Opinnäytetyö (AMK)

Tietotekniikka

Sulautetut ohjelmistot

2015

Timo Salomäki

PARTIKKELISYSTEEMIN SUUNNITTELU JA OHJELMOINTI

TURUN AMMATTIKORKEAKOULU

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautetut ohjelmistot

2015 | 38

Tiina Ferm

Timo Salomäki

PARTIKKELISYSTEEMIN SUUNNITTELU JA OHJELMOINTI

Opinnäytetyön tarkoituksena oli suunnitella ja toteuttaa Windows-pohjainen partikkelisysteemi. Partikkelisysteemi on tarkoitettu satunnaisuutta sisältävien luonnonilmiöiden ja erikoistehosteiden, kuten esimerkiksi sateen, sumun tai galaksien simuloimiseen. Ohjelmakoodi kirjoitettiin C#-ohjelmointikielellä käyttäen apuna MonoGame-ohjelmistokehystä ja Visual Studio -ohjelmankehitysympäristöä.

Työssä tarkasteltiin partikkelisysteemien historiaa ja perusrakennetta, jonka pohjalta toteutettiin yleiskäyttöinen ja rajapintojen avulla laajennettavissa oleva partikkelisysteemi. Perustoiminnallisuuden lisäksi toteutettiin erilaisia lähettimiä ja vaikuttimia, jotka mahdollistivat monimutkaisempien tehosteiden luomisen. Partikkelitehosteiden luomista ja muokkausta varten kehitettiin myös erillisenä sovelluksena toimiva Windows-pohjainen partikkelieditori, jota käyttämällä kaikki partikkelisysteemin ominaisuudet ovat testattavissa ilman ohjelmakoodin kirjoittamista. Editori sisältää partikkelisysteemiä reaaliaikaisesti ajavan livenäkymän, johon ominaisuuslistan kautta tehdyt parametrien muutokset välittyvät automaattisesti.

Itse partikkelisysteemin toteutuksen lisäksi työssä keskityttiin sen optimointiin ja suorituskyvyn testaukseen. Tätä varten kehitettiin työkalu, joka automatisoi partikkelitehosteiden muistinkäytön ja prosessorikuormituksen monitoroinnin. Paljon partikkeleja sisältävien tehosteiden kohdalla simulaation pullonkaulaksi osoittautui muistinkäyttö, jonka optimoimiseksi tehtiin erilaisia muutoksia ohjelmakoodiin ja arkkitehtuuriin. Testaustyökalu mahdollistaa myös uusien testien kirjoittamisen ja olemassa olevien laajentamisen eri rajapintoja hyödyntämällä.

Yksinkertaisen partikkelisysteemin luominen oli verraten helppo tehtävä, mutta jatkokehityksen ja laajennettavuuden mahdollistavat arkkitehtuurilliset ratkaisut osoittautuivat suorituskykyongelmien lisäksi työn suurimmiksi haasteiksi. Olio-ohjelmoinnin sääntöjä ja erilaisia suunnittelumalleja hyödyntämällä on kuitenkin mahdollista ylläpitää ja kehittää monimutkaistakin ohjelmakoodikokonaisuutta. Työn puitteissa hyödynnetyistä suunnittelumalleista objektiallas osoittautui toteutuksen haasteellisuudesta huolimatta yhdeksi tärkeimmäksi elementiksi suorituskyvyn kannalta. Luokan geneerinen toteutus mahdollistaa sen suoran uudelleenkäytön myös tulevissa projekteissa.

ASIASANAT:

MonoGame, XNA, C#, Windows, partikkelisysteemi, simulaatio, peliohjelmointi

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information Technology | Embedded Software

2015 | 38

Tiina Ferm

Timo Salomäki

DESIGN AND IMPLEMENTATION OF A PARTICLE SYSTEM

The purpose of my thesis was to plan and implement a particle system that supports Windows operating system. The particle system is meant to simulate fuzzy natural phenomena and special effects like rain, fog and galaxies. Programming was done using C# language and MonoGame Framework. Visual Studio was used as a development environment.

After studying the history and basic architecture of particle systems, a generic and easily extensible particle system was implemented. In addition to basic features, different emitters and modifiers were created to support the creation of more complex particle effects. A Windows-based particle editor was also programmed to aid in creation and testing of all the particle system's features. The editor effectively removes the need to write a single line of code to create effects. Particle system is simulated inside the editor in real time so that all the changes made through a property list is immediately visible in the live view.

In addition to the design and implementation of the particle system, extra attention was paid to optimization and performance testing. A separate tool for automating memory and processor monitoring was developed. The bottleneck in particle heavy effects was shown to be memory handling and some changes in code and overall architecture were made to fix the problem. The dedicated testing and benchmarking tool can be extended using interfaces.

Creating a simple particle system was a rather easy task but planning for a future-proof and extendable architecture was one of the biggest challenges on top of the problems with performance. By following the rules of object oriented programming and making use of software design patterns, it's possible to work effectively with a complex programming project. The most interesting and valuable pattern used in programming was the object pool pattern that fixed several performance issues. The generic implementation of the object pool can be directly used in future projects.

KEYWORDS:

MonoGame, XNA, C#, Windows, particle system, simulation, game programming

SISÄLTÖ

KÄYTETYT LYHENTEET

1 JOHDANTO **1**

2 TEORIA **3**

2.1 Partikkelisysteemin rakenne 5

2.2 Partikkelisysteemi osana pelimoottoria 7

2.3 Pelimoottorin toiminta 8

2.4 Työkalut 10

2.5 Optimointi 12

3 PARTIKKELISYSTEEMIN TOTEUTUS **15**

3.1 ParticleSystem 15

3.2 ParticleEffect 16

3.3 ParticleEmitter 17

3.4 ParticleFactory 20

3.5 Vaikuttimet 20

4 PARTIKKELIEDITORI **24**

4.1 Livenäkymä 25

4.2 Ominaisuusikkuna 25

4.3 Tiedostoformaatti 26

4.4 Serialisoinnin ongelmat 27

5 OPTIMOINTI **28**

5.1 Object Pool 29

5.2 Suorituskykytestaus 32

6 YHTEENVETO **36**

LÄHTEET **38**

KUVAT

Kuva 1. Star Trek II: The Wrath of Khan -elokuvassa käytetty partikkelitehoste. (Martin 1999). 4
Kuva 2. Pelisilmukka koostuu kahdestä päämetodista, joiden kokonaissuoritusaika tulee olla maksimissaan noin 16 ms, jotta pelin päivitystaajuus pysyy tavoitellussa 60 FPS:ssä. 9
Kuva 3. Editorissa asetettu pysäytyspiste on korostettu punaisella. Pysäytyspisteen jälkeen virheenkäsittelijällä on liikuttu manuaalisesti kaksi koodiriviä alaspäin. Seuraavaksi ajettava koodirivi on korostettu keltaisella. 12
Kuva 4. Opinnäytetyössä toteutetulla partikkelisysteemillä luodusta räjäytysanimaatiosta otettuja ruutukaappauksia. 17
Kuva 5. Lumisade-efekti on toteutettu tuulivaikutinta hyödyntäen. 22
Kuva 6. MouseModifier vetää partikkeleja hiiren osoittimen suuntaan. 23
Kuva 7. Partikkelitehoste muokattavana partikkelieditorissa. 24
Kuva 8. List-luokka käyttää sisäisessä toteutuksessa taulukkoa (Array). 30
Kuva 9. Ero alkion poistamisessa listan alku- tai loppupäästä. 31

TAULUKOT

Taulukko 1. Partikkelilähettimen muodot. 19
Taulukko 2. Ominaisuusikkunan laajennettu tyyppituki. 25
Taulukko 3. IBenchmark-rajapinnan metodit ja tapahtumakäsittelijät. 34

KÄYTETYT LYHENTEET

2D	Kaksiulotteinen
3D	Kolmiulotteinen
C++	Bjarne Stroutstrupin vuonna 1983 kehittämä oliopohjainen ohjelmointikieli
C#	Microsoftin 2000-luvun alussa kehittämä olio-ohjelmointikieli
C	Bell Labsissa vuonna 1972 kehitetty imperatiivinen ohjelmointikieli
CLR	Common Language Runtime on Microsoftin .Net Frameworkin virtuaalikonekomponentti, joka huolehtii .NET –sovellusten suorittamisesta.
DirectX	Microsoftin kehittämä ohjelmointirajapinta tietokoneohjelman ja laitteiston välille
DLL	Dynamic Link Library. Ajonaikainen kirjasto on tapa jakaa ohjelmakoodia ja dataa eri ohjelmien kesken.
ECMA	European Computer Manufacturers Association on kansainvälinen standardointiorganisaatio, joka keskittyy informaatio- ja kommunikaatioteknologioiden sekä kuluttajaelektroniikan standardointiin.
FPS	Frames Per Second. Ruudun päivitystaajuus kokonaisina ruudun uudelleenpiirtoina sekunnissa.
iOS	Applen kehittämä mobiilikäyttöjärjestelmä
OpenGL	Open Graphics Library on Silicon Graphicsin kehittämä monialustainen ohjelmointirajapinta
OS X	Applen kehittämä Unix-pohjainen työpöytäkäyttöjärjestelmä
RGB	Red, green, blue. RGB-värimalli on väritila, jossa muodostetaan eri värejä punaisen, vihreän ja sinisen väristä valoa sekoittamalla.
XNA	“XNA is Not an Acronym”. XNA on Microsoftin peliohjelmointikirjasto.

1 JOHDANTO

Tämän opinnäytetyön aiheena on reaaliaikaiseen simulaatioon tarkoitetun partikkelisysteemin toteutus suunnitteluvaiheesta testaukseen ja editorin rakentamiseen. Tavoitteena on rakentaa helposti laajennettava ja arkkitehtuuriltaan selkeä kaksiulotteisten erikoistehosteiden tuottamiseen tarkoitettu systeemi, joka toimii itsenäisesti tai valmiin pelimoottorin osana. Työssä keskitytään erityisesti partikkelisysteemin arkkitehtuuriin ja sen eri osien toimintaan sekä systeemin kehityksessä kohdattaviin haasteisiin ja niihin esitettäviin ratkaisuihin.

Partikkelisysteemi voi pelistä riippuen olla kaksi- tai kolmiulotteinen. Näiden systeemien toteutuksen erot eivät ole periaatteessa kovin suuret, mutta kolmiulotteinen systeemi on matemaattisesti haastavampi ja se vaatii tietokoneelta enemmän suorituskykyä. Kolmiulotteisen grafiikan ohjelmointi on myös huomattavasti haastavampaa, joten opinnäytetyön aiheena olevan partikkelimoottorin käyttö rajataan kaksiulotteisiin tehosteisiin. Käyttökohteiksi sopivatkin hyvin esimerkiksi tasohyppelypelit ja erilaiset ylhäältä tai sivulta kuvatut simulaatiot.

Pelimoottoriin upotettavan komponentin lisäksi projektin aikana toteutetaan Windows-käyttöjärjestelmälle suunnattu graafisen käyttöliittymän käsittävä partikkelieditori, jonka avulla pelin grafiikoista vastaava artisti pystyy toteuttamaan haluamansa tehosteet ilman ohjelmointiosaamista. Editorin on tarkoitus korvata kokonaan käsin tehtävä ohjelmointityö, joten sitä käytettäessä on mahdollista hyödyntää kaikkia partikkelisysteemin tarjoamia ominaisuuksia. Tehosteita suunnittelevan artistin ei ole välttämätöntä tehdä yhteistyötä itse pelinkehittäjien kanssa muuten kuin käyttöönotossa ja testauksessa. Uusien tehosteiden luominen voidaan aloittaa täysin tyhjästä tai hyödyntämällä aiemmin luotua tehostetta. Keskeneräiset tehosteet on mahdollista tallentaa tiedostona kovalevylle ja lähettää tarvittaessa toiselle henkilölle muokattavaksi. Editorin tarkoituksena on toimia kehityksen edetessä myös testiympäristönä, sillä työn rajauksen vuoksi partikkelisysteemiä varten ei luoda erillistä testipeliä.

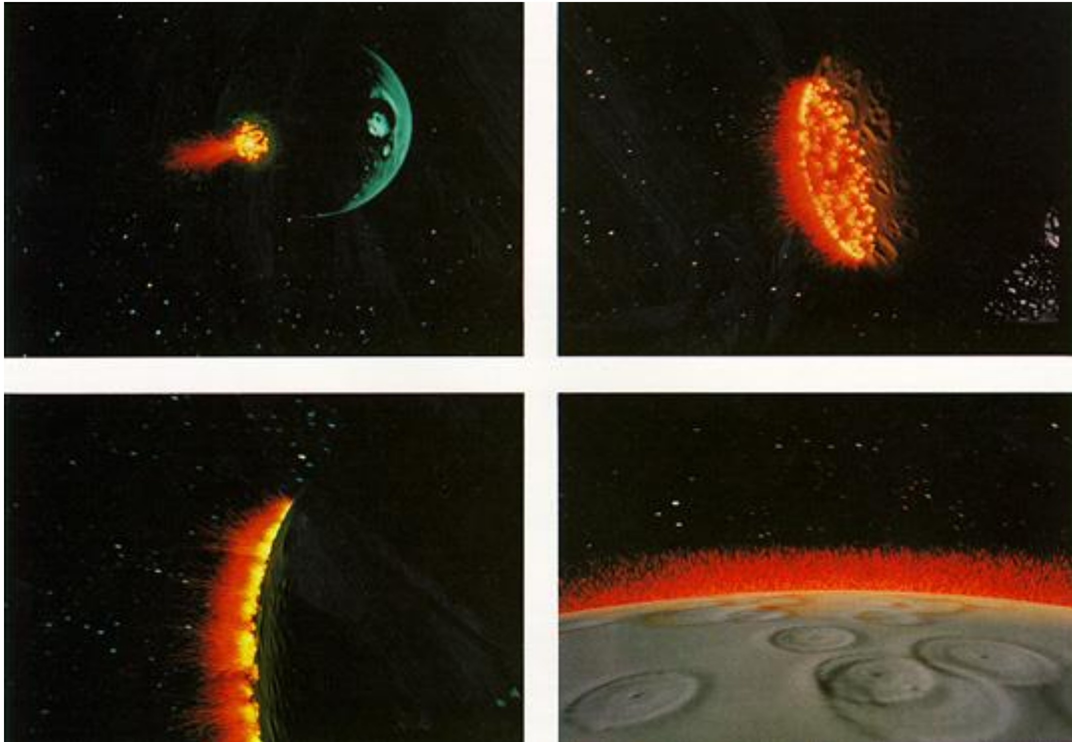
Projektin päätarkoituksena ei ole luoda kaikkein tehokkainta ratkaisua partikkelitehosteiden simulointiin, vaan valmiin tuotteen helppokäyttöisyys ja ylläpidettävyys jatkokehitettävyyttä unohtamatta ovat projektin tärkeimpiä anteja.

2 TEORIA

Partikkelisysteemi-termiä käytettiin ensimmäisen kerran vuonna 1983 kuvaamaan Start Trek II: The Wrath of Khan -elokuvassa näkyvän kuvasarjan toteutustapaa. Kyseinen kuvasarja sisältää planeetan, jonka pinnalla räjähtävä pommi aiheuttaa tulimyrskyn, joka lopulta nielaisee koko planeetan sisäänsä. (Kuva 1.) Tehosteen tekijä W. T. Reeves loi työllään perustan nykyaikaisille partikkelisysteemeille, mutta partikkelisysteemi on tietokonegrafiikkaan liittyvänä terminä varsin löyhästi määritelty ja sitä onkin käytetty kuvaamaan mallinnus- ja renderöintitekniikoita sekä animaatiotyyppejä. (Martin 1999.) Termin käytön sekavuudesta huolimatta kaikilla partikkelisysteemeillä on Reevesin (1983) mukaan kaksi yhteistä piirrettä:

1) Partikkelisysteemi koostuu yksittäisistä partikkeleista, jotka sisältävät joitain partikkeliin suoraan tai epäsuoraan vaikuttavia ominaisuuksia. Partikkelit voivat olla yksinkertaisia primitiivejä, kuten pisteitä tai viivoja, mutta niitä voidaan kuvata myös monimutkaisemmilla muodoilla tai kuvilla.

2) Partikkelien ominaisuudet määritellään jonkin asteista satunnaisuutta hyödyntäen, jolloin partikkelisysteemin toiminta ei ole ennustettavissa ja saavutetut tehosteet ovat luonnollisempia. Satunnaisuutta voidaan kontrolloida asettamalla tietyt rajat, jolloin hallinta pysyy jossain määrin toteuttajan käsissä. Rajat voidaan määritellä esim. prosentuaalisena heittona jostain alkuarvosta, tai minimi- ja maksimiarvoilla, joiden väliltä ominaisuuksien arvot arvotaan. (Reeves 1983).



Kuva 1. Star Trek II: The Wrath of Khan -elokuvassa käytetty partikkelitehoste. (Martin 1999).

Erilaiset tehosteet, kuten sade, sumu ja räjähdykset ovat tyypillisiä esimerkkejä partikkelisysteemin avulla simuloiduista ilmiöistä, ja nykypäivänä ne ovatkin olennainen osa pelien ja elokuvien visuaalisen ilmeen luomisessa. Partikkelisysteemin tehosteet muodostuvat erinäköisistä, kokoisista ja värisistä objekteista, joita voidaan simuloida yhtäaikaisesti kymmeniä tai satoja tuhansia (pelkästään partikkeleista koostuvissa simulaatioissa jopa miljoonia). Jotkin tehosteet vaativat hyvin suunniteltuina vain muutaman partikkeliobjektin käyttöä, mutta monimutkaisemmat tehosteet saattavat vaatia suuria määriä pieniä partikkeleja uskottavan lopputuloksen saavuttamiseksi. Esimerkiksi simuloitaessa vettä voidaan yhden partikkelin kuvitella vastaavan yhtä vesimolekyyliä. Tällöin partikkelien tulee olla erittäin pieniä, ja niitä tulee olla simulaatiossa niin monta, että yksittäisiä ”molekyylejä” ei pysty erottamaan toisistaan. Partikkeleista koostuvan massan tulee näyttää yhtenäiseltä erilaisia roiskeita lukuun ottamatta.

2.1 Partikkelisysteemin rakenne

Käyttötarkoituksesta riippuen partikkelien prosessointi ja piirtäminen tapahtuu joko jaetusti prosessin ja näytönohjaimen varassa, tai lähes kokonaan (objektien luomista ja näytönohjaimelle siirtämistä lukuun ottamatta) jälkimmäisen varassa. Tehokkaimmat partikkelisysteemit käyttävät prosessoria vain systeemin ja sen osien initialisointiin, jonka jälkeen tarvittava data siirretään näytönohjaimelle, jossa sitä käsitellään ja piirretään ilman prosessorin puuttumista asiaan. Näytönohjaimella tapahtuvan päivityksen toteutuksessa käytetään shader-kieltä, sillä näytönohjain ei osaa suoraan tulkita pelin prosessorin varaisten komponenttien ohjelmointiin käytettyä ohjelmakoodia. Shader-kieliä on olemassa useita, mutta vain muutama niistä on saavuttanut laajemman suosion. Reaaliaikaiseen renderöintiin tarkoitettujen kielien valinnassa on myös otettava huomioon kohdealusta, sillä kielet on sidottu johonkin tiettyyn grafiikkarajapintaan. Esimerkiksi Microsoftin DirectX-rajapinnan shader-kieli on nimeltään High-Level Shader Language. Se on C-ohjelmointikielen kaltainen kieli, jota voidaan hyödyntää esim. Windows Phone-, Xbox One- ja Windows-pelinkehityksessä. (Varcholik 2014, 8.)

Partikkelisysteemin arkkitehtuuri vaihtelee paljon sen käyttötarkoituksen mukaan. Monimutkaisempien tehosteiden luomiseen, laajennettavaksi tarkoitettu systeemi vaatii korkeamman abstraktiotason, kun taas yksinkertaiseen peliin riittää huomattavasti joustamattomampi toteutus. Vuosien varrella reaaliaikaisten pelikäyttöön tarkoitettujen systeemien rakenne on kuitenkin perusosiltaan vakiintunut.

Partikkeli

Partikkelisysteemin pienin yksittäinen osa on partikkeli. Tehokkaassa systeemissä partikkelia kuvaava luokka sisältää vain välttämättömimmät ominaisuudet sen päivitystä ja piirtämistä varten, sillä luokan instansseja voi olla samanaikaisesti koneen muistissa jopa kymmeniä tuhansia. Luokan tulee sisältää vähintään

sijaintia ja nopeutta kuvaavat kentät, jotta partikkeli voidaan piirtää ruudulle, mutta lähes poikkeuksetta ominaisuuksia on enemmän.

Eräs ensimmäisistä partikkelistysteemin kuvanneista henkilöistä, W. T. Reeves (1983), määritteli partikkeliobjektille kahdeksan ominaisuutta, joiden avulla yksinkertaisten tuli- ja räjähdyssimulaatioden tekeminen oli mahdollista. Partikkelin sijainnin (position) ja nopeuden (velocity) lisäksi Reeves määritteli ominaisuudet väri (color), muoto (shape), koko (size) ja läpinäkyvyys (transparency) kuvaamaan partikkelin ulkonäköä. Partikkeliobjektien elinikää kontrolloi ikä- ja maksimiikä -ominaisuudet, joiden avulla on mahdollista muuttaa esim. läpinäkyvyyttä, häivyttäen partikkelin pois näkyvistä elinkaarensa aikana. (Reeves 1983.) Nykyaikaisiin partikkelisysteemeihin verrattuna erot ovat hyvin pienet, mutta muodon sijaan partikkelin ulkonäöstä vastaa yleensä jokin kuva eli kaksiulotteinen tekstuuri. Partikkelilla voi myös olla jotain muita ominaisuuksia, mutta Reevesin listaamien ominaisuuksien käyttäminen riittää varsin monipuolisten partikkelisysteemien luomiseen.

Partikkelilähetin

Tyypillinen partikkelisysteemi sisältää yhden tai useamman lähettimen (emitter), jonka pääasiallinen tehtävä on partikkelien luominen ja alkuarvojen, kuten nopeuden ja sijainnin asettaminen. Lähettimen luodessa partikkeleja jatkuvana virtana, syntyy siitä eräänlainen suihku. Usein lähetintä voi myös muokata lähettämään partikkeleja tietyin aikavälein, jolloin suihku on katkonainen. (Shiffman 2015).

Tehokkuuden ja olio-ohjelmoinnin periaatteiden näkökulmasta yhden lähettimen tulisi lähettää vain yhdennäköisiä partikkeleja, joka johtaa siihen että monimutkaisempien tehosteiden luomiseen vaaditaan yleensä useita partikkelilähettimiä. Esimerkiksi räjähdystehosteen luomiseen voidaan käyttää useita eri tavoin käyttäytyviä lähettimiä, jotka alkavat ja loppuvat hieman eri aikoihin sulavan tehosteen aikaansaamiseksi. Seuraavassa on kuvattu erään räjähdysefektin kulku ja sen sisältämät lähettimet.

Räjähdys alkaa erittäin nopealla ja kirkkaalla välähdyksellä, joka on tehosteen ensimmäinen lähetin. Tehokkaimmin toteutettuna välähdys on yksi partikkeli, joka piirretään näytölle tehosteen alussa muutaman ruudun ajan. Välähdyksen jälkeen alkaa itse räjähdysosa. Tehosteen keskiosasta lähtee liikkeelle eri suuntiin nopeasti laajenevia pilvimäisiä partikkeleja, joiden väri vaihtuu niiden elinajan mukaan lähes valkoisesta kirkkaan keltaisesta oranssin, punaiseen ja harmaaseen. Värin vaihduttua harmaaseen, partikkelit himmenevät hitaasti pois näkyviltä. Yhdellä lähettimellä voidaankin värejä sopivasti käyttämällä simuloida tulenlieskat ja niistä aiheutuva savupilvi. Paineaallon luomiseen riittää yksinkertaisimmillaan yksi rengasmaisen tekstuurin omaava lähetin. Keskiosasta lähtevä nopeasti laajeneva ja himmenevä partikkeli säilyttää skaalautuessaan rengasmaisen muotonsa ja näkyy niin lyhyen aikaa ruudulla, että sen ulkonäkö ei ole erityisen tärkeä. Yleensä tehosteisiin halutaan mahdollisimman paljon näyttävyyttä pienillä partikkelimäärillä, joten räjähdystä voidaan vielä koristella esimerkiksi kipinöillä käyttäen tehosteen alkupuolella käynnistettävää suihkua, joka lähettää suurella nopeudella kirkkaasti säteileviä pisteitä eri suuntiin.

2.2 Partikkelisysteemi osana pelimoottoria

Pelimoottori on sovelluskehityksessä käytettävä kirjasto, joka on suunniteltu videopelien luomista ja ohjelmointia varten. Yleensä täysiverinen moottori sisältää pelinkehityksessä vaaditut ominaisuudet ja apuvälineet niiden toteuttamiseen. Tällaisia komponentteja ovat mm. renderöintimoottori 2D- tai 3D-grafiikan piirtämistä varten, fysiikkamoottori, äänikirjastot, verkkokirjastot sekä partikkelisysteemi ja erilaiset animointia helpottavat kirjastot. (Thorn 2011, 9-19.) Pelimoottori koostuu käytännössä ajonaikaisesta komponentista ja erilaisista pelinkehitystä tukevista työkaluista. Partikkelisysteemi voi olla pelimoottoriin sisäänrakennettu komponentti, mutta markkinoilla on myös erillisinä komponentteina myytäviä systeemejä, joiden käyttöönotto tapahtuu olemassa olevaan pelimoottoriin integroimalla. Tällainen partikkelisysteemi toimitetaan yleensä DLL-tiedostona tai lähdekoodipakettina, jolloin ohjelmakoodin kääntäminen on sovelluskehittäjän vas-

tuulla. Joidenkin systeemien mukana toimitetaan tehosteiden luomiseen käytettävä editori, jonka tallentamat tehostetiedostot voidaan lukea suoraan peliin ajonaikaisen komponentin kautta. Yksinkertaisemmissa ratkaisuissa on kuitenkin mahdollista, että tehosteet luodaan suoraan ohjelmakoodiin tai yksinkertaisempaan konfiguraatiotiedostoon.

Vaikka partikkelitehosteita alettiin nähdä elokuvissa jo 80-luvun alkupuolelta lähtien, ovat pelikäyttöön tarkoitetut reaaliaikaiset partikkelisysteemit suorituskyvyn kannalta erittäin vaativia. Tästä syystä peleissä alettiin nähdä vakavasti otettavia tehosteita vasta 90-luvun lopusta lähtien, kun NVIDIA julkaisi teknisesti ottaen ensimmäisen näytönohjaimen, GeForce 256:n, PC-alustalle (NVIDIA 2015). Näytönohjainten saapuminen markkinoille mahdollisti aivan uudenlaisten pelinkehitystekniikoiden synnyn ja laskennan siirtyminen pois prosessorilta mahdollisti puolestaan yhä monimutkaisempien partikkelitehosteiden luomisen.

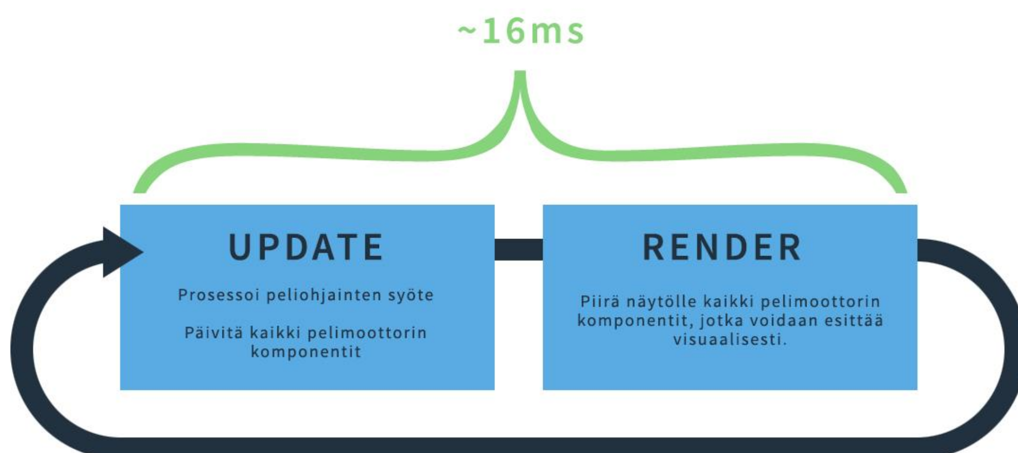
2.3 Pelimoottorin toiminta

Nykyaikaisen pelimoottorin perustana toimii lähes poikkeuksetta pelisilmukka, joka on käynnissä pelin alusta sen lopettamiseen asti. Silmukan päätarkoitus on erottaa toisistaan pelin ajallinen edistyminen ja käyttäjän syöte. (Nystrom 2014). Pelisilmukka näkyy pelaajalle parhaiten ruudunpäivityksen kautta, sillä pelin sisältö on piirrettävä näytölle säännöllisin väliajoin. Ruudunpäivityksen nopeutta kuvataan yleisesti arvolla, joka kertoo kuinka monta kertaa sekunnissa uuden ruudun piirtäminen tapahtuu. Piirtotaajuus (FPS) on keskeinen osa pelimoottorin toimintaa, sillä liian alhainen piirtotaajuus rikkoo liikkeen illuusion, kun taas rajusti vaihteleva taajuus häiritsee peliin keskittymistä.

Viime vuosien aikana 60 FPS:n päivitystaajuus on noussut eräänlaiseksi standardiksi PC-pelinkehityksessä. Ihmissilmä näkee helposti eron 30:n ja 60:n FPS:n välillä, varsinkin jos päivitystaajuuden eroja vertaillaan päivittämällä puolet ruudusta alemmalla taajuudella. Mitä korkeampi taajuus on, sitä enemmän tietokoneelta vaaditaan tehoja. 60 FPS onkin varsin optimaalinen taajuus, jossa simulaatio koetaan tarpeeksi sulavaksi tuntuakseen realistiselta. (Bakaus 2014.)

Monet nykyaikaiset pelimoottorit ovat joustavia päivitystaajuuden suhteen, joten taajuus voi kevyellä kuormituksella nousta jopa tuhansiin päivityksiin sekunnissa. Tällöin nopeammat tietokoneet pystyvät ajamaan peliä sulavammin ilman rajoituksia. Rajoittamaton päivitystaajuus tuottaa kuitenkin erilaisia haasteita pelin päivityksen synkronoinnissa, koska pelin tulee edelleen pyöriä sulavasti nopeutumatta tai hidastumatta välillä. Jokaisella päivityskierroksella lasketaan edellisestä päivityksestä kulunut aika ja päivitetään kaikki pelin osat käyttäen laskuissa hyväksi delta-aikaa (Nystrom 2015).

Pelisilmukka (Kuva 2.) on periaatteeltaan hyvin yksinkertainen, sillä jokaisella kierroksella kutsutaan päivitysmetodia, joka prosessori käyttäjän peliohjainten tai muiden syötelaitteiden kautta antaman syötteen ja tekee tarvittavat toimenpiteet. Päivityssilmukassa myös päivitetään kaikkien pelin komponenttien tila. (Nystrom 2015.) Esimerkiksi perinteisessä matopelissä pelaajan painaessa näppäimistöllä vasenta nuolinäppäintä, mato muuttaa suuntansa vasemmalle. Syötteen prosessoinnin jälkeen madon sijainti päivittyy ja lopuksi pelin kaikki osat piirretään näytölle uuteen paikkaan. Pelaajan antaessa käskyjä näppäimistöllä, ne prosessoidaan seuraavassa päivityssilmukassa. Jos kierroksen aikana ei ole tullut käskyjä syötelaitteilta, peli päivittää tilansa ilman ulkopuolisia muutoksia.



Kuva 2. Pelisilmukka koostuu kahdesta päämetodista, joiden kokonaissuoritusaika tulee olla maksimissaan noin 16 ms, jotta pelin päivitystaajuus pysyy tavoitellussa 60 FPS:ssä.

2.4 Työkalut

Tässä opinnäytetyössä toteutettava partikkelisysteemi ja sen käyttöä helpottava editori toteutetaan C#-ohjelmointikieltä ja MonoGame-ohjelmistokehystä hyödyntäen. MonoGame toimii moottorin runkona luoden yhteyden Microsoftin laitteistorajapinnan DirectX:n ja pelin välille.

Perinteisesti pelinkehityksessä on käytetty matalan tason ohjelmointikieltä, jotta ohjelmakoodi suoritetaan mahdollisimman laiteläheisesti. C ja C++ ovat todennäköisesti yleisimmät peliohjelmoinnissa käytettävät ohjelmointikielet, mutta tietokoneiden, pelikonsolien ja varsinkin mobiililaitteiden tehojen lisääntyessä myös korkeamman tason ohjelmointikielet ovat kasvattaneet suosiotaan. C# on Microsoftin vuonna 2000 julkistama yleiskäyttöinen, tyyppiturvallinen, olio-ohjelmointikieli. Sitä käytetään yleensä ohjelmoitaessa Windows-käyttöjärjestelmässä ajettavaa ohjelmakoodia, mutta koska kieli on ECMA-standardoitu, voidaan sitä ajaa millä tahansa alustalla. (Microsoft Corporation 2013, 15.) C#-koodin ajamiseen vaaditaan kullekin alustalle erikseen ohjelmoitu ajoympäristö. Windows-alustalla ympäristönä toimii Microsoftin kehittämä Common Language Runtime (CLR), mutta yksi tunnetuimpia muille alustoille suunnatuista ajoympäristöistä on avoimen lähdekoodin Mono-projekti (Albahari & Albahari 2012, 3). Projektin avulla C#-ohjelmakoodia voidaan ajaa useilla kolmannen osapuolen alustoilla, mukaan lukien Linux, OS X, Android ja iOS.

XNA Framework on Microsoftin vuonna 2006 julkaisema peliohjelmointikirjasto, jota voi käyttää .Net Frameworkin tukemilla ohjelmointikielillä. Kirjaston avulla pelinkehittäjien on mahdollista julkaista pelejä Windows-pohjaisille alustoille, Windowsille, Windows Phonelle, Xbox 360:lle ja samana vuonna XNA:n kanssa julkaistulle musiikkisoittimelle, Zunelle. Kirjasto päivitettiin viimeisen kerran syksyllä 2011 versioon 4.0 ja kaksi vuotta myöhemmin Microsoft tiedotti kehityksen lopettamisesta medialle vuotaneessa sisäisessä tiedotteessaan (Hruska 2013). XNA:n ympärille on kuitenkin vuosien aikana rakentunut aktiivinen yhteisö ja siihen pohjautuvia avoimen lähdekoodin projekteja alettiin kehittää jo ennen alku-

peräisen projektin kehityksen virallista lopettamista. Aktiivisin näistä on MonoGame, jonka ensimmäinen versio julkaistiin vuonna 2009. Projektin tarkoituksena on toteuttaa XNA Frameworkin viimeisen version ominaisuudet yksi yhteen, laajentaen alustatukea myös muille kuin Microsoftin laitteille ja käyttöjärjestelmille. Kirjastoa käytetään pääosin C#-ohjelmointikielellä, joten muilla kuin Windows-pohjaisilla alustoilla ohjelmakoodin tulkinta tapahtuu Mono-projektin ajoympäristön avulla. Windows-toteutuksessa hyödynnettävä, erityisesti peliohjelmoinnissa käytettävää DirectX-ohjelmointirajapintaa ei ole käytössä muilla alustoilla, joten projekti tukee myös OpenGL:ää, joka on Silicon Graphicsin vuonna 1992 kehittämä laitteistoriippumaton ohjelmointirajapinta (Muhammad 2013, 7). Koska MonoGamen tarkoitus on olla identtinen vastine XNA:lle, on sen nimiavaruuksien, luokkien ja metodien nimeämiset kopioitu suoraan esikuvaltaan. Näin ollen XNA Frameworkia hyödyntävä peli voidaan ihannetapauksessa muuntaa MonoGame-peliksi ilman mitään muutoksia ohjelmakoodiin.

Visual Studio

Visual Studio on Microsoftin kehittämä ohjelmankehitysympäristö, joka tarjoaa sovelluskehittäjille erilaisia tuottavuustyökaluja projektinhallinnasta, syntaksiväritykseen ja koodintäydennykseen. Ohjelman avulla voidaan luoda sovelluksia niin työpöytä- kuin web- ja mobiiliympäristöihinkin. Visual Studion kautta ajettavat projektit on mahdollista ajaa virheenjäljittäjän (debugger) kautta, jolloin ajonaikaisen virhetilanteen sattuessa kehitysympäristö näyttää tarkalleen missä kohtaa virhe tapahtui ja minkälainen virheilmoitus siitä aiheutui. Sovelluskehittäjä pääsee käsiksi virheen hetkellä vallinneeseen ohjelman tilaan, jolloin muistissa olevien muuttujien ja olioiden sisältöä voidaan tarkastella vapaasti.

Virheenkäsittelijällä on mahdollista asettaa ohjelmakoodiin editorin kautta pysäytyspisteitä (break point), jotka aiheuttavat ohjelman suorituksen siirtymisen virheenjäljittäjään (Kuva 3). Virheenkäsittelijän avulla ohjelmakoodissa voidaan liikkua rivi riviltä ja seurata ohjelman kulkua metodista ja luokasta toiseen.

```
84      public void Update()
85      {
86          if (IsActive && !IsPaused)
87          {
88              // Check if the effect has run its course
89              if (emitterCounter >= EffectLength)
90              {
91                  switch (EffectType)
92                  {
93                      case LoopMode.Loop:
94                          Reset();
95                          break;
96                  }
97              }
98
99              if (++watchCounter >= watchUpdate)
100             {
101                 watch.Reset();
102                 watch.Start();
103             }
104
105             // Go through all the emitters in the particle system and update them
106             foreach(var emitter in emitters)
107             {
108                 // If the emitter is not yet active,
109                 if (emitter.State == ParticleEmitter.EmitterState.Static)
110                 {
111                     // Check if the emitter is supposed to be activated
112                     if (emitterCounter == emitter.StartTime)
113                     {
114                         emitter.State = ParticleEmitter.EmitterState.Dynamic;
115                     }
116
117                     else { }
118                 }
```

Kuva 3. Editorissa asetettu pysäytyspiste on korostettu punaisella. Pysäytyspisteen jälkeen virheenkäsittelijällä on liikuttu manuaalisesti kaksi koodiriviä alaspäin. Seuraavaksi ajettava koodirivi on korostettu keltaisella.

2.5 Optimointi

Ohjelmistokehitysprosessissa tehtävässä optimoinnissa on tärkeää, että tehtyjen muutosten vaikutus kyetään mittaamaan mahdollisimman tarkasti. Jos käytettävissä ei ole tarkkoja työkaluja tai testausta varten kehitettyjä ohjelmia, ei silmämääräisesti pysty mitenkään määrittelemään millisekuntien heittoa suuntaan tai toiseen. Oikeassa kohdassa suunnitellusti tehdyt optimoinnit erottavat hyvän ja erinomaisen pelin tai sovelluksen toisistaan. Liian aikaiset optimoinnit (premature optimization) saattavat tehdä ohjelmakoodista erittäin vaikealukuista ja ylläpidonkannalta haastavaa. Aikaisin tehdyt optimoinnit voivat myös osoittautua turhiksi jos niiden kohteena olleet komponentit muuttuvat liikaa, tehden kaiken työn turhaksi.

Suorituskyvyltään tehokkaita ja helposti ylläpidettäviä sovelluksia kehitettäessä on hyvä hyödyntää yleiskäyttöisiä, käytännössä toimivia, uudelleenkäytettäviä ja oliosuuntautuneita ohjelmointilähtökohtia, eli suunnittelumalleja (Eriksson & Penker 2000, 221). Suunnittelumallit rakentuvat pitkällä aikavälillä kertyneen käytännön kokemuksen kautta saatuun tietoon jonkin oliopohjaisen suunnittelun tavanomaisesta tilanteesta (Gamma, Helm, Johnson & Vlissides 2003).

Object pool

Object pool (objektiallas) -suunnittelumallin tarkoituksena on parantaa suorituskykyä ja muistinkäyttöä uudelleenkäyttämällä olioita ennalta määrätyn kokoisesta ”altaasta” sen sijaan, että olioita (objekteja) luotaisiin ja tuhottaisiin yksitellen. (Nystrom 2015) Peliohjelmoinnissa mallin antama hyöty tulee hyvin esille, sillä muistinkäyttö jopa tuhansia olioita käsiteltäessä on merkittävässä roolissa pelin sujuvuuden kannalta. Suuri ero suorituskyvyssä havaitaan myös ohjelmointikielissä, joissa automaattinen muistinhallinta tekee manuaalisen muistinhallinnan vaikeaksi tai jopa mahdottomaksi ja hidastaa ohjelman suoritusta. C# ja Java ovat peliohjelmoinnissa käytettäviä kieliä, joiden muistinhallintaan kuuluva roskienkeruuoperaatio vaikeuttaa tasaisen päivitystaajuuden ylläpitoa. Alemman tason ohjelmointikielissä kuten C++:ssa muistialueiden varaaminen ja vapauttaminen voidaan tehdä manuaalisesti, jolloin peliohjelmoijan ei tarvitse varautua automaattisten työkalujen aiheuttamiin ongelmiin. (Wikibooks.org 2015, 32.)

C-kielestä eroten C#:ssa olio tuhoutuu vasta kun roskienkeruu (Garbage Collection) päättää vapauttaa sen omistaman muistialueen muiden käyttöön. Ennen muistialueen vapautusta, roskienkerääjä tarkistaa kaikki mahdolliset viittaukset objektiin ja jos yksikin viittaus löytyy, objekti jätetään muistiin eikä muistipaikkaa vapauteta. (Albahari & Albahari, 490–491.) Roskienkeruu ei tuota suuria ongelmia toisaikaisesti toimivissa liiketoimintasovelluksissa, sillä niiden suorituskyky ei yleensä kärsi merkittävästi muutaman millisekunnin viiveestä jonka kyseinen proseduuri aiheuttaa. Peli- ja simulaatiokäytössä ylimääräinen viive päivityssyklissä

aiheuttaa helposti ongelmia, joten moottorin suunnittelussa tulee lähtökohtaisesti minimoida roskienkeruun tarve.

Ratkaisuna ongelmaan on luoda ja initialisoida valmiiksi suuri määrä olioita, jotka käytön jälkeen siirretään tuhoamisen sijaan omaan listaansa uudelleenkäytettäväksi ja alustetaan tarvittaessa. On myös mahdollista sisällyttää altaassa olevaan luokkaan jonkinlainen kenttä, joka kertoo sen käytön loppumisesta, mutta geneerisessä ratkaisussa tulee olla mahdollista tallettaa minkä tahansa tyyppisiä olioita altaaseen. Käytännössä tämä tarkoittaa sitä, että jos pelissä käytetään objektiallasta esimerkiksi metsän puiden säilyttämiseen muistissa, tulee metsän luomisessa varata tila sadoille tai tuhansille olioille. Altaan luomisessa ei vielä alusteta yhtään sen sisältämää luokan instanssia, vaan niitä otetaan käyttöön sitä mukaa kun metsän puita tulee pelaajan näkyviin ja niitä piirretään. Kun käyttäjä siirtyy metsässä paikasta toiseen, vapautetaan näkyvistä kadonneiden puiden oliot takaisin käytettävien listalle odottamaan uudelleenalustamista.

Objektiallas voidaan tarpeesta riippuen toteuttaa staattisena tai dynaamisena, eli altaassa olevien olioiden lukumäärää voidaan joissain tapauksissa kasvattaa ajon aikana. Jos altaassa olevien olioiden lukumäärä on staattinen ja pelimoottorin pitäisi alustaa kapasiteettia suurempi lukumäärä objekteja, joudutaan objektien alustaminen joko keskeyttää ja odottaa uusien vapautumista, tai uudelleenalustaa ”vanhimpia” olioita niin, että ne katoavat näytöltä kesken kaiken. Kummassakin ratkaisussa on omat huonot puolensa, joten jos käyttökohde niin sallii, voi altaasta tehdä dynaamisen. Tällöin objektialtaaseen voidaan sen luomisenkin jälkeen lisätä uusia olioita tarpeen vaatiessa.

3 PARTIKKELISYSTEEMIN TOTEUTUS

Tässä opinnäytetyössä toteutettu partikkelisysteemi pyrittiin toteuttamaan hyödyntäen olemassa olevaa teoriaa niin partikkelisysteemien kuin olio-ohjelmoinnin perusteidenkin osalta. Systeemin rakenne on suunniteltu niin, että jatkokehitys ja laajentaminen on mahdollista erilaisten rajapintojen ja abstraktioiden ansiosta. Partikkelisysteemin luomisessa otettiin myös huomioon vaatimus helposta liittämisestä olemassa olevaan pelimoottoriin.

3.1 ParticleSystem

ParticleSystem on luokka, joka sisältää kaikki pelissä käytettävät tehosteet, toimien liittymäkohtana pelimoottorin ja partikkelitehosteiden välillä. Se sisältää tarvittavat toiminnot tehosteiden lataamiseen, luomiseen sekä päivittämiseen ja piirtämiseen. Pelin suorituksen aikana luokan kaksi tärkeintä metodia kontrolloivat kaikkien partikkelitehosteiden päivitystä ja piirtämistä. Seuraavassa on pelisilmukan kaksi päävaihetta kuvattuna ParticleSystem-luokan näkökulmasta:

1) Update-metodissa ParticleSystem-luokan pääasiallinen tehtävä on välittää päivityskäsky kaikille sen sisältämille tehosteille.

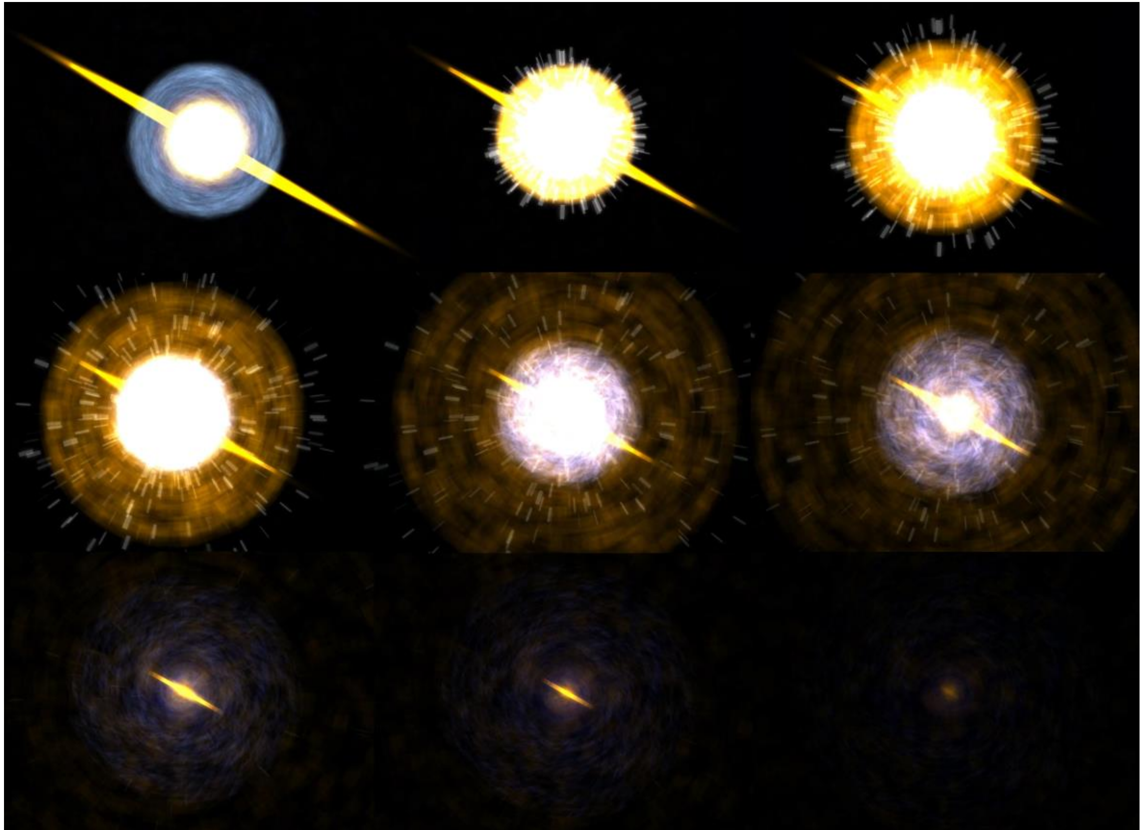
2) Draw-metodi huolehtii partikkelisysteemin näkymisestä näytöllä, joten pääohjelman on kutsuttava sitä aina kun partikkelitehosteet piirretään näytölle. Piirtokäskyn mukana toimitetaan parametrina Graphics-luokan instanssi, jonka ParticleSystem-luokka välittää eteenpäin kaikille sisältämilleen tehosteille. Graphics-olion avulla yksittäiset partikkelisysteemin tehosteet piirtävät sisältämänsä partikkelit näytölle.

3.2 ParticleEffect

ParticleEffect on luokka, joka sisältää tiedot yhden tehosteen esittämiseksi ja simuloimiseksi. Yksittäinen instanssi luokasta voi olla esimerkiksi pelissä näytettävä sade-efekti, jolloin luokka sisältää tehosteessa tarvittavat tekstuurit (graafinen esitys), yksittäisten partikkelisuihkujen ajoitukset sekä logiikan tehosteen ajamiseksi. ParticleEffect-luokan tarjoaman rajapinnan kautta partikkelisysteemin käyttäjille annetaan mahdollisuus käynnistää, tauottaa ja lopettaa tehosteen suoritus. Käyttäjä voi myös ladata valmiita tehosteita tiedostosta tai luoda manuaalisesti uusia tehosteita ohjelmakoodin kautta.

Partikkelisysteemiin on toteutettu kolme ajoitukseltaan erilaista tehostetyyppiä, joista kukin sopii omanlaistensa tehosteiden simuloimiseen. Seuraavassa on lyhyt kuvaus tehosteista, niiden eroista ja käyttökohteista:

- Once-tyyppi on varattu määrättyinä ajankohtina toistettaville tehosteille. Esimerkiksi räjähdysanimaatio (Kuva 4) näytetään vain silloin kun ruudulla tuhoutuu jokin objekti. Kun tehoste on ajettu alusta loppuun, aloitetaan sen toisto uudestaan vasta uuden räjähdyksen tapahtuessa.
- Loop-tyyppiä käytetään joidenkin monimutkaisempien tehosteiden kanssa, kun tehosteen halutaan jatkavan toistoa kunkin animaatiokierroksen jälkeen. Tehoste jatkaa käynnistämisen jälkeen partikkelitehosteen toistamista niin kauan kunnes suoritus tauotetaan tai lopetetaan kokonaan. Kierrosten välissä voidaan myös määrittää pidettäväksi tietyn mittainen tauko millisekunneissa.
- Continuous-tehostetyyppi sopii erityisesti pitkään kestäviin, verraten yksinkertaisiin tehosteisiin. Esimerkiksi pelin taustagrafiikoiden päälle piirrettävä lumisade on hyvä esimerkki Continuous-tyypin tehosteesta. Lumisade alkaa tehosteen käynnistyessä ruudun yläreunasta, eikä se vaadi mitään hienosäädettyjä ajoituksia tai lukuisia partikkelisuihkuja. Tästä johtuen sitä ei tarvitse käynnistyksen jälkeen enää hallinnoida, vaan tehoste elää omaa elämäänsä sen pysäyttämiseen asti.



Kuva 4. Opinnäytetyössä toteutetulla partikkelisysteemillä luodusta räjäytysanimaatiosta otettuja ruutukaappauksia.

3.3 ParticleEmitter

ParticleEffect-luokan sisältämät partikkelilähettimet (ParticleEmitter) toteutettiin mahdollisimman joustaviksi ja laajennettaviksi, jotta monipuolisten tehoisteiden luominen olisi mahdollista. ParticleEmitter-luokan tehtävinä on yleisen partikkelisysteemimallin mukaisesti luoda uusia partikkeleja, päivittää aiemmin luodut partikkelit, piirtää kaikki elossa olevat partikkelit näytölle sekä palauttaa partikkelisysteemiä varten toteutettuun objektialtaaseen elinkaarensa lopussa olevat oliot. Suorituskyvyn kannalta kyseessä on koko moottorin kriittisin osa, sillä kaikkein aikaavievimmät toimenpiteet tehdään luokan kautta. Esimerkiksi partikkelien päivityksessä yhdenkin millisekunnin nopeutus kokonaisajassa mahdollistaa useiden satojen partikkelien lisäämisen systeemiin.

Satunnaisuus partikkeleja lähetettäessä on toteutettu minimi- ja maksimiarvoilla, jotka määrittelevät selkeät rajat kunkin ominaisuuden satunnaisuudelle. Partikkelitehosteita luova henkilö voi esimerkiksi määritellä alkunopeuden 0–100, jolloin partikkelien alkunopeudet jakaantuvat oletuksena tasaisesti määritellylle välille. Satunnaislukujen luomiseen on käytetty .Net Frameworkin sisältämää Random-luokkaa, jonka avulla on mahdollista arpoa niin kokonaislukuja kuin desimaalilukujakin. Jos mitään satunnaisuutta ei haluta, tulee minimi- ja maksimiarvot asettaa saamaksi. Tällöin kaikki luodut partikkelit sisältävät kyseisen arvon halutun ominaisuuden kohdalla.

Työssä haluttiin toteuttaa mahdollisimman monikäyttöinen partikkelilähetin, joten pelkän kaksiulotteisessa koordinaatistossa sijaitsevan pisteen käyttö oli liian suuri rajaus haluttujen tehosteiden luomisen kannalta. Lähetintä varten luotiinkin useita erilaisia muotoja, joita käyttämällä partikkelien syntypaikkaa voidaan muuttaa tarpeen mukaan. Uusia muotoja on mahdollista ohjelmoida käyttämällä sitä varten tehtyä rajapintaa, joten taulukossa 1 kuvatut lähettimen muodot ovat vain systeemin mukana tulevat oletusmuodot.

Taulukko 1. Partikkelilähettimen muodot.

Tyyppi	Selite
Piste	Geometriassa piste on yksinkertaisin olio, jolla muut oliot voidaan määritellä (Etälukio 2015). Sen sijaintia kaksiulotteisella tasolla kuvataan x- ja y-koordinaateilla ja kaikki pistemäisen lähetysalustan synnyttämät partikkelit saavat alkusijainniksi tämän kyseisen pisteen koordinaatit.
Viiva	Suora on kahden pisteen välille määritelty jana, jota käytettäessä luodut partikkelit saavat alkukoordinaattinsa jonkin janalla olevan pisteen mukaan. Partikkelisuihkun lähetyssuunta voidaan määrittää olevan kohtisuorassa viivan kulloinkin vallitsevaa suuntaa kohtaan. Tällöin partikkelien lähtösuunta päivittyy aina sen mukaan mihin viiva kulloinkin osoittaa. Oletuksena partikkelit luodaan satunnaisiin pisteisiin viivan varrelle, mutta ne voidaan myös määrittää syntyväksi tasaisin välimatkoin toisistaan. Jos esimerkiksi keralla luodaan kymmenen partikkelia, sijoitetaan ensimmäinen viivan alkupisteeseen, jonka jälkeen kukin uusi partikkeli sijoittuu kymmenesosan viivan pituudesta kauemmaksi edellisestä partikkelista.
Ellipsi	Ellipsi on monipuolinen lähetysalusta, sillä sitä käytettäessä partikkelit voivat saada useita erilaisia alkukoordinaatteja asetuksista riippuen. Partikkelien luonti voidaan rajoittaa ellipsin sisällä oleviin pisteisiin, tai ellipsin määrittelevän käyrän pisteille. Viivan tavoin ellipsissä voidaan määrittää erilaisia lähtösuuntia ja pisteitä.
Suorakulmio	Suorakulmio on alue, jonka sisällä tai reunoilla voi syntyä partikkeleja. Se on toiminnaltaan pitkälti ellipsin kaltainen, erojen ollessa lähinnä lähetysalustan muodossa.

3.4 ParticleFactory

Uusien partikkelien luonti on kaikkien mahdollisten aloitusarvojen ja lähetysalustojen vuoksi varsin monimutkainen prosessi. Se on myös oma selkeä kokonaisuutensa, joka olio-ohjelmoinnin perusperiaatteiden vuoksi on projektissa erillisenä luokkana.

Käytännössä ParticleEmitter-luokka tarkistaa jokaisella päivityskierroksella uusien partikkelien luomisen tarpeen ja sen onko partikkeleja jo partikkelisuihkulle määritelty maksimimäärä liikkeellä. Jos partikkeleja tarvitaan lisää, lähettää ParticleEmitter kutsun ParticleFactory-luokalle (partikkelitehdas). Metodikutsun mukana välitetään haluttu partikkelien lukumäärä, jonka perusteella partikkelitehdas initialisoi partikkelit tietyillä aloitusarvoilla uudelleenkäytettäviksi objektialtaan kautta.

ParticleFactory-luokalla on vain yksi käyttötarkoitus, mutta sen toiminnallisuus on mietitty pitkälle hyvän suorituskyvyn saavuttamiseksi. Kukin partikkeli tarvitsee syntyessään ominaisuuksilleen useita aloitusarvoja, joiden asettaminen tehokkaasti ja mahdollisimman vähän resursseja käyttäen on haastavaa. Esimerkiksi satunnaista aloitusväriä laskiessa ohjelmakoodissa luodaan kolme satunnaislukua punaiselle, vihreälle ja siniselle (RGB) värikanavalle, joiden avulla XNA:n Color-luokasta luodaan uusi instanssi, joka taas asetetaan partikkeliobjektin väriarvoksi. Käytännössä jokainen satunnaisluvun ja luokkainstanssin luominen on oma aikaa ja resursseja vievä toimenpiteensä, joita halutaan tehokkuuden maksimoimiseksi mahdollisimman vähän ja harvoin. Ohjelmakoodissa onkin muutamia optimointivaiheessa tehtyjä ratkaisuja, jotka keventävät luomisprosessia merkittävästi.

3.5 Vaikuttimet

Projektin alkuvaiheissa partikkelit kulkivat luomisensa jälkeen varsin yllätyksettömästi ennalta määrättyyn suuntaan elinkaarensa loppuun asti. Yksinkertaisissa

tehosteissa tämä rajoitus ei aiheuttanut ongelmia, mutta jos partikkelisysteemin haluttiin reagoivan esimerkiksi sen isäntänä toimivan pelimoottorin tapahtumiin, ei se ollut järkevällä tavalla mahdollista. Oli siis selvää, että partikkelien kulkuun ja ominaisuuksiin tulisi voida puuttua jonkinlaisen rajapinnan kautta vielä niiden luomisen jälkeenkin.

Modifier-luokat eli vaikuttimet ovat objekteja, joita voidaan rajattomasti liittää olemassaolevaan ParticleEffect- tai ParticleEmitter-instanssiin. ParticleEffectiin liitettäessä vaikuttimet muokkaavat tehosteen sisältämiä partikkelisuihkuja, kun taas ParticleEmitteriin liitettäessä vaikutuksen alaisena ovat yksittäisen partikkelisuihkun sisältämät partikkelit. IEmitterModifier- ja IParticleModifier-rajapinnat kuvaavat kunkin vaikuttimen perustoiminnot jotka niistä periytyvien luokkien tulee toteuttaa.

IParticleModifier

IParticleModifier on ParticleEmitteriin liitettävä rajapinta partikkeleja muokkaavalle vaikuttimelle. Se sisältää vain kaksi metodia, jotka rajapinnan toteuttavien luokkien on tarjottava käyttöön. Update-metodi vastaanottaa parametrina viitteen Particle-olioon ja muokkaa sitä halutulla tavalla. Lähettimen sisältämiä vaikuttimia kutsutaan jokaisen partikkelin päivityksen lopuksi, joten vaikuttimet vaikuttavat partikkeleihin jokaisella päivityskierroksella.

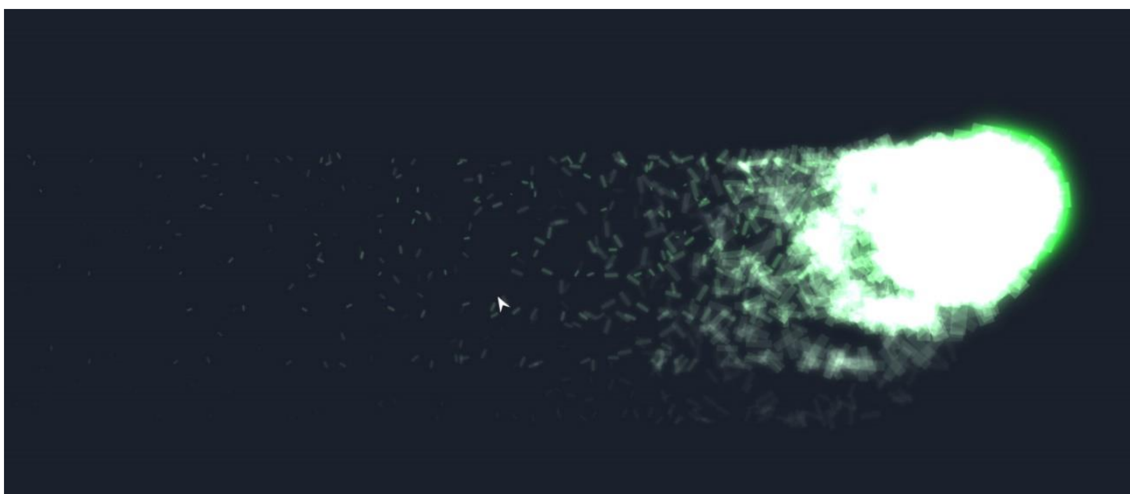
UpdateModifier-metodi mahdollistaa sen, että kun kaikki partikkelit on prosessoitu vaikuttimen läpi, vaikutin voi päivittää omat arvonsa. Esimerkkinä tästä on tuulivaikutin, joka simuloi satunnaisesti muuttuvaa tuulenvirettä. Jokaisella päivityskierroksella tuulivaikutin puhaltaa partikkeleja satunnaisella voimalla satunnaiseen suuntaan ja päivityskierroksen jälkeen se päivittää itsensä muuttamalla tuulen kovuutta ja suuntaa (Kuva 5). Vaikuttimet toimivat niiden toteuttaman rajapinnan puitteissa, joten ne voivat sisältää mitä tahansa lisädataa, kunhan pakolliset Update- ja UpdateModifier-metodit on toteutettu. Tämän työn puitteissa projektiin

on luotu kolme IParticleModifier-rajapinnasta periytyvää partikkelivaikutinta, joista yksi on edellä kuvattu tuulivaikutin.

Kuva 5. Lumisade-efekti on toteutettu tuulivaikutinta hyödyntäen.

GravityModifier on esimerkki yksinkertaisesta vaikuttimesta, jolla on varsin suuri merkitys partikkelisysteemissä. Suuressa osassa pelimaailmoista on jonkinlainen painovoima ja partikkelitehosteiden tulee todennäköisesti totella samoja fysiikan lakeja muiden peliobjektien kanssa. Partikkelisysteemin ydinosasta on tarkoituksella jätetty pois painovoiman kaltaiset tapauskohtaisesti tarvittavat ominaisuudet, koska pelinkehittäjiä ei haluta sitouttaa yhteen tiettyyn toteutukseen. Tämän vuoksi systeemin mukana tarjotaan erillisenä irtonaisena osana yksinkertainen toteutus painovoimavaikuttimesta. Vaikuttimen ainoa määriteltävä kenttä on painovoiman voimakkuus, jota käytetään hyväksi partikkelien nopeusvektorin muokkaamisessa. Partikkelin painon ollessa suurempi sen nopeus ruudun alaosassa sijaitsevaa maata kohti kasvaa nopeammin kuin kevyemmän partikkelin. UpdateModifier-metodi on painovoimavaikuttimessa tyhjä, koska painovoima ei oletettavasti koskaan muutu. Vaikuttimen ei siis ole tarvetta päivittää itseään päivityssyklien välissä.

MouseModifier on monimutkaisempi esimerkki siitä, mitä kaikkea vaikuttimilla voi tehdä. Kun kyseinen vaikutin on käytössä, käyttäjä voi painaa hiiren vasemman napin pohjaan jossain kohtaa näyttöä ja kaikki lähettyvillä olevat partikkelit alkavat siirtyä partikkelin painosta riippuen määrätyn suuruisella kiihtyvällä liikkeellä hiiren osoittimen suuntaan (Kuva 6). Vetovoiman vaikutus katoaa heti, kun hiiren painike päästetään irti.

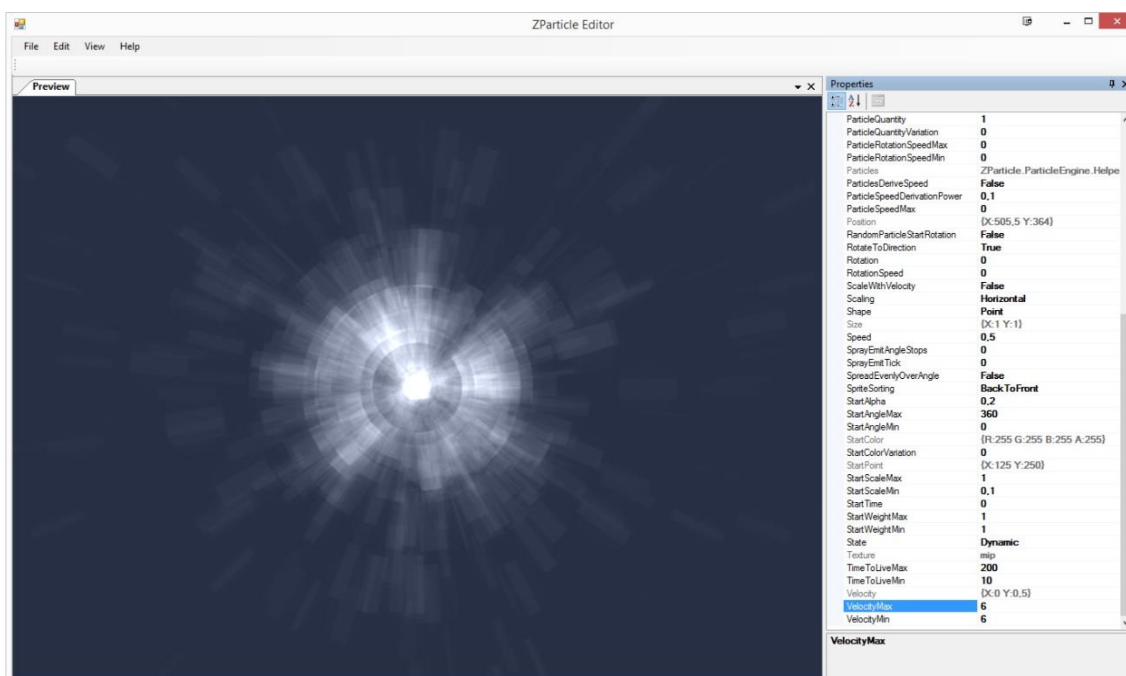


Kuva 6. MouseModifier vetää partikkeleja hiiren osoittimen suuntaan.

Vaikuttimien avulla partikkelisysteemin toimintaa voidaan laajentaa helposti koskematta ollenkaan sen kriittisiin osiin. Ne ovat verrattavissa työpöytäsovelluksista tuttuihin liitännäisiin, jotka toimivat vuorovaikutuksessa isäntäsovelluksen kanssa. Vaikuttimia käytettäessä rajoitteena on lähinnä pelinkehittäjän mielikuvitus, sillä partikkeli- ja partikkelilähetinobjektit ovat julkisten ominaisuuskenttien kautta täysin kehittäjän hallittavissa. Jatkokehityksessä on tarkoitus entisestään lisätä vaikuttimien mahdollisuuksia puuttua esimerkiksi piirtometodissa tapahtuviin asioihin.

4 PARTIKKELIEDITORI

Tässä opinnäytetyössä toteutettiin Windows-käyttöjärjestelmälle editori partikkelisysteemin testausta ja tehosteiden luomista varten (Kuva 7). Editori on Windows Forms -käyttöliittymäkirjastoa hyödyntävä C#-ohjelma, ja se koostuu irrallisista osista, joita käyttäjä voi halutessaan järjestää uudelleen (docking). Kyseinen komponenttien osittelu on tuttu monista ohjelmointiympäristöistä, kuten Visual Studiosta ja Eclipsestä sekä Photoshopin ja After Effectsin kaltaisista graafisen- ja videosuunnittelun työkaluista. Ohjelman sulkeutuessa kunkin ikkunan sijainti ja koko tallennetaan kovalevylle, jotta seuraavalla avauksella kaikki osiot ovat samassa paikassa kuin mihin ne edellisellä kerralla jäivät. Näin kukin editoria käyttävä henkilö pystyy kustomoimaan sen oman mielensä mukaisesti. Irralliset osiot myös mahdollistavat tehokkaan työskentelyn usean näytön kanssa. Esimerkiksi kolmen näytön kanssa työskentelevä henkilö voi pitää partikkelisysteemiä esittävän livenäkymän mahdollisimman suurena keskimmäisellä näytöllä ja ominaisuusluetteloa sekä muita ikkunoita sivunäytöille sijoiteltuina.



Kuva 7. Partikkelitehoste muokattavana partikkelieditorissa.

4.1 Livenäkymä

Editorin tärkeimmät komponentit ovat livenäkymä ja ominaisuuseditori. Livenäkymän avulla artisti näkee reaaliaikaisesti parikkelisysteemiin ominaisuusikkunan kautta tehdyt muutokset, jolloin tehosteen muokkaus on nopeaa ja tarkkaa. Näkymän pohjana toimii yksinkertainen MonoGamen päälle rakennettu pelimoottori, jonka kautta sen sisältävää partikkelisysteemiä voidaan muokata. Livenäkymän toteutuksessa suurin haaste oli pelimoottorin silmukan ja piirtorajapintojen liittäminen Windows Forms -kontrollien toimintalogiikkaan, sillä MonoGamen pelinäkymää ei ole tarkoitettu pelisilmukkatoteutuksensa vuoksi piirrettäväksi tapahtumapohjaisen Windows-ohjelman sisällä.

4.2 Ominaisuusikkuna

Ominaisuusikkuna on näkymä, jonka kautta livenäkymän partikkelisysteemiä voidaan muokata. Toteutuksen pohjana toimii .Net Frameworkin mukana tuleva PropertyGrid-kontrolli, joka tarjoaa rajapinnan minkä tahansa objektin julkisiin kenttiin. Linkitys PropertyGridin ja siinä valittuna olevan objektin välillä on automaattinen, joten mikä tahansa muutos päivittyy heti itse kohteeseen. Partikkelimoottorin tapauksessa kontrollia on laajennettu tukemaan erilaisia MonoGame:n tyyppejä, jotka on lueteltu taulukossa 2.

Taulukko 2. Ominaisuusikkunan laajennettu tyyppituki.

Tyyppi	Selite
Color	Partikkelien värit ovat Microsoft.XNA.Graphics.Color-tyyppisiä olioita, joiden määrittely editorissa on helpointa tehdä klassisen värivalintakontrollin (color picker) avulla. PropertyGrid tukee oletuksena .Net Frameworkin Color-objektia, mutta MonoGamen vastaava objekti on nimestään huolimatta eri tyyppinen, eikä kontrolli osaa oletuksena esittää sitä muokattavassa muodossa.

Position	Position kuvaa tehosteen sijaintia näytön vasempaan yläkulmaan (koordinaatit 0,0) nähden. Vastaavaa editorikontrollia käytetään myös muissa kentissä, joissa halutaan määrittää jokin piste kaksiulotteisella tasolla. Tällä hetkellä editointi tapahtuu kahden x- ja y-koordinaatteja vastaavien tekstiruutujen kautta, mutta myöhemmässä versiossa on tarkoitus toteuttaa kontrolli jolla haluttu piste voidaan valita eräänlaisen koordinaattikentän kautta hiirellä osoittamalla ja klikkaamalla.
StartAngle EndAngle	Oletuksena partikkelilähetin sinkoaa partikkeleja joka suuntaan, mutta systeemi tukee myös partikkelien lähettämistä tietylle sektorille. Sektorin voi määrittää graafisesti ympyränmuotoista sektorivalitsinta käyttäen, tai kirjoittamalla tekstiruutuihin asteissa sektorin rajaavat alku- ja loppukulmat.

4.3 Tiedostoformaatti

Partikkelieditorin tarkoituksena on toimia niin testialustana kuin pelissä käytettävien tehosteiden luontityökalunakin. Jotta editorissa luotuja tehosteita voidaan käyttää uudelleen jossain toisessa käyttökohteessa, tulee ne voida tallentaa kovalevylle tiedostoksi. Tehosteiden tallennus ja avaaminen on toteutettu partikkelisysteemin sisällä .Net Frameworkissa valmiina olevilla objektien serialisointia varten kehitetyillä kirjastoilla. Editorissa luodut tehosteet tallennetaan XML-formaattiin pääluokan (ParticleEffect) ja kaikkien sen sisältämien alaluokkien julkiset kentät serialisoimalla. Itse partikkelitehosteen luonnetta kuvaavien ominaisuuksien lisäksi tallennustiedosto sisältää kunkin partikkelilähettimen sisältämän tekstuurin, joten tehosteen jakamiseen ja käyttöönottoon riittää vain yhden tiedoston siirtäminen paikasta toiseen.

4.4 Serialisoinnin ongelmat

Vaikka .Net Framework sisältää erittäin laajan tuen erilaisten tietotyyppien serialisoimiselle, ei se siltikään tue monimutkaisempien kolmansien osapuolien luokkien serialistointia. Tästä johtuen esimerkiksi partikkelitekstuurien serialisointi ja deserialistointi tuli ohjelmoida erikseen partikkelieditorin tarpeita varten. Ilman serialisointituen lisäämistä tekstuureja ei olisi voinut jakaa osana tehosteen sisältämää XML-tiedostoa, vaan ne olisi tullut säilyttää esimerkiksi samassa kansiossa itse tehostetiedoston kanssa. Mielenkiintoiseksi kuvatiedostojen serialisoinnin tekee se, että MonoGamea varten niiden tulee olla jo valmiiksi sen tukemassa muodossa, eli XNB-tiedostona.

5 OPTIMOINTI

Partikkelisysteemin perustoiminnallisuuden valmistuttua suuri osa kehitystyöstä keskittyi systeemin eri komponenttien optimoimiseen. Muistinhallinnan sekä prosessorin ja näytönohjaimen käytön suhteen tehokkaampi ohjelmakoodi mahdollistaa monimutkaisempien tehosteiden luomisen pienemmällä koneteholla. Eräs mielenkiintoisimmista tehokkuutta lisäävistä optimoinneista on muistinkäyttöä ja roskienkeruuta merkittävästi vähentävä objektiallas-suunnittelumalli, josta tehtiin projektin tarpeisiin sopiva toteutus.

Joissain tapauksissa tarvittavien partikkelien määrä on mahdoton arvioida etukäteen, joten ennalta allokoitu partikkeliolioiden lukumäärä objektialtaassa saattaa osoittautua tehosteen intensiivisimmässä vaiheessa liian vähäiseksi. Tämän työn tarkoituksena oli luoda mahdollisimman hyvin tilanteeseen mukautuva partikkelisysteemi, joten niissä tapauksissa kun alussa allokoitujen olioiden määrä ei vastaa tarvetta, suurennetaan objektialtaan kokoa lennossa. Näin ollen vapaiden partikkelien loppuessa luodaan uusia olioita tarpeen mukaan ja lisätään ne vapaana olevien partikkelien listaan. Vaikka dynaamisella objektialtaalla ratkaistaan edellä kuvattu ongelma, on uusien olioiden luominen kuitenkin vaativa toimenpide suorituskyvyn kannalta. Onkin parasta arvioida tarvittava partikkelimäärä mahdollisimman tarkasti ja mieluummin ylimitoitetusti, jottei tule vastaan tilannetta jossa luodaan suuria määriä olioita yhden päivityskierroksen aikana. Paljon partikkeleja vaativan tehosteen luonnissa tulee välttää liian pienen muistialueen varaamista, sillä objektialtaan laajentaminen ja uusien partikkelien alustaminen hidastaa simulaatiota, jonka seurauksena objektiallas-suunnittelumallin hyöty menetetään hetkellisesti.

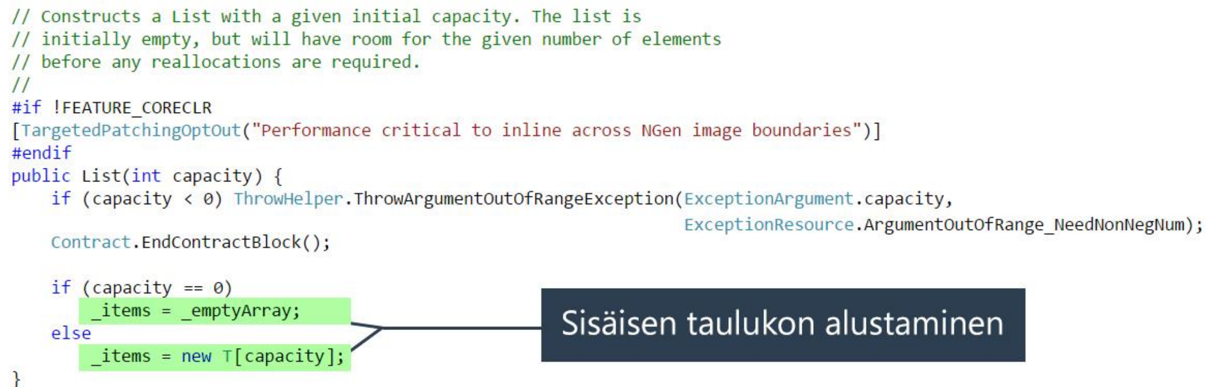
5.1 Object Pool

Partikkelisysteemin käyttöön toteutettu objektiallas (ParticlePool) sisältää kaksi eri oliolistaa, joiden nimet ovat Pool ja InUse. Pool on lista, joka sisältää käytettävissä olevat vapaat partikkelioliot. Oliot ovat joko käyttämättömiä ja valmiiksi alustettuja, tai aiemmin käytettyjä mutta vapaiksi merkittyjä. Elinkaarensa päässä olevat partikkelit merkitään vapautuneiksi ja siirretään Pooliin odottamaan uudelleenkäyttöä. InUse on lista, jonka partikkelioliot ovat parhaillaan käytössä ja näkyvät ruudulla. Listojen lisäksi luokka sisältää metodeja, joilla objektiallas-suunnittelumallin toiminnallisuus toteutetaan.

ParticlePool()

Luokan rakentaja [*ParticlePool(int initialCapacity, int initialCount)*] vastaanottaa parametrina kaksi kokonaislukuarvoa. InitialCapacityn arvoa käytetään molempien partikkelilistojen kapasiteetin alustamiseen. Vaikka .Net Frameworkissa List-luokka on dynaamisesti kasvava taulukko, on sen kapasiteetin varaamisesta hyötyä suorituskyvyn kannalta. List-luokan sisäinen toteutus hyödyntää taulukkoa, joka on kooltaan staattinen (Kuva 8), joten jokainen alkion lisääminen listaan vaatii listan sisäisessä käytössä olevan taulukon uudelleenalustamista (Microsoft Corporation 2015). Kun kaikki taulukon alkiot sijaitsevat muistissa peräkkäin, on niiden arvojen hakeminen nopeampaa, koska prosessori tallentaa välimuistiinsa jokaisella lukukerralla jonkin verran kohteena olevaa muistialuetta edeltävää ja seuraavaa aluetta. Prosessorin ei siis tarvitse jokaisella lukukerralla lukea dataa RAM-muistista, vaan se voi löytää kyseisen muistiosoitteen sisällön omasta välimuististaan. (Toub, Ostrovsky & Yildiz 2008.) Näin ollen eheiden, muistissa peräkkäin sijaitsevien lista- ja taulukkoalkioiden tärkeys muodostuu varsin suureksi. Mitä vähemmän prosessori joutuu kuluttamaan aikaa RAM-muistin kanssa kommunikoimiseen, sitä nopeammin partikkelisysteemi kykenee päivittämään ja piirtämään itsensä.

InitialCount-arvoa käytetään Pool-listan täyttämiseen partikkeliobjekteilla. Vaikka listat on alustettu halutulla kapasiteetilla, eivät ne sisällä yhtään oikeaa partikkelioliota vaan ainoastaan niille varatun tilan. Yleensä initialCount on sama kuin kapasiteetti, mutta erityistapauksia varten kumpikin arvo on mahdollista määritellä erikseen.

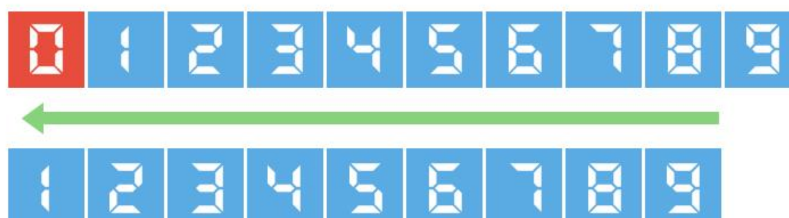


Kuva 8. List-luokka käyttää sisäisessä toteutuksessa taulukkoa (Array).

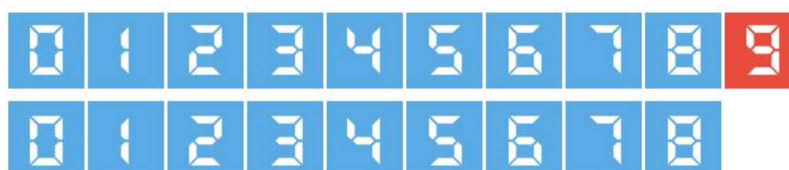
Particle GetObject()

GetObject on metodi, joka palauttaa kutsujalle vapaana olevan, tai juuri luodun ja altaaseen lisätyn partikkeliobjektin. Normaalista poiketen partikkeliobjekti palautetaan listan viimeisestä alkiosta. Tämä johtuu siitä, että .Net Frameworkin List-luokan toteutuksessa alkion poistaminen listalta johtaa luokan sisäisen taulukon uudelleenjärjestämiseen (Microsoft Corporation 2015). Jos GetObject-metodi palauttaisi ensimmäisen alkion, tapahtuisi taustalla koko listan siirtäminen yhdellä indeksillä taaksepäin (Kuva 9).

Listan alkupäästä poistettaessa, jokainen poistokohdan jälkeinen alkio siirretään yhden alkion vasemmalle.



Listan loppupäästä poistettaessa, alkioita ei tarvitse siirtää. Operaatio on huomattavasti nopeampi ja vaatii vähemmän resursseja



Kuva 9. Ero alkion poistamisessa listan alku- tai loppupäästä.

void ReturnObjects(List<int> particlesToReturn)

ReturnObjects-metodi vastaa käytöstä poistuneiden partikkeliolioiden palauttamisesta altaaseen. Parametrina vastaanotetaan lista kokonaislukuja, jotka ovat viittauksia käytössä olevan partikkelilistan alkioihin. Metodin sisällä kunkin indeksin kohdalla oleva partikkeliolio poistetaan InUse-listalta ja lisätään takaisin altaaseen. ParticleEmitter on ainoa luokka, joka kutsuu tätä metodia, sillä sen tehtäviin kuuluu käytöstä poistuneiden partikkelien palauttaminen altaaseen.

void DisposeAllObjects

DisposeAllObjects on metodi, jota kutsutaan kun ParticleEmitter ja sen sisältämä partikkeliallas tuhotaan, eli poistetaan käytöstä. On hyvän ohjelmointitavan mukaista poistaa viittaukset kaikkiin käytöstä poistuneisiin olioihin, jotta .Net Frameworkin roskienkeruu osaa poistaa ne muistista. Sisäisesti metodi on toteutettu niin, että se kutsuu kunkin altaassa olevan olion Dispose-metodia, mutta vain

siinä tapauksessa että olio toteuttaa IDisposable-rajapinnan. Jos olio ei toteuta kyseistä rajapintaa, ei sitä voi erikseen tuhota. Roskienkeruu kuitenkin siivoaa olion muistista jossain vaiheessa, koska kaikki viittaukset siihen on poistettu.

5.2 Suorituskykytestaus

Tämän työn kohteena olevan partikkelisysteemin kehitys keskittyi pitkälti yli projektin puolenvälin vain puuttuvien ominaisuuksien ohjelmoimiseen ja olemassa olevien toimintojen vakauden varmistamiseen. Projektin ensimmäiset kunnolliset suorituskykytestaukset ja optimoinnit tehtiin vasta sitten, kun kaikki vaatimusmäärittelyssä listatut ominaisuudet olivat valmiita ja todettu toimiviksi. Yksi päivityskierros partikkelimoottorissa kestää oletuksena maksimissaan 16 ms piirtäminen mukaan laskettuna, joten kierrosajasta ei ole varaa menettää millisekuntiakaan turhan tai epäonnistuneen optimoinnin tuloksena.

Projektin pääasiallisena kehitysympäristönä toimiva Visual Studio sisältää monenlaisia työkaluja koodin staattiseen analysointiin ja suorituskykytestaukseen. Työkaluilla seurattiin projektin edistyessä sekä muistinkäyttöä että prosessorin kuormitusta säiekohtaisesti. Kun partikkelisysteemiä ajettiin jonkin aikaa analyysityökalun seuratessa haluttuja kohteita, voitiin koodin ongelmakohtiin pureutua rivi riviltä pullonkaulat löytäen. Aivan projektin alussa suorituskykyä kuitenkin testattiin varsin alkeellisin menetelmin.

Stopwatch

Yksinkertaisin tapa mitata kunkin partikkelisysteemin osion kuluttamaa aikaa on käyttää hyväksi .Net Frameworkiin sisältyvää Stopwatch-luokkaa. Stopwatch kykenee mittaamaan kulunutta aikaa yhden prosessorin aikayksikön (tick) tarkkuudella ja se on uudemmilla tietokoneilla erittäin tarkka, koska sen sisäinen toteutus käyttää Kernel32-kirjaston korkean resoluution tuloksia palauttavaa QueryPerformanceFrequency-metodia (Microsoft Corporation 2015). Tätä tekniikkaa käytet-

tiin paljon esimerkiksi yksittäisen partikkelilähettimen päivityskierroksen kulutta-
man ajan tarkasteluun. Tuloksena saatu aika oli helppo esittää kehitysympäristön Output-ikkunassa tai näyttää suoraan partikkelieditorissa. Koodin muuttaminen suorituskykytestausta varten oli kuitenkin alkuvaiheen väliaikainen ratkaisu, joten projektin myöhemmissä vaiheissa sujuvampaa testausta varten luotiin erillinen testausprojekti.

Benchmark

Benchmark on partikkelisysteemiä ja sen osia hyödyntävä testausprojekti, joka automatisoi mahdollisimman suuren osan testauksesta ja tulosten kirjaamisesta sekä analysoinnista. Vaikka Benchmark-projektin avulla on mahdollista testata moottorin yleistä toimivuutta, on sen pääasiallinen tarkoitus toimia suorituskykytestauksen työkaluna.

Projektin avulla luodut testit ajetaan automaattisesti ennalta määriteltyjä tehosteita käyttämällä ja koko testin ajalta tallennetaan tärkeitä mittaustuloksia myöhempää analysointia varten. Testituloksia tulkitsemalla voi määritellä uusimpien muutoksien vaikutuksen niin muistinkäyttöön kuin kunkin päivitys- ja piirtometodin kuluttamaan aikaan. Jotta mittaustuloksista olisi jotain hyötyä, tulee niiden pysyä perustoiminnoiltaan ja mittaustekniikoiltaan samana vaikka partikkelisysteemi itsessään muuttuukin koko ajan. Tällöin Benhcmark-projektin ensimmäisillä versioilla saadut testitulokset ovat verrattavissa uusimpien versioiden tuloksiin.

Tarkkojen testitulosten saamiseksi testiohjelman vaikutus testattavana olevaan sovellukseen tai kirjastoon on mahdollisimman pieni. Minimaalinen vaikutus on saavutettu käyttämällä datan keräämisessä tehokkaita ja tarkoitukseen sopivia tietovarastoja sekä muutamia optimointikeinoja joilla testauksesta ja seurannasta huolehtiva koodi tuottaa minimaalisen määrän roskienkeruuta vaativia olioita muistiin. Testidatan keräämisestä ja tallentamisesta vastaavat metodit on profiloitu tarkasti kolmannen osapuolen analyysityökalulla, jotta suorituskyky on tasainen ja pahimmat testattavaan systeemiin vaikuttavat pullonkaulat on korjattu.

IBenchmark

Suorituskykytestauksen perustana on IBenchmark-rajapinta, joka määrittää yhteiset pelisäännöt kaikille testeille. Opinnäytetyön puitteissa rajapinnasta on periytetty vain yksi luokka (BasicBenchmark), mutta rajapinnan käyttö mahdollistaa monimutkaisempien testityyppien luomisen tulevaisuudessa. IBenchmark määrittelee ainoastaan kaikki tarpeelliset metodit, jotta itse testejä ajava sovellus pystyy ajamaan kaikki testit läpi niiden toteutuksesta riippumatta. Seuraavassa on kuvailtu IBenchmark-rajapinnan metodit sekä tapahtumankäsittelijät ja niiden käyttö:

Taulukko 3. IBenchmark-rajapinnan metodit ja tapahtumakäsittelijät.

Metodi	Selite
Start	Testit käynnistetään testisovelluksen toimesta.
Pause	Testit ajetaan automaattisesti, mutta niiden suoritus on mahdollista keskeyttää väliaikaisesti. Tällöin kaikki testiin kuuluvat partikkelitehosteet pysähtyvät ja mittaustulosten tallennus keskeytyy.
Stop	Testit voidaan lopettaa ennen niiden loppumista. Stop-metodi huolehtii olioiden tuhoamisesta.
Update	Benchmarkin tulee kyetä mittaamaan päivityksen ajoitukset mahdollisimman tarkasti, joten kaikki pääohjelmalta tulevat päivityskäskyt kulkevat sen kautta. Benchmark ohjaa päivityskäskyn sisältämälleen partikkelisysteemille, joka arkkitehtuurin mukaisesti hoitaa kaikkien sisältämiensä olioiden päivityksen. Update-metodi mittaa Stopwatch-luokan avulla päivityskierroksen kuluttamaa aikaa ja tallentaa muistissa olevaan taulukkoon kunkin kierroksen muistinkäytön ja partikkelien lukumäärän. Muistinkäytön mittaamisessa käytetään .Net:in GarbaceCollector-luokkaa,

	jonka GetTotalMemory-metodi palauttaa sovelluksen kokonaismuistinkäytön tavuissa. Metodia kutsuttaessa forceFullCollection-parametrille annetaan arvoksi false, jotta metodi palauttaa välittömästi muistin tilan, eikä odota seuraavaa roskienkeruukierrosta.
Draw	Piirtometodissa mitataan sen suorittamiseen kuluvaa aikaa, koska työn puitteissa ei ollut mahdollista kehittää näytönohjaimen toimintaan liittyviä seurantametodeja. .Net Frameworkin oliot sijaitsevat tietokoneen RAM-muistissa, kun taas näytönohjaimelle piirrettäväksi siirretyt oliot sijaitsevat näytönohjaimen muistissa.
BenchmarkFinished	Tapahtumakäsittelijä joka ilmoittaa testin loppumisesta pääohjelmalle. Testin pituus määritellään ennen sen aloittamista päivityssyklien lukumääränä. Jos testin pituus on 600 sykliä, kestää se yleisesti ottaen noin 10 sekuntia.
GetReport	GetReport-metodi palauttaa kaikki mittaustulokset valmiiksi prosessoituina pääohjelmalle. Metodi laskee muistissa olevista datataulukoista mittaustulosten keskiarvon sekä minimi- ja maksimiarvot koko testin ajalta. Jos FullResults-kenttä on true, palautetaan datan mukana myös kaikki taulukoiden sisältämä raakadata myöhempää käyttöä varten.

6 YHTEENVETO

Työn tuloksena syntyi partikkelisysteemi ja sen hyödyntämiseen rakennettu editori Windows-käyttöjärjestelmälle. Partikkelisysteemin voi lisätä pienellä lisätyöllä MonoGamea hyödyntävään pelimoottoriin tai simulaatioon ja se tukee käytännössä myös MonoGamen esikuvana toimivaa XNA-peliohjelmointikirjastoa yhteisen nimeämiskäytännön ja identtisen toiminnallisuuden vuoksi. MonoGamen käytön myötä projekti on periaatteessa alustariippumaton, mutta testaus suoritettiin vain Windowsin työpöytäympäristössä ja Windows-pohjaisella tabletilla. Tabletilla projektin loppuvaiheessa suoritetussa testauksessa kävi ilmi, että suorituskyvyn tulee olla parempi mobiilialustoilla. Tämä on tulevaisuudessa mahdollista ratkaista erilaisilla kehittäjän asettamilla alustakohtaisilla laatuasetuksilla. Mobiililaitteella voidaan tarvittaessa rajoittaa simulaation tarkkuutta tai grafiikan näyttävyyttä, jotta systeemin toimintaan tarvitaan vähemmän tehoja isäntälaitteelta.

Työn puitteissa oli tarkoitus luoda jonkinlainen testipeli tai simulaatio jonka kautta partikkelimoottorin olisi nähnyt sen mahdollisessa käyttökohteessa. Projektin laajuus kuitenkin yllätti varsinkin editorin osalta, joten itse pelimoottoriin liittämistä ei ole työn puitteissa päästy testaamaan. Partikkelieditorissa moottori toimii eräänlaisessa hiekkalaatikossa ilman rajoituksia, mutta pelimoottorin osana sen täytyy toimia kurinalaisemmin, jotta kaikki resurssit eivät mene partikkelitehosteiden simuloimiseen ja piirtämiseen. Partikkelieditorissa saa kuitenkin hyvän kuvan siitä kuinka paljon erilaisia vaihtoehtoja ja mahdollisuuksia tehosteiden luomisessa on, ja kuinka paljon eri asetukset voivat muuttaa lopullisen tehosteen ulkonäköä.

Toimivan partikkelisysteemin luominen on varsin triviaali tehtävä, mutta tehokkuuden, laajennettavuuden ja uudelleenkäytettävyyden varmistaminen vaatii syvällisempää asiantuntemusta ja tutustumista peliohjelmoinnin ja yleensä olio-ohjelmoinnin suunnittelumalleihin. Reaaliaikaisen simulaation kanssa työskennellessä haasteeksi muodostuu myös lyhyt aikaikkuna, jonka aikana on suoriuduttava niin prosessorin kuin näytönohjaimenkin varassa olevista toimenpiteistä.

Kaikista tehtävistä suoriutuminen 16 ms:n aikana vaatii syvällisempää tuntemusta esimerkiksi .Net Frameworkin luokkakirjastoista, sillä väärin valitulla tietotyypillä voi olla suuri vaikutus pelin suorituskykyyn. Suuria objektimääriä käsitellessä niiden liikkeet niin muistin kuin prosessorinkin välillä tulee olla tarkkaan tiedossa ja otettu huomioon ohjelmakoodissa.

Partikkelisysteemin kehittäminen jatkuu vielä tämän työn palautuksen jälkeen, sillä se on testeissä ja kehityksen aikana osoittautunut rakenteeltaan helposti laajennettavaksi ja lupaavaksi peli- tai simulaatiokäyttöä ajatellen. Seuraavat kehityskohteet ovat pelimoottoriin tai ulkopuoliseen työkaluun liittämistä helpottavat rajapinnat ja niiden vaatimat muutokset.

LÄHTEET

Albahari, J. & Albahari, B. 2012. C# 5.0 In a Nutshell. Sebastopol: O'Reilly Media, Inc.

Bakaus, P. 2014. The Illusion of Motion. Viitattu 12.1.2015 https://paulbakaus.com/tutorials/performance/the-illusion-of-motion/

Eriksson, H., Penker, M. 2000. Business Modeling with UML: Business Patterns at Work. Hoboken: John Wiley & Sons, Inc.

Etälukio 2015. Pitkä matematiikka. Viitattu 30.3.2015 http://www02.oph.fi/etalukio/pitka_matematiikka/kurssi3/maa3_teoria1.html

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 2003. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley

Hruska, J. 2013. Microsoft kills Xbox 360/PC cross-platform development, declares DirectX "no longer evolving". Viitattu 14.1.2015 http://www.extremetech.com/gaming/147289-microsoft-kills-xbox-360pc-cross-platform-development-declares-directx-no-longer-evolving

Martin, A. 1999. Particle Systems. Viitattu 5.2.2015 http://web.cs.wpi.edu/~matt/courses/cs563/talks/psys.html

Microsoft Corporation 2015. Reference Source .Net Framework 4.5.2. Viitattu 23.3.2015 http://referencesource.microsoft.com/

Microsoft Corporation 2013. C# Language Specification Version 5.0. Viitattu 23.3.2015 https://msdn.microsoft.com/en-us/library/ms228593.aspx

Muhammad, M. 2013. OpenGL Development Cookbook. Birmingham: Packt Publishing

NVIDIA Corporation 2015. GeForce 256 The World's First GPU. Viitattu 26.3.2015 http://www.nvidia.com/page/geforce256.html

Nystrom, R. 2014. Game Programming Patterns. Viitattu 3.1.2015 http://gameprogrammingpatterns.com/game-loop.html

Reeves, W. T. 1983. Particle Systems - A Technique for Modeling a Class of Fuzzy Objects. Computer Graphics 17:3, 359.376

Shiffman, D. 2012. The Nature of Code. Viitattu 2.2.2015 http://natureofcode.com/book/chapter-4-particle-systems/

Thorn, A. 2011. Game Engine Design and Implementation. Burlington: Jones & Bartlett Learning, LLC

Toub, S., Ostrovsky, I., Yildiz H. 2008. False Sharing. Viitattu 25.3.2015 https://msdn.microsoft.com/en-us/magazine/cc872851.aspx

Varcholik, P. 2014. Real-Time 3D Rendering with DirectX® and HLSL: A Practical Guide to Graphics Programming. Boston: Addison-Wesley

Wikibooks.org 2015. C++ Programming. Viitattu 19.3.2015 http://upload.wikimedia.org/wikipedia/commons/e/e9/CPlusPlusProgramming.pdf