Tampereen yliopisto

Severi Kujala

GRAFIIKKAOHJELMOINTIA ERI LÄHESTYMISTAVOILLA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Lokakuu 2019

TIIVISTELMÄ

Severi Kujala: Grafiikkaohjelmointia eri lähestymistavoilla
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Lokakuu 2019

Tässä kandidaatintyössä perehdytään eri abstraktiotasoille sijoittuviin ohjelmointirajapintoihin, jotka ovat tarkoitettu tietokonegrafiikan piirtoon. Tarkastelu rajataan matalan tason rajapintaan ja korkeamman tason rajapintaan. Koska vaihtoehtoja on lukuisia, rajataan lisäksi API:en lukumäärä yhteen kutakin tasoa kohti. API:en on myös oltava C- ja C++-kielien kanssa yhteensopivia. Tavoitteena on selvittää rajapintojen eroja ja yhtäläisyyksiä.

Rajapinnoiksi on valittu Simple DirectMedia Layer (SDL) ja Open Graphics Library (OpenGL). Tutkittavia asioita ovat rajapintojen erityispiirteet sekä miten ne eroavat esimerkiksi muista samalla abstraktiotasolla sijaitsevista rajapinnoista. Työssä vertaillaan myös esimerkiksi rajapintojen syntaksia ja käyttökohteita.

Työssä havaitaan, että vertailuja on jonkin verran erilaisilla foorumeilla, mutta tämän työn kaltaisia analyyseja ei ole. Lisäksi havaitaan, että matalan ja vähän korkeamman abstraktiotason rajapinnat eroavat huimasti toisistaan sekä ominaisuuksiiltaan että käytöltään. Vertailun myötä myös rajapinnan valinta tiettyä käyttötarkoitusta kohden selkeytyy entisestään. Matalan tason rajapinta sopii siis tarkkuutta ja tehokkkuutta vaativiin ohjelmiin, kuten 3D-peleihin, mutta pieniin projekteihin ja esimerkiksi vain 2D-elementtejä vaativaan ohjelmiin korkeamman tason rajapinta on aikaa ja vaivaa säästävä vaihtoehto.

Avainsanat: tietokonegrafiikka, API, OpenGL, SDL

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

API	ohjelmointirajapinta (Application Programming Interface)
CPU	suoritin, prosessori (Central Processing Unit)
GLEW	Kirjasto OpenGL-funktioiden lataamiseen (Graphics Library Extension Wrangler)
GLSL	OpenGL:n sävytinkieli (Graphics Library Shading Language)
GPU	grafiikkasuoritin, -prosessori (Graphics Processing Unit)
SDL	Simple DirectMedia Layer
VBO	objekti kulmapisteiden säilömiseen GPU:lle lähettämistä varten (Vertex Buffer Object)

1 JOHDANTO

Tietokoneohjelmistot muuttuvat jatkuvasti visuaalisemmiksi tietokoneiden ja ohjelmien kehittyessä. Aikaisemmin suoritin (Central Processing Unit, CPU) piirsi kuvan näytölle. Tällöin ei kuitenkaan jäänyt kovin paljon aikaa muulle toiminnalle, kuten logiikan suorittamiselle. Ongelma ratkaistiin kehittämällä piirtoa varten erillinen komponentti: grafiikkaprosessori eli GPU (Graphics Processing Unit). GPU on myös eräänlainen suoritin, mutta siinä missä CPU sisältää vain muutaman ytimen, GPU sisältää satoja tai jopa tuhansia ytimiä. Arkikielessä grafiikkaprosessorista käytetään nimitystä näytönohjain, jonka osa GPU vain oikeastaan onkin.

GPU:lle voidaan antaa käskyjä, kuten CPU:llekin. Tätä varten on olemassa erilaisia ohjelmointirajapintoja (Application Programming Interface, API), joita kaikki grafiikkaa sisältävät sovelluksetkin hyödyntävät. Jotta ohjelmoijan ei tarvitsisi keskittyä ohjelmoimaan alkeellisia ja yksinkertaisia käskyjä, on tehty myös korkeamman abstraktiotason kirjastoja.

Tässä kandidaatintyössä perehdytään eri abstraktiotasoille sijoittuviin ohjelmointirajapintoihin, jotka on tarkoitettu tietokonegrafiikan piirtoon, sekä vertaillaan näitä esimerkiksi ominaispiirteiden ja niiden käyttökohteiden kautta. Tarkastelu rajataan matalan tason rajapintaan ja korkeamman tason rajapintaan. Koska vaihtoehtoja on lukuisia, rajataan lisäksi API:en lukumäärä yhteen kutakin tasoa kohti. API:en on myös oltava C- ja C++-kielien kanssa yhteensopivia.

Työn 2. luku käsittelee eri grafiikkarajapintojen sisäisesti käyttämiä algoritmeja kuvan piirtoon ja eri kappaleiden hallintaan. 3. ja 4. luku kertovat eri abstaktiotasojen rajapinnoista alkaen helpommin ymmärrettävästä eli korkeamman tason grafiikka-API:sta. Luvussa 5 vertaillaan kahta valittua rajapintaa sekä edeltävissä luvuissa esitetyn pohjalta että esimerkkiohjelmien kautta. Tässä hyödynnetään myös luvussa 2 esitettyä teoriaa. Luku 6 on nimensä mukaisesti työn yhteenveto.

2 KESKEISET ALGORITMIT JA TEKNIIKAT

Tietokonegrafiikalla on vahva perusta matematiikassa. Kaikki osa-alueet sisältävät paljon yksinkertaista geometriaa ja monimutkaisia kompleksilukuja, kuten kvaternioita. Geometriaa käytetään muun muassa kappaleiden hallintaan ja kvaternioita rotaatioiden yhteydessä.

Monet 2D-grafiikan menetelmät pätevät myös 3D-grafiikassa tai ainakin toimivat pohjana joillekin 3D-grafiikan metelmille. Tästä syystä tässäkin työssä teoriaa käsitellään enimmäkseen 2D-tapauksien kautta. Vaikka kolmas ulottuvuus tuokin vain uuden muuttujan, jotkin algoritmit saattavat monimutkaistua huomattavasti.

Tämän luvun tarkoituksena on valottaa tietokonegrafiikan perusperiaatteita sekä erilaisia tekniikoita, jotka auttavat ymmärtämään myöhemmin esitettävän vertailun taustaa paremmin. Teoria pohjautuu vahvasti Antti Puhakan teokseen *3D-grafiikka* [23] sen yksinkertaisen ja helposti omaksuttavan ilmaisutavan vuoksi. Lisänä on käytetty julkaisuja tekniikoiden varhaisista versioista sekä näiden uudemmista toteutuksista.

2.1 Koordinaatistot

Kappaleiden solmujen eli kulmapisteiden (vertex) paikan ilmaisemiseen käytetään yleensä x-, y- ja z-koordinaatteja. Tällaista koordinaatistoa, jossa koordinaattiakselit ovat kohtisuorassa toisiinsa nähden, sanotaan *karteesiseksi koordinaatistoksi*. Origo eli keskipiste sijaitsee akseleiden leikkauskohdassa. [23, s. 30–31] Paikan koordinaatit voidaan esittää myös vektorimuodossa:

$$\begin{bmatrix} x \\ y \end{bmatrix} [23, s.30].$$

Tietokonegrafiikassa on kuitenkin muutamia poikkeuksia liittyen tavallisimpiin matemaattisiin koordinaatistoihin. Esimerkiksi kuvaruudussa pikselien positiivinen y-akseli ei kasvakaan ylöspäin vaan tavallisesti juuri päinvastaiseen suuntaan. 3D-grafiikassa kaikki akselit saattavat olla eri järjestyksessä ja kasvaa päinvastaiseen suuntaan, mikä riippuu pitkälti grafiikka-API:n toteutuksesta. Esimerksiksi OpenGL:ssa x-akseli osoittaa oikealle, y-akseli ylös ja z-akseli katsojaa kohti [17, luku 4]. OpenGL:sta kerrotaan lisää myöhemmissä luvuissa.

Erilaiset kappaleet, kuten pallot ja nelikulmiot, sijaitsevat maailmankoordinaatistossa (world space), joka kuvaa niiden suhdetta eli esimerkiksi etäisyyttä toisiinsa. Kappaleilla on myös oma objektikoordinaatistonsa (local object space), joka kuvaa kappaleen solmujen välisiä suhteita. Objektikoordinaatiston origo sijaitsee kappaleen keskipisteessä. [23, s. 167]

2.2 Transformaatiot

Kappaleita joudutaan joskus siirtelemään, suurentamaan ja kiertämään jonkin akselin ympäri. Näistä operaatioista käytetään yleisnimitystä transformaatio eli muunnos (transformation). Muunnoksia ovat esimerkiksi translaatio eli siirtomuunnos (translation), skaalaus (scale) ja rotaatio eli kierto (rotation). Näissä kaikissa kolmessa operaatiossa käytetään tavallisesti matriiseja tai vektoreita. [23, s.91–92] Koska 3D-grafiikassa muunnoksiin lisätään yleensä vain z-koordinaatti, esitellään tämän työn muunnokset 2D-muunnoksina.

Translaatio saadaan yksinkertaisesti lisäämällä vakioita koordinaatteihin:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix},$$

missä x' ja y' ovat muutoksen jälkeiset koordinaatit sekä a ja b vakioita. [23, s. 91]

Skaalaus suoritetaan yhden tai useamman koordinaattiakselin suhteen kertomalla koordinaatit muunnosmatriisilla. Tällöin saadaan

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix},$$

missä x' ja y' ovat muutoksen jälkeiset koordinaatit sekä a ja b muutoskertoimia. Mikäli kuvio tai kappale haluttaisiin peilata, kaikille solmuille suoritettaisiin skaalaus, jossa peilausakselia vastaavan muutoskertoimen arvoksi asetetaan -1. [23, s. 91–92]

Kierrossa sen sijaan hyödynnetään perusaritmetiikan lisäksi trigonometriaa. Rotaatiomatriisi $R(\theta)$ on muotoa

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix},$$

missä θ on kiertokulma. Kertomalla rotaatiomatriisi koordinaattivektorilla saadaan

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta & -y\sin\theta \\ x\sin\theta & y\sin\theta \end{bmatrix} .[23, s.91]$$

Erityisesti 3D-avaruudessa kierto voidaan toteuttaa tehokkaamin: käytetään kvaternioita. 2D-tapauksessa riittäisi tavalliset yksikkökompleksiluvulla kertominen, mikä siis vastaa kiertoa origon ympäri. 3D-tapauksessa *Eulerin kaava* $e^{\imath\theta} = \cos\theta + \imath\sin\theta$ ei riitä, vaan tarvitaan kaikkia imaginaariyksikköjä $\imath$, $\jmath$ ja k. [23, s. 111]

2.3 Homogeeniset koordinaatit

Edellä esiteltyt transformaatiot voidaan toteuttaa matriiseilla lukuunottamatta translaatiota. On kuitenkin kätevämpää laskea kaikki muunnokset samalla tavalla, sillä näin voidaan yhdistää muunnokset yhdeksi matriisiksi. Skaalaus ja rotaatio eivät tuota tässä ongelmia, sillä ne perustuvat kertolaskuun. Translaatio sen sijaan perustuu yhteenlaskuun, minkä vuoksi translaatiota ei voida käyttää yhdistettäessä matriiseja. [23, s. 96]

Tätä varten onkin kehitetty tapa, jolla saadaan myös translaatiot suoritettua matriiseja käyttämällä. Pisteet on täten esitettävä *homogeenisinä koordinaatteina*. 2D-avaruudessa tämä tapahtuu lisäämällä koordinaattivektoriin kolmas koordinaatti: w. Perustapauksessa pätee $w = 1$, mutta ei koskaan $w = 0$, sillä nollalla ei voi jakaa. [23, s. 97]

James F. Blinn ja Martin E. Newell toteavat artikkelissaan *clipping using homogenous coordinates*, että mallia kuvataan w:n avulla neljännessä ulottuvuudessa, ja se *projisoidaan* suoralle $w = 1$ eli takaisin 3D-koordinaatistoon. Tapaus $w = 0$ johtaa sen sijaan äärettömyyteen. [5] Jos jokainen koordinaatti jaetaan w:lla, jako jätetään tällöin suorittamatta määrittelemättömän laskutuloksen välttämiseksi. Tapahtuma, jossa suoritetaan w:lla jako, kutsutaan *perspektiivijaoksi*[23, s. 196].

Transformaatiomatriiseiksi saadaan nyt

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix},$$

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} ja$$

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

missä $T(t_x, t_y)$ on translaatiomatriisi, $S(s_x, s_y)$ on skaalausmatriisi ja $R(\theta)$ on rotaatiomatriisi. [23, s. 101] 3D-avaruudessa käytetään samaa ideaa ja lisätään ylimääräinen koordinaatti w sekä matriiseihin ylimääräinen rivi ja sarake.

2.4 Viivasegmentin piirto ja Bresenhamin algoritmi

Oleellista tietokonegrafiikassa on erilaisten viivojen eli *viivasegmenttien* piirtäminen. Yksinkertaisin tapa olisi käydä kaikki pikselit läpi pisteiden ($x1$, $y1$) ja ($x2$, $y2$) välillä. Kuitenkaan tämä algoritmi ei pysty sellaisenaan piirtämään muita kuin x- ja y-akselien suuntaisia viivoja. Algoritmiin voidaan lisätä suoran yhtälö $y = ax + b$, jolloin voidaan valita oikea pikseli verraten matemaattiseen suoraan, joka leikkaa molemmat pisteet. Tämä algoritmi osaa jo piirtää viivasegmentin, mutta kertolaskusta johtuen se on melko hidas. Eräs vaihtoehtoinen ja paljon nopeampi algoritmi, joka ei turvaudu kerto- tai jakolaskuihin, on nimeltään *Bresenhamin algoritmi* [6].

Bresenhamin algoritmin vahvuuksia ovat sen nopeus ja se, että algoritmi ei käytä kertolaskuja eikä liukulukulaskentaa. Algoritmin toiminta on selitetty hyvin Antti Puhakan kirjassa *3D-grafiikka*. [23]

Koska muut tapaukset ovat samoja lukuunottamatta paria muutosta, tarkastellaan aluksi tilannetta, jossa suoran kulmakerroin $0 < k < 1$. Tällöin viiva sijaistee koordinaatiston ensimmäisessä kahdeksanneksessa. VIivasegmentin piirrossa joudutaan valitsemaan yleensä kahden pikselin väliltä: $(x + 1, y)$ tai $(x + 1, y + 1)$. Jotta jatkuvasti ei valita esimerkiksi ylempää pikseliä ja vääristetä suoraa, otetaan käyttöön virhetermit e ja d. Merkitään $\Delta x = x_q - x_p$ sekä $\Delta y = y_q - y_p$, ja asetetaan virhetermien alkuarvoiksi $e = 0$ ja $d = 2\Delta y\Delta x$. [23, s. 155]

Mikäli valitaan ylempi pikseli, asetetaan $e' = e + k$. Jos taas valitaan edellisen pikselin viereinen pikseli eli $(x+1, y)$, asetetaan $e' = e+m-1$. Myös virhetermin d arvoa muutetaan piirretyn pikselin mukaan. Ensimmäisessä tapauksessa $d' = d + 2\Delta y$ ja jälkimmäisessä $d' = d + 2\Delta y - 2\Delta x$. [23, s. 155–156]

Virhetermien arvoa muutetaan joka x-koordinaatin arvon iteraatiolla. Termi d määrää, mihin pikseliin siirrytään. Tapauksessa $d \leq 0$ siirrytään viereiseen $(x + 1, y)$, ja asetaan d:n uudeksi arvoksi $d' = d + 2\Delta y$. Muutoin siirrytään ylempään $(x + 1, y + 1)$, ja asetetaan $d' = d + 2\Delta y - 2\Delta x$. [23, s. 155–156]

Kuten edellä todettiin, muut kuin ensimmäisen kahdeksanneksen tapaus saadaan piirrettyä verrattain pienillä muutoksilla. Viivasegmentit, joille $0 \leq m \leq 1$, voidaan piirtää

peilaamalla viiva jonkin suoran suhteen. Esimerkiksi tilanteessa $|\Delta y| > |\Delta x|$ viiva piirretään vaihtamalla x ja y keskenään. Jos taas $\Delta x < 0$, x korvataan sen negaatiolla $-x$. Myös tapauksessa $\Delta y < 0$, koordinaatti korvataan negaatiollaan. [23, s. 157]

Tästä syntyy kuitenkin Puhakan mukaan ongelma: piirrettyjen pikseleiden sijainnit eivät ole identtiset, mikäli piirtosuunta vaihdetaan päinvastaiseksi. Tämä kuitenkin voidaan ratkaista päättämällä piirtosuunta esimerkiksi Δx:n etumerkin perusteella. [23, s. 157]

2.5 Monikulmion täyttö

Vaikka Bresenhamin algoritmilla voidaan piirtää viivoja tehokkaasti, useimmat kappaleet tai *mallit* (model) ovat “umpinaisia” eli kokonaan väritettyjä. Kappale, kuten monikulmio, pitää siis täyttää jotenkin. Tähän tarkoitukseen on olemassa erilaisia menetelmiä, joista ehkä tunnetuimmat ovat *tulvatäyttö* ja *vaakarivitäyttö*.

Tulvatäyttö (flood fill) on eräs yksinkertaisimmista täyttöalgoritmeista. Klassisin versio tästä on 8:n naapurin täyttö (8-neighbour fill), jossa väritetään alkaen valitusta pikselistä ja siirrytään sitten sen ympärillä oleviin pikseleihin. Tätä toistetaan myös naapuripikseleille. Voidaan havaita, että algoritmi on rekursiivinen ja onkin Andrew Glassnerin mielestä “yksi hienoimmista rekursioesimerkeistä tietokonegrafiikassa”. [15] Vaikka tulvatäyttö on myös yksinkertainen toteuttaa, algoritmina se on kuitenkin melko tehoton, minkä vuoksi se ei välttämättä reaaliaikaiseen täyttöön sovikaan [23, s. 159–160].

Vaakarivitäytössä käytetään aivan toisenlaista lähestymistapaa. Nimensä mukaisesti vaakarivitäytössä pikselit väritetään rivi kerrallaan. Puhakka nimeää algorimille kolme eri vaihetta:

1. Etsitään rivin ja monikulmion reunojen väliset leikkauspisteet.
2. Järjestetään leikkauspisteet kasvavan x-koordinaatin mukaan.
3. Käydään leikkauspisteiden välit läpi, ja väritetään joka toinen väli. [23, s. 160]

Vaakarivitäytössä ylläpidetään listoja A ja B. Listaan A järjestetään monikulmion eivaakasuorat reunaviivat alempien päätepisteiden y-koordinaatin mukaan nousevaan järjestykseen. Joka täyttökierroksella listaan B tallennetaan reunaviivat, jotka leikkaavat käsiteltävän vaakarivin, niiden leikkauspisteen x-koordinaatin mukaan. [23, s. 160–161] Listojen sisältö muuttuu algoritmin edetessä vaakariveittäin. Puhakka kuitenkin huomauttaa, että esimerkiksi listan B sisältö ei yksinkertaisten kuvioiden kohdalla muutu. Tämä johtuu siitä, että reunaviivat eivät pääse leikkaamaan toisiaan. [23, s. 160–161]

2.6 Leikkausalgoritmit

Kuviot ja mallit eivät aina mahdu kokonaan piirtoikkunaan. Tällöin malli on *leikattava* (clip). Leikkausalgoritmien perusajatuksena on leikata yksittäisiä viivoja tai kokonaisia

kuvioita pienemmiksi joko *hylkäämällä* (reject) tai *hyväksymällä* (accept) osia tarkastelun kohteesta. Yksittäisten pisteiden tapauksessa rajojen ulkopuolelle jääneiden kulmapisteiden tilalle generoidaan “rajapisteitä” aivan piirtoikkunan rajalle. Tällöin myöhemmässä tarkastelussa voidaan huomioida vähemmän pisteitä ja nopeuttaa operaatioita esimerkiksi erillisen pikselikohtaisen tarkastelun sijaan.

Yksi varhaisimmista ja tunnetuimmista leikkausalgoritmeista on Danny Cohenin ja Ivan Sutherlandin mukaan nimetty Cohen–Sutherland-algoritmi. Se tarjoaa tehokkaan tavan leikata viivoja suorakulmion muotoista piirtoikkunaa varten. Algoritmissa tehdään aluksi testejä, joilla selvitetään leikkauskohtien laskemisen tarpeellisuus. Tämä tehdään kaikille päätepistepareille. Näiden testien avulla voidaan triviaalisti hyväksyä isoja ja kokonaan ruudulle mahtuvia kuvioita lähes kokonaan sekä triviaalisti hylätä paljon pisteitä pienen piirtoikkunan tapauksessa. [12, s. 113]

3 KORKEAN TASON GRAFIIKKA-API

Kuten aikaisemmin todettiin, grafiikka-API:eja ja -kirjastoja on paljon erilaisia. Toiset ovat helppokäyttöisiä, toiset vaikeampia oppia, mutta ne tarjoavat enemmän ominaisuuksia ja tarkemman grafiikan hallinnan matalan tason rajapintoihin nähden. C- ja C++-kielten grafiikkakirjastot ovat pääasiassa melko matalan tason kirjastoja johtuen kielten luonteesta, mutta ne voidaan silti jaotella korkean ja matalan tason rajapintoihin.

Tarkemmin esiteltäväksi korkean tason API:ksi on valittu SDL. Valinta on tehty SDL:n ominaisuuksien ja helppokäyttöisyyden perusteella.

3.1 Erilaisia korkean tason rajapintoja

Korkean tason graafisen ohjelmoinnin rajapintoja ovat esimerkiksi Qt, GTK+ ja SDL. Qt on alustariippumaton grafiikan ja muun median sekä verkon ohjelmointiin tarkoitettu ohjelmistokehys [3]. Se soveltuukin hyvin esimerkiksi graafisten käyttöliittymien toteuttamiseen. Qt on toteutettu käyttäen C++-kieltä [3].

Hieman samankaltainen on myös GTK+. GTK+ (GIMP toolkit [41]) on alun perin GIMP:n (GNU Image Manipulation Program [39]) hyödyntämä rajapinta. Sen uudemmilla versioilla on toteutettu muun muassa Linux-käyttöjärjestelmien GNOME-työpöytäympäristö [40]. Toisin kuin Qt, GTK+ ei *widgettien* eli erilaisten käyttöliittymäelementtien lisäksi sisällä muita ominaisuuksia, kuten verkko-ohjelmointimoduuleja [41].

SDL (Simple DirectMedia Layer) on kahdesta edellä mainitusta enemmän Qt:n kaltainen. Moduuleita on useita, ja ne mahdollistavat äänentoiston, verkko-ohjelmoinnin sekä hiiren, näppäimistön ja peliohjainten painallussignaalien käsittelyn. Qt:n tavoin myös SDL on alustariippumaton ja tukee yleisimpiä käyttöjärjestelmiä Windowsista Androidiin. [16]

Useat korkean tason grafiikka-API:t tai -kirjastot hyödyntävät muita käyttöjärjestelmästä riippuvia rajapintoja ja toimivat näin *sovittimina* (wrapper) matalamman tason vastaaviille. Esimerkiksi linux-käyttöjärjestelmässä useat kirjastot hyödyntävät X Window System -nimistä rajapintaa. [16] X Window System eli X11 hoitaa tarvittavien ikkunoiden luomisen ja niihin liittyvät operaatiot, kuten käyttäjän syötteet ja ikkunaan piirron [7]. Windowsin vastaava rajapinta on nimeltään DirectX [20].

3.2 SDL:n ominaispiirteitä

SDL on jaettu useampaan alijärjestelmään. Alijärjestelmiä ovat esimerkiksi video, audio, säikeet, ajastimet sekä alijärjestelmä erilaisille syötetapahtumille [26]. Näiden lisäksi SDL sisältää erikseen ladattavia moduuleita esimerkiksi äänentoistoon (SDL_mixer) [31], verkko-ohjelmointiin (SDL_net) [32], fonttien käsittelemiseen (SDL_ttf, true type font) [37] sekä kuvatiedostojen lataamiseen (SDL_image) [29].

SDL ei kuitenkaan sellaisenaan tue esimerkiksi ympyrän piirtoa. Tähän tarkoitukseen on olemassa standardin ulkopuolisia lisäosia. Ympyrän ja monen muun geometrisen kuvion piirtoon käytetään SDL_gfx-lisäosaa [9]. Mikäli esimerkiksi ympyrän piirtoon ei haluta käyttää lisäosia, on vastaava toiminnallisuus tehtävä ohjelmistolla piirtämällä ympyrän kehälle riittävästi viivoilla yhdistettyjä pisteitä. Tällöin saadaan vaikutelma ympyrästä, vaikka todellisuudessa kuvio onkin monikulmio.

Yleisesti käytettyjä SDL:n versioita on olemassa kaksi: 1.2 ja 2.0. Merkittävin ero näiden versioiden välillä on, että SDL 1.2 on toteutettu pääasiassa käyttämään CPU:ta renderöintiin. Tätä kutsutaan ohjelmistopohjaiseksi renderöinniksi (software rendering). SDL 2.0 taas hyödyntää laitteistokiihdytystä (hardware acceleration). [25] Laitteistokiihdytyksessä suurin osa piirtotaakasta siirretään GPU:lle, joka osaa suorittaa grafiikan vaativat toisteiset, mutta yksinkertaiset, operaatiot tehokkaasti.

SDL on toteutettu käyttäen C-kieltä, ja se tukee luonnollisesti näin myös C++-kieltä. Joillekin kielille on kuitenkin saatavissa *kielisidoksia* (language binding). Vaikka jokin kieli ei suoraan tukisikaan SDL:a, on silti mahdollista käyttää SDL-kirjastoa lisäämällä erillistä koodia liimaksi kielen ja kirjaston välille. Kielisidoksen kautta tuettuja kieliä ovat esimerkiksi Rust, Lua, Pascal, Python ja C#. [27]

3.3 SDL-ohjelmointi käytännössä

Jotta SDL:n eri alijärjestelmiä ja moduuleja voidaan käyttää, on SDL-ympäristö ensin alustettava. Ohjelmassa 3.1 on esimerkki alustuksesta yksinkertaisessa ohjelmassa, jossa SDL-ikkuna on kapsuloitu *Window*-luokaksi.

```
1  bool Window::init() {
2
3      // init subsystems
4      if ( SDL_Init(SDL_INIT_VIDEO) != 0) {
5          std::cerr << "SDL: "
6                    << SDL_GetError()
7                    << "\n";
8          return false;
9      }
10     mWindow = SDL_CreateWindow(mTitle.c_str(),
11                                SDL_WINDOWPOS_CENTERED,
12                                SDL_WINDOWPOS_CENTERED,
13                                mWidth,
14                                mHeight,
15                                0);
16
17     if (mWindow == nullptr) {
18         std::cerr << "SDL: "
19                   << SDL_GetError()
20                   << "\n";
21         return false;
22     }
23     mRenderer = SDL_CreateRenderer(mWindow,
24                                    SDL_RENDERER_PRESENTVSYNC,
25                                    flag);
26
27     if (mRenderer == nullptr) {
28         std::cerr << "SDL: "
29                   << SDL_GetError()
30                   << "\n";
31         return false;
32     }
33
34     return true;
35 }
```

Ohjelma 3.1. *Esimerkki Window-luokan alustuksesta.*

Ohjelmassa 3.1 muuttujat *mWindow*, *mRenderer* ja *mTitle* ovat luokan *Window* jäsenmuuttujia. Edellisistä *mWindow* ja *mRenderer* ovat osoittimia tietueihin (struct), jotka ovat tyypeiltään *SDL_Window* ja *SDL_Renderer*. SDL käyttää näitä renderöijän tilan [35] ja ikkunan hallintaan. SDL-alijärjestelmät on mahdollista alustaa yksittäin käyttäen funktiota *SDL_Init* ja operaattoria *bitwise OR* yhdistämään haluttujen järjestelmien enumerointeja [30]. Tämä tapa osoittautuu varsin kömpelöksi, mikäli tarkoitus onkin alustaa useampia alijärjestelmiä. SDL tarjoaakin huomattavasti siistimmän vaihtoehdon enumeroinnin avulla: *SDL_INIT_EVERYTHING* [30].

Kenties keskeisin osa graafista ohjelmaa on päivityssilmukka (update loop), jossa uuden kuvan (frame) piirto tapahtuu. Ennen uuden kuvan piirtoa vanha on pyyhittävä puskurista (buffer) pois. Kuvan valmistuttua se lähetetään näytölle piirrettäväksi. Tästä aiheutuu kuitenkin ongelma: kuvaa vasta piirretään, mutta samaa dataa saatetaan lukea toisaalla. Useimmiten tämä ratkaistaan käyttämällä kahta eri puskuria ja vaihtelemalla puskuria, jonka data näytetään näytöllä. [36] Ohjelmaan 3.2 on toteutettu päivityssilmukka, joka piirtää punaisen neliön ikkunan vasempaan yläkulmaan käyttäen edellä kuvattua menetelmää.

```
1  SDL_Rect rectangle{0, 0, 200, 200};
2
3  while (true) {
4
5      // clear screen
6      SDL_SetRenderDrawColor(ptrToRenderer, 255, 255, 255, 255);
7      SDL_RenderClear(ptrToRenderer);
8
9      // draw to buffer
10     SDL_SetRenderDrawColor(ptrToRenderer, 255, 0, 0, 255);
11     SDL_RenderDrawRect(ptrToRenderer, &rectangle);
12     SDL_RenderFillRect(ptrToRenderer, &rectangle);
13
14     // update screen
15     SDL_RenderPresent(ptrToRenderer);
16
17 }
```

Ohjelma 3.2. *Esimerkki päivityssilmukasta.*

Ohjelmassa 3.2 puskureiden vaihto tapahtuu kutsumalla funktiota *SDL_RenderPresent* [36]. Piirron kohteena oleva puskuri on myös tyhjennettävä ennen piirtoa. Tämä suoritetaan asettamalla renderöitäväksi väriksi haluttu taustaväri ja kutsumalla *SDL_RendererClear*-funktiota. [33]

4 MATALAN TASON GRAFIIKKA-API

Ohjelmointi matalan tason grafiikka-API:lla tarkoittaa käytännössä keskustelua GPU:n kanssa. Rajapinnat piilottavat vain laitteistotason yksilöllisyydet ja oikeastaan kaikki muut on ohjelmoijan hallittavissa. Lisäksi laitteisto on aina paljon nopeampi kuin vastaavan toiminnallisuuden ohjelmisto. Ohjelmoimalla mahdollisimman tarkasti laitteiston näkökulmasta myös ohjelman tehokkuus kasvaa huomattavasti.

Luku keskittyy matalan tason rajapinnoista OpenGL:an. Valinta on tehty lähteiden, selkeyden ja omaksuttavuuden perusteella. Koska matalan tason rajapintaa hyödyntävä ohjelma on paljon korkeamman tason vastaavaa pidempi ja monimutkaisempi, on esitetyt ohjelmat pyritty pitämään mahdollisimman yksinkertaisina ja selkeinä. Tästä syystä koodiesimerkkejä ei ole läheskään kaikista suorituksen vaiheista.

4.1 Yleisiä piirteitä matalalla tasolla

Matalan tason grafiikka-API:t vaativat ohjelmoijalta paljon enemmän kuin korkean tason vastaavat. Ohjelmoijan on muun muassa pidettävä itse huolta piirtodatan, esimerkiksi solmujen, säilömisestä erilaisissa puskureissa ja sen lähettämisestä GPU:lle. Vastapainona tämä antaa kuitenkin paljon tarkemmat hallintamahdollisuudet sekä piirtotavan että itse piirrettävien *primitiivien* suhteen.

Primitiivit ovat matalan tason API:eille yleinen ominaisuus. Ne kuvaavat erilaisia yksinkertaisia kuvioita, joita API:n voi määrätä piirtämään piirtodatasta. Tällaisia ovat esimerkiksi pisteet, viivat, kolmiot, monikulmiot ja kolmionauhat (triangle strip) [17, sanasto]. Näistä yleisin on kuitenkin kolmio, joka on yksinkertaisin monikulmio. Lisäksi jos kolmion yhtä pistettä liikutetaan, vain sen asento muuttuu, toisin kuin esimerkiksi neliön tapauksessa, jossa neliö “taittuu”. Kolmioista muodostetaan usein kolmioverkkoja (triangle mesh), joka tarkoittaa käytännössä listaa tai taulukkoa (array).

Matalan abstraktiotason rajapinnoille on yhteistä myös sävyttimet (shader). Sävyttimet ovat pieniä ohjelmia, joita GPU:n lukuisat ytimet suorittavat samanaikaisesti kaikille mallin kulmapisteille tai piirtoikkunan pikseleille [17, kohta 1.1].

Sävyttimien lisäksi API:eilla on tietty *liukuhihna* (pipeline), joka käsittää mallin kulmapisteiden päätymisen välimuistista osaksi piirtoikkunan pikseleitä. Liukuhihna voi esimerkiksi sisältää kulmapisteiden määrittämisen, *kulmapistesävyttimet* (vertex shader), rasteroin-

nin ja *pikselisävyttimet* (pixel shader, fragment shader) [24]. Liukuhihnan tarkoituksena on nopeuttaa datan käsittelyä ottamalla välittömästi uutta dataa liukuhihnan vaiheeseen, kun edellinen on käsitelty sen sijaan, että odotettaisiin datan kulkua kaikkien vaiheiden läpi.

4.2 Eri rajapintoja

Matalan tason rajapinnat ovat lähes poikkeuksetta 3D-rajapintoja. Näistä tunnetuimmat lienevät Direct3D, Metal, Vulkan ja OpenGL. Direct3D on osa Microsoftin DirectX-rajapintaa [20] ja Windowsille kehitetyt graafiset 3D-ohjelmat käyttävät yleensä juuri Direct3D-rajapintaa piirtoon. Metal sen sijaan on Applen kehittämä rajapinta. Metal on tarkoitettu vain Applen tuotteisiin, ja se on toteutettu käyttäen C++14:ta [4].

Vulkanin ja OpenGL:n standardit ovat Khronos Groupin ylläpitämiä [42]. Toisin kuin Direct3D ja Metal, Vulkan ja OpenGL ovat alustariippumattomia, mutta ehkä eniten linux-käyttöjärjestelmissä käytettyjä. Molemmat ovat paljon käytettyjä ja luotettavia, mutta Vulkan on jonkin verran näistä kahdesta monimutkaisempi.

4.3 OpenGL:n erityispiirteitä

Monet OpengGL:n ominaisuudet ovat löydettävissä myös muista rajapinnoista, varsinkin Direct3D muistuttaa paljon OpenGL:a. Kuten muutkin tunnetuimmat rajapinnat, myös OpengGL sisältää liukuhihnan, oman sävytinkielen sekä erilaisia primitiivejä [17, kohdat 1.1 ja 1.3].

OpenGL ei kuitenkaan riitä yksinään tietokoneen ruudulle piirtämiseen. OpenGL ei sisällä itse piirtoikkunaa, vain työkalut sille piirtämiseen. Piirtoikkunan luontiin käytetäänkin erillistä kirjastoa, joka myös yleensä sisältää funktion *piirtokontekstin* luomiseen [14].

Kohdassa 3.1 esiteltiin joukko tähän tarkoitukseen sopivia kirjastoja, joista monipuolisin ja verrattain helppokäyttöisin lienee SDL2.0. SDL-linkin käytännön toteutuksesta kerrotaan lisää kohdassa 4.4.

4.3.1 Renderöinnin liukuhihna

Aikaisemmin mainittiin grafiikka-API:eilla olevan tietty liukuhihna renderöinnnin suhteen. OpenGL:n liukuhihna on 9-osainen, ja se on esitetty kokonaisuudessaan kuvassa 4.1. Aluksi kulmapisteet läheteään kulmapistesävyttimelle, ja suoritetaan sävyttimen sisältämät operaatiot. Tämän jälkeen suoritetaan valinnaiset vaiheet: tessellaatio ja geometriasävytys. Tessellaatiota käytetään tavallisesti yksityiskohtien lisäämiseen lisäämällä kulmapisteiden määrää [17, kohta 8.1]. Geometriasävyttimellä voidaan vähentää tai lisätä

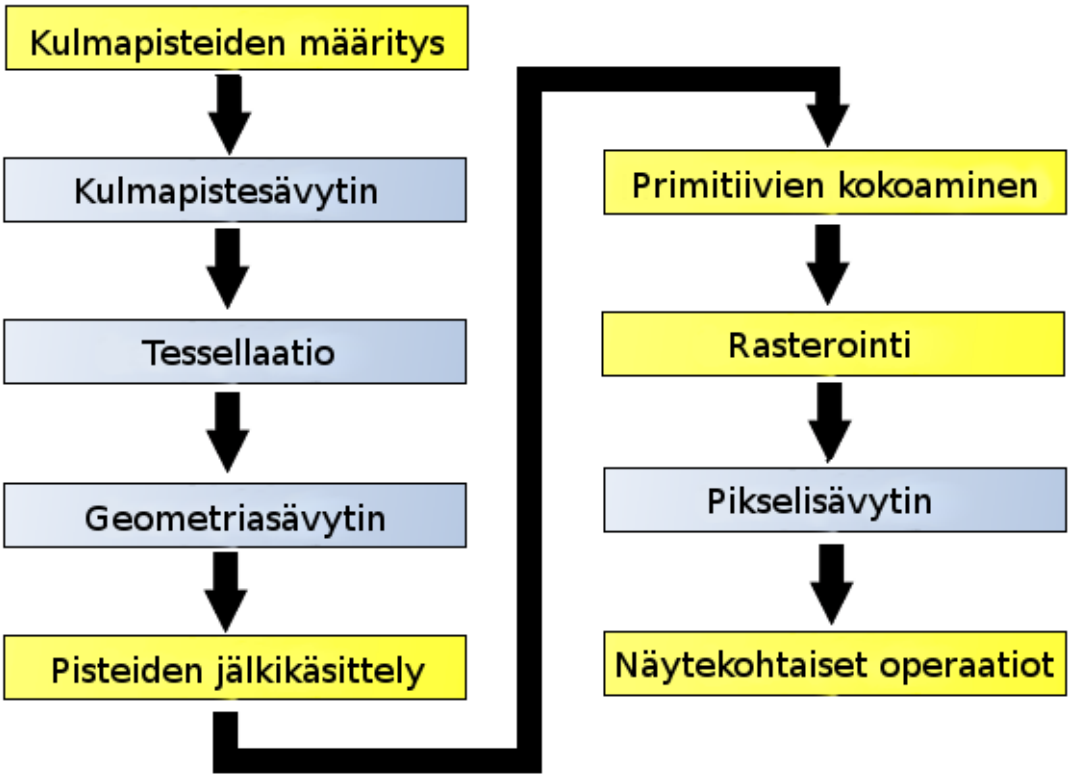
primitiivien määrää sekä muuttaa niitä toisiksi primitiiveiksi, esimerkiksi kolmioista pisteiksi [17, kohta 3.4].

Tämän jälkeen suoritetaan kulmapisteiden jälkikäsittely. Jälkikäsittely sisältää esimerkiksi leikkaukset ja perspektiivijaon, joita käsiteltiin kohdissa 2.6 ja 2.3, sekä transformaation ikkunakoordinaatistoon. [24]

Jälkikäsittelyä seuraa primitiivien kokoaminen. Tässä vaiheessa primitiivit muutetaan lopulliseen muotoonsa. Esimerkiksi jos ohjelmoija käytti piirtämiseen kolmionauhoja, lopputuloksena syntyy kolmioita. Ohjelmoijan valinnat siis vaikuttavat tämän vaiheen suoritustapaan. [24]

Seuraavaksi suoritetaan rasterointi ja *fragmenttien* lähetys pikselisävyttimelle. Rasteroinnissa ikkunakoordinaatistoon transformoidut mallit muutetaan fragmenteiksi, jotka ovat kokoelma erilaista dataa, jota tarvitaan myöhemmin pikseleiden käsittelyssä [13]. Tällaista dataa ovat muun muassa sijainti ikkunakoordinaatistossa sekä edellisten vaiheiden sävyttimiltä tullut data. Kuten tessellaatio ja geometriasävytin, myös pikselisävytin on valinnainen vaihe. [24] Tätä vaihetta harvemmin jätetään kuitenkaan tekemättä. Pikseli- eli fragmenttisävytin muuttaa fragmentit pikselidataksi eli esimerkiksi väriksi ja syvyystiedoksi [24].

Viimeinen renderöintiliukuhihnan vaihe on näytekohtaisten operaatioiden suorittaminen. Käytännössä tämä tarkoittaa joukkoa erilaisia testejä, joilla selvitään, minkä värinen kustakin pikselistä tulee. Testejä ovat omistajuustesti, saksitesti, siluettitesti (stencil test) ja syvyystesti. Omistajuustestissä tarkistetaan, onko fragmentin pikseli OpenGL:n omistama. Jos ei, fragmentti hylätään, sillä sen alueella oleva pikseli voi olla esimerkiksi toisen ikkunan alueella. Saksitesti toimii kuin leikkausalgoritmit, mutta pisteiden sijaan testaakin, mitkä fragmentit ovat piirtoikkunan alueella. Mikäli siluettitesti on asetettu tehtäväksi, testataan onko testin arvo käyttäjän määräämän mukainen verratessa siluettipuskurissa olevaan näytteeseen. Mikäli testi epäonnistuu, fragementti hylätään. Syvyystestissä sen sijaan testataan, mitkä fragmentit peittyvät toisten fragmenttien alle ja voidaan näin karsia pois. [24]



Kuva 4.1. *OpenGL:n piirron liukuhihna, muokattu lähteestä [24]*

Edellä selitetyistä vaiheista kaikki ohjelmoitavat ja muokattavat on merkitty kuvaan 4.1 sinisellä pohjalla. Keltaisella pohjalla merkityt ovat ei-muokattavia (fixed-function) vaiheita, jotka laitteisto eli GPU suorittaa. [24]

4.3.2 GLSL

Erilaisten sävyttimien tekoon OpenGL tarjoaa oman kielensä: GSLS:n (OpenGL Shading Language) [18]. GLSL muistuttaa C-kieltä, mutta sisältää tietokonegrafiikan vaatimia komponentteja. Kuten C, GLSL on käännettävä kieli, mutta kääntäminen tapahtuu vasta OpenGL-ohjelman suorituksen aikana *glCompileShader*-funktiolla [17, kohta 6.2.1]. Lisäksi, C-kielestä poiketen, rekursio on kokonaan kielletty [17, kohta 6.1].

Jotkin tavallisissa kielissä vain luokkina toteutetut datatyypit ovat GLSL:ssa “sisäänrakennettuja”. Esimerkiksi matriisit ja vektorit sisältyvät kieleen. [17, kohta 6.1]. Muita datatyyppejä ovat esimerkiksi etumerkittömät ja etumerkilliset kokonaisluvut (unsigned int ja signed int), jotka molemmat ovat 32-bittisiä, ja liukulukutyypit ovat *float* (32-bittinen) ja *double* (64-bittinen) [18, s. 28, 86].

4.4 Ohjelmointi käytännössä

OpenGL on yksi helpoimmin omaksuttavista 3D-grafiikan rajapinnoista. OpenGL:n vanhemmissa versioissa on ollut nykyistäkin helpompi *kiinteiden funktioiden liukuhihna* (fixed function pipeline). Tällä viitataan tiettyihin aikaisemmin ohjelmoitamattomiin liukuhihnan vaiheisiin, jotka on uudemmissa versioissa korvattu vapaasti muokattavilla sävyttimillä. [11]

Kuten SDL:kin, OpenGL tukee yleisimpiä ohjelmointikieliä. Merkittävimpiä kieliä ovat C, C++ [22] ja muut C:n sukuiset kielet, kuten C# ja Java. Näiden lisäksi tuettuja ovat Ada, Fortran, Haskell sekä tunnetuimmat yleiskäyttöiset skriptikielet, kuten Python, Perl ja Ruby. [19]

4.4.1 Esimerkkiluokan *Window* alustus

Alustus on paljon monimutkaisempi moniin korkean tason rajapintoihin verrattuna. OpenGL:n SDL-linkin kautta tehtävä alustus koostuu kahdesta vaiheesta: *OpenGL-kontekstin luominen* ja *funktioiden lataus*. Molemmat vaiheet vaativat alusta- ja kielikohtaista ohjelmointia, mistä johtuen kielisidoksissa tai apukirjastoissa nämä on abstrahoitu pois. [14]

OpenGL-konteksti tarkoittaa tilakonetta, joka pitää kirjaa renderöintiin tarvittavista tiloista ja datasta. Voidaankin sanoa, että konteksti on OpenGL itsessään. Konteksteja voi myös luoda useampia esimerkiksi yhtä säiettä kohti, mutta yhtä kontekstia ei voi käyttää useammasta säikeestä samanaikaisesti. [21] Tästä johtuen OpenGL soveltunee parhaiten suoritettavaksi yhdellä säikeellä. Tämä ei kuitenkaan estä esimerkiksi käyttäjän syötteiden ja muiden OpenGL:stä irrallaan olevian operaatioiden rinnakkaistamista.

Eri näytönohjainvalmistajat ovat tehneet omat toteutuksensa OpenGL:sta tiettyjä näytönohjaimia varten. Tämä johtaa siihen, että OpenGL-funktio-osoittimet on *ladattava* ennen funktioiden käyttöä alustakohtaisesti. [14] Funktioiden lataukseen suositellaan käytettäväksi apukirjastoa, kuten GLEW:a (OpenGL Extension Wrangler). GLEW on alustariippumaton ja C/C++-yhteensopiva [45], minkä vuoksi se on mainio lisä SDL–OpenGL-ohjelmaan.

Ohjelmaan A.1 on toteutettu OpenGL-versio aiemmin esitetystä ohjelmasta 3.1. Voidaan havaita, että alijärjestelmien alustus ja ikkunan luonti ovat lähes identtiset aikaisempaan versioon verrattuna. Ennen ikkunan luontia on kuitenkin alustettava erilaisia OpenGL:n vaatimia puskureita. Joissakin apukirjastoissa ja kielisidoksissa nämä on piilotettu käyttäjältä abstraktioiden alle [14].

Ohjelmassa A.1 puskureiden ja ikkunan alustusta seuraa aiemmin mainittu OpenGL-kontekstin luominen, mikä tapahtuu SDL:ssa yksinkertaisella funktiolla *SDL_GL_CreateContext* [28]. Tämän jälkeen suoritetaan funktioiden lataus käyttäen GLEW:n omaa alustusfunktiota *glewInit* [46]. Viimeinen vaihe tässä alustusfunktiossa on syvyystestien kytkeminen päälle.

4.4.2 Muu alustus

Luokan *Window* lisäksi sävyttimet ja erilaiset kulmapistepuskurit on myös alustettava. Sävyttimien alustus sisältää sävytinohjelmien kääntämisen ja kulmapisteiden *attribuuttien* indeksien asettamisen datapuskureille kulmapistesävytintä varten. Attribuutteja ovat esimerkiksi kulmapisteiden sijainnit. [17, kohta 3.1]

Kulmapistepuskurit ovat puskureita, joiden sisältö lähetetään GPU:lle käsiteltäväksi. OpenGL:ssa käytetään termiä kulmapistepuskuriobjekti (vertex buffer object, VBO). Termin nimestä voidaan jo havaita, että näiden puskurien alustus tarkoittaa niiden täyttämistä kulmapisteillä tai niihin liittyvällä muulla datalla. Tämän lisäksi OpenGL:lle on kerrottava indeksien siirrokset, mikäli annettu data sisältää pisteiden sijainnin lisäksi jotain muuta dataa [43].

Tässä alaluvussa esitetyt alustukset sisällytetään usein eri luokkiin ohjelmoijan itsetehdyssä koodissa. Näitä luokkia esitellään tarkemmin rajapintojen vertailun yhteydessä kohdassa 5.2.

5 SDL–OPENGL-VERTAILU

Jokaisella grafiikka-API:lla on omat ominaisuutensa, heikkoutensa ja vahvuutensa. Eroavaisuuksia voidaan havaita paljonkin, vaikka rajapinnoilla olisi esimerkiksi samat käyttökohteet. Kun tarkastellaan kahta eri abstraktiotasoilla sijaitsevia rajapintoja, eroavaisuudet kulminoituvat rajapintojen ominaisuuksiin, kirjoitettavan koodin luonteeseen sekä itse rajapinnan käyttöön ja käyttökohteisiin.

Tässä luvussa vertaillaan eri tavoin aikaisemmin esiteltyjä rajapintoja eli SDL:a ja OpenGL:a. Tarkastelussa hyödynnetään myös rajapintojen taustana olevaa tietokonegrafikan teoriaa.

5.1 Ominaisuudet

SDL ja OpenGL eroavat merkittävästi ominaisuuksiltaan. Toisessa on grafiikan lisäksi paljon muutakin funktionaalisuutta ja toinen keskittyy vain renderöintiin. Renderöintiominaisuuksissa on jonkin verran jopa päällekkäisyyksiä, sillä sisäisesti SDL käyttääkin renderöintiin OpenGL:a linux-järjestelmillä, mistä onkin mainintoja esimerkiksi SDL–FAQ-osiossa SDL:n dokumentaatiossa [8] ja eri foorumeilla. Tarkastelemalla lähdekoodia voidaankin havaita, että joissain funktioissa on käytetty OpenGL-funktioita.

Alaluvussa 3.2 kerrottiin muun muassa SDL:n eri alijärjestelmistä ja moduuleista. Siinä missä SDL sisältää moduuleja esimerkiksi äänentoistoon ja verkko-ohjelmointiin, OpenGL sisältää puhtaasti vain renderöintityökalut. OpenGL, kuten kohdassa 4.3 todettiin, ei ota kantaa piirtoikkunan luontiin ja hallintaan, vaan tarvitsee tähän apukirjaston.

SDL ei tietenkään tarvitse apukirjastoa, sillä se sisältää myös ikkunan hallinnan, ja sitä käytetäänkin myös OpenGL:n apuna kohdan 4.3 mukaan. Voidaan sanoa, että SDL on kokonaisen graafisen ohjelman näkökulmasta itsenäinen kirjasto: sillä voidaan toteuttaa esimerkiksi peli, joka käyttää ääniefektejä ja sisältää verkon yli tapahtuvan moninpelin käyttämättä mitään muuta kuin SDL-yhteensopivia moduuleja.

5.1.1 Kolmion renderöinti -esimerkki

SDL itsessään ei kuitenkaan sisällä aivan kaikkea renderöintiin liittyen. OpenGL:ssa yksi tärkeimpiä primitiivejä oleva kolmio tuottaa SDL:ssa vaikeuksia renderöinnin suhteen.

SDL toki mahdollistaa mielivaltaisten viivojen piirron pisteiden välille [34], mutta täytettyä kolmiota ei olekaan triviaalia piirtää. Viivat ja esimerkiksi suorakulmiot kuuluvat SDL:n 2D laitteistokiihdytettyyn renderöinti-API:in, mutta täytetylle kolmiolle sen sijaan ei ole vastaavaa tukea [1]. Tällöin on turvauduttava SDL_gfx-lisäosaaan [10] tai itsetehtyyn ohjelmistopohjaiseen renderöijään.

2-ulotteinen täytetty kolmio saadaan käyttämällä kohdassa 2.5 esiteltyä vaakarivitäyttöä. Voidaan esimerkiksi luoda luokka *Triangle*, joka ottaa parametreikseen kolmion kärkipisteet. Vaakarivitäyttöalgoritmin 1. vaihe suoritetan näiden pisteiden perusteella. 3D-kolmiokin on mahdollista renderöidä käyttäen pohjana SDL:a ja erilaisia algoritmeja. Kolmion asentoa on pystyttävä muuttamaan, jolloin tarvitaankin jo kohdassa 2.3 esitettyjä homogeenisen koordinaatiston transformaatioita.

Luvussa 2 esitetyn teorian perusteella voitaisiin toteuttaa ohjelmistopohjainen renderöijä, joka osaisi myös piirtää esimerkiksi Blenderilla tehtyjä 3D-mallinnuksia. Aikaisemmin mainittuun kolmiorenderöijään tarvittaisiin kuitenkin vielä muutamia lisäominaisuuksia, kuten kohdan 2.6 leikkausalgoritmeja ja tämän työn ulkopuolelle jääviä tekniikoita.

OpenGL:ssa kolmion renderöinti on ohjelmoijalle täysin triviaalia. Tämä johtuu siitä, että kolmio on OpenGL:n yksi primitiiveistä [17, kohta 1.3]. OpenGL ei kuitenkaan osaa piirtää kolmiota ilman aikaisemmin mainittuja täyttöä ja leikkausta. Grafiikkasuoritinvalmistajat lienevätkin toteuttaneet OpenGL:an vastaavia algoritmeja.

5.1.2 Yhteensopivuus

Sekä OpenGL että SDL ovat alustariippumattomia, kuten kohdissa 3.2 ja 4.2 mainittiin. Molemmat ovat yhteensopivia tunnetuimpien käyttöjärjestelmien kanssa. Perustuen kohtiin 3.2 ja 4.4 tuetut ohjelmointikieletkin ovat samoja muutamaa poikkeusta lukuunottamatta. OpenGL:lla tuettuja kieliä on kuitenkin enemmän.

Molemmista on olemassa eri versioita, jotka eivät välttämättä kuitenkaan ole yhteensopivia alustan kanssa. Esimerkiksi OpenGL:n uudemmat versiot eivät tietenkään ole tuettuja vanhemmilla järjestelmillä.

5.2 Ohjelmointi

Ohjelmoinnin ero SDL:n ja OpenGL:n välillä on ilmeinen. Koska OpenGL on matalamman tason API, on selvää, että ohjelmakoodi on monimutkaisempaa ja funktiot vähemmän abstrakteja.

Ohjelmoinnin erojen tarkastelu on hajoitettu kahteen osaan: käytännön ohjelmoinnin vertailuun ja luokkajakoon. Käytännön ohjelmoinnin vertailussa tarkastellaan aiemmin työssä esiteltyjä alustusfunktioita rinnakkain ja nimetään suurimmat erot päivityssilmukoissa.

Luokkajako-osiossa tarkastellaan tarkemmin kutakin rajapintaa käyttävän ohjelman rakenteellisia vaatimuksia.

5.2.1 Alustus ja päivityssilmukka

Kuten kohdissa 4.4 ja 3.2 kerrottiin, OpenGL ja SDL ovat C- ja C++-kielten kanssa yhteensopivia. Kuitenkin näiden rajapintojen funktiot eroavat merkittävästi. Tarkastelemalla alustusta käsitteleviä ohjelmia 3.1 ja A.1 voidaan havaita, että pääpiirteittäin alustus on samanlainen. OpenGL:n tapauksessa alustukseen kuuluu myös puskureiden alustus ja SDL:ssa funktioiden lataus on korvattu tyyppiä SDL_Renderer olevan tietuen alustuksella. OpenGL:n alustukseen sisältyy myös kontekstin luominen, jota SDL:ssa ei ohjelmoijan tarvitse tehdä.

Alustuksien vähäinen eroavaisuus on kuitenkin olemassa vain luokassa *Window*. OpenGL nimittäin vaatii myös sävyttimien ja erilaisten kulmapistepuskurien alustuksen, mitkä selitettiin kohdassa 4.4.2.

Päivityssilmukan, jossa mallien ja kappaleiden renderöinti tapahtuu, osalta OpenGL on myös paljon monimutkaisempi. Ohjelman 3.2 mukaan SDL:n päivityssilmukka sisältää tiivistetysti vain funktiokutsut piirtoikkunan siivoamiselle, kappaleiden renderöimiselle puskuriin ja itse ikkunan päivittämiselle.

OpenGL sen sijaan vaatii sävyttimien aktivoinnin funktiolla *glUseProgram* [44] ja kulmapisteiden attribuuttien, joista kerrottiin kohdassa 4.4.2, lähettämisen GPU:lle. Mikäli halutaan esimerkiksi 1. persoonan vaikutelma, on myös tehtävä erilaisia transformaatioita ja koordinaattimuunnoksia malleille. Näitä käsiteltiin luvussa 2.

5.2.2 Ohjelman hajottaminen luokiksi

Mikäli ohjelman on tarkoitus tehdä muutakin kuin renderöidä yksi kolmio, on järkevää hajottaa ohjelma eri luokiksi. SDL:n tapauksessa luokiksi muodostuvat *Model*, *Input*, *Window* ja *Control*. Luokka *Model* sisältää kaiken informaation piirrettävästä mallista: koon, värin ja sijainnin. *Model* sisältää tietenkin myös piirtofunktion. Luokka *Input* käsittelee nimensä mukaisesti käyttäjän syötteitä, joihin kuuluvat ohjelman sulkemisen komennot sekä mallin liikuttaminen, mikäli se on ohjelmassa tarkoitettua. *Window* on sama kuin aikaisemminkin esitelty luokka *Window*. Sen tehtävä on abstrahoida SDL:n ikkunan hallinta. *Control* on ikään kuin alusta, jolla muut operaatiot tapahtuvat ja sisältääkin muiden luokkien alustamisen sekä itse päivityssilmukan.

OpenGL-versio vastaavasta sisältää muutaman muunkin luokan. SDL-ohjelman luokkien lisäksi tarvitaan *Mesh*, *Shader* ja *Camera*. *Shader* vastaa sävytinohjelmien kääntämisestä ja suorittamisesta. Luokan tärkeimmät funktiot ovat alustusfunktioiden lisäksi funktiot perspektiivin asetukselle ja sävyttimen sitomiselle (bind) eli aktivoinnille. Luokan *Shader* toimintaa päivityssilmukassa kuvattiin kohdassa 5.2.1. *Shader* toimii siis linkkinä

OpenGL-ohjelman ja sävytinohjelmien välillä. Objekteja *Shader* voidaan luoda esimerkiksi yksi kappale jokaista mallia kohden.

Luokka *Mesh* suorittaa kulmapistepuskurien alustuksen, mikä mainittiinkin jo kohdassa 4.4.2. Luokan *Model* tavoin myös *Mesh* sisältää piirtofunktion. Erona on, että *Model* sisältää luokan *Mesh* piirtofunktion ja sävyttimien sitomisen tälle mallille sekä kulmapisteiden attribuuttien lähettämisen sävytinohjelmille. Luokan *Mesh* piirtofunktio taas sisältää itse kulmapisteiden sitomisfunktion *glBindVertexArray* ja OpenGL-piirtofunktion *glDrawArrays*. *Model* voi omistaa yhden tai useamman objektin *Mesh*. Näin saadaan esimerkiksi kokonainen maisema ladattua yhdeksi malliksi.

Mainituista OpenGL-ohjelman luokista *Camera* on jokseenkin valinnainen. Kolmiota piirrettäessä tätä luokkaa ei tarvita, mutta mikäli näkymää halutaan muuttaa, tarvitaan *Camera* havainnollistamaan ja päivittämään näkökulmaa käyttäjän syötteiden perusteella. Myös SDL-ohjelmaan voidaan lisätä jonkinlainen kamera-luokka, mutta tällöin on luultavasti tarkoitus jo tehdä ohjelmistopohjainen 3D-renderöijä.

5.3 Käyttö ja käyttökohteet

SDL tarjoaa erinomaiset mahdollisuudet laitteistokiihdytetyn 2D-grafiikan piirtoon ja hallintaan. Tästä syystä se sopiikin erilaisten 2D-pelien, kuten *Space Invaders* tai *Demonstar*, tekemiseen. SDL on myös hyvin dokumentoitu ja tutoriaalisarjoja löytyy esimerkiksi Youtubesta. API:n dokumentaationa voi käyttää esimerkiksi SDL Wikiä [38].

Kuitenkaan graafisten käyttöliittymien tekemiseen SDL ei sovellu. SDL tarjoaa vain viivojen, pisteiden ja nelikulmioiden sekä erilaisten tekstuurien piirtomahdollisuuden, mutta ei näitä hyödyntäviä helppokäyttöisiä luokkia. Käyttöliittymien teko on siis mahdollista, mutta erittäin työlästä. Tähän tarkoitukseen aikaisemmin mainitut GTK+ ja erityisesti Qt sopivat paremmin.

Kuten SDL 2.0 Wikissa ja aikaisemmissakin luvuissa kerrotaan, SDL:sta itsestään ei löydy tarvittavaa funktionaalisuutta 3D-grafiikkaan [16]. Rinnalle vaaditaan jokin matalamman tason rajapinta käsittelemään 3D-dataa, jolloin SDL toimii piirtoalustana ja hoitaa piirtoikkunan käsittelyn. SDL sopiikin hyvin esimerkiksi OpenGL:n kanssa käytettäväksi, jolloin muodostuu kohdan 4.3 mukainen linkki. Tällöin SDL hoitaa ikkunan ja käyttäjän syötteiden hallinnan ja OpenGL itse renderöinnin.

OpenGL sen sijaan on monipuolinen renderöimiseen ja verrattain helppo omaksua. Varsinkin linux-käyttöjärjestelmälle OpengGL:n vaatiman ohjelmointiympäristön ja eri kirjastojen asentaminen on hyvin dokumentoitu esimerkiksi erilaisten tutoriaalien muotoon. Tästä syystä OpenGL tarjoaakin erinomaiset mahdollisuudet 3D-grafiikasta kiinnostuneelle, vaikka aikaisempaa taustaa grafiikkaohjelmoinnista ei olisikaan.

Linux-käyttöjärjestelmille suunnattujen ohjelmien renderöinnissä käytetään usein juuri OpenGL:a. Näistä esimerkkeinä ovat 3D-mallinnusohjelma Blender [2] ja monien pelien

linux-toteutukset.

Vaikka OpenGL tunnetaan 3D-rajapintana, voi sitä silti käyttää 2D-grafiikkaan. 2D-grafiikkaan on toki olemassa helppokäyttöisempiäkin kirjastoja, mutta OpenGL mahdollistaa samat asiat kuin 3D-grafiikassakin: tarkan hallinnnan ja näyttävät efektit. Samalla ohjelmoijan vastuu ja työmäärä kasvavat.

5.4 Kooste

Vertailun tulosten ja rajapintojen erillisten tarkastelujen havainnollistamiseen on tehty kooste, jossa eroja tarkastellaan eri osa-alueiden mukaan. Tämä kooste on nähtävillä taulukossa 5.1.

Taulukko 5.1. *kooste SDL:n ja OpenGL:n eroista.*

	SDL	OpenGL
Tuetut kielet	Rust, Lua, Pascal, Python, C#, C, C++	C, C++, C#, Java, Ada, Fortran, Haskell, Python, Perl, Ruby
Tuetut alustat	Windows, linux, Mac OS, Android	Windows, linux, Mac OS, Android
Vastaavia rajapintoja	GTK+, Qt	Metal, Direct3D, Vulkan
Ominaisuudet renderöinnin lisäksi	verkko, ääni, kuvanlataus, fontit	-
Abstraktiotaso	melko matala	matala
Alustuksen koko	35 riviä	49 riviä (apukirjastojen avulla)
Tuetut perusprimitiivit	neliöt, viivat, pisteet	neliöt, viivat, pisteet, kolmiot
Käyttökohdeohjelmat	2D-pelit	Blender, 3D-pelit

Taulukkoon 5.1 on koottu erot muun muassa tuetuista kielistä ja alustoista, vastaavanlaisista rajapinnoista, lisäominaisuuksista, alustuksen koosta koodiriveinä sekä saatavilla olevista primitiiveistä ja käyttökohteista. Perusprimitiiveillä tarkoitetaan tässä primitiivejä, jotka eivät muodostu toisista primitiiveistä. Tämä sulkee pois esimerkiksi kolmionauhat, jotka voidaan tietenkin muodostaa kolmioista.

6 YHTEENVETO

Työn tarkoituksena oli perehtyä kahteen eri rajapintaan, joista kumpikin sijaitsivat eri abstraktiotasoilla. Tällä pyrittiin luomaan kaksi erilaista näkökulmaa grafiikkaohjelmointiin. Käsiteltäviä piirteitä olivat erot muihin vastaaviin API:eihin, tuetut kielet sekä itse ohjelmointi ja syntaksi. Tämän jälkeen rajapintoja vertailtiin keskenään. Vertailun tukena käytettiin työn lukua 2, jossa esitettiin erilaisia algoritmeja ja tekniikoita, joita käytetään rajapinnoissa sisäisesti tai rajapintaa käyttävän ohjelmoijan hyödyntämänä.

Eroja löytyi varsin paljon jo siitäkin syystä, että verrattavat rajapinnat olivat erityyppisiä: toinen sekä renderöintiin että muuhun toiminnallisuuteen tarkoitettu ja toinen pelkästään renderöintiin. Ohjelmointi OpenGL:lla on suhteellisen tehotonta johtuen sen monimutkaisuudesta, mutta toisaalta se kykenee vaativampaan renderöintiin kuin SDL. Erot esimerkiksi tuettujen kielten määrässä johtunevat yksinkertaisesti rajapinnan suosiosta. OpenGL on nimittäin huomattavasti tunnetumpi kuin SDL ja tukeekin enemmän kieliä. Abstraktiotaso vaikuttaa myös paljon eroihin, mutta renderöintiominaisuuksien laajuuden kautta.

Rajapintoja käsitteleviä lähteitä löytyi melko paljon. Nämä kuitenkin olivat yleensä osa API:n dokumentaatiota, kuten SDL Wiki [38] ja OpenGL:n viitesivut [44], tai rajapinnasta ja sen käytöstä kertova erillinen kirja, kuten *OpenGL superbible: comprehensive tutorial and reference* [17]. Teoriaa käsiteltiin sekä yleisteoksen, kuten kirjan *3D-grafiikka* [23], että yksittäisistä algoritmeista kertovien tieteellisten julkaisujen avulla. Yleensä muut kuin tieteelliset julkaisut saattavat olla epäluotettavia, mutta koska API:t ovat paljon käytettyjä, voidaan esimerkiksi dokumentaatioon luottaa.

Muita SDL–OpenGL-vertailuja on saatavilla pääasiassa internetissä. Vertailuja voi löytää erilaisilta foorumeilta, joilla on myös muiden vastaavien rajapintojen välisiä vertailuja. Tieteellisiä kirjoituksia aiheesta ei kuitenkaan juuri löydy.

Tutkimusta voisi edistää huomattavasti erilaisten testiympäristöjen avulla ja selvittää rajapintojen tehokkuuksia esimerkiksi alustusnopeuden ja piirtoikkunan maksimipäivitystaajuuden avulla vertaamalla näitä renderöitävien primitiivien määrään. Tämä ei ole kandidaatintyön rajoissa mielekästä, ja työ luultavasti kelpaisikin tällöin jo diplomityöksi.

LÄHTEET

[1] *2D Accelerated Rendering*. URL: https://wiki.libsdl.org/CategoryRender (viitattu 14. 08. 2019).

[2] *About [Blender]*. URL: https://www.blender.org/about/ (viitattu 14. 08. 2019).

[3] *About Qt*. 26. toukokuuta 2019. URL: https://wiki.qt.io/About_Qt (viitattu 04. 07. 2019).

[4] Apple inc. *Metal shading language specification*. Versio 2.2. 29. toukokuuta 2019. URL: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf (viitattu 03. 08. 2019).

[5] J. F. Blinn ja M. E. Newell. Clipping using homogenous coordinates. *ACM SIGGRAPH Computer Graphics* 12 (3 1978), 245–251.

[6] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4 (1 1965), 25–30.

[7] A. Coopersmith. *X Window System Concepts*. 14. heinäkuuta 2013. URL: https://www.x.org/wiki/guide/concepts/#xisclientserver (viitattu 07. 07. 2019).

[8] *FAQ: Development*. URL: https://wiki.libsdl.org/FAQDevelopment (viitattu 15. 08. 2019).

[9] ferzkopp. *SDL_gfx / SDL2_gfx*. 2. tammikuuta 2016. URL: http://www.ferzkopp.net/wordpress/2016/01/02/sdl_gfx-sdl2_gfx/ (viitattu 08. 07. 2019).

[10] ferzkopp. *I:/Sources/sdl2gfx/SDL2_gfxPrimitives.c File Reference*. URL: http://www.ferzkopp.net/Software/SDL2_gfx/Docs/html/_s_d_l2__gfx_primitives_8c.html#aa828ac3c3fe5ce610c5ad1516dfb6f3f (viitattu 13. 08. 2019).

[11] *Fixed Function Pipeline*. 10. huhtikuuta 2015. URL: https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline (viitattu 09. 08. 2019).

[12] J. D. Foley, A. van Dam, S. K. Feiner ja J. F. Hughes. *Computer graphics: principles and practice*. 2. painos. Addison-Wesley, 1997.

[13] *Fragment*. Englanti. 7. lokakuuta 2017. URL: https://www.khronos.org/opengl/wiki/Fragment (viitattu 04. 08. 2019).

[14] *Getting started*. 4. heinäkuuta 2019. URL: https://www.khronos.org/opengl/wiki/Getting_Started (viitattu 03. 08. 2019).

[15] A. Glassner. Fill 'Er Up! *IEEE Computer Graphics and Applications* 21 (2001), 78–85.

[16] *Introduction to SDL 2.0*. URL: https://wiki.libsdl.org/Introduction (viitattu 04. 07. 2019).

[17] R. S. W. Jr., G. Sellers ja N. Haemel. *OpenGL superbible: comprehensive tutorial and reference*. 7. painos. USA Boston: Addison-Wesley Professional, 2015.

[18] J. Kessenich, D. Baldwin ja R. Rost. *The OpenGL Shading Language*. 9. toukokuuta 2017. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf (viitattu 05. 08. 2019).

[19] *Language bindings*. 7. kesäkuuta 2018. URL: https://www.khronos.org/opengl/wiki/Language_bindings (viitattu 08. 08. 2019).

[20] Microsoft. *Where is the DirectX SDK?* 31. toukokuuta 2018. URL: https://docs.microsoft.com/en-us/windows/win32/directx-sdk--august-2009- (viitattu 07. 07. 2019).

[21] *OpenGL Context*. 17. syyskuuta 2018. URL: https://www.khronos.org/opengl/wiki/OpenGL_Context (viitattu 09. 08. 2019).

[22] *OpenGL Overview*. URL: https://www.opengl.org/about/ (viitattu 09. 08. 2019).

[23] A. Puhakka. *3D-grafiikka*. Helsinki: Talentum, 2008.

[24] *Rendering Pipeline Overview*. 8. huhtikuuta 2019. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (viitattu 03. 08. 2019).

[25] *SDL 1.2 to 2.0 Migration Guide*. URL: https://wiki.libsdl.org/MigrationGuide (viitattu 10. 07. 2019).

[26] *SDL 2.0 API by Category*. URL: https://wiki.libsdl.org/APIByCategory (viitattu 08. 07. 2019).

[27] *SDL Language Bindings*. URL: https://libsdl.org/languages.php (viitattu 11. 07. 2019).

[28] *SDL_GL_CreateContext*. URL: https://wiki.libsdl.org/SDL_GL_CreateContext (viitattu 18. 08. 2019).

[29] *SDL_image 2.0*. 19. kesäkuuta 2019. URL: https://www.libsdl.org/projects/SDL_image/ (viitattu 15. 08. 2019).

[30] *SDL_Init*. URL: https://wiki.libsdl.org/SDL_Init (viitattu 15. 08. 2019).

[31] *SDL_mixer 2.0*. 26. lokakuuta 2018. URL: https://www.libsdl.org/projects/SDL_mixer/ (viitattu 15. 08. 2019).

[32] *SDL_net 2.0*. 3. tammikuuta 2016. URL: https://www.libsdl.org/projects/SDL_net/ (viitattu 15. 08. 2019).

[33] *SDL_RenderClear*. URL: https://wiki.libsdl.org/SDL_RenderClear (viitattu 18. 08. 2019).

[34] *SDL_RenderDrawLine*. URL: https://wiki.libsdl.org/SDL_RenderDrawLine (viitattu 14. 08. 2019).

[35] *SDL_Renderer*. URL: https://wiki.libsdl.org/SDL_Renderer (viitattu 15. 08. 2019).

[36] *SDL_RenderPresent*. URL: https://wiki.libsdl.org/SDL_RenderPresent (viitattu 15. 08. 2019).

[37] *SDL_ttf 2.0*. 26. lokakuuta 2018. URL: https://www.libsdl.org/projects/SDL_ttf/ (viitattu 15. 08. 2019).

[38] *Simple DirectMedia Layer*. URL: https://wiki.libsdl.org/FrontPage (viitattu 16. 08. 2019).

[39] The GIMP Team. *GIMP*. URL: https://www.gimp.org/ (viitattu 04. 07. 2019).

[40] The GTK Team. *API Documentation*. URL: https://www.gtk.org/documentation.php (viitattu 22. 09. 2019).

[41] The GTK Team. *What is GTK, and how can I use it?* URL: https://www.gtk.org/ (viitattu 15. 08. 2019).

[42] The Khronos Group inc. *Khronos Membership*. 2019. URL: https://www.khronos.org/members/ (viitattu 15. 08. 2019).

[43] The Khronos Group inc ja 3Dlabs Inc. *glVertexAttribPointer*. 2014. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribPointer.xhtml (viitattu 18. 08. 2019).

[44] The Khronos Group inc ja 3Dlabs Inc. *Khronos Membership*. 2014. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/ (viitattu 16. 08. 2019).

[45] *The OpenGL Extension Wrangler Library*. URL: http://glew.sourceforge.net/ (viitattu 09. 08. 2019).

[46] *The OpenGL Extension Wrangler Library, Initializing GLEW*. URL: http://glew.sourceforge.net/basic.html (viitattu 13. 08. 2019).

A OHJELMAT

```
1  bool Window::init() {
2
3      // init subsystems
4      if ( SDL_Init(SDL_INIT_EVERYTHING) != 0) {
5          std::cerr << "SDL: "
6                    << SDL_GetError()
7                    << "\n";
8          return false;
9      }
10
11     // init buffers
12     SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 8);
13     SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
14     SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
15     SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 8);
16
17     SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE, 32);
18     SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 16);
19     SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
20
21     mWindow = SDL_CreateWindow(TITLE.c_str(),
22                                SDL_WINDOWPOS_CENTERED,
23                                SDL_WINDOWPOS_CENTERED,
24                                WIDTH,
25                                HEIGHT,
26                                SDL_WINDOW_OPENGL);
27
28     if (mWindow == nullptr) {
29         std::cerr << "SDL: "
30                   << SDL_GetError()
31                   << "\n";
32         return false;
33     }
34
```

```
35      mContext = SDL_GL_CreateContext(mWindow);
36
37      // function loading
38      GLenum error = glewInit();
39      if (error != GLEW_OK) {
40          std::cerr << "GLEW: "
41                    << glewGetErrorString(error)
42                    << "\n";
43          return false;
44      }
45
46      glEnable(GL_DEPTH_TEST);
47
48      return true;
49  }
```

Ohjelma A.1. *Esimerkki Window-luokan alustuksesta OpenGL:lla.*