview. Diaesitys. Saatavissa: http://media.steampowered.com/apps/valve/2015/Pierre-Loup_Griffais_and_John_McDonald_Vulkan.pdf [viitattu 10.5.2016].

Sellers, G., Richard, S. & Haemel N. 2014. OpenGL SuperBible. Comprehensive Tutorial and Reference. 6. painos. Boston: Addison-Wesley.

Sherrod, A. 2008. Game Graphics Programming. 1. painos. Boston: Cengage Learning.

Shreiner, D., Sellers, G., Kessenich, J. & Licea-Kane, B. 2013. OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 4.3. 8. painos. Boston: Addison-Wesley.

Stercus engine repository. 2016. Lähdekoodi. Saatavissa: https://github.com/uraani/StercusEngine [viitattu 10.5.2016].