jamk.fi

Näytönohjain CPU:n laskennan suorittajana

Markus Järvinen

Opinnäytetyö
Marraskuu 2015
Tekniikan ja liikenteen ala
Insinööri (AMK), Ohjelmistotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Järvinen, Markus	Opinnäytetyö, AMK	24.11.2015
	Sivumäärä 25	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty:

Työn nimi
Näytönohjain CPU:n laskennan suorittajana

Tutkinto-ohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
Ari Rantala

Toimeksiantaja(t)

Tiivistelmä

Näytönohjaimia käytetään perinteisesti 2D- tai 3D-grafiikan piirtämiseen, mutta nykyisin näytönohjaimilla voidaan suorittaa grafiikan piirron lisäksi muutakin laskentaa. Tätä näytönohjaimen laskentakapasiteetin hyödyntämistä kutsutaan GPGPU-tekniikaksi ja sitä voidaan käyttää mm. OpenCL-ohjelmointirajapinnan avulla. Näytönohjaimet ovat keskusprosessoreihin verrattuna hyvin tehokkaita rinnakkaislaskennassa. GPGPU-tekniikkaa on hyödynnetty piirtämisen apuvälineenä esim. peleissä piirrettävien kohteiden tietoja päivittämiseen näytönohjaimessa.

Työn tavoitteena oli selvittää näytönohjaimen hyödyntämistä keskusprosessorin rinnalla tietokonepelien yhteydessä. Opinnäytetyössä luotiin pelimoottorin pohja, jonka avulla luotiin vokselipohjainen simulaatio. Simulaatio toteutettiin OpenCL-ohjelmointirajapinnan avulla näytönohjaimelle sopivaksi suunniteltuna. Lisäksi verrattiin näytönohjaimesta saatuja tuloksia keskusprosessorilla suoritettavaan vastaavan toteutukseen. Opinnäytetyöllä ei ollut toimeksiantajaa vaan kyseessä oli henkilökohtainen tutkimusprojekti GPGPU-tekniikan hyödyntämisestä.

Toteutus kirjoitettiin C++-kielellä Ubuntu Linux -käyttöjärjestelmäympäristössä. Toteutuksessa hyödynnettiin myös keskusprosessorin monisäietukea.

Simulaation avulla ei onnistuttu keräämään suoritukseen liittyviä tuloksia, joten toteutus jäi siltä osin vajaaksi. Tiedonsiirroista saadut tulokset rajoittuivat vain yhteen näytönohjaimeen, mutta siitä saadut tiedot toimivat kohtuullisen hyvin suunnan antajina näytönohjaimen hyödyntämisessä. Opinnäytetyön tulokset osoittavat, että tiedonsiirto näytönohjaimelle on hidasta näytönohjaimen tarjoamaan muistimäärään nähden.

Tulosten avulla pääteltiin, että painopisteenä tulisi olla GPGPU-tekniikan hyödyntäminen näytönohjaimessa jo olevalle datalle ja tiedonsiirtojen minimointi näytönohjaimen ja tietokoneen keskusmuistin välillä.

Suurien datakokoelmien käsittely näytönohjaimella kestäisi liian kauan, mikäli tulokset tulisi siirtää pois näytönohjaimelta käsittelyn jälkeen. Nämä tiedonsiirrot laskisivat kuvanpiirtämisten määrän alle 30 kuvanpiirtoon sekunnissa, jota ei voida hyväksyä monissa nykyajan peleissä.

Avainsanat
näytönohjain, vokseli, prosessori, OpenCL, OpenGL, moniydinprosessori

Muut tiedot

jamk.fi

Description

Author(s) Järvinen, Markus	Type of publication Bachelor's thesis	Date 24.11.2015
	Number of pages 25	Language of publication: finnish
		Permission for web publication:

Title of publication
Applying CPU computing for GPU

Degree programme
Software Engineering

Supervisor(s)
Rantala, Ari

Assigned by

Description

Graphics processing units are normally used for drawing 2D- or 3D-graphics; however, nowadays graphics processing units can be utilized for different kinds of computing besides graphics rendering. This type of graphics processing unit computing is referred to as GPGPU and it can be implemented with the usage of APIs like OpenCL. Graphics processing units are more efficient compared to central processing units when it comes to parallelized computing. GPGPU has been utilized in conjunction with rendering in games where the statuses of drawn entities are updated through the usage of GPGPU.

The main goal of the thesis was to study utilizing GPGPU alongside the central processing unit in video games. For thesis purposes a simple game engine was build which was used to create voxel based simulation. The simulation was adapted for it to be run on graphics processing units with the OpenCL API. The performance results received from the usage of GPGPU were to be compared to the results received from central processing unit which could also run the simulation. The thesis did not have an assigner(s) and was a personal research project on utilizing GPGPU.

Implementation was done with the C++ language and the operating system used was Ubuntu Linux. Central processing unit's multithreading capabilities were utilized during the development.

Results could not be measured through the simulation which left parts of the goals of the thesis unobtained. Results received from data transfers were limited for single graphics processing unit; however, the results gained from it could be used as guidelines when using GPGPU. The results showed that data transfers are slow compared to the memory size offered by the graphics processing unit.

Through the results it was deduced that when using GPGPU the implementations should focus using the data residing within the graphics processing unit and minimize data transfers between central processing unit and graphics processing unit.

Keywords (subjects)
graphics processing unit, central processing unit, GPGPU, OpenCL, OpenGL

Miscellanous

Taulukot

1 Johdanto

Pelit ovat maailmanlaajuisesti erittäin merkittävä bisnes ja pelien yleistyminen, lisääntyminen ja monipuolistuminen mobiililaitteilla, kuten älypuhelimilla (Android, iPhone, Windows Phone) sekä tableteilla, perinteisten pelikonsoleiden (Nintendo, Playstation, Xbox) sekä tietokoneiden rinnalle, ovat nostaneet pelien merkityksen merkittäväksi viihdeteollisuudenalaksi ympäri maailman. Peliteollisuus Suomessakin on jo noussut miljardin euron liikevaihdon tasolle ja kasvun lisääntymistä on jo ennustettu. (S. Suvanto, 2014)

Erilaisia pelityyppejä ovat esimerkiksi pulma-, seikkailu-, toiminta-, ammunta-, rooli-, tasoloikka- ja strategiapelit. Pelityyppien välillä on yleensä myös paljon eroa pelilaitteen tehovaatimuksissa esim. pulmapelit voivat olla tehovaatimuksiltaan yleensä melko vaatimattomia, kun taas toiminta– sekä strategiapelit voivat vaatia paljon tehoja fysiikan, keinotekoisen älyn ja pelin graafisen ulkoasun vuoksi.

Normaalisti pelien toimintatapa on pohjautunut yksisäikeiseen toimintaan. Sen tehokas toimintakyky joko pohjautuu out-of-order suoritukseen (Out-of-Order Execution, OoOE), jossa suoritettavia komentoja pyritään suorittamaan siten, että kun komentojen vaatimat tiedot ovat valmiita käytettäväksi ja riippuvuuksia tietoihin liittyen ei ole jäljellä, voidaan kyseiset komennot suorittaa samalla kun aikaa vievät edelliset komennot ovat suoritettavina. OoOE tehokkuus riippuu prosessorin laskukyvystä sekä itse koodin riippuvaisuuksista edellisiin komentoihin. Pullonkauloja voi esiintyä liikaa koodissa, mitkä haittaavat OoOE suoritusta.

Toinen vaikuttava tekijä yksisäikeisten pelien toimintaan on prosessorin taajuus, mikä kääntyy nopeudeksi. Koska taajuutta ei voi nostaa jatkuvasti ylöspäin laskentatehon kasvattamiseksi laitteiston vikaantumisen vuoksi, kuten tietokoneen materiaalien fyysisten vaurioiden takia esim. komponenttien lämpövaurioiden takia, siirryttiin moniydinprosessoreiden käyttöönottoon ja hyödyntämiseen. Laskennan jakaminen mahdollistaa nopeamman suorituksen sekä helpommin skaalattavan ja tehon nostattavan laitteiston, mutta vaatii

enemmän tiedon synkronisointia komentosarjojen välissä, jotta tiedot pysyisivät samana komentojen eri vaiheissa. Moniydinprosessoreiden hyödyntämistä on myös yritetty helpottaa tarjoamalla niiden hyödyntämiseen tarkoitettuja kirjastoja kuten OpenMP sekä Intelin Intel TBB (Thread Building Blocks), jotka helpottavat olemassa olevan koodin kääntämistä moniydinprosessoreille pienemmällä vaivalla. Nykyajan pelit ovat soveltaneet useampaa prosessoriydintä toteutuksessaan, vaikkakin tuki kehittäjien puolelta on ollut vaihtelevaa.

Moniydinprosessoreiden lisäksi on myös kehitetty näytönohjaimia hyödyntäviä GPGPU (General-purpose computing on graphics processing units)-ohjelmointirajapintoja, joilla pyritään hyödyntämään näytönohjaimien laskentakykyä muussa kuin 3D-grafiikan piirrossa. Tietokonepelien yhteydessä GPGPU-tekniikka on painottunut visuaalisten efektien simulointiin savusta partikkeleihin, mutta kyseiset efektit eivät vaikuta itse pelin toiminnallisuuteen ollenkaan. Tämän työn tarkoituksena oli pyrkiä selvittämään GPGPU-tekniikan käyttöä tietokonepelien toiminnallisuuden vaikuttamiseen sekä sen tehokkuuden ja hyödyllisyyden selvittämiseen niin itsenäisesti kuin prosessorin rinnallakin sekä siihen liittyviä ongelmia ja vaatimuksia.

2 Tietoperusta

2.1 *Muistityypit*

2.1.1 RAM

RAM (Random-Access Memory) on tietokoneen ajon aikana käytettävissä oleva muisti. RAM:iin voidaan ohjelmien suorittamisen aikana tallentaa ja lukea tietoa, joka on paljon nopeampaa, kuin kovalevyiltä lukeminen ja niihin kirjoittaminen ohjelmien ajon aikana. Tietokoneohjelmien tulisi käyttää RAM:ia sillä prosessorien sisäänrakennetut muistimäärät ovat todennäköisesti liian vähäiset ohjelman tarvittavaan ja nopeasti saatavaan ja käsiteltävään tietomäärän nähden.
Yleisin RAM-tyyppi on DRAM (Dynamic Random-Access Memory) pohjainen DDR SDRAM.

2.1.2 VRAM

VRAM (Video Random-Access Memory) on näytönohjaimissa sovellettu muistityyppi. Nykyisten näytönohjainten VRAM-tyyppi on GDDR, joka pohjautuu DDR SDRAM muistiin. VRAM:in määrää ei voi kasvattaa normaalin RAM:in tapaisesti, sillä se on pysyvästi liitettynä näytönohjaimeen.

2.2 *Näytönohjain*

2.2.1 GPU

GPU (Graphics Processor Unit) on näytönohjain, jonka pääkäyttötarkoitus on grafiikan piirtämisessä näyttölaitteille. Näytönohjaimet painottuvat rinnakkaiseen laskentaan toteutuksessaan, mikä mahdollistaa tehokkaan toteutuksen grafiikan piirtämiseen.
Grafiikan piirtämiseen sovellettavia laitteita oli jo olemassa, mutta ne olivat erikoistuneita vain tiettyjen grafiikan piirtämisen osien hyödyntämiseen. Vasta v. 1999 Nvidia ja ATI julkaisivat kuluttajakäyttöön nykytermin mukaiset

näytönohjaimet, joissa kaikki grafiikan piirtoon liittyvät toiminnot tapahtuvat näytönohjaimessa. (C. McClanahan, 2011)

2.2.2 GPGPU

GPGPU tarkoittaa näytönohjaimen soveltamista yleiskäyttöisen laskentaan (General-purpose computing on graphics processing units). Tämän suuntautumisen avulla voidaan soveltaa näytönohjaimia muuhunkin kuin grafiikan piirtämiseen, johon se on alun perin suunniteltu. Tämän hetken tunnetuimmat ja käytetyimmät GPGPU-tekniikkaan soveltuvat ohjelmointirajapinnat ovat OpenCL sekä Nvidian CUDA.

2.2.3 OpenGL

OpenGL on laitteistosta riippumaton ohjelmointirajapinta 2D- sekä 3D-grafiikan piirtämiseen. Ohjelmointirajapintaa tukevien laitteiden kuitenkin tulee tarjota omat ajurinsa OpenGL:n virallisten määritelmien mukaisesti toteutettuna.
OpenGL:n alkuperäinen kehittäjä oli SGI (aikaisemmin Silicon Graphics), jonka kehittäminen siirtyi v. 2006 Khronos Groupin alaiseksi. OpenGL-rajapintaa käytetään tietokonepelien piirtämiseen sekä teollisuuden grafiikkasovellusten yhteydessä. OpenGL:n piirtämistapa perustuu tilakoneeseen missä OpenGL:lle määritellään piirtämiseen liittyvät asetukset ja tilat joiden perusteella piirtämiskomennot toimivat.
Tämän lisäksi OpenGL voi ajaa varjostin (eng. Shader) ohjelmia virallisesti OpenGL 2.0:sta lähtien. Varjostin-ohjelmien avulla grafiikkakoodaaja voi määritellä erilaisia piirtotapoja, mitä näytönohjain suorittaa. (The Khronos Group, 2015)

2.2.4 OpenCL

OpenCL on ohjelmointirajapinta, jolla pyritään luomaan ohjelmia, joita voidaan suorittaa erilaisissa laskentayksiköissä, kuten prosessoreissa että näytönohjaimissa.
OpenCL:n alkuperäinen kehittäjä oli Apple Inc, joka on siirtynyt Khronos Groupin alaiseksi jatkokehittämiseen. OpenCL:n avulla voidaan hyödyntää tietokoneen erilaisia laskentayksiköitä, joita saatettaisiin käyttää normaalisti

tietyn tyyppiseen laskentaan, kuten näytönohjaimet, joita käytetään grafiikkasovellusten grafiikan piirtämiseen ja antamalla kyseisille laskentayksiöille niiden normaalista laskentatehtävistä poikkeavat tehtävät. OpenCL:n pääasiallisena tarkoituksena on suorittaa paljon rinnakkaista laskettavaa vaativia tehtäviä sekä toteutuksia. Tämän avulla esimerkiksi näytönohjaimien laskentayksikköjä saadaan tehokkaasti käyttöön GPGPU-tekniikan periaatteella, sillä ne juuri perustuvat rinnakkaiseen laskentaan 2D-kuvien muodostamiseen 2D- sekä 3D-mallien pohjalta. (The Khronos Group, 2015)

2.2.5 CUDA

CUDA (Compute Unified Device Architecture) on Nvidian näytönohjaimille suunnattu GPGPU-tekniikalle suunniteltu ohjelmointirajapinta. Nvidia on toteuttanut GPGPU-pohjaisia sovelluksia CUDA:n avulla esim. PhysX, joka on reaaliaikainen fysiikkamoottori, jota monet tunnetut pelit sekä pelimoottorit hyödyntävät kuten pelimoottoreista Unreal Engine. PhysX oli alun perin oli suunniteltu erillisille fysiikan käsittely yksiköille (PPU, Physics Processing Unit) suoritettavaksi, mutta sitä hyödynnetään nykyisin näytönohjaimissakin. Näytönohjaimien hyödyntäminen mahdollistaa täten erikoistuneiden laskentayksiköiden kuten PPU:n korvaamisen ja helpottaa tietokoneen laskentakyvyn parantamista vain lisäämällä sopivia näytönohjaimia. PhysX:ää sovelletaan graafisten efektien simuloinnissa suurimmaksi osaksi. (Nvidia Corporation, 2015)

2.3 *Vokseli*

Vokseli (eng. Voxel) on elementti 3D-kuvassa joka koostuu useista kerroksista 2D-kuvia. Normaalissa 2D-kuvassa pikseli on vokselin vastine 3D-kuvaan nähden. Jokainen kerros esittää tietyltä syvyydeltä napattua kuvaa. Vokseliin voidaan varastoida 2D-kuvan pikselin tapaisesti tietoa, joita voidaan hyödyntää erilaisissa sovelluksissa, kuten lääketieteessä magneettikuvien yhteydessä. Peleissä sitä on hyödynnetty maaston kuvaamisessa ja piirtämisessä sekä geometrian muotojen määrittämisessä. Tunnetuin vokselipohjainen peli on Minecraft. (Voxel, 2015)

3 GPGPU-tekniikan ongelmien tarkastelu tietokonepeleissä

3.1 *Tiedonsiirtonopeudet ja muistin hyödyntäminen*

Vaikka näytönohjain pystyisi laskemaan tehokkaasti normaalisti prosessorilla laskettavia laskuja ja toimintoja, sen hyötykäyttöön vaikuttaa RAM:in ja VRAM:in välisessä tiedonvaihdossa/synkronoinnissa syntyvä viive. Tämän vuoksi pitää pyrkiä tarkastelemaan ja selvittämään eri tasoisten näytönohjainten tiedonsiirtokykyä ja määrittämään tiedonsiirtoon liittyvät suositukset tehokkaan laskennan vuoksi.
Muistin määrä näytönohjaimissa on vaihtelevaa ja tällä hetkellä vallitsevat VRAM määrät ovat n. 1-4 GB (Steam Hardware & Software Survey, 2015; Standalone Hardware Stats, 2015), joka myös vaikuttaa GPGPU:n käyttöön sekä grafiikan piirtämiseen sillä piirtämiseen kuulumaton data vie tilaa piirtämiseen liittyvältä datalta, mikä saattaa aiheuttaa pullonkaulan jos piirtämiseen liittyvää dataa joudutaan poistamaan näytönohjaimen VRAM:sta GPGPU laskennan ajaksi ja siirtämään sen takaisin laskentojen jälkeen.

3.2 *Näytönohjaimen hyödyntäminen ei-visuaalisissa efekteissä*

GPGPU-tekniikkaa käytetään yleisesti pelien yhteydessä graafisten efektien lisäämiseen tai parantamiseen esim. lisäämällä partikkeleita, jotka säilyvät näytönohjaimen muistissa. Kyseisten efektien tilaa päivitetään myös näytönohjaimen puolella ilman, että prosessori vaikuttaa asiaan ollenkaan. Samaa suoritetaan savun simuloinnissa sekä piirtämisessä. Tämä on käytännöllistä, mutta ei anna tarpeeksi hyvää kuvaa GPGPU:n soveltuvuudesta muuhun kuin puhtaasti graafisten efektien simuloinnista ja piirtämisestä, sillä pelin toiminnallisuuteen kyseiset visuaaliset efektit eivät juuri vaikuta. Tarkoituksena onkin pyrkiä suorittamaan pelin toiminnallisuuteen vaikuttavaa laskentaa, jonka voi havainnollistaa visuaalisena efektinä.

3.3 *GPGPU:n hyödyllisyys verrattuna prosessoriin ja yhdessä*

Koska näytönohjaimen laskentatapa ja tehokkuus ei ole verrattavissa normaaliin prosessorilla suoritettavaan laskentaan suoraan, pitää näytönohjaimella suoritettava laskenta purkaa yksinkertaisempiin palasiin, joita voidaan ajaa rinnakkain ilman datakisoja, jotka voivat aiheuttaa vääristyneen tiedon lukemista ja kirjoittamista. Tämä saattaa vaatia useamman askeleen verrattuna prosessoreilla suoritettaviin vastaaviin laskentoihin, mutta mahdollistaa suurien tietomäärien käsittelyn kerralla, mikä on GPGPU-tekniikan käyttötarkoitus. Paljon jakautuva koodipolku on hitaampaa suorittaa näytönohjaimella kuin prosessorilla.

3.4 *Ulkopuolisten kirjastojen toteutukset*

Pelimoottorit voivat hyödyntää muita kirjastoja, jotka saattavat myös hyödyntää GPGPU-tekniikkaa. Tällöin saattaa syntyä tilanteita, joissa esim. näytönohjaimen muistin käytöstä syntyy ongelmia tiedonsiirtojen kanssa. Tässä työssä ei kuitenkaan tarkastella kyseistä tilannetta.

3.5 *Hyväksyttävä määrä kuvaruutuja sekunnissa*

Työn tavoitteena oli pyrkiä tarkastelemaan tuloksia eri skaaloilla GPGPU:n tehokkuudesta kun alarajana on 30 kuvan piirtämistä sekunnissa. Tämän rajan alittaminen vaikuttaa pelikokemukseen siten, että kuvien vaihtuminen on selkeästi havaittavissa videopelien yhteydessä ja mahdollisesti vaikuttaa itse pelin pelaamiseen, sillä pelin pyörimisnopeus joko hidastuu ennalta määrätyn kuvaruudun päivitysnopeuteen nähden tai sisältää suuria aukkoja päivitysten välissä. Tämä taas vaikuttaa pelaajan haluamien toimintojen toteuttamiseen, jotka suoritetaan syöttölaitteiden, kuten näppäimistön pohjalta. Eli yhden päivityksen ja päivityksen tuloksen piirtämisen suorittamiseen saisi kulua vain 33 millisekuntia, kun pyritään havainnollistamaan päivityksen tulokset piirrettynä ruudulle 30 kertaa sekunnissa.

4 GPGPU-tekniikan hyötykäytön selvittäminen

Opinnäytetyössä sovellettiin OpenCL-ohjelmointirajapintaa mittauksissa GPGPU:n tehokkuudesta ja näytönohjain kohtaisia optimointeja ei toteutettu. Mittauksissa käytetään itsetehtyä pelimoottorin pohjaa. Tämän työn aikana ei käytetty kovalevyä sovelluksen apuna.

4.1 *RAM:in ja VRAM:in välinen viive*

Mitataan näytönohjaimeen liittyvät kirjoitus- ja lukunopeudet skaalaamalla käsiteltävää datamäärää kunnes siirtonopeudet ovat selvillä. Tarkoituksena on selvittää onko tämä viive kuinka haitallinen GPGPU:n hyötykäyttöön. Viiveen ollessa suuri, se haittaisi nopeasti suoritettavan, mutta yksinkertaisen laskennan hyötykäyttöä. Mittauksessa käytetään 50 megatavua muistia siirtoaikojen mittauksessa, joka täytetään valmiiksi arvoilla. Muisti sijaitsee RAM:iin sijoitettuna.

Toteutus suoritetaan yksinkertaisella silmukalla.

```
for(unsigned int i = 0; i < OPENCLMEMORYTEST; ++i)
{
        boost::property_tree::ptree t;
        cl_int error = queue->enqueueWrite(wr);
        while(CL_SUCCESS != this->clContext->getEventProfilingInfo(p_info))
        {}

        p_info.name = CL_PROFILING_COMMAND_END;
        p_info.value = &timeEnd;
        while(CL_SUCCESS != this->clContext->getEventProfilingInfo(p_info))
        {}

        t.add("bytes",wr.size);
        t.add("time",timeEnd - timeStart);
        t.add("average",(float)wr.size / (timeEnd - timeStart));
        tree.add_child("memory_transfer.results.write_results.result",t);
        wr.size *= 0.5f;
        p_info.value = &timeStart;
        p_info.name = CL_PROFILING_COMMAND_START;
    }
    wr.size = b_info.size;
    p_info.value = &timeStart;
    p_info.name = CL_PROFILING_COMMAND_START;
    for(unsigned int i = 0; i < OPENCLMEMORYTEST; ++i)
    {
        boost::property_tree::ptree t;
        cl_int error = queue->enqueueRead(wr);
        while(CL_SUCCESS != this->clContext->getEventProfilingInfo(p_info))
        {}
```

```
        p_info.name = CL_PROFILING_COMMAND_END;
        p_info.value = &timeEnd;
        while(CL_SUCCESS != this->clContext->getEventProfilingInfo(p_info))
        {}

        t.add("bytes",wr.size);
        t.add("time",timeEnd - timeStart);
        t.add("average",(float)wr.size / (timeEnd - timeStart));
        tree.add_child("memory_transfer.results.read_results.result",t);
        wr.size *= 0.5f;
        p_info.value = &timeStart;
        p_info.name = CL_PROFILING_COMMAND_START;
    }
```

queue-muuttuja on OpenCL:n komentojonon (Command Queue) kääritsijä, joka ottaa vastaan suoritettavia käskyjä. Mittauskäskyjen aloitukset ja lopetukset saadaan getEventProfilingInfo()-funktion avulla p_info-muuttuja sisältää tarvittavat muuttujat eri tapahtumien tiedusteluun. Tällä hetkellä OpenCL tukee vain suoritukseen liittyvien aikojen tiedusteluun, kuten komennon aloitus- ja lopetusajat.
Lopuksi tiedot tallennetaan Boost-kirjaston tarjoamaan tietorakenteeseen nimeltä property tree.

4.2 *Datapakettien mitoittaminen*

Kun viive on saatu mitoitettua pyritään mitoittamaan siirrettävien/muutettavien datapakettien koko sellaisiksi, että näytönohjain saisi suoritettua datapaketteja koskevat laskut ja toiminnot yhtä datakokonaisuutta kohden. Samaan aikaan valmistellaan ja siirretään seuraava laskettava erä näytönohjaimen muistiin ja luetaan edellisten laskujen tulokset pois mahdollisesti näytönohjaimelta prosessorin ohjaamana. Tämä on suhteellisen tärkeä vaihe, sillä tehokkaan GPGPU laskennan suorittaminen riippuu datan siirtokyvystä varsinkin kun kyseessä on tietokonepelit, joissa ajan käyttö on rajoitettu.

Myös datapakettien määrä ja koko myös vaikuttavat lopputulokseen siirtonopeuden lisäksi, jos kerralla käsiteltävän, siirrettävän ja luettavan datapakettien koko vaikuttaa piirtämisessä käytettävän tiedon tilapäiseen siirtoon pois näytönohjaimen muistista, voi se nostaa yhden päivityksen ja piirtämiseen liittyvän ajan liian korkeaksi, kun tarvittavat piirtämiseen liittyvät tiedot palautetaan takaisin näytönohjaimelle. Tämä myös saattaa osoittautua haitalliseksi ajureiden puolelta muistin varaamiseen ja vapauttamiseen

liittyvien ajankäytön vuoksi. Tarkoituksena on mittaustulosten perusteella päätellä, onko näytönohjaimen muistimäärä liian pieni jaettuna GPGPU:lle ja grafiikan piirtämiselle.
Datapakettien mitoitus toteutetaan ennen varsinaisen ohjelman ajamista, joka sitten hyödyntää tuloksia toteutuksessaan parhaansa mukaan.

4.3 *Vokselipohjainen nesteen simulointi*

Tässä työssä käytetään vokseleita nesteen simuloinnissa. Vokseleiden avulla voidaan helposti suorittaa simulointi ja vokseleiden vaatimat muistimäärät kattavat työssä tarkasteltavat tiedonsiirtoon liittyvät ongelmat.

4.3.1 Vokselien määritelmät

Vokselin perusmuoto kuvataan näin:

- koko: 12 tavua jaettuna kolmeen (3) 32-bittiseen muuttujaan
- Vokseliryhmä tulee olemaan kooltaan 32^3

Vokselin muotoja ei ole määritelty vokseliin itse vaan se toimii säiliönä, josta luetaan sen muoto ja kyseisen muodon yhteydessä esiintyvät arvot.
Vokselin perustietoihin kuuluu:

- ID, jonka arvo on välillä 0 - 16383. Nolla (0) tarkoittaa, että vokseli on tyhjä
- Lämpömuokkausmerkki, joka on välillä 0 – 1. Toimii korvamerkkinä lämmön päivittämisen yhteydessä
- Muokkausmerkki, joka on välillä 0 – 1. Toimii korvamerkkinä siirtymien päivittämisen yhteydessä
- Olomuoto, jonka arvo on välillä 0 – 2, 0 = kiinteä, 1 = nestemäinen, 2 = kaasu
- Lämpötila, jonka arvo on välillä 0 – 16383 ja yksikkönä Kelvin

Olomuodon perusteella suoritetaan kyseisille olomuodoille sopivat toiminnot. Vedellä ja kaasulla on samat ominaisuudet.

- Korkeus, joka on välillä 0 – 255
- Virtaussuunta X-tasolla, joka on välillä 0 – 239 ja tarkkuutena 1,5 astetta
- Virtaussuunta Y-tasolla, joka on välillä 0 – 239 ja tarkkuutena 1,5 astetta
- Nopeus, joka on välillä 0 - (2^{32} - 1) ja nopeus mm/s

- Täyttösuunta, joka määritellään arvoilla 0 – 5
- Muokkaustyyppi, joka määritellään arvoilla 0 - 3

Kiinteälle vokselille ei ole määritelty erikoisominaisuuksia.

4.3.2 Vokselityyppi

Vokselityyppissä kuvataan tyypin eri ominaisuuksia seuraavasti.

- Yleisominaisuudet (attributes), jotka ovat bittikenttiä vokselityypin ominaisuuksista
- Tiheys
- Kiehumispiste
- Sulamispiste

4.3.3 Vokselien päivittäminen

Vokselien tilan päivittämistä suoritetaan seuraavasti:
Vokselilähetin lähettää joko lämpöä tai tietyn tyyppistä vokselia tietyillä tiedoilla (esim. nestettä tietyllä nopeudella tiettyyn suuntaan). Lämpöä lähetetään tietystä pisteestä säteilynä ympärille, joka imeytyy vokseleihin sekä välittää sitä naapurivokseleille. Tämä päivitys suoritetaan pidemmän ajan yhteydessä siihen pisteeseen asti, että lämpöä ei imeydy tai varastoidu mihinkään vokseliin, jolloin lähetin poistetaan päivitettävistä lähettimistä. Tämän vakiintumisen jälkeen tilanteeseen voi vaikuttaa toinen lähetinlähde, vokselien tyypin muuttuminen tai lähettimen poistaminen.
Toteutus käy vokseliryhmän jokaisen vokselin lävitse lähettimen sijainnista aloittaen pallon laajentumisen tapaisesti leviämällä ympäristöön päivittämällä vokselien tietoja ja muuttamalla mahdollisesti vokselien olomuotoa.

Jos lähetin synnyttää esim. nestettä, tallennetaan nesteen etupään vokseliryhmä, josta tutkitaan nesteen mahdollinen eteneminen vokseliryhmässään sekä naapuriryhmissä. Tällä tavoin saadaan pidettyä päivitykseen liittyvien ryhmien määrä pienempänä, kuin jos tutkittaisiin jokainen ryhmä, jossa on liikkeellä olevaa nestettä, joiden määrä saattaa kasvaa huomattavasti nesteen etenemisen jatkuessa.

Nesteen etenemisen päivittämistä tutkitaan kuution sivuja vastaavien naapurivokselien tilatietoja hyödyntämällä. Päivityksen alussa tarkastetaan

nesteen etenemisen kärkipisteet sekä mahdollisten loppupisteiden muodostuminen lähettimen sammuttamisen jälkeen. Näiden pisteiden avulla nesteen etenemistä päivitetään nopeuden sekä nesteen korkeuden avulla naapurivokseleihin.

4.3.4 GPGPU-tekniikan sovellus vokseleilla

Tietokone luo vokseleiden avulla virtuaalimaailman ja lisää sinne nesteiden alkulähteet. Näiden purojen etenemistä simuloidaan ja samalla mitataan näytönohjaimen suoriutuminen kyseisestä tehtävästä. Simuloinnissa toteutetaan ennakointia, kosketuksen tunnistusta sekä siihen reagointia aivan kuten peleissä pelihahmojen ja kohteiden väleillä.

4.3.5 Prosessoripohjainen sovellus vokseleilla

GPGPU:n sovellus mukautettuna prosessorille. Näytönohjaimen ja prosessorin tuloksia vertaillaan sekä toteutukseen liittyviä eroavaisuuksia. Prosessorin etuna on, että datasiirtoja ei tarvitse tehdä eri muistien välillä ja käytettävän muistin määrä voi olla suuri mutta tämän työn aikana pyritään pitämään ohjelman muistimäärä 4 Gt:n tasolla.

4.3.6 Prosessori ja GPGPU-pohjainen sovellus vokseleilla

Prosessorin sekä näytönohjaimen hyödyntäminen yhtä aikaa jaetuilla resursseilla, missä tiedon yhdessä pitävyys on tärkeää ja tiedon siirtonopeus merkittävässä osassa pelin piirtonopeuden takia. Toteutuksessa pyritään ennakoimaan tulevia tapahtumia sekä niiden mahdollisia vaikutuksia muihin vokseliryhmiin, jotta vokseliryhmien päivitykset tapahtuisivat samalla suorittajalla.

4.4 *Piirrettyjen kuvien määrä sekunnissa tarkastelu*

Simuloinnin tulee käyttää vähemmän aikaa kuin 33 ms (millisekunti), jotta muut mahdolliset pelin osa-alueet, kuten tekoälyn käsittely, voisivat realistisesti toteutua, lisätään 10 ms mittaustuloksiin. Näin myös nähdään, onko ohjelman suorittamassa koodissa tai käsiteltävässä tietomäärässä

ongelmia.

4.5 *Säikeistys prosessorilla*

Käytettävien säikeiden määrä lasketaan seuraavasti:

hyödynnettävät säikeet = prosessorin maksimi säiemäärä * 0.75 - 1

Koska PC suorittaa samanaikaisesti muita ohjelmia, pitää nämä ottaa myös huomioon ja jättää näille taustaohjelmille resursseja hyödynnettäviksi. Pelien käynnissä ollessa ei yleensä ajeta raskaita ohjelmia, minkä vuoksi pelit voivat olettaa käyttävänsä paljon laitteiston tarjoamia resursseja ilman, että pelin suoriutuminen kärsisi muiden ohjelmien vuoksi. Yksi säie poistetaan sillä se vastaa pelimoottorin omaa säiettä.

Suoritettavien säikeiden sovellus toteutetaan seuraavasti:

- Pääsäie, joka aktivoi itse pelin oman silmukan ja suorittaa moottorin funktioita, jotka olisivat kevyitä kuten näppäimistön lukemista ja sen tallentamista peliä varten.
- Pelisäie, joka suorittaa itse pelin koodin sekä piirtämisen
- Muut säikeet, jotka odottavat tehtävän antoja pelisäikeeltä

Nämä säikeet käynnistetään suorittamaan silmukkaa, jonka sisällä ne suorittavat seuraavat toiminnot.

1. Tarkista, onko tietyn tyyppistä tehtävän suorittajaa olemassa.
2. Jos ei ole, säikeestä tulee kyseisten tehtävien suorittaja, ja pyrkii suorittamaan kyseisen suorittajan tehtäviä kunnes kyseisiä tehtäviä ei ole jäljellä.
3. Säie suorittaa yleisiä tehtävänantoja
4. Jos säie ei löydä yhtään suoritettavaa tehtävää, se suorittaa X määrän kokonaisluvun yhteenlaskuja ja aloittaa tehtävien haun alusta.
5. Jos säie oli suorittanut aktiivisen odottamisen kuvattuna 4. askeleessa, menee säie lepotilaan ennalta määrätyksi ajaksi.
6. Lepotilan loputtua aloitetaan tutkiminen alusta.

Vaiheiden 1. ja 2. suoritukset ovat lukittuna mutex-lukoilla, jonka avulla

huolehditaan siitä, että säikeet eivät suorita samoja tehtäviä ja että tietyn tyyppisille tehtäville olisi ainoastaan yksi suorittaja (esim. kovalevyltä lataukset sekä muiden laitteistojen kanssa keskustelu).
Toteutuksen tulee tukea tehtävien määrittelyä sen määräämillä tavoilla, jotta säikeytys toimisi.

4.6 *Tehtävän määritelmä*

Tehtävä on määritelty seuraavasti:

```
class Task
{

public:
    void* object;
    void* taskarguments;
    char taskType;
    void (*taskExecute)(Task*);
    void (*taskCallback)(Task*);
};
```

- object-muuttuja sisältää mahdollisesti jonkin luokan olion osoittimen. Tämä voidaan muuttaa takaisin oikean tyyppiseksi olioksi suoritettavassa funktiossa.
- taskarguments-muuttuja sisältää osoittimen funktion argumentteihin, mikäli niitä hyödynnetään.
- taskType-muuttuja kuvastaa suoritettavan tehtävän tyyppiä, jota hyödynnetään tehtävien lajitteluissa sopiville tyypeille. Vaihtoehtoina ovat TASK, EXTERNAL_TASK, DATA_TRANSFER_TASK, SHARED_TASK. TASK on normaali, prosessorilla suoritettava toiminto, EXTERNAL_TASK on toisella laitteella suoritettava toiminto kuten näytönohjaimella, DATA_TRANSFER_TASK tarkoittaa tiedonsiirtoa kovalevyltä, ja SHARED_TASK voidaan suorittaa joko prosessorilla tai toisella laitteella kuten näytönohjaimella.
- taskExecute-muuttuja on suoritettavan tehtävän funktio-osoitin, joka toimii tehtävän aloituspisteenä.
- taskCallback-muuttuja toimii vapaaehtoisena funktio-osoittimena, joka suoritetaan tehtävän suorituksen jälkeen, mikäli se on käytössä.

Kun ohjelman yksi osio on lisännyt tarpeeksi suoritettavia tehtäviä, kannattaa nämä suorittaa loppuun, ennen kuin siirrytään ohjelman seuraavaan vaiheeseen, jotta seuraavan vaiheen tiedon hyödyntämiset olisivat ajantasaiset. Tämän synkronoiminen voidaan suorittaa funktiolla nimeltä executeTasks(), joka suorittaa kaikki listassa olevat tehtävät muiden säikeiden ohella. Funktio löytyy jokaisesta pelimoottorin ajettavien osien luokista. Pelimoottorin toteutus ei tällä hetkellä priorisoi tiedon siirron vaatimusta, ennen kuin kyseisten tietojen hyödyntävä tehtävä suoritetaan. Tämä kyllä voidaan toteuttaa tehtävän taskCallback-funktio-osoittimella.

5 Tulokset

5.1 *Tiedonsiirtonopeudet*

Tulokset on kerätty Nvidian GeForce GTX 560 Ti-näytönohjaimella, jonka valmistajana on ASUSTeK Computer Inc. Ajurin versio on Nvidian 346.47 ja OpenCL-versio on 1.1.
Mittaustulokset on kerätty ASUS-valmistajan Nvidia 560 Ti-näytönohjaimella.
Kirjoitusmittaustulokset (Tiedonsiirron kirjoitusnopeudet)

Tavumäärä (kt)	Aika (µs)	Hyötysuhde
50000	8682,656	5.758606
25000	4280,896	5.839899
12500	2147,456	5.820841
6250	1066,304	5.861368
3125	478,784	6.526952
1562,5	240,8	6.488787
781,25	119,552	6.534813
390,625	61,152	6.387772
195,312	31,968	6.10961
97,656	17,344	5.630535
48,828	7,84	6.228061
24,414	5,024	4.859475
12,207	3,2	3.814687
6,103	2,272	2.68618
3,051	1,792	1.702567
1,525	1,568	0.9725766
0,762	1,472	0.5176631
0,381	1,408	0.2705966
0,19	1,376	0.1380814
0,095	1,344	0.07068452
0,047	1,376	0.03415698
0,023	1,376	0.01671512
0,011	1,344	0.008184524
0,005	1,344	0.003720238
0,002	1,344	0.001488095

Taulukko 1: Tiedonsiirron kirjoitusnopeudet

Mittaustietojen perusteella voidaan sanoa, että tiedonkirjoituksessa esiintyvä viive on vähintään n. 1.3 – 1.6 mikrosekuntia ja pienimmän kirjoituksen yhteydessä esiintyvä suurin tiedonkirjoitusmäärä n. 1.5 kilotavun luokkaa. Paras kirjoitussuhde tavoitetaan vasta 390 kilotavun siirron yhteydessä, minkä kirjoittamiseen menee n. 60 mikrosekuntia.

Lukumittaustulokset (Tiedonsiirron lukunopeudet)

Tavumäärä (kt)	Aika (µs)	Hyötysuhde
50000	8486,88	5.891447
25000	4458,368	5.607433
12500	2186,368	5.717244
6250	1067,744	5.853463
3125	512	6.103516
1562,5	239,552	6.522592
781,25	131,36	5.947396
390,625	61,28	6.374429
195,312	32,064	6.091317
97,656	17,472	5.589286
48,828	10,24	4.768359
24,414	6,496	3.758313
12,207	4,544	2.6864
6,103	3,648	1.672971
3,051	3,2	0.9534375
1,525	3,008	0.5069814
0,762	2,848	0.2675562
0,381	2,848	0.1337781
0,19	2,752	0.0690407
0,095	2,752	0.03452035
0,047	2,752	0.01707849
0,023	2,752	0.008357558
0,011	2,752	0.003997093
0,005	2,752	0.001816861
0,002	2,752	0.0007267442

Taulukko 2: Tiedonsiirron lukunopeudet

Tiedon lukunopeudessa minimiviive lukemisessa näyttäisi olevan n. 2,75 mikrosekuntia, mikä taas tarkoittaa sitä, että kirjoittaminen on n. kaksi kertaa nopeampaa suorittaa kuin lukeminen minimi viiveen puolesta. Mutta paras lukusuhde vastaa parasta kirjoitussuhdetta eli 390 kilotavua luetaan n. 60 mikrosekunissa, joten tämä ei tulisi olemaan ongelmana.

6 Johtopäätökset

6.1 *Tiedonsiirron ongelmat*

Kuten tiedonsiirtotaulukoista (Tiedonsiirron kirjoitusnopeudet; Tiedonsiirron lukunopeudet) näkee, on tiedonsiirto hidasta. Pelkästään 50 megatavun siirtäminen kestää 7 millisekuntia ja lukeminen 8 millisekuntia, mikä vaikeuttaa selvästi näytönohjaimen hyödyntämistä tehokkaasti prosessorin rinnalla. Vaikka saisikin pidettyä näytönohjaimen suorittamassa tehokkaasti koodia niin tiedon siirtäminen RAM:in ja VRAM:in välillä rajoittaa näytönohjaimen hyödyntämistä prosessorin rinnalla tehtävissä missä tiedon synkronointi on tärkeässä osassa pelin päivitysten kannalta. Täten tiedon siirtoa tulisi minimoida ja kokonaan välttää mikäli mahdollista.

6.2 *GPGPU:n tehokkuus*

GPGPU-laskennan hyödyntämistä ei onnistuttu mittaamaan aiotulla toteutuksella. Tämä johtui toteutuksen jäädessä keskeneräiseksi ajan loppumisen sekä liian aikaisten optimointi yritysten vuoksi.

6.3 *Yhteenveto*

Vaikka toteutuksen avulla tehdyt mittaukset jäivät välistä sekä näytteiden määrä jäädessä pieneksi, antaa alun tiedonsiirtonopeudet kuitenkin suuntaa antavan kuvan siitä, että minkälaisia toteutuksia tulisi hyödyntää näytönohjaimella. Suurien tietomäärien siirtely olisi haitallista pelille, mikä alkaisi näkyä pelin päivitysnopeudessa selkeästi. Pelkästään 100 Mt siirtely yhteen suuntaan laskisi näytettävien kuvien määrän alle 60 kuvaruutuun sekunnissa ottamatta huomioon muita päivityksessä tapahtuvia toimintoja.

Näytönohjaimien hyödyntäminen peleissä todellakin kannattaa painottaa siten, että suoritettavat toiminnot käyttävät näytönohjaimella sijaitsevaa dataa, kuten piirtämiseen liittyvää dataa.

Tulevaisuudessa GPGPU:n hyödyntämisen parantamista varten

laiteistonkehittäjät voisivat parantamaan tiedonsiirron nopeuksia prosessorin ja näytönohjaimen välillä.

7 Lähteet

CUDA. Nvidia Corporation. Viitattu 19.10.2015. http://www.geforce.com/hardware/technology/cuda

McClanahan, C. History and Evolution of GPU Architecture. Georgia Tech College of Computing 27.3.2011. Viitattu 19.10.2015. http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf

OpenGL. The Khronos Group. Viitattu 19.10.2015. https://www.opengl.org/about/

OpenCL. The Khronos Group. Viitattu 19.10.2015. https://www.khronos.org/opencl/

OpenCL History. Wikipedia. Viitattu 19.10.2015. https://en.wikipedia.org/wiki/OpenCL#History

PhysX. Nvidia Corporation. Viitattu 19.10.2015. http://www.geforce.com/hardware/technology/physx

Standalone Hardware Stats 2015-09. Unity Technologies 9.2015. Viitattu 20.10.2015. http://hwstats.unity3d.com/pc/mem.html

Steam Hardware & Software Survey: Septemper 2015. Valve Corporation 9.2015. Viitattu 20.10.2015. http://store.steampowered.com/hwsurvey

Suvanto, S. 2014. Peliteollisuus on jo miljardibisnestä Suomessa. Yle Uutiset 3.4.2014. Viitattu 19.10.2015. http://yle.fi/uutiset/peliteollisuus_on_jo_miljardibisnesta_suomessa/7169571

Voxel. Wikipedia. Viitattu 19.10.2015. https://en.wikipedia.org/wiki/Voxel